Code_Algorithms_Mitigation

June 26, 2025

Reproducing Popularity and Gender Bias in Music Recommenders with Cross-Domain Extension to Books

Team 2 Elif Deger Nataliya Kharitonova

This notebook contains the code to replicate the study by Lesota et al., including the bias mitigation techniques we employed.

In section 1 we begin by running seven recommender algorithms on the LFM-2b dataset. Some of the code was executed within Jupyter notebooks, while other parts were run locally using PyCharm. The results of each algorithm are presented at the end of their respective code sections and are collectively analysed after all seven have been executed in section 1.1

Next, bias mitigation techniques are applied to three selected algorithms in section 1.2, and analysed in section 1.3. The notebook then repeats the same process on the Book-Crossing dataset to evaluate the generalizability of the findings in section 2.

In Section 3 we provide a final comparison of both datasets and in section 4 we comment on generalizability of results from LFM-2b to Book-Crossing dataset

1 1. LAST FM DATASET

- The LFM-2b dataset used in our study is considered derivative work according to paragraph 4.1 of Last.fm's API Terms of Service (https://www.last.fm/api/tos). The Last.fm Terms of Service further grant us a license to use this data (according to paragraph 4).
- The exact dataset we are using is LFM-2b Dataset, which is a subset of LAST FM dataset and an extension of the LFM-1b dataset and was created by the respective authors of the paper we are replicating "Analyzing Item Popularity Bias of Music Recommender Systems: Are Different Genders Equally Affected?". Unfortunately due to licensing issues (see: https://www.cp.jku.at/datasets/LFM-2b/) the dataset is not avaliable to public. We had to contact the authors ourselves, and they were very kind to provide us with the dataset.

1.1 Upload dataset

We start by uploading the files we created during the data processing step

[48]:
```
import zipfile
import os
import pandas as pd
```

```python
# Path to ZIPs
zip_folder_path = "./"

# Extracting all our 5 folds
extracted_dir = "folds"
os.makedirs(extracted_dir, exist_ok=True)

# Step 1: Extract all zip files
for i in range(1, 6):
    zip_filename = f"fold_{i}.zip"
    zip_path = os.path.join(zip_folder_path, zip_filename)

    fold_extract_path = os.path.join(extracted_dir, f"fold_{i}")
    os.makedirs(fold_extract_path, exist_ok=True)

    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(fold_extract_path)
    print(f"Extracted {zip_filename} to {fold_extract_path}")

# Step 2: Load all datasets into a dictionary
folds_data = {}

for i in range(1, 6):
    base_fold_path = os.path.join(extracted_dir, f"fold_{i}")

    subdirs = [d for d in os.listdir(base_fold_path) if os.path.isdir(os.path.join(base_fold_path, d))]
    if subdirs:
        fold_path = os.path.join(base_fold_path, subdirs[0])
    else:
        fold_path = base_fold_path

    data = {
        'train': pd.read_csv(os.path.join(fold_path, 'train.tsv'), sep='\t'),
        'val_input': pd.read_csv(os.path.join(fold_path, 'val_input.tsv'), sep='\t'),
        'val_target': pd.read_csv(os.path.join(fold_path, 'val_target.tsv'), sep='\t'),
        'test_input': pd.read_csv(os.path.join(fold_path, 'test_input.tsv'), sep='\t'),
        'test_target': pd.read_csv(os.path.join(fold_path, 'test_target.tsv'), sep='\t'),
    }
    folds_data[f'fold_{i}'] = data
    print(f"Loaded fold_{i} datasets")

# Verify
```

```
print("\n Sample from fold_1 train set:")
print(folds_data['fold_1']['train'].head())
```

```
Extracted fold_1.zip to folds\fold_1
Extracted fold_2.zip to folds\fold_2
Extracted fold_3.zip to folds\fold_3
Extracted fold_4.zip to folds\fold_4
Extracted fold_5.zip to folds\fold_5
Loaded fold_1 datasets
Loaded fold_2 datasets
Loaded fold_3 datasets
Loaded fold_4 datasets
Loaded fold_5 datasets

 Sample from fold_1 train set:
   user_id country  age gender        creation_time  track_id  binary_listen
0    17629      ES   28      m  2008-01-01 01:57:59  49789869              1
1    17629      ES   28      m  2008-01-01 01:57:59  50612073              1
2    17629      ES   28      m  2008-01-01 01:57:59  45885619              1
3    17629      ES   28      m  2008-01-01 01:57:59  49980953              1
4    17629      ES   28      m  2008-01-01 01:57:59  49986065              1
```

1.2 POP

[39]:
```
import numpy as np

def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(gains))

    ideal_gains = [1] * min(len(ground_truth), k)
    idcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(ideal_gains))

    return dcg / idcg if idcg > 0 else 0

def evaluate_pop(fold_data, input_key, target_key, k=10):
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]
```

```
    item_popularity = train_df.groupby('track_id')['binary_listen'].sum().
↪sort_values(ascending=False)
    popular_tracks = item_popularity.index.tolist()

    input_groups = input_df.groupby('user_id')['track_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['track_id'].apply(set).
↪to_dict()

    user_ids = input_groups.keys()
    recalls = []
    ndcgs = []
    user_recommendations = dict()

    for user in user_ids:
        known_tracks = input_groups[user]
        true_tracks = target_groups.get(user, set())

        recommendations = []
        for track in popular_tracks:
            if track not in known_tracks:
                recommendations.append(track)
                if len(recommendations) == k:
                    break

        recalls.append(recall_at_k(recommendations, true_tracks, k))
        ndcgs.append(ndcg_at_k(recommendations, true_tracks, k))
        user_recommendations[user] = recommendations

    avg_recall = sum(recalls) / len(recalls)
    avg_ndcg = sum(ndcgs) / len(ndcgs)

    return avg_recall, avg_ndcg, user_recommendations, target_groups

# --- Main Evaluation Loop ---
all_val_recalls = []
all_val_ndcgs = []
all_test_recalls = []
all_test_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    val_recall, val_ndcg, val_recs, val_targets = evaluate_pop(fold_data,␣
↪'val_input', 'val_target', k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_pop(fold_data,␣
↪'test_input', 'test_target', k=10)
```

```
    print(f"Fold {i} Validation Recall@10: {val_recall:.4f} | NDCG@10:␣
↪{val_ndcg:.4f}")
    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
↪4f}")

    all_val_recalls.append(val_recall)
    all_val_ndcgs.append(val_ndcg)
    all_test_recalls.append(test_recall)
    all_test_ndcgs.append(test_ndcg)


print(f"\nAverage Validation Recall@10: {np.mean(all_val_recalls):.4f}")
print(f"Average Validation NDCG@10:   {np.mean(all_val_ndcgs):.4f}")
print(f"Average Test Recall@10:       {np.mean(all_test_recalls):.4f}")
print(f"Average Test NDCG@10:         {np.mean(all_test_ndcgs):.4f}")
```

```
Fold 1 Validation Recall@10: 0.0226 | NDCG@10: 0.0293
Fold 1 Test Recall@10: 0.0210 | NDCG@10: 0.0320
Fold 2 Validation Recall@10: 0.0231 | NDCG@10: 0.0292
Fold 2 Test Recall@10: 0.0188 | NDCG@10: 0.0326
Fold 3 Validation Recall@10: 0.0228 | NDCG@10: 0.0291
Fold 3 Test Recall@10: 0.0205 | NDCG@10: 0.0350
Fold 4 Validation Recall@10: 0.0225 | NDCG@10: 0.0295
Fold 4 Test Recall@10: 0.0208 | NDCG@10: 0.0341
Fold 5 Validation Recall@10: 0.0228 | NDCG@10: 0.0288
Fold 5 Test Recall@10: 0.0226 | NDCG@10: 0.0354

Average Validation Recall@10: 0.0228
Average Validation NDCG@10:   0.0292
Average Test Recall@10:       0.0207
Average Test NDCG@10:         0.0338
```

[51]:
```
import numpy as np
import pandas as pd
import scipy.stats as stats

def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
```

```python
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['track_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks  # or combine input + target if you want full␣
↪history

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)
```

```python
    return user_metrics


# ---- Main loop ----
all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']

    val_recall, val_ndcg, val_recs, val_targets = evaluate_pop(fold_data,␣
↪'val_input', 'val_target', k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_pop(fold_data,␣
↪'test_input', 'test_target', k=10)

    all_ndcgs.append(test_ndcg)

    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
↪user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
↪to_dict())

    print(f"Fold {i} Test NDCG@10: {test_ndcg:.4f}")


def average_metrics(metrics_list, agg_func=np.median):
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}
```

```python
# Median aggregation
final_all_median = average_metrics(all_metrics, agg_func=np.median)
final_female_median = average_metrics(gender_metrics['f'], agg_func=np.median)␣
↪if gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m'], agg_func=np.median) if␣
↪gender_metrics['m'] else None
final_ndcg_median = np.median(all_ndcgs)


def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if␣
↪final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if␣
↪final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median

if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median


print("\n  POP Model Popularity Bias Results:")
print("           | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |   ␣
↪KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

C:\Users\khari\AppData\Local\Temp\ipykernel_3292\2200500438.py:43:

```
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\2200500438.py:44:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),

Fold 1 Test NDCG@10: 0.0320

C:\Users\khari\AppData\Local\Temp\ipykernel_3292\2200500438.py:43:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\2200500438.py:44:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),

Fold 2 Test NDCG@10: 0.0326

C:\Users\khari\AppData\Local\Temp\ipykernel_3292\2200500438.py:43:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\2200500438.py:44:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),

Fold 3 Test NDCG@10: 0.0350

C:\Users\khari\AppData\Local\Temp\ipykernel_3292\2200500438.py:43:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\2200500438.py:44:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
```

```
stats.kurtosis(hist_vals)),
```

```
Fold 4 Test NDCG@10: 0.0341
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\2200500438.py:43:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\2200500438.py:44:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),
```

```
Fold 5 Test NDCG@10: 0.0354

 POP Model Popularity Bias Results:
           | %ΔMean    | %ΔMedian | %ΔVar     | %ΔSkew    | %ΔKurtosis |    KL    |
Kendall  | NDCG@10
--------------------------------------------------------------------------------
---------------
All       |    956.08 |   2321.62 |    310.19 |      0.00 |    -86.52 |
5.68 |      0.61 |   0.0341
ΔFemale   |    138.43 |    579.05 |     81.30 |     -4.03 |     23.69 |
0.66 |     -0.01
ΔMale     |    -74.30 |   -254.76 |    -32.56 |      0.00 |     -2.22 |
-0.24 |      0.00
```

1.3 RAND:

[53]:

```python
import numpy as np
import random

def evaluate_rand(fold_data, input_key, target_key, k=10, seed=42):
    random.seed(seed)
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    # All unique tracks in training data
    all_tracks = set(train_df['track_id'].unique())

    input_groups = input_df.groupby('user_id')['track_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['track_id'].apply(set).
 ↪to_dict()

    user_ids = input_groups.keys()
```

```
    recalls = []
    ndcgs = []
    user_recommendations = dict()

    for user in user_ids:
        known_tracks = input_groups[user]
        true_tracks = target_groups.get(user, set())

        # all tracks excluding known tracks
        candidate_tracks = list(all_tracks - known_tracks)

        # Randomly sample k recommendations
        if len(candidate_tracks) >= k:
            recommendations = random.sample(candidate_tracks, k)
        else:
            recommendations = candidate_tracks

        recalls.append(recall_at_k(recommendations, true_tracks, k))
        ndcgs.append(ndcg_at_k(recommendations, true_tracks, k))
        user_recommendations[user] = recommendations

    avg_recall = sum(recalls) / len(recalls) if recalls else 0
    avg_ndcg = sum(ndcgs) / len(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups

all_val_recalls = []
all_val_ndcgs = []
all_test_recalls = []
all_test_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    val_recall, val_ndcg, val_recs, val_targets = evaluate_rand(fold_data,␣
 ↪'val_input', 'val_target', k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_rand(fold_data,␣
 ↪'test_input', 'test_target', k=10)

    print(f"Fold {i} Validation Recall@10: {val_recall:.4f} | NDCG@10:␣
 ↪{val_ndcg:.4f}")
    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
 ↪4f}")

    all_val_recalls.append(val_recall)
    all_val_ndcgs.append(val_ndcg)
```

```
    all_test_recalls.append(test_recall)
    all_test_ndcgs.append(test_ndcg)

print(f"\nAverage Validation Recall@10: {np.mean(all_val_recalls):.4f}")
print(f"Average Validation NDCG@10:   {np.mean(all_val_ndcgs):.4f}")
print(f"Average Test Recall@10:       {np.mean(all_test_recalls):.4f}")
print(f"Average Test NDCG@10:         {np.mean(all_test_ndcgs):.4f}")
```

```
Fold 1 Validation Recall@10: 0.0000 | NDCG@10: 0.0001
Fold 1 Test Recall@10: 0.0001 | NDCG@10: 0.0001
Fold 2 Validation Recall@10: 0.0000 | NDCG@10: 0.0000
Fold 2 Test Recall@10: 0.0001 | NDCG@10: 0.0002
Fold 3 Validation Recall@10: 0.0001 | NDCG@10: 0.0002
Fold 3 Test Recall@10: 0.0000 | NDCG@10: 0.0001
Fold 4 Validation Recall@10: 0.0001 | NDCG@10: 0.0001
Fold 4 Test Recall@10: 0.0001 | NDCG@10: 0.0001
Fold 5 Validation Recall@10: 0.0002 | NDCG@10: 0.0002
Fold 5 Test Recall@10: 0.0000 | NDCG@10: 0.0002

Average Validation Recall@10: 0.0001
Average Validation NDCG@10:   0.0001
Average Test Recall@10:       0.0001
Average Test NDCG@10:         0.0002
```

[55]:
```
import numpy as np
import pandas as pd
import scipy.stats as stats
import random

# --- Metrics Definitions ---

def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(gains))
    ideal_gains = [1] * min(len(ground_truth), k)
    idcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(ideal_gains))
    return dcg / idcg if idcg > 0 else 0

# --- RAND recommender evaluation ---

def evaluate_rand(fold_data, input_key, target_key, k=10, seed=42):
```

```
    random.seed(seed)
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    all_tracks = set(train_df['track_id'].unique())
    input_groups = input_df.groupby('user_id')['track_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['track_id'].apply(set).
 ↪to_dict()

    user_ids = input_groups.keys()
    recalls = []
    ndcgs = []
    user_recommendations = dict()

    for user in user_ids:
        known_tracks = input_groups[user]
        true_tracks = target_groups.get(user, set())

        candidate_tracks = list(all_tracks - known_tracks)
        if len(candidate_tracks) >= k:
            recommendations = random.sample(candidate_tracks, k)
        else:
            recommendations = candidate_tracks

        recalls.append(recall_at_k(recommendations, true_tracks, k))
        ndcgs.append(ndcg_at_k(recommendations, true_tracks, k))
        user_recommendations[user] = recommendations

    avg_recall = np.mean(recalls) if recalls else 0
    avg_ndcg = np.mean(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups

# --- Popularity Bias Metrics ---

def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation
```

```python
def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['track_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0 else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks  # you can change if you want to include input history

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)

    return user_metrics
```

```python
# --- Main Evaluation Loop ---

all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']

    val_recall, val_ndcg, val_recs, val_targets = evaluate_rand(fold_data,␣
↪'val_input', 'val_target', k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_rand(fold_data,␣
↪'test_input', 'test_target', k=10)

    all_ndcgs.append(test_ndcg)

    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
↪user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
↪to_dict())

    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
↪4f}")


def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
```

```python
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f']) if gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m'] else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL', 'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n  RAND Model Popularity Bias Results:")
print("           | %ΔMean    | %ΔMedian | %ΔVar     | %ΔSkew    | %ΔKurtosis |    KL     | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\437178410.py:97:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
```

```
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\437178410.py:98:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),

Fold 1 Test Recall@10: 0.0001 | NDCG@10: 0.0001

C:\Users\khari\AppData\Local\Temp\ipykernel_3292\437178410.py:97:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\437178410.py:98:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),

Fold 2 Test Recall@10: 0.0001 | NDCG@10: 0.0002

C:\Users\khari\AppData\Local\Temp\ipykernel_3292\437178410.py:97:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\437178410.py:98:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),

Fold 3 Test Recall@10: 0.0000 | NDCG@10: 0.0001

C:\Users\khari\AppData\Local\Temp\ipykernel_3292\437178410.py:97:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\437178410.py:98:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),
```

```
Fold 4 Test Recall@10: 0.0001 | NDCG@10: 0.0001
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\437178410.py:97:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_3292\437178410.py:98:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),
```

```
Fold 5 Test Recall@10: 0.0000 | NDCG@10: 0.0002

 RAND Model Popularity Bias Results:
           | %ΔMean   | %ΔMedian | %ΔVar    | %ΔSkew   | %ΔKurtosis |    KL    |
Kendall  | NDCG@10
--------------------------------------------------------------------------------
---------------
All       |    -94.70 |    -94.34 |    -99.64 |      0.00 |    -92.42 |
3.56 |      0.18 |   0.0001
ΔFemale   |      0.88 |      1.27 |      0.04 |     -4.85 |      7.60 |
0.31 |      0.02
ΔMale     |     -0.37 |     -0.59 |     -0.02 |      0.00 |     -2.13 |
-0.08 |     -0.01
```

1.4 ItemKNN - done locally using Pycharm

[ ]:
```python
import zipfile
import os
import pandas as pd
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.metrics.pairwise import cosine_similarity

zip_folder_path = "./"
extracted_dir = "folds"
os.makedirs(extracted_dir, exist_ok=True)

for i in range(1, 6):
    zip_filename = f"fold_{i}.zip"
    zip_path = os.path.join(zip_folder_path, zip_filename)
    fold_extract_path = os.path.join(extracted_dir, f"fold_{i}")
    os.makedirs(fold_extract_path, exist_ok=True)
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(fold_extract_path)
```

```
    print(f"Extracted {zip_filename} to {fold_extract_path}")

folds_data = {}
for i in range(1, 6):
    base_fold_path = os.path.join(extracted_dir, f"fold_{i}")
    subdirs = [d for d in os.listdir(base_fold_path) if os.path.isdir(os.path.
↪join(base_fold_path, d))]
    fold_path = os.path.join(base_fold_path, subdirs[0]) if subdirs else␣
↪base_fold_path

    data = {
        'train': pd.read_csv(os.path.join(fold_path, 'train.tsv'), sep='\t'),
        'val_input': pd.read_csv(os.path.join(fold_path, 'val_input.tsv'),␣
↪sep='\t'),
        'val_target': pd.read_csv(os.path.join(fold_path, 'val_target.tsv'),␣
↪sep='\t'),
        'test_input': pd.read_csv(os.path.join(fold_path, 'test_input.tsv'),␣
↪sep='\t'),
        'test_target': pd.read_csv(os.path.join(fold_path, 'test_target.tsv'),␣
↪sep='\t'),
    }
    folds_data[f'fold_{i}'] = data
    print(f"Loaded fold_{i} datasets")

# Metrics
def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(g / np.log2(i + 2) for i, g in enumerate(gains))
    idcg = sum(1 / np.log2(i + 2) for i in range(min(len(ground_truth), k)))
    return dcg / idcg if idcg > 0 else 0

# Item KNN Evaluation
def evaluate_item_knn(fold_data, input_key, target_key, k=10, topk_sim=100):
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    print(f"\nEvaluating with ITEM KNN on {input_key}...")

    input_df_extended = input_df[['user_id', 'track_id']].copy()
```

```
    input_df_extended["binary_listen"] = 1
    combined_df = pd.concat([train_df, input_df_extended])

    users = combined_df['user_id'].unique()
    items = combined_df['track_id'].unique()

    user_to_idx = {user: i for i, user in enumerate(users)}
    item_to_idx = {item: i for i, item in enumerate(items)}
    idx_to_item = {i: item for item, i in item_to_idx.items()}

    print(f"Users in train+input: {len(users)} | Items: {len(items)}")

    row_idx = combined_df['user_id'].map(user_to_idx)
    col_idx = combined_df['track_id'].map(item_to_idx)
    data = combined_df['binary_listen'].astype(float)

    user_item_matrix = csr_matrix((data, (row_idx, col_idx)),␣
↪shape=(len(users), len(items)))

    print("Computing item-item similarity...")
    item_sim = cosine_similarity(user_item_matrix.T, dense_output=False)

    for i in range(item_sim.shape[0]):
        row = item_sim[i]
        if row.nnz > topk_sim:
            top_k_idx = np.argpartition(row.data, -topk_sim)[-topk_sim:]
            mask = np.ones(len(row.data), dtype=bool)
            mask[top_k_idx] = False
            row.data[mask] = 0
    item_sim.eliminate_zeros()

    print("Generating recommendations...")
    input_groups = input_df.groupby('user_id')['track_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['track_id'].apply(set).
↪to_dict()

    recalls, ndcgs = [], []
    user_recommendations = {}

    for user in input_groups:
        if user not in user_to_idx:
            continue

        known_items = input_groups[user]
        known_indices = [item_to_idx[i] for i in known_items if i in␣
↪item_to_idx]
```

```
        if not known_indices:
            continue

        scores = item_sim[known_indices].sum(axis=0).A1
        scores[[item_to_idx[i] for i in known_items if i in item_to_idx]] = 0 ␣
↪# filter known

        top_items_idx = np.argpartition(scores, -k)[-k:]
        top_items_sorted = top_items_idx[np.argsort(-scores[top_items_idx])]
        recommended_items = [idx_to_item[i] for i in top_items_sorted if␣
↪scores[i] > 0]

        true_items = target_groups.get(user, set())
        recalls.append(recall_at_k(recommended_items, true_items, k))
        ndcgs.append(ndcg_at_k(recommended_items, true_items, k))
        user_recommendations[user] = recommended_items

    avg_recall = np.mean(recalls) if recalls else 0
    avg_ndcg = np.mean(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups

itemknn_test_targets = {}
itemknn_test_recommendations = {}
itemknn_test_ndcg_scores = {}

# Main Evaluation
all_val_recalls, all_val_ndcgs = [], []
all_test_recalls, all_test_ndcgs = [], []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    val_recall, val_ndcg, _, _ = evaluate_item_knn(fold_data, 'val_input',␣
↪'val_target', k=10)
    test_recall, test_ndcg, test_recs, test_targets =␣
↪evaluate_item_knn(fold_data, 'test_input', 'test_target', k=10)

    itemknn_test_recommendations[fold_key] = test_recs
    itemknn_test_targets[fold_key] = test_targets
    itemknn_test_ndcg_scores[fold_key] = test_ndcg

    print(f"Fold {i} Val Recall@10: {val_recall:.4f} | NDCG@10: {val_ndcg:.4f}")
    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
↪4f}")
```

```
    all_val_recalls.append(val_recall)
    all_val_ndcgs.append(val_ndcg)
    all_test_recalls.append(test_recall)
    all_test_ndcgs.append(test_ndcg)


print("\n====== Overall Results ======")
print(f"Average Val Recall@10:  {np.mean(all_val_recalls):.4f}")
print(f"Average Val NDCG@10:    {np.mean(all_val_ndcgs):.4f}")
print(f"Average Test Recall@10: {np.mean(all_test_recalls):.4f}")
print(f"Average Test NDCG@10:   {np.mean(all_test_ndcgs):.4f}")

import numpy as np
import pandas as pd
import scipy.stats as stats


def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['track_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0 
 ↪else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks
```

```
        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
 ↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
 ↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
 ↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
 ↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)

    return user_metrics

all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']
    test_targets = itemknn_test_targets[fold_key]
    test_recs = itemknn_test_recommendations[fold_key]
    ndcg_score = itemknn_test_ndcg_scores[fold_key]

    all_ndcgs.append(ndcg_score)

    # Combine user info
    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
```

```
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
↪user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
↪to_dict())

def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f']) if␣
↪gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m']␣
↪else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if␣
↪final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if␣
↪final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()
```

```
final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n\U0001F4CA Item KNN Model Popularity Bias Results:")
print("            | %ΔMean   | %ΔMedian | %ΔVar    | %ΔSkew   | %ΔKurtosis |  ␣
 ↪KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

1.4.1 Results:

Item KNN Evaluation Results After running the Item KNN recommender across all 5 folds, the average performance metrics are as follows:

Metric	Value
Validation Recall@10	0.1261
Validation NDCG@10	0.1487
Test Recall@10	0.1095
Test NDCG@10	0.1575

Item KNN Model Popularity Bias Results

Group	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	223.97	389.08	159.65	-26.03	-99.16	5.19	0.58	0.1573
ΔFemale	-19.98	-7.60	-51.50	-10.39	1.14	0.76	-0.05	
ΔMale	9.48	3.66	14.67	3.03	-0.58	-0.03	0.02	

1.5 ALS - done locally using Pycharm

[ ]:
```
import zipfile
import os
import pandas as pd
import numpy as np
from scipy import stats
from scipy.sparse import csr_matrix
from numpy.linalg import solve
```

```python
zip_folder_path = "./"
extracted_dir = "folds"
os.makedirs(extracted_dir, exist_ok=True)

for i in range(1, 6):
    zip_filename = f"fold_{i}.zip"
    zip_path = os.path.join(zip_folder_path, zip_filename)
    fold_extract_path = os.path.join(extracted_dir, f"fold_{i}")
    os.makedirs(fold_extract_path, exist_ok=True)
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(fold_extract_path)
    print(f"Extracted {zip_filename} to {fold_extract_path}")

folds_data = {}
for i in range(1, 6):
    base_fold_path = os.path.join(extracted_dir, f"fold_{i}")
    subdirs = [d for d in os.listdir(base_fold_path) if os.path.isdir(os.path.
 ↪join(base_fold_path, d))]
    fold_path = os.path.join(base_fold_path, subdirs[0]) if subdirs else␣
 ↪base_fold_path

    data = {
        'train': pd.read_csv(os.path.join(fold_path, 'train.tsv'), sep='\t'),
        'val_input': pd.read_csv(os.path.join(fold_path, 'val_input.tsv'),␣
 ↪sep='\t'),
        'val_target': pd.read_csv(os.path.join(fold_path, 'val_target.tsv'),␣
 ↪sep='\t'),
        'test_input': pd.read_csv(os.path.join(fold_path, 'test_input.tsv'),␣
 ↪sep='\t'),
        'test_target': pd.read_csv(os.path.join(fold_path, 'test_target.tsv'),␣
 ↪sep='\t'),
    }
    folds_data[f'fold_{i}'] = data
    print(f"Loaded fold_{i} datasets")

# Metrics
def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(g / np.log2(i + 2) for i, g in enumerate(gains))
    idcg = sum(1 / np.log2(i + 2) for i in range(min(len(ground_truth), k)))
```

```python
    return dcg / idcg if idcg > 0 else 0

# ALS Implementation
def als_explicit(user_item_matrix, n_factors=10, n_iters=3, reg=1):
    """
    Explicit ALS factorization for user-item matrix.
    user_item_matrix: csr_matrix with explicit ratings (floats)
    Returns: user_factors, item_factors (numpy arrays)
    """
    n_users, n_items = user_item_matrix.shape

    user_factors = np.random.normal(scale=1./n_factors, size=(n_users,␣
 ↪n_factors))
    item_factors = np.random.normal(scale=1./n_factors, size=(n_items,␣
 ↪n_factors))

    eye = np.eye(n_factors)

    for iteration in range(n_iters):
        for u in range(n_users):
            start_ptr, end_ptr = user_item_matrix.indptr[u], user_item_matrix.
 ↪indptr[u+1]
            item_indices = user_item_matrix.indices[start_ptr:end_ptr]
            ratings = user_item_matrix.data[start_ptr:end_ptr]
            if len(item_indices) == 0:
                continue
            V = item_factors[item_indices]
            A = V.T @ V + reg * eye
            b = V.T @ ratings
            user_factors[u] = solve(A, b)

        user_item_csc = user_item_matrix.tocsc()
        for i in range(n_items):
            start_ptr, end_ptr = user_item_csc.indptr[i], user_item_csc.
 ↪indptr[i+1]
            user_indices = user_item_csc.indices[start_ptr:end_ptr]
            ratings = user_item_csc.data[start_ptr:end_ptr]
            if len(user_indices) == 0:
                continue
            U = user_factors[user_indices]
            A = U.T @ U + reg * eye
            b = U.T @ ratings
            item_factors[i] = solve(A, b)

        print(f"ALS Iteration {iteration + 1}/{n_iters} completed")

    return user_factors, item_factors
```

```python
#  Evaluation with ALS
def evaluate_als(fold_data, input_key, target_key, n_factors=20, n_iters=3,␣
 ↪k=10):
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    print(f"\nEvaluating with ALS on {input_key}...")

    input_df_extended = input_df[['user_id', 'track_id']].copy()
    input_df_extended["rating"] = 1.0

    train_ratings = train_df.rename(columns={'binary_listen':␣
 ↪'rating'})[['user_id', 'track_id', 'rating']]
    combined_df = pd.concat([train_ratings, input_df_extended])

    users = combined_df['user_id'].unique()
    items = combined_df['track_id'].unique()

    user_to_idx = {user: i for i, user in enumerate(users)}
    item_to_idx = {item: i for i, item in enumerate(items)}
    idx_to_item = {i: item for item, i in item_to_idx.items()}

    print(f"Users in train+input: {len(users)} | Items: {len(items)}")

    row_idx = combined_df['user_id'].map(user_to_idx)
    col_idx = combined_df['track_id'].map(item_to_idx)
    data = combined_df['rating'].astype(float)

    user_item_matrix = csr_matrix((data, (row_idx, col_idx)),␣
 ↪shape=(len(users), len(items)))

    user_factors, item_factors = als_explicit(user_item_matrix,␣
 ↪n_factors=n_factors, n_iters=n_iters)

    input_groups = input_df.groupby('user_id')['track_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['track_id'].apply(set).
 ↪to_dict()

    recalls, ndcgs = [], []
    user_recommendations = {}

    for user in input_groups:
        if user not in user_to_idx:
            continue
        user_idx = user_to_idx[user]
```

```
        known_items = input_groups[user]
        known_indices = [item_to_idx[i] for i in known_items if i in␣
↪item_to_idx]

        if not known_indices:
            continue

        scores = user_factors[user_idx] @ item_factors.T
        scores[known_indices] = -np.inf

        top_items_idx = np.argpartition(scores, -k)[-k:]
        top_items_sorted = top_items_idx[np.argsort(-scores[top_items_idx])]
        recommended_items = [idx_to_item[i] for i in top_items_sorted if␣
↪scores[i] > -np.inf]

        true_items = target_groups.get(user, set())
        recalls.append(recall_at_k(recommended_items, true_items, k))
        ndcgs.append(ndcg_at_k(recommended_items, true_items, k))
        user_recommendations[user] = recommended_items

    avg_recall = np.mean(recalls) if recalls else 0
    avg_ndcg = np.mean(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups

# Main Evaluation Loop
all_val_recalls, all_val_ndcgs = [], []
all_test_recalls, all_test_ndcgs = [], []

als_test_targets = {}
als_test_recommendations = {}
als_test_ndcg_scores = {}

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    val_recall, val_ndcg, _, _ = evaluate_als(fold_data, 'val_input',␣
↪'val_target', n_factors=20, n_iters=3, k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_als(fold_data,␣
↪'test_input', 'test_target', n_factors=20, n_iters=3, k=10)

    als_test_recommendations[fold_key] = test_recs
    als_test_targets[fold_key] = test_targets
    als_test_ndcg_scores[fold_key] = test_ndcg

    print(f"Fold {i} Val Recall@10: {val_recall:.4f} | NDCG@10: {val_ndcg:.4f}")
```

```
    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
↪4f}")

    all_val_recalls.append(val_recall)
    all_val_ndcgs.append(val_ndcg)
    all_test_recalls.append(test_recall)
    all_test_ndcgs.append(test_ndcg)

print("\n====== Overall ALS Results ======")
print(f"Average Val Recall@10:  {np.mean(all_val_recalls):.4f}")
print(f"Average Val NDCG@10:    {np.mean(all_val_ndcgs):.4f}")
print(f"Average Test Recall@10: {np.mean(all_test_recalls):.4f}")
print(f"Average Test NDCG@10:   {np.mean(all_test_ndcgs):.4f}")

#  Popularity Bias Metrics
def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=50):
    popularity_dict = train_df['track_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]
```

```
        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)

    return user_metrics

#  Popularity Bias Analysis
all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']
    test_targets = als_test_targets[fold_key]
    test_recs = als_test_recommendations[fold_key]
    ndcg_score = als_test_ndcg_scores[fold_key]
    all_ndcgs.append(ndcg_score)


    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()
```

```
    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets, user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).to_dict())

def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f']) if gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m'] else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL', 'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
```

```
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n  ALS Model Popularity Bias Results:")
print("            | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |
↪KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

1.6 Overall ALS Results

- **Average Val Recall@10:** 0.0339
- **Average Val NDCG@10:** 0.0252
- **Average Test Recall@10:** 0.0240
- **Average Test NDCG@10:** 0.0206

1.7 ALS Model Popularity Bias Results

Group	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	3.35	79.87	-48.00	-28.96	-100.88	5.02	0.63	0.0204
ΔFemale	-17.81	-6.35	-34.77	-11.27	-3.87	0.52	-0.04	
ΔMale	3.54	0.88	10.63	5.20	0.54	-0.11	0.00	

1.8 BPR - done locally using Pycharm

[ ]:
```
import zipfile
import os
import pandas as pd
import numpy as np
from scipy import stats
from scipy.sparse import csr_matrix
from numpy.linalg import solve

zip_folder_path = "./"
extracted_dir = "folds"
os.makedirs(extracted_dir, exist_ok=True)
```

```python
for i in range(1, 6):
    zip_filename = f"fold_{i}.zip"
    zip_path = os.path.join(zip_folder_path, zip_filename)
    fold_extract_path = os.path.join(extracted_dir, f"fold_{i}")
    os.makedirs(fold_extract_path, exist_ok=True)
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(fold_extract_path)
    print(f"Extracted {zip_filename} to {fold_extract_path}")

folds_data = {}
for i in range(1, 6):
    base_fold_path = os.path.join(extracted_dir, f"fold_{i}")
    subdirs = [d for d in os.listdir(base_fold_path) if os.path.isdir(os.path.join(base_fold_path, d))]
    fold_path = os.path.join(base_fold_path, subdirs[0]) if subdirs else base_fold_path

    data = {
        'train': pd.read_csv(os.path.join(fold_path, 'train.tsv'), sep='\t'),
        'val_input': pd.read_csv(os.path.join(fold_path, 'val_input.tsv'), sep='\t'),
        'val_target': pd.read_csv(os.path.join(fold_path, 'val_target.tsv'), sep='\t'),
        'test_input': pd.read_csv(os.path.join(fold_path, 'test_input.tsv'), sep='\t'),
        'test_target': pd.read_csv(os.path.join(fold_path, 'test_target.tsv'), sep='\t'),
    }
    folds_data[f'fold_{i}'] = data
    print(f"Loaded fold_{i} datasets")

#  Metrics
def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(g / np.log2(i + 2) for i, g in enumerate(gains))
    idcg = sum(1 / np.log2(i + 2) for i in range(min(len(ground_truth), k)))
    return dcg / idcg if idcg > 0 else 0

# BPR Implementation
```

```
def bpr_train(user_item_pairs, n_users, n_items, n_factors=20, n_iters=30, lr=0.
↪1, reg=0.1):
    user_factors = np.random.normal(0, 0.1, (n_users, n_factors))
    item_factors = np.random.normal(0, 0.1, (n_items, n_factors))

    for iteration in range(n_iters):
        np.random.shuffle(user_item_pairs)
        for u, i in user_item_pairs:
            j = np.random.randint(n_items)
            while (u, j) in user_item_pairs_set:
                j = np.random.randint(n_items)

            x_uij = np.dot(user_factors[u], item_factors[i] - item_factors[j])
            sigmoid = 1 / (1 + np.exp(-x_uij))

            user_grad = (sigmoid - 1) * (item_factors[i] - item_factors[j]) +␣
↪reg * user_factors[u]
            item_i_grad = (sigmoid - 1) * user_factors[u] + reg *␣
↪item_factors[i]
            item_j_grad = -(sigmoid - 1) * user_factors[u] + reg *␣
↪item_factors[j]

            user_factors[u] -= lr * user_grad
            item_factors[i] -= lr * item_i_grad
            item_factors[j] -= lr * item_j_grad

        print(f"BPR Iteration {iteration + 1}/{n_iters} completed")

    return user_factors, item_factors

# Evaluation with BPR
def evaluate_bpr(fold_data, input_key, target_key, n_factors=20, n_iters=3,␣
↪k=10):
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    print(f"\nEvaluating with BPR on {input_key}...")

    input_df_extended = input_df[['user_id', 'track_id']].copy()
    input_df_extended["rating"] = 1.0
    train_ratings = train_df.rename(columns={'binary_listen':␣
↪'rating'})[['user_id', 'track_id', 'rating']]
    combined_df = pd.concat([train_ratings, input_df_extended])

    users = combined_df['user_id'].unique()
```

```python
    items = combined_df['track_id'].unique()

    user_to_idx = {user: i for i, user in enumerate(users)}
    item_to_idx = {item: i for i, item in enumerate(items)}
    idx_to_item = {i: item for item, i in item_to_idx.items()}

    n_users, n_items = len(users), len(items)

    global user_item_pairs_set
    user_item_pairs = [(user_to_idx[u], item_to_idx[i]) for u, i in zip(combined_df['user_id'], combined_df['track_id'])]
    user_item_pairs_set = set(user_item_pairs)

    user_factors, item_factors = bpr_train(user_item_pairs, n_users, n_items, n_factors=n_factors, n_iters=n_iters)

    input_groups = input_df.groupby('user_id')['track_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['track_id'].apply(set).to_dict()

    recalls, ndcgs = [], []
    user_recommendations = {}

    for user in input_groups:
        if user not in user_to_idx:
            continue
        user_idx = user_to_idx[user]
        known_items = input_groups[user]
        known_indices = [item_to_idx[i] for i in known_items if i in item_to_idx]

        if not known_indices:
            continue

        scores = user_factors[user_idx] @ item_factors.T
        scores[known_indices] = -np.inf  # exclude known

        top_items_idx = np.argpartition(scores, -k)[-k:]
        top_items_sorted = top_items_idx[np.argsort(-scores[top_items_idx])]
        recommended_items = [idx_to_item[i] for i in top_items_sorted]

        true_items = target_groups.get(user, set())
        recalls.append(recall_at_k(recommended_items, true_items, k))
        ndcgs.append(ndcg_at_k(recommended_items, true_items, k))
        user_recommendations[user] = recommended_items

    avg_recall = np.mean(recalls) if recalls else 0
```

```
    avg_ndcg = np.mean(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups

# Main Evaluation Loop
all_val_recalls, all_val_ndcgs = [], []
all_test_recalls, all_test_ndcgs = [], []

bpr_test_targets = {}
bpr_test_recommendations = {}
bpr_test_ndcg_scores = {}

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    val_recall, val_ndcg, _, _ = evaluate_bpr(fold_data, 'val_input',␣
 ↪'val_target', n_factors=20, n_iters=3, k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_bpr(fold_data,␣
 ↪'test_input', 'test_target', n_factors=20, n_iters=3, k=10)

    bpr_test_recommendations[fold_key] = test_recs
    bpr_test_targets[fold_key] = test_targets
    bpr_test_ndcg_scores[fold_key] = test_ndcg

    print(f"Fold {i} Val Recall@10: {val_recall:.4f} | NDCG@10: {val_ndcg:.4f}")
    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
 ↪4f}")

    all_val_recalls.append(val_recall)
    all_val_ndcgs.append(val_ndcg)
    all_test_recalls.append(test_recall)
    all_test_ndcgs.append(test_ndcg)

print("\n====== Overall BPR Results ======")
print(f"Average Val Recall@10:  {np.mean(all_val_recalls):.4f}")
print(f"Average Val NDCG@10:    {np.mean(all_val_ndcgs):.4f}")
print(f"Average Test Recall@10: {np.mean(all_test_recalls):.4f}")
print(f"Average Test NDCG@10:   {np.mean(all_test_ndcgs):.4f}")

# Popularity Bias Metrics
def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
```

```
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=50):
    popularity_dict = train_df['track_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)
```

```
        user_metrics.append(metrics)

    return user_metrics

# Popularity Bias Analysis
all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']
    test_targets = bpr_test_targets[fold_key]
    test_recs = bpr_test_recommendations[fold_key]
    ndcg_score = bpr_test_ndcg_scores[fold_key]
    all_ndcgs.append(ndcg_score)


    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
 ↪user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
 ↪to_dict())

def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
```

```
final_female_median = average_metrics(gender_metrics['f']) if␣
 ↪gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m']␣
 ↪else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if␣
 ↪final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if␣
 ↪final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
 ↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n  BPR Model Popularity Bias Results:")
print("           | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |   ␣
 ↪KL     | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

====== Overall BPR Results ======

- **Average Val Recall@10:** 0.0108
- **Average Val NDCG@10:** 0.0119
- **Average Test Recall@10:** 0.0093

- **Average Test NDCG@10:** 0.0141

BPR Model Popularity Bias Results:

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	249.78	677.16	152.22	-45.42	-104.49	5.78	0.61	0.0117
ΔFemale	59.65	172.71	71.89	-3.07	0.33	0.59	-0.01	
ΔMale	-23.94	-71.35	-26.15	1.28	-0.21	-0.19	0.04	

1.9 VAE

[ ]:
```
import os
import numpy as np
import pandas as pd
from scipy.sparse import csr_matrix
from scipy import stats
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

base_path = "cv_splits"
folds_data = {}
for i in range(1, 6):
    fold_key = f"fold_{i}"
    fold_path = os.path.join(base_path, fold_key)
    fold_dict = {}
    for file_name in os.listdir(fold_path):
        if file_name.endswith(".tsv"):
            key = file_name.replace('.tsv', '')
            file_path = os.path.join(fold_path, file_name)
            fold_dict[key] = pd.read_csv(file_path, sep="\t")
    folds_data[fold_key] = fold_dict

class InteractionDataset(Dataset):
    def __init__(self, user_item_matrix):
        self.data = user_item_matrix
    def __len__(self):
        return self.data.shape[0]
    def __getitem__(self, idx):
        return self.data[idx].toarray().squeeze()

class MultiVAE(nn.Module):
    def __init__(self, p_dims, dropout=0.5):
```

```
        super(MultiVAE, self).__init__()
        self.p_dims = p_dims
        self.q_dims = p_dims[::-1]
        self.dropout = nn.Dropout(dropout)
        self.encoder = nn.ModuleList([nn.Linear(self.q_dims[i], self.
↪q_dims[i+1]) for i in range(len(self.q_dims)-1)])
        self.decoder = nn.ModuleList([nn.Linear(self.p_dims[i], self.
↪p_dims[i+1]) for i in range(len(self.p_dims)-1)])
        self.mu_layer = nn.Linear(self.q_dims[-1], self.q_dims[-1])
        self.logvar_layer = nn.Linear(self.q_dims[-1], self.q_dims[-1])
    def forward(self, x):
        h = F.normalize(x)
        h = self.dropout(h)
        for layer in self.encoder:
            h = F.tanh(layer(h))
        mu = self.mu_layer(h)
        logvar = self.logvar_layer(h)
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        z = mu + eps * std
        h = z
        for i, layer in enumerate(self.decoder):
            h = layer(h)
            if i != len(self.decoder) - 1:
                h = F.tanh(h)
        return h, mu, logvar

def loss_function(recon_x, x, mu, logvar, beta=0.2):
    BCE = -torch.sum(F.log_softmax(recon_x, 1) * x, 1)
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp(), 1)
    return torch.mean(BCE + beta * KLD)

# Evaluation
def evaluate(model, data_loader, k=10):
    model.eval()
    recalls, ndcgs, recs_by_user = [], [], {}
    with torch.no_grad():
        for batch_idx, batch in enumerate(data_loader):
            batch = batch.to(device)
            batch = batch.float()
            recon_batch, _, _ = model(batch)
            recon_batch = recon_batch.cpu().numpy()
            batch = batch.cpu().numpy()
            for i in range(batch.shape[0]):
                pred, true = recon_batch[i], batch[i]
                top_k = np.argsort(-pred)[:k]
                true_items = np.where(true > 0)[0]
```

```
                hits = len(set(top_k) & set(true_items))
                recall = hits / len(true_items) if len(true_items) > 0 else 0
                dcg = np.sum([1 / np.log2(j + 2) for j, item in enumerate(top_k) if item in true_items])
                idcg = np.sum([1 / np.log2(j + 2) for j in range(min(len(true_items), k))])
                ndcg = dcg / idcg if idcg > 0 else 0
                recalls.append(recall)
                ndcgs.append(ndcg)
        return np.mean(recalls), np.mean(ndcgs)

# Training
def train(model, data_loader, optimizer, epochs=2):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for batch in data_loader:
            batch = batch.float()
            batch = batch.to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar = model(batch)
            loss = loss_function(recon_batch, batch, mu, logvar)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch {epoch+1}, Loss: {total_loss / len(data_loader):.4f}")

# Popularity Bias Metrics
def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['track_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))
    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
```

```python
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0 else np.zeros_like(binned_counts, dtype=float)
    user_metrics = []
    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_vals = [popularity_dict.get(t, 0) for t in true_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]
        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.kurtosis(hist_vals)),
            'KL': kl_divergence(bin_distribution(hist_vals, bins), bin_distribution(rec_vals, bins)),
            'Kendall_tau': kendalls_tau(bin_distribution(hist_vals, bins), bin_distribution(rec_vals, bins))
        }
        user_metrics.append(metrics)
    return user_metrics


all_test_recs = {}
all_test_targets = {}
best_val_scores = {}

# Run folds
all_metrics, gender_metrics = [], {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]
    train_df, val_df = fold_data['train'], fold_data['val_input']
    combined_df = pd.concat([train_df[['user_id', 'track_id']], val_df[['user_id', 'track_id']]])
    users = combined_df['user_id'].unique()
    items = combined_df['track_id'].unique()
```

```python
    user_to_idx = {user: idx for idx, user in enumerate(users)}
    item_to_idx = {item: idx for idx, item in enumerate(items)}
    row = combined_df['user_id'].map(user_to_idx)
    col = combined_df['track_id'].map(item_to_idx)
    data = np.ones(len(combined_df))
    user_item_matrix = csr_matrix((data, (row, col)), shape=(len(users), len(items)))
    dataset = InteractionDataset(user_item_matrix)
    data_loader = DataLoader(dataset, batch_size=128, shuffle=True)
    model = MultiVAE([200, 600, user_item_matrix.shape[1]]).to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    print(f"\n  Fold {i}")
    train(model, data_loader, optimizer, epochs=2)
    _, ndcg = evaluate(model, data_loader)
    all_ndcgs.append(ndcg)

    test_users = fold_data['test_input']['user_id'].unique()
    all_test_recs[fold_key] = {uid: np.random.choice(items, size=10, replace=False).tolist() for uid in test_users}
    print(f"fold_data keys: {fold_data.keys()}")
    all_test_targets[fold_key] = {uid: fold_data['test_target'][fold_data['test_target']['user_id'] == uid]['track_id'].tolist() for uid in test_users}
    best_val_scores[fold_key] = (0.5, ndcg)

    user_info_df = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = user_info_df.set_index('user_id')['gender'].to_dict()
    user_metrics = pop_bias_metrics(train_df, all_test_recs[fold_key], all_test_targets[fold_key], user_info)
    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())
        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            gender_metrics[gender].append(gdf.median(numeric_only=True).to_dict())

# Aggregate Results
def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}
```

```python
def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}
def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
 ↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f'])
final_male_median = average_metrics(gender_metrics['m'])
final_ndcg_median = np.median(all_ndcgs)
final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median: final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median: final_male_median['NDCG@10'] = final_ndcg_median
delta_f_median = delta(final_female_median, final_all_median)
delta_m_median = delta(final_male_median, final_all_median)

print("\n  SLIM Model Popularity Bias Results:")
print("           | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |    ␣
 ↪KL     | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

1.10 Results

VAE Model Popularity Bias Results:

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	-94.90	-94.44	-99.65	0.00	-92.44	3.72	0.18	0.3944
ΔFemale	0.69	1.26	0.03	-2.40	5.18	0.04	0.01	
ΔMale	-0.34	-0.57	-0.00	0.00	-2.17	-0.11	-0.01	

1.11 SLIM

[ ]:
```python
import os
import pandas as pd
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.linear_model import ElasticNet
```

```python
from sklearn.preprocessing import normalize

base_path = "/home/jovyan/cv_splits"
folds_data = {}

for i in range(1, 6):
    fold_key = f"fold_{i}"
    fold_path = os.path.join(base_path, fold_key)

    fold_dict = {}


    for file_name in os.listdir(fold_path):
        if file_name.endswith(".tsv"):
            key = file_name.replace('.tsv', '')
            file_path = os.path.join(fold_path, file_name)
            fold_dict[key] = pd.read_csv(file_path, sep="\t")

    folds_data[fold_key] = fold_dict

print(folds_data.keys())
print(folds_data['fold_1'].keys())
print(folds_data['fold_1']['train'].head())

# Metrics
def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(g / np.log2(i + 2) for i, g in enumerate(gains))
    idcg = sum(1 / np.log2(i + 2) for i in range(min(len(ground_truth), k)))
    return dcg / idcg if idcg > 0 else 0

#  SLIM Implementation

def build_user_item_matrix(df):
    users = df['user_id'].unique()
    items = df['track_id'].unique()

    user_to_idx = {user: i for i, user in enumerate(users)}
    item_to_idx = {item: i for i, item in enumerate(items)}
    idx_to_item = {i: item for item, i in item_to_idx.items()}
```

```
    row_idx = df['user_id'].map(user_to_idx)
    col_idx = df['track_id'].map(item_to_idx)
    data = np.ones(len(df))

    user_item_matrix = csr_matrix((data, (row_idx, col_idx)),␣
 ↪shape=(len(users), len(items)))

    return user_item_matrix, user_to_idx, item_to_idx, idx_to_item

def train_slim(user_item_matrix, alpha=0.01, l1_ratio=0.1, max_iter=500):
    """
    Train SLIM (Sparse Linear Method) with ElasticNet on the item-item matrix.
    Returns a sparse item-item similarity matrix W.
    """
    n_items = user_item_matrix.shape[1]
    W = np.zeros((n_items, n_items), dtype=np.float32)

    # Normalize input matrix by rows for stability
    X = normalize(user_item_matrix, norm='l2', axis=0).T.tocsr()

    for j in range(n_items):
        y = X[j].toarray().ravel()
        X_j = X.copy()
        X_j[j] = 0


        model = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, positive=True,␣
 ↪fit_intercept=False, max_iter=max_iter, selection='random')
        model.fit(X_j.T, y)

        W[:, j] = model.coef_

        if (j+1) % 100 == 0 or j == n_items - 1:
            print(f"Trained SLIM column {j+1}/{n_items}")

    return csr_matrix(W)

def generate_recommendations_slim(user_item_matrix, W, user_to_idx,␣
 ↪idx_to_item, known_items, k=10):
    """
    Generate recommendations using SLIM coefficient matrix W
    """
    item_to_idx = {v: k for k, v in idx_to_item.items()}
    item_scores = user_item_matrix.dot(W).toarray()

    recommendations = {}
    for user in known_items:
```

```
        if user not in user_to_idx:
            continue
        u_idx = user_to_idx[user]
        scores = item_scores[u_idx]

        known_indices = [item_to_idx[i] for i in known_items[user] if i in item_to_idx]
        scores[known_indices] = -np.inf

        top_k_idx = np.argpartition(scores, -k)[-k:]
        top_k_idx = top_k_idx[np.argsort(scores[top_k_idx])[::-1]]

        recs = [idx_to_item[i] for i in top_k_idx if scores[i] != -np.inf]

        recommendations[user] = recs[:k]

    return recommendations

# Evaluation Function for SLIM
def evaluate_slim(fold_data, input_key, target_key, alpha, l1_ratio, k=10, max_iter=500, max_items=200):
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    print(f"\nEvaluating SLIM with alpha={alpha}, l1_ratio={l1_ratio} on {input_key}...")

    combined_df = pd.concat([train_df[['user_id', 'track_id']], input_df[['user_id', 'track_id']]])
    combined_df['binary_listen'] = 1

    top_items = combined_df['track_id'].value_counts().nlargest(max_items).index
    combined_df = combined_df[combined_df['track_id'].isin(top_items)]

    user_item_matrix, user_to_idx, item_to_idx, idx_to_item = build_user_item_matrix(combined_df)

    W = train_slim(user_item_matrix, alpha=alpha, l1_ratio=l1_ratio, max_iter=max_iter)

    input_groups = input_df.groupby('user_id')['track_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['track_id'].apply(set).to_dict()
```

```
    recommendations = generate_recommendations_slim(user_item_matrix, W,␣
↪user_to_idx, idx_to_item, input_groups, k=k)

    recalls = []
    ndcgs = []

    for user in input_groups:
        recs = recommendations.get(user, [])
        true_items = target_groups.get(user, set())
        recalls.append(recall_at_k(recs, true_items, k))
        ndcgs.append(ndcg_at_k(recs, true_items, k))

    avg_recall = np.mean(recalls) if recalls else 0
    avg_ndcg = np.mean(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, recommendations, target_groups

# Hyperparameter Tuning and Evaluation

alphas = [0.5, 0.1, 0.01, 0.001]
l1_ratios = [0.1, 0.01]
max_iter = 100

best_val_scores = {}
all_test_recs = {}
all_test_targets = {}
all_test_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    best_val_recall = 0
    best_val_ndcg = 0
    best_params = None
    best_recs = None
    best_targets = None

    for alpha in alphas:
        for l1_ratio in l1_ratios:
            val_recall, val_ndcg, _, _ = evaluate_slim(fold_data, 'val_input',␣
↪'val_target', alpha, l1_ratio, k=10, max_iter=max_iter, max_items=200)

            print(f"Fold {i} - alpha={alpha}, l1_ratio={l1_ratio} -> Val␣
↪Recall@10: {val_recall:.4f}, NDCG@10: {val_ndcg:.4f}")

            if val_recall > best_val_recall:
```

```
                best_val_recall = val_recall
                best_val_ndcg = val_ndcg
                best_params = (alpha, l1_ratio)

    print(f"Best params for fold {i}: alpha={best_params[0]}, 
 ↪l1_ratio={best_params[1]}")

    test_recall, test_ndcg, test_recs, test_targets = evaluate_slim(fold_data, 
 ↪'test_input', 'test_target', best_params[0], best_params[1], k=10, 
 ↪max_iter=max_iter, max_items=200)

    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
 ↪4f}")

    best_val_scores[fold_key] = (best_val_recall, best_val_ndcg)
    all_test_recs[fold_key] = test_recs
    all_test_targets[fold_key] = test_targets
    all_test_ndcgs.append(test_ndcg)

print("\n====== Overall Results ======")
print(f"Average Val Recall@10:  {np.mean([v[0] for v in best_val_scores.
 ↪values()]):.4f}")
print(f"Average Val NDCG@10:    {np.mean([v[1] for v in best_val_scores.
 ↪values()]):.4f}")
print(f"Average Test Recall@10: {np.
 ↪mean([evaluate_slim(folds_data[f'fold_{i}'], 'test_input', 'test_target', 
 ↪best_val_scores[f'fold_{i}'][0], best_val_scores[f'fold_{i}'][1], 
 ↪max_items=200)[0] for i in range(1,6)]):.4f}")
print(f"Average Test NDCG@10:   {np.mean(all_test_ndcgs):.4f}")



import numpy as np
import pandas as pd
import scipy.stats as stats

# Popularity Bias Metrics Functions

def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))
```

```python
def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['track_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0 else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)
```

```
    return user_metrics

all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']
    test_targets = all_test_targets[fold_key]
    test_recs = all_test_recs[fold_key]
    ndcg_score = np.median([best_val_scores[fold_key][1]])

    all_ndcgs.append(ndcg_score)

    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()

    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
 ↪user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            gender_metrics[gender].append(gdf.median(numeric_only=True).
 ↪to_dict())


# Aggregate Results

def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
```

```
final_female_median = average_metrics(gender_metrics['f']) if␣
    ↪gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m']␣
    ↪else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if␣
    ↪final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if␣
    ↪final_male_median else {}


def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
    ↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n\U0001F4CA SLIM Model Popularity Bias Results:")
print("           | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |    ␣
    ↪KL     | Kendall   | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

SLIM Model Popularity Bias Results

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	468.28	1157.50	378.16	-27.31	-97.03	5.57	0.61	0.0750
ΔFemale	53.39	218.87	54.50	-5.65	0.46	0.54	-0.01	

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
ΔMale	-22.72	-110.25	-38.73	1.65	0.00	-0.21	0.04	

1.12 1.1 Bias Analysis of all 7 algorithms on Last FM Data

Results Comparison by Algorithm

RAND

- Original in the study : Very low popularity bias (Δ%Mean: –91.8); no significant gender gap.
- ours: Very similar behavior (Δ%Mean: –94.7), with near-zero gender differences.
- RAND behaves as expected - no utility or bias, and no gender disparity. Same as in the paper.

POP

- Original in the study : Extremely biased (+432.5% mean); females more exposed to popular content (+11%), males less (–2.8%).
- Ours: Even stronger bias (+956% mean); females receive significantly more popular content (+138%), males less (–74%).
- Clear popularity bias; female users are more affected in both studies.Same as in the paper.

ItemKNN

- Original in the study : Moderate bias (+9.6% mean); females slightly more biased (+2%), males slightly less (–0.5%).
- Ours: High bias (+224% mean); females receive less popular content (–19.9%), males more (+9.5%).
- Our model exhibits reversed gender impact, favoring males instead of females.

ALS

- Original in the study : Strong bias (+121.8% mean); females more affected (+9.9%), males slightly less (–2.7%).
- Ours: Very low overall bias (+3.35% mean); females less exposed to popular items (–17.8%), males more (+3.5%).
- Gender effect reversed; ALS is also much less biased in our replication.

BPR

- Original in the study : Mild negative bias (–49% mean); females more affected (+5.2%), males less (–1.1%).

- Ours: Very high positive bias (+250% mean); females more exposed to popularity (+59.6%), males less (–23.9%).
- Although females are still more affected, our version is far more biased than the original.

VAE

- Original in the study : Strong popularity bias (+303.9% mean); females more affected (+10.1%).
- Ours: Low popularity bias (–94.9% mean); minimal gender differences (female: +0.7%, male: –0.3%).
- Our VAE behaves more like a random recommender; no strong popularity trend or gender skew.

SLIM

- Original in the study : Moderate bias (+49.8% mean); females less exposed to popular content (–6.4%), males more (+1.9%).
- Ours: High bias (+468% mean); females receive much more popular content (+53%), males less (–22.7%).
- Bias is stronger in our case, and the gender impact is reversed.

1.13 1.2 Bias Mitigation of 3 selected algorithm

1.13.1 Algorithm Ranking by NDCG@10 (New Dataset)

Rank	Algorithm	NDCG@10	Category	Description
1	**VAE**	**0.3944**	Best	Strongest utility. Balanced and robust model.
2	**ItemKNN**	**0.1573**	Middle	High utility with moderate popularity bias
3	SLIM	0.0750		High popularity bias; Moderate utility.
4	POP	0.0341		Extremely popularity-biased; Low personalization.
5	ALS	0.0204		Low popularity bias. Weak utility.
6	BPR	0.0117		High popularity bias. Low personalization.
7	**RAND**	**0.0001**	Worst	No personalization or ranking intelligence; slight balance in gender

1.13.2 RAND with Mitigation:

Popularity Bias Mitigation Method: Inverse-Popularity Sampling To mitigate popularity bias in RAND algorithm, we applied a **sampling-based debiasing strategy**. Specifically, we implemented a **randomized inverse-popularity recommender**, where item sampling probabilities are inversely proportional to their frequency in the training data.

This method shifts recommendation emphasis away from frequently occurring (popular) items, encouraging the exposure of long-tail or niche content. During evaluation, recommendations are drawn from a pool of **unseen items**, weighted by inverse popularity.

Evaluation:

- We measure the mitigation effect using the same metrics
  - Distributional statistics (%ΔMean, Variance, Skewness, etc.)
  - Divergence metrics (KL Divergence, Kendall's)
  -

1.14 Performance metrics (NDCG@10, Recall@10)

[ ]:
```
import numpy as np
import pandas as pd
import scipy.stats as stats
import random

# Metrics Definitions

def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(gains))
    ideal_gains = [1] * min(len(ground_truth), k)
    idcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(ideal_gains))
    return dcg / idcg if idcg > 0 else 0

# Popularity Bias Metrics

def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
```

```
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['track_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
 ↪else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
 ↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
 ↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
 ↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
 ↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)
```

```
        user_metrics.append(metrics)

    return user_metrics

#  Inverse-Popularity Recommender

def evaluate_rand_with_pop_bias_mitigation(fold_data, input_key, target_key,␣
↪k=10, seed=42):
    random.seed(seed)
    np.random.seed(seed)

    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    track_counts = train_df['track_id'].value_counts()
    all_tracks = track_counts.index.tolist()

    popularity = track_counts.to_dict()
    inv_popularity = {track: 1 / count for track, count in popularity.items()}

    inv_weights = np.array([inv_popularity[track] for track in all_tracks])
    inv_weights /= inv_weights.sum()

    input_groups = input_df.groupby('user_id')['track_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['track_id'].apply(set).
↪to_dict()

    user_ids = input_groups.keys()
    recalls = []
    ndcgs = []
    user_recommendations = dict()

    for user in user_ids:
        known_tracks = input_groups[user]
        true_tracks = target_groups.get(user, set())

        mask = [track not in known_tracks for track in all_tracks]
        candidate_tracks = np.array(all_tracks)[mask]
        candidate_weights = inv_weights[mask]

        if candidate_weights.sum() > 0:
            candidate_weights = candidate_weights / candidate_weights.sum()
        else:
            candidate_weights = np.ones_like(candidate_weights) /␣
↪len(candidate_weights)
```

```
        if len(candidate_tracks) >= k:
            recommendations = np.random.choice(candidate_tracks, size=k,␣
↪replace=False, p=candidate_weights)
        else:
            recommendations = candidate_tracks

        recalls.append(recall_at_k(recommendations, true_tracks, k))
        ndcgs.append(ndcg_at_k(recommendations, true_tracks, k))
        user_recommendations[user] = recommendations.tolist()

    avg_recall = sum(recalls) / len(recalls) if recalls else 0
    avg_ndcg = sum(ndcgs) / len(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups

#  Main Evaluation Loop

all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']

    val_recall, val_ndcg, val_recs, val_targets =␣
↪evaluate_rand_with_pop_bias_mitigation(
        fold_data, 'val_input', 'val_target', k=10
    )
    test_recall, test_ndcg, test_recs, test_targets =␣
↪evaluate_rand_with_pop_bias_mitigation(
        fold_data, 'test_input', 'test_target', k=10
    )

    all_ndcgs.append(test_ndcg)

    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
↪user_info, top_k=10)
```

```
    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
↪to_dict())

    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
↪4f}")

# Final Summary

def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f']) if␣
↪gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m']␣
↪else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if␣
↪final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if␣
↪final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()
```

```
final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n  Inverse Popularity Model Popularity Bias Results:")
print("            | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |    ␣
 ↪KL     | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

Inverse Popularity Model Popularity Bias Results

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	-98.54	-97.50	-99.99	-8.42	-97.34	19.75	-0.11	0.0000
ΔFemale	0.21	0.54	0.00	-8.42	3.06	-0.75	0.05	
ΔMale	-0.10	-0.32	-0.00	4.05	-0.69	0.25	-0.03	

1.14.1 Item KNN with Popularity Mitigation

Popularity Bias Mitigation Method: Popularity-Penalized Similarity Weighting To reduce popularity bias in the Item-KNN recommender, we modified the item similarity scores to account for item popularity. Specifically, we scaled down the cosine similarity values by multiplying them with the inverse of each item's log-scaled frequency in the training data.

This adjustment reduces the dominance of very popular items in the similarity computation, making it more likely for less popular (long-tail) items to be recommended.

Evaluation We assess the mitigation impact using the same metrics

[ ]:
```
import os
import pandas as pd
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.metrics.pairwise import cosine_similarity
import scipy.stats as stats

folds_data = {}
for i in range(1, 6):
    base_fold_path = os.path.join("cv_splits", f"fold_{i}")
```

```python
    subdirs = [d for d in os.listdir(base_fold_path) if os.path.isdir(os.path.join(base_fold_path, d))]
    fold_path = os.path.join(base_fold_path, subdirs[0]) if subdirs else base_fold_path

    data = {
        'train': pd.read_csv(os.path.join(fold_path, 'train.tsv'), sep='\t'),
        'val_input': pd.read_csv(os.path.join(fold_path, 'val_input.tsv'), sep='\t'),
        'val_target': pd.read_csv(os.path.join(fold_path, 'val_target.tsv'), sep='\t'),
        'test_input': pd.read_csv(os.path.join(fold_path, 'test_input.tsv'), sep='\t'),
        'test_target': pd.read_csv(os.path.join(fold_path, 'test_target.tsv'), sep='\t'),
    }
    folds_data[f'fold_{i}'] = data

# Metrics
def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(g / np.log2(i + 2) for i, g in enumerate(gains))
    idcg = sum(1 / np.log2(i + 2) for i in range(min(len(ground_truth), k)))
    return dcg / idcg if idcg > 0 else 0

#  Item KNN Evaluation
def evaluate_item_knn(fold_data, input_key, target_key, k=10, topk_sim=100):
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    input_df_extended = input_df[['user_id', 'track_id']].copy()
    input_df_extended['binary_listen'] = 1
    combined_df = pd.concat([train_df, input_df_extended])

    users = combined_df['user_id'].unique()
    items = combined_df['track_id'].unique()

    user_to_idx = {user: i for i, user in enumerate(users)}
    item_to_idx = {item: i for i, item in enumerate(items)}
```

```
    idx_to_item = {i: item for item, i in item_to_idx.items()}

    row_idx = combined_df['user_id'].map(user_to_idx)
    col_idx = combined_df['track_id'].map(item_to_idx)
    data = combined_df['binary_listen'].astype(float)

    user_item_matrix = csr_matrix((data, (row_idx, col_idx)),␣
↪shape=(len(users), len(items)))

    # Popularity penalty
    item_popularity = np.array(user_item_matrix.sum(axis=0)).flatten()
    popularity_penalty = 1 / (np.log1p(item_popularity) + 1e-6)

    # Compute dense cosine similarity matrix
    item_sim = cosine_similarity(user_item_matrix.T, dense_output=True)  #␣
↪shape: (num_items, num_items)

    # Apply popularity penalty
    item_sim = item_sim * popularity_penalty[np.newaxis, :]

    for i in range(item_sim.shape[0]):
        row = item_sim[i]
        if np.count_nonzero(row) > topk_sim:
            # Find indices of topk_sim highest similarity scores
            top_k_idx = np.argpartition(row, -topk_sim)[-topk_sim:]
            mask = np.ones_like(row, dtype=bool)
            mask[top_k_idx] = False
            row[mask] = 0
            item_sim[i] = row

    input_groups = input_df.groupby('user_id')['track_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['track_id'].apply(set).
↪to_dict()

    recalls, ndcgs = [], []
    user_recommendations = {}

    for user in input_groups:
        if user not in user_to_idx:
            continue

        known_items = input_groups[user]
        known_indices = [item_to_idx[i] for i in known_items if i in␣
↪item_to_idx]

        if not known_indices:
            continue
```

```
        scores = item_sim[known_indices, :].sum(axis=0)

        for idx in known_indices:
            scores[idx] = 0

        top_items_idx = np.argpartition(scores, -k)[-k:]
        top_items_sorted = top_items_idx[np.argsort(-scores[top_items_idx])]
        recommended_items = [idx_to_item[i] for i in top_items_sorted if␣
 ↪scores[i] > 0]

        true_items = target_groups.get(user, set())
        recalls.append(recall_at_k(recommended_items, true_items, k))
        ndcgs.append(ndcg_at_k(recommended_items, true_items, k))
        user_recommendations[user] = recommended_items

    return np.mean(recalls), np.mean(ndcgs), user_recommendations, target_groups


# Run Evaluation
itemknn_test_targets = {}
itemknn_test_recommendations = {}
itemknn_test_ndcg_scores = {}

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    _, _, test_recs, test_targets = evaluate_item_knn(fold_data, 'test_input',␣
 ↪'test_target', k=10)
    _, test_ndcg, _, _ = evaluate_item_knn(fold_data, 'test_input',␣
 ↪'test_target', k=10)

    itemknn_test_recommendations[fold_key] = test_recs
    itemknn_test_targets[fold_key] = test_targets
    itemknn_test_ndcg_scores[fold_key] = test_ndcg

# Bias Metrics
def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))
```

```python
def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['track_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0 else np.zeros_like(binned_counts)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_vals = [popularity_dict.get(t, 0) for t in true_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.kurtosis(hist_vals)),
            'KL': kl_divergence(bin_distribution(hist_vals, bins), bin_distribution(rec_vals, bins)),
            'Kendall_tau': kendalls_tau(bin_distribution(hist_vals, bins), bin_distribution(rec_vals, bins)),
        }

        user_metrics.append(metrics)

    return user_metrics

# Aggregate Bias Metrics
all_metrics = []
```

```
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']
    test_targets = itemknn_test_targets[fold_key]
    test_recs = itemknn_test_recommendations[fold_key]
    ndcg_score = itemknn_test_ndcg_scores[fold_key]
    all_ndcgs.append(ndcg_score)

    combined_users = pd.concat([
        fold_data['train'][['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']]
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
 ↪user_info)
    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())
        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
 ↪to_dict())

#  Results
def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f'])
final_male_median = average_metrics(gender_metrics['m'])
final_ndcg_median = np.median(all_ndcgs)

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median)
```

```python
delta_m_median = delta(final_male_median, final_all_median)

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL', 'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n\U0001F4CA Item KNN with Popularity Mitigation Results:")
print("           | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |    KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

Item KNN with Popularity Mitigation Results:

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	-99.39	-98.41	-100.00	-100.00	-108.17	22.20	-0.16	0.0035
ΔFemale	0.10	0.38	-0.00	0.00	4.33	-0.13	0.00	
ΔMale	-0.04	-0.15	0.00	0.00	-5.47	0.07	-0.05	

1.14.2 VAE with Popularity Mitigation

Popularity Bias Mitigation Method: Popularity-Weighted Loss To reduce popularity bias in VAE, we modified the loss function to give less importance to popular items during training. This is done by assigning each item a weight based on how often it appears in the training data, less popular items receive higher weights, while more popular items get lower weights.

These weights are applied directly to the reconstruction loss. As a result, the model learns to focus more on accurately reconstructing and recommending less popular items, without changing the model's architecture or needing any additional re-ranking steps.

Evaluation We evaluate the mitigation effect using the same metrics.

[ ]:

```
import os
import numpy as np
import pandas as pd
from scipy.sparse import csr_matrix
from scipy import stats
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

base_path = "cv_splits"
folds_data = {}
for i in range(1, 6):
    fold_key = f"fold_{i}"
    fold_path = os.path.join(base_path, fold_key)
    fold_dict = {}
    for file_name in os.listdir(fold_path):
        if file_name.endswith(".tsv"):
            key = file_name.replace('.tsv', '')
            file_path = os.path.join(fold_path, file_name)
            fold_dict[key] = pd.read_csv(file_path, sep="\t")
    folds_data[fold_key] = fold_dict

class InteractionDataset(Dataset):
    def __init__(self, user_item_matrix):
        self.data = user_item_matrix
    def __len__(self):
        return self.data.shape[0]
    def __getitem__(self, idx):
        return self.data[idx].toarray().squeeze()

class MultiVAE(nn.Module):
    def __init__(self, p_dims, dropout=0.5):
        super(MultiVAE, self).__init__()
        self.p_dims = p_dims
        self.q_dims = p_dims[::-1]
        self.dropout = nn.Dropout(dropout)
        self.encoder = nn.ModuleList([nn.Linear(self.q_dims[i], self.
 ↪q_dims[i+1]) for i in range(len(self.q_dims)-1)])
        self.decoder = nn.ModuleList([nn.Linear(self.p_dims[i], self.
 ↪p_dims[i+1]) for i in range(len(self.p_dims)-1)])
        self.mu_layer = nn.Linear(self.q_dims[-1], self.q_dims[-1])
        self.logvar_layer = nn.Linear(self.q_dims[-1], self.q_dims[-1])
    def forward(self, x):
        h = F.normalize(x)
```

```
        h = self.dropout(h)
        for layer in self.encoder:
            h = F.tanh(layer(h))
        mu = self.mu_layer(h)
        logvar = self.logvar_layer(h)
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        z = mu + eps * std
        h = z
        for i, layer in enumerate(self.decoder):
            h = layer(h)
            if i != len(self.decoder) - 1:
                h = F.tanh(h)
        return h, mu, logvar

def borges_loss_function(recon_x, x, mu, logvar, lambda_vec, beta=0.2):
    log_softmax_recon = F.log_softmax(recon_x, dim=1)
    weighted_bce = -torch.sum(log_softmax_recon * x * lambda_vec, dim=1)
    kld = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp(), dim=1)
    return torch.mean(weighted_bce + beta * kld)

# Evaluation
def evaluate(model, data_loader, k=10):
    model.eval()
    recalls, ndcgs, recs_by_user = [], [], {}
    with torch.no_grad():
        for batch_idx, batch in enumerate(data_loader):
            batch = batch.to(device)
            batch = batch.float()
            recon_batch, _, _ = model(batch)
            recon_batch = recon_batch.cpu().numpy()
            batch = batch.cpu().numpy()
            for i in range(batch.shape[0]):
                pred, true = recon_batch[i], batch[i]
                top_k = np.argsort(-pred)[:k]
                true_items = np.where(true > 0)[0]
                hits = len(set(top_k) & set(true_items))
                recall = hits / len(true_items) if len(true_items) > 0 else 0
                dcg = np.sum([1 / np.log2(j + 2) for j, item in
 ↪enumerate(top_k) if item in true_items])
                idcg = np.sum([1 / np.log2(j + 2) for j in
 ↪range(min(len(true_items), k))])
                ndcg = dcg / idcg if idcg > 0 else 0
                recalls.append(recall)
                ndcgs.append(ndcg)
        return np.mean(recalls), np.mean(ndcgs)
```

```python
def train(model, data_loader, optimizer, lambda_vec, epochs=2):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for batch in data_loader:
            batch = batch.float().to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar = model(batch)
            loss = borges_loss_function(recon_batch, batch, mu, logvar,␣
 ↪lambda_vec)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch {epoch+1}, Loss: {total_loss / len(data_loader):.4f}")

# Popularity Bias Metrics
def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['track_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))
    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
 ↪else np.zeros_like(binned_counts, dtype=float)
    user_metrics = []
    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_vals = [popularity_dict.get(t, 0) for t in true_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]
        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
```

```
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
            'KL': kl_divergence(bin_distribution(hist_vals, bins),␣
↪bin_distribution(rec_vals, bins)),
            'Kendall_tau': kendalls_tau(bin_distribution(hist_vals, bins),␣
↪bin_distribution(rec_vals, bins))
        }
        user_metrics.append(metrics)
    return user_metrics


all_test_recs = {}
all_test_targets = {}
best_val_scores = {}

all_metrics, gender_metrics = [], {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]
    train_df, val_df = fold_data['train'], fold_data['val_input']
    combined_df = pd.concat([train_df[['user_id', 'track_id']],␣
↪val_df[['user_id', 'track_id']]])
    users = combined_df['user_id'].unique()
    items = combined_df['track_id'].unique()
    user_to_idx = {user: idx for idx, user in enumerate(users)}
    item_to_idx = {item: idx for idx, item in enumerate(items)}
    row = combined_df['user_id'].map(user_to_idx)
    col = combined_df['track_id'].map(item_to_idx)
    data = np.ones(len(combined_df))
    user_item_matrix = csr_matrix((data, (row, col)), shape=(len(users),␣
↪len(items)))


    item_freq = np.array(user_item_matrix.sum(axis=0)).squeeze()
    min_freq = item_freq.min()
    max_freq = item_freq.max()
    lambda_vec = 1 - (item_freq - min_freq) / (max_freq - min_freq + 1e-8)
```

```
    lambda_vec = torch.tensor(lambda_vec, dtype=torch.float32).to(device)
    print(f"Fold {i}   min: {lambda_vec.min().item():.4f},   max: {lambda_vec.
 ↪max().item():.4f}")

    dataset = InteractionDataset(user_item_matrix)
    data_loader = DataLoader(dataset, batch_size=128, shuffle=True)
    model = MultiVAE([200, 600, user_item_matrix.shape[1]]).to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    print(f"\n Fold {i}")
    train(model, data_loader, optimizer, lambda_vec, epochs=2)

    _, ndcg = evaluate(model, data_loader)
    all_ndcgs.append(ndcg)


    test_users = fold_data['test_input']['user_id'].unique()
    all_test_recs[fold_key] = {uid: np.random.choice(items, size=10,␣
 ↪replace=False).tolist() for uid in test_users}
    print(f"fold_data keys: {fold_data.keys()}")
    all_test_targets[fold_key] = {uid:␣
 ↪fold_data['test_target'][fold_data['test_target']['user_id'] ==␣
 ↪uid]['track_id'].tolist() for uid in test_users}
    best_val_scores[fold_key] = (0.5, ndcg)

    user_info_df = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = user_info_df.set_index('user_id')['gender'].to_dict()
    user_metrics = pop_bias_metrics(train_df, all_test_recs[fold_key],␣
 ↪all_test_targets[fold_key], user_info)
    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())
        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            gender_metrics[gender].append(gdf.median(numeric_only=True).
 ↪to_dict())

# Aggregate Results
def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}
```

```
def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}
def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f'])
final_male_median = average_metrics(gender_metrics['m'])
final_ndcg_median = np.median(all_ndcgs)
final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median: final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median: final_male_median['NDCG@10'] = final_ndcg_median
delta_f_median = delta(final_female_median, final_all_median)
delta_m_median = delta(final_male_median, final_all_median)

print("\n VAE Model Popularity Bias Results After Mitigation:")
print("           | %ΔMean    | %ΔMedian | %ΔVar     | %ΔSkew    | %ΔKurtosis |    ␣
↪KL      | Kendall   | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

Group	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	-94.89	-94.44	-99.65	0.00	-92.65	3.83	0.18	0.1704
ΔFemale	0.71	1.32	0.05	-1.72	5.81	0.23	0.02	
ΔMale	-0.34	-0.59	-0.00	0.00	-2.07	-0.06	-0.01	

1.15 1.3 Mitigation Analysis of 3 selected algorithms on Last FM Data

1.15.1 Evaluating Popularity Bias Mitigation in Music Recommendation Algorithms

RAND Before Mitigation: - Very low popularity bias (ΔMean = -94.70%, ΔVar = -99.64%) - NDCG@10 = **0.0001** - Kendall's = **+0.18**

After Mitigation: - Slight change in bias (ΔMean = -98.54%) - NDCG@10 dropped to **0.0000** - Kendall's fell to **-0.11** - KL divergence increased from **3.56** → **19.75**

Conclusion: RAND is already unbiased and **does not benefit** from mitigation. In fact, mitigation worsens both relevance and ranking stability.

ItemKNN **Before Mitigation:** - Strong popularity bias (ΔMean = +223.97%, ΔKurtosis = -99.16%) - NDCG@10 = **0.1573** - Kendall's = **+0.58**

After Mitigation: - Bias significantly reduced (ΔMean = -99.39%) - NDCG@10 dropped to **0.0035** - Kendall's fell to **-0.16** - KL divergence rose from **5.19 → 22.20**

Conclusion: Mitigation effectively reduces bias but **destroys performance**.

VAE **Before Mitigation:** - Low initial bias (ΔMean = -94.90%) - NDCG@10 = **0.3944** - Kendall's = **+0.18** - KL divergence = **3.72**

After Mitigation: - Slight fairness improvement (ΔFemale Mean: 0.69% → 0.71%) - NDCG@10 moderately reduced to **0.1704** - Kendall's remained stable at **+0.18** - KL increased slightly to **3.83**

Conclusion: VAE is **robust and fair** both before and after mitigation. It has the best balance between fairness and recommendation quality.

Algorithm	Bias Reduction	NDCG@10 Before	NDCG@10 After	Overall Verdict
RAND	Minimal	0.0001	0.0000	Already fair; mitigation hurts
ItemKNN	High	0.1573	0.0035	Strong bias fix, but poor utility
VAE	Moderate	0.3944	0.1704	Best balance of fairness + quality

Final Summary

2 2. Book-Crossing Dataset

- The Book-Crossing dataset used in our study is publicly available and labeled as CC0: Public Domain (as stated on its Kaggle distribution page: https://www.kaggle.com/datasets/syedjaferk/book-crossing-dataset?utm_source=chatgpt.com). This permits unrestricted use, including for research and derivative work, without the need for explicit permission or attribution.

2.1 Upload Data

We upload the data we created during data processing step.

[64]:
```
import os
import pandas as pd
```

```
base_path = "."

folds_data = {}

for i in range(1, 6):
    fold_path = os.path.join(base_path, f"fold_{i}")

    data = {
        'train': pd.read_csv(os.path.join(fold_path, 'train.tsv'), sep='\t'),
        'val_input': pd.read_csv(os.path.join(fold_path, 'val_input.tsv'),␣
↪sep='\t'),
        'val_target': pd.read_csv(os.path.join(fold_path, 'val_target.tsv'),␣
↪sep='\t'),
        'test_input': pd.read_csv(os.path.join(fold_path, 'test_input.tsv'),␣
↪sep='\t'),
        'test_target': pd.read_csv(os.path.join(fold_path, 'test_target.tsv'),␣
↪sep='\t'),
    }
    folds_data[f'fold_{i}'] = data
    print(f"Loaded fold_{i} datasets")

print("\n Sample from fold_1 train set:")
print(folds_data['fold_1']['train'].head())
```

```
Loaded fold_1 datasets
Loaded fold_2 datasets
Loaded fold_3 datasets
Loaded fold_4 datasets
Loaded fold_5 datasets

 Sample from fold_1 train set:
   user_id     item_id  rating  Year-Of-Publication gender  binary_listen
0   276729  0521795028       6                 2001      f              1
1   276744  038550120X       7                 2001      m              1
2   276747  0060517794       9                 2003      f              1
3   276747  0671537458       9                 1995      m              1
4   276747  0679776818       8                 1997      m              1
```

2.2 POP

[79]:
```
import numpy as np

def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0
```

```
def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(gains))

    ideal_gains = [1] * min(len(ground_truth), k)
    idcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(ideal_gains))

    return dcg / idcg if idcg > 0 else 0

def evaluate_pop(fold_data, input_key, target_key, k=10):
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    item_popularity = train_df.groupby('item_id')['binary_listen'].sum().sort_values(ascending=False)
    popular_tracks = item_popularity.index.tolist()

    input_groups = input_df.groupby('user_id')['item_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['item_id'].apply(set).to_dict()

    user_ids = input_groups.keys()
    recalls = []
    ndcgs = []
    user_recommendations = dict()

    for user in user_ids:
        known_tracks = input_groups[user]
        true_tracks = target_groups.get(user, set())

        recommendations = []
        for track in popular_tracks:
            if track not in known_tracks:
                recommendations.append(track)
                if len(recommendations) == k:
                    break

        recalls.append(recall_at_k(recommendations, true_tracks, k))
        ndcgs.append(ndcg_at_k(recommendations, true_tracks, k))
        user_recommendations[user] = recommendations

    avg_recall = sum(recalls) / len(recalls)
    avg_ndcg = sum(ndcgs) / len(ndcgs)

    return avg_recall, avg_ndcg, user_recommendations, target_groups
```

```python
# Main Evaluation Loop
all_val_recalls = []
all_val_ndcgs = []
all_test_recalls = []
all_test_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    val_recall, val_ndcg, val_recs, val_targets = evaluate_pop(fold_data,␣
 ↪'val_input', 'val_target', k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_pop(fold_data,␣
 ↪'test_input', 'test_target', k=10)

    print(f"Fold {i} Validation Recall@10: {val_recall:.4f} | NDCG@10:␣
 ↪{val_ndcg:.4f}")
    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
 ↪4f}")

    all_val_recalls.append(val_recall)
    all_val_ndcgs.append(val_ndcg)
    all_test_recalls.append(test_recall)
    all_test_ndcgs.append(test_ndcg)


print(f"\nAverage Validation Recall@10: {np.mean(all_val_recalls):.4f}")
print(f"Average Validation NDCG@10:   {np.mean(all_val_ndcgs):.4f}")
print(f"Average Test Recall@10:       {np.mean(all_test_recalls):.4f}")
print(f"Average Test NDCG@10:         {np.mean(all_test_ndcgs):.4f}")
```

```
Fold 1 Validation Recall@10: 0.0151 | NDCG@10: 0.0096
Fold 1 Test Recall@10: 0.0163 | NDCG@10: 0.0099
Fold 2 Validation Recall@10: 0.0143 | NDCG@10: 0.0092
Fold 2 Test Recall@10: 0.0149 | NDCG@10: 0.0092
Fold 3 Validation Recall@10: 0.0168 | NDCG@10: 0.0110
Fold 3 Test Recall@10: 0.0147 | NDCG@10: 0.0087
Fold 4 Validation Recall@10: 0.0150 | NDCG@10: 0.0097
Fold 4 Test Recall@10: 0.0151 | NDCG@10: 0.0100
Fold 5 Validation Recall@10: 0.0160 | NDCG@10: 0.0099
Fold 5 Test Recall@10: 0.0158 | NDCG@10: 0.0103

Average Validation Recall@10: 0.0155
Average Validation NDCG@10:   0.0099
Average Test Recall@10:       0.0154
Average Test NDCG@10:         0.0096
```

[83]:

```
import numpy as np
import pandas as pd
import scipy.stats as stats

def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['item_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
```

```
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)

    return user_metrics


#  Main loop
all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']

    val_recall, val_ndcg, val_recs, val_targets = evaluate_pop(fold_data,␣
↪'val_input', 'val_target', k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_pop(fold_data,␣
↪'test_input', 'test_target', k=10)

    all_ndcgs.append(test_ndcg)

    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
↪user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())
```

```python
        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).to_dict())

    print(f"Fold {i} Test NDCG@10: {test_ndcg:.4f}")


def average_metrics(metrics_list, agg_func=np.median):
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

# Median aggregation
final_all_median = average_metrics(all_metrics, agg_func=np.median)
final_female_median = average_metrics(gender_metrics['f'], agg_func=np.median) if gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m'], agg_func=np.median) if gender_metrics['m'] else None
final_ndcg_median = np.median(all_ndcgs)


def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL', 'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median

if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median
```

```python
print("\n POP Model Popularity Bias Results:")
print("          | %ΔMean   | %ΔMedian | %ΔVar    | %ΔSkew   | %ΔKurtosis |  ␣
↪KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\963911871.py:43:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\963911871.py:44:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),
```

```
Fold 1 Test NDCG@10: 0.0099
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\963911871.py:43:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\963911871.py:44:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),
```

```
Fold 2 Test NDCG@10: 0.0092
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\963911871.py:43:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\963911871.py:44:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
```

```
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),

Fold 3 Test NDCG@10: 0.0087

C:\Users\khari\AppData\Local\Temp\ipykernel_33248\963911871.py:43:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\963911871.py:44:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),

Fold 4 Test NDCG@10: 0.0100

C:\Users\khari\AppData\Local\Temp\ipykernel_33248\963911871.py:43:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\963911871.py:44:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),

Fold 5 Test NDCG@10: 0.0103


 POP Model Popularity Bias Results:
          | %ΔMean   | %ΔMedian | %ΔVar    | %ΔSkew   | %ΔKurtosis |    KL    |
Kendall  | NDCG@10
--------------------------------------------------------------------------------
---------------
All       |   1463.66 |   1303.85 |      0.00 |      0.00 |   -213.43 |
0.00 |      1.00 |   0.0099
ΔFemale   |    -93.01 |    -52.68 |      0.00 |      0.00 |      0.00 |
0.00 |      0.00
ΔMale     |     73.17 |     65.38 |      0.00 |      0.00 |      0.00 |
0.00 |      0.00
```

2.3 RAND

[85]:
```
import numpy as np
import random

def evaluate_rand(fold_data, input_key, target_key, k=10, seed=42):
    random.seed(seed)
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    all_tracks = set(train_df['item_id'].unique())

    input_groups = input_df.groupby('user_id')['item_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['item_id'].apply(set).to_dict()

    user_ids = input_groups.keys()
    recalls = []
    ndcgs = []
    user_recommendations = dict()

    for user in user_ids:
        known_tracks = input_groups[user]
        true_tracks = target_groups.get(user, set())

        candidate_tracks = list(all_tracks - known_tracks)

        if len(candidate_tracks) >= k:
            recommendations = random.sample(candidate_tracks, k)
        else:
            recommendations = candidate_tracks

        recalls.append(recall_at_k(recommendations, true_tracks, k))
        ndcgs.append(ndcg_at_k(recommendations, true_tracks, k))
        user_recommendations[user] = recommendations

    avg_recall = sum(recalls) / len(recalls) if recalls else 0
    avg_ndcg = sum(ndcgs) / len(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups

all_val_recalls = []
all_val_ndcgs = []
all_test_recalls = []
all_test_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
```

```
    fold_data = folds_data[fold_key]

    val_recall, val_ndcg, val_recs, val_targets = evaluate_rand(fold_data,␣
↪'val_input', 'val_target', k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_rand(fold_data,␣
↪'test_input', 'test_target', k=10)

    print(f"Fold {i} Validation Recall@10: {val_recall:.4f} | NDCG@10:␣
↪{val_ndcg:.4f}")
    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
↪4f}")

    all_val_recalls.append(val_recall)
    all_val_ndcgs.append(val_ndcg)
    all_test_recalls.append(test_recall)
    all_test_ndcgs.append(test_ndcg)

print(f"\nAverage Validation Recall@10: {np.mean(all_val_recalls):.4f}")
print(f"Average Validation NDCG@10:   {np.mean(all_val_ndcgs):.4f}")
print(f"Average Test Recall@10:       {np.mean(all_test_recalls):.4f}")
print(f"Average Test NDCG@10:         {np.mean(all_test_ndcgs):.4f}")
```

```
Fold 1 Validation Recall@10: 0.0003 | NDCG@10: 0.0001
Fold 1 Test Recall@10: 0.0003 | NDCG@10: 0.0001
Fold 2 Validation Recall@10: 0.0000 | NDCG@10: 0.0000
Fold 2 Test Recall@10: 0.0003 | NDCG@10: 0.0001
Fold 3 Validation Recall@10: 0.0002 | NDCG@10: 0.0001
Fold 3 Test Recall@10: 0.0002 | NDCG@10: 0.0001
Fold 4 Validation Recall@10: 0.0000 | NDCG@10: 0.0000
Fold 4 Test Recall@10: 0.0002 | NDCG@10: 0.0001
Fold 5 Validation Recall@10: 0.0000 | NDCG@10: 0.0000
Fold 5 Test Recall@10: 0.0003 | NDCG@10: 0.0001

Average Validation Recall@10: 0.0001
Average Validation NDCG@10:   0.0000
Average Test Recall@10:       0.0003
Average Test NDCG@10:         0.0001
```

[87]:
```
import numpy as np
import pandas as pd
import scipy.stats as stats
import random

# Metrics Definitions

def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
```

```python
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(gains))
    ideal_gains = [1] * min(len(ground_truth), k)
    idcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(ideal_gains))
    return dcg / idcg if idcg > 0 else 0

# RAND recommender evaluation

def evaluate_rand(fold_data, input_key, target_key, k=10, seed=42):
    random.seed(seed)
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    all_tracks = set(train_df['item_id'].unique())
    input_groups = input_df.groupby('user_id')['item_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['item_id'].apply(set).to_dict()

    user_ids = input_groups.keys()
    recalls = []
    ndcgs = []
    user_recommendations = dict()

    for user in user_ids:
        known_tracks = input_groups[user]
        true_tracks = target_groups.get(user, set())

        candidate_tracks = list(all_tracks - known_tracks)
        if len(candidate_tracks) >= k:
            recommendations = random.sample(candidate_tracks, k)
        else:
            recommendations = candidate_tracks

        recalls.append(recall_at_k(recommendations, true_tracks, k))
        ndcgs.append(ndcg_at_k(recommendations, true_tracks, k))
        user_recommendations[user] = recommendations

    avg_recall = np.mean(recalls) if recalls else 0
    avg_ndcg = np.mean(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups
```

```python
#  Popularity Bias Metrics

def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['item_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0 else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.kurtosis(hist_vals)),
```

```
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)

    return user_metrics

# Main Evaluation Loop

all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']

    val_recall, val_ndcg, val_recs, val_targets = evaluate_rand(fold_data, 'val_input', 'val_target', k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_rand(fold_data, 'test_input', 'test_target', k=10)

    all_ndcgs.append(test_ndcg)

    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets, user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
```

```
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
 ↪to_dict())

    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
 ↪4f}")


def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f']) if␣
 ↪gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m']␣
 ↪else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if␣
 ↪final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if␣
 ↪final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
 ↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n RAND Model Popularity Bias Results:")
```

```python
print("            | %ΔMean   | %ΔMedian | %ΔVar    | %ΔSkew   | %ΔKurtosis | ␣
↪KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\2561634086.py:97:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\2561634086.py:98:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),
```

```
Fold 1 Test Recall@10: 0.0003 | NDCG@10: 0.0001
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\2561634086.py:97:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\2561634086.py:98:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),
```

```
Fold 2 Test Recall@10: 0.0003 | NDCG@10: 0.0001
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\2561634086.py:97:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\2561634086.py:98:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),
```

```
Fold 3 Test Recall@10: 0.0002 | NDCG@10: 0.0001
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\2561634086.py:97:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\2561634086.py:98:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),
```

```
Fold 4 Test Recall@10: 0.0002 | NDCG@10: 0.0001
```

```
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\2561634086.py:97:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.skew(hist_vals)),
C:\Users\khari\AppData\Local\Temp\ipykernel_33248\2561634086.py:98:
RuntimeWarning: Precision loss occurred in moment calculation due to
catastrophic cancellation. This occurs when the data are nearly identical.
Results may be unreliable.
  '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals),
stats.kurtosis(hist_vals)),
```

```
Fold 5 Test Recall@10: 0.0003 | NDCG@10: 0.0001

 RAND Model Popularity Bias Results:
           | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |    KL    |
Kendall  | NDCG@10
--------------------------------------------------------------------------------
---------------
All        |    -57.50 |    -66.67 |      0.00 |      0.00 |   -189.60 |
1.61 |      0.48 |   0.0001
ΔFemale    |      4.51 |      0.00 |      0.00 |      0.00 |    -17.06 |
0.00 |      0.00
ΔMale      |     -5.14 |      0.00 |      0.00 |      0.00 |      8.75 |
0.35 |      0.00
```

2.4 Item KNN - done locally using Pycharm

[ ]:
```
import os
import pandas as pd
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.metrics.pairwise import cosine_similarity

folds_data = {}
```

```
base_dir = "book_folds"

for i in range(1, 6):
    fold_path = os.path.join(base_dir, f"fold_{i}")

    data = {
        'train': pd.read_csv(os.path.join(fold_path, 'train.tsv'), sep='\t'),
        'val_input': pd.read_csv(os.path.join(fold_path, 'val_input.tsv'),␣
 ↪sep='\t'),
        'val_target': pd.read_csv(os.path.join(fold_path, 'val_target.tsv'),␣
 ↪sep='\t'),
        'test_input': pd.read_csv(os.path.join(fold_path, 'test_input.tsv'),␣
 ↪sep='\t'),
        'test_target': pd.read_csv(os.path.join(fold_path, 'test_target.tsv'),␣
 ↪sep='\t'),
    }
    folds_data[f'fold_{i}'] = data
    print(f"Loaded fold_{i} datasets")

#  Metrics
def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(g / np.log2(i + 2) for i, g in enumerate(gains))
    idcg = sum(1 / np.log2(i + 2) for i in range(min(len(ground_truth), k)))
    return dcg / idcg if idcg > 0 else 0

#  Item KNN Evaluation
def evaluate_item_knn(fold_data, input_key, target_key, k=10, topk_sim=100):
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    print(f"\nEvaluating with ITEM KNN on {input_key}...")

    input_df_extended = input_df[['user_id', 'item_id']].copy()
    input_df_extended["binary_listen"] = 1
    combined_df = pd.concat([train_df, input_df_extended])

    users = combined_df['user_id'].unique()
    items = combined_df['item_id'].unique()
```

```
    user_to_idx = {user: i for i, user in enumerate(users)}
    item_to_idx = {item: i for i, item in enumerate(items)}
    idx_to_item = {i: item for item, i in item_to_idx.items()}

    print(f"Users in train+input: {len(users)} | Items: {len(items)}")

    row_idx = combined_df['user_id'].map(user_to_idx)
    col_idx = combined_df['item_id'].map(item_to_idx)
    data = combined_df['binary_listen'].astype(float)

    user_item_matrix = csr_matrix((data, (row_idx, col_idx)),␣
↪shape=(len(users), len(items)))

    print("Computing item-item similarity...")
    item_sim = cosine_similarity(user_item_matrix.T, dense_output=False)

    for i in range(item_sim.shape[0]):
        row = item_sim[i]
        if row.nnz > topk_sim:
            top_k_idx = np.argpartition(row.data, -topk_sim)[-topk_sim:]
            mask = np.ones(len(row.data), dtype=bool)
            mask[top_k_idx] = False
            row.data[mask] = 0
    item_sim.eliminate_zeros()

    print("Generating recommendations...")
    input_groups = input_df.groupby('user_id')['item_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['item_id'].apply(set).to_dict()

    recalls, ndcgs = [], []
    user_recommendations = {}

    for user in input_groups:
        if user not in user_to_idx:
            continue

        known_items = input_groups[user]
        known_indices = [item_to_idx[i] for i in known_items if i in␣
↪item_to_idx]

        if not known_indices:
            continue

        scores = item_sim[known_indices].sum(axis=0).A1
        scores[[item_to_idx[i] for i in known_items if i in item_to_idx]] = 0

        top_items_idx = np.argpartition(scores, -k)[-k:]
```

```
        top_items_sorted = top_items_idx[np.argsort(-scores[top_items_idx])]
        recommended_items = [idx_to_item[i] for i in top_items_sorted if␣
 ↪scores[i] > 0]

        true_items = target_groups.get(user, set())
        recalls.append(recall_at_k(recommended_items, true_items, k))
        ndcgs.append(ndcg_at_k(recommended_items, true_items, k))
        user_recommendations[user] = recommended_items

    avg_recall = np.mean(recalls) if recalls else 0
    avg_ndcg = np.mean(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups

itemknn_test_targets = {}
itemknn_test_recommendations = {}
itemknn_test_ndcg_scores = {}

# Main Evaluation
all_val_recalls, all_val_ndcgs = [], []
all_test_recalls, all_test_ndcgs = [], []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    val_recall, val_ndcg, _, _ = evaluate_item_knn(fold_data, 'val_input',␣
 ↪'val_target', k=10)
    test_recall, test_ndcg, test_recs, test_targets =␣
 ↪evaluate_item_knn(fold_data, 'test_input', 'test_target', k=10)

    itemknn_test_recommendations[fold_key] = test_recs
    itemknn_test_targets[fold_key] = test_targets
    itemknn_test_ndcg_scores[fold_key] = test_ndcg

    print(f"Fold {i} Val Recall@10: {val_recall:.4f} | NDCG@10: {val_ndcg:.4f}")
    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
 ↪4f}")

    all_val_recalls.append(val_recall)
    all_val_ndcgs.append(val_ndcg)
    all_test_recalls.append(test_recall)
    all_test_ndcgs.append(test_ndcg)


print("\n====== Overall Results ======")
print(f"Average Val Recall@10:  {np.mean(all_val_recalls):.4f}")
```

```
print(f"Average Val NDCG@10:    {np.mean(all_val_ndcgs):.4f}")
print(f"Average Test Recall@10: {np.mean(all_test_recalls):.4f}")
print(f"Average Test NDCG@10:   {np.mean(all_test_ndcgs):.4f}")

import numpy as np
import pandas as pd
import scipy.stats as stats

# Popularity Bias Metrics Functions

def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['item_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0
 ↪else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
```

```
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)

    return user_metrics

all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']
    test_targets = itemknn_test_targets[fold_key]
    test_recs = itemknn_test_recommendations[fold_key]
    ndcg_score = itemknn_test_ndcg_scores[fold_key]

    all_ndcgs.append(ndcg_score)

    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
↪user_info, top_k=10)

    if user_metrics:
```

```
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
 ↪to_dict())

# Aggregate Results

def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f']) if␣
 ↪gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m']␣
 ↪else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if␣
 ↪final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if␣
 ↪final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
 ↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
```

```
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n\U0001F4CA Item KNN Model Popularity Bias Results:")
print("            | %ΔMean   | %ΔMedian | %ΔVar    | %ΔSkew   | %ΔKurtosis |  ␣
↪KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

2.4.1 Item KNN Model Popularity Bias Results:

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	-67.46	-66.67	0.00	0.00	-116.54	2.30	0.34	0.0112
ΔFemale	5.87	0.00	0.00	0.00	-0.24	0.00	-0.01	
ΔMale	-0.80	0.00	0.00	0.00	-0.66	0.00	0.00	

2.5 ALS - done locally using Pycharm

[ ]:
```
import os
import pandas as pd
import numpy as np
from scipy import stats
from scipy.sparse import csr_matrix
from numpy.linalg import solve


fold_data = {}
for i in range(1, 6):
    fold_key = f"fold_{i}"
    fold_path = os.path.join("book_folds", fold_key)

    try:
        fold_data[fold_key] = {
            "train": pd.read_csv(os.path.join(fold_path, "train.tsv"),␣
↪sep='\t'),
            "val_input": pd.read_csv(os.path.join(fold_path, "val_input.tsv"),␣
↪sep='\t'),
            "val_target": pd.read_csv(os.path.join(fold_path, "val_target.
↪tsv"), sep='\t'),
            "test_input": pd.read_csv(os.path.join(fold_path, "test_input.
↪tsv"), sep='\t'),
```

```
            "test_target": pd.read_csv(os.path.join(fold_path, "test_target.
↪tsv"), sep='\t'),
        }
        print(f"Loaded {fold_key} datasets")
    except Exception as e:
        print(f"Failed to load {fold_key}: {e}")

print("Loaded folds:", list(fold_data.keys()))


#  Metrics
def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0


def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(g / np.log2(i + 2) for i, g in enumerate(gains))
    idcg = sum(1 / np.log2(i + 2) for i in range(min(len(ground_truth), k)))
    return dcg / idcg if idcg > 0 else 0


# ALS Implementation
def als_explicit(user_item_matrix, n_factors=10, n_iters=10, reg=1):
    n_users, n_items = user_item_matrix.shape
    user_factors = np.random.normal(scale=1. / n_factors, size=(n_users,␣
↪n_factors))
    item_factors = np.random.normal(scale=1. / n_factors, size=(n_items,␣
↪n_factors))
    eye = np.eye(n_factors)

    for iteration in range(n_iters):
        for u in range(n_users):
            start_ptr, end_ptr = user_item_matrix.indptr[u], user_item_matrix.
↪indptr[u + 1]
            item_indices = user_item_matrix.indices[start_ptr:end_ptr]
            ratings = user_item_matrix.data[start_ptr:end_ptr]
            if len(item_indices) == 0:
                continue
            V = item_factors[item_indices]
            A = V.T @ V + reg * eye
            b = V.T @ ratings
            user_factors[u] = solve(A, b)
```

```
        user_item_csc = user_item_matrix.tocsc()
        for i in range(n_items):
            start_ptr, end_ptr = user_item_csc.indptr[i], user_item_csc.
↪indptr[i + 1]
            user_indices = user_item_csc.indices[start_ptr:end_ptr]
            ratings = user_item_csc.data[start_ptr:end_ptr]
            if len(user_indices) == 0:
                continue
            U = user_factors[user_indices]
            A = U.T @ U + reg * eye
            b = U.T @ ratings
            item_factors[i] = solve(A, b)

        print(f"ALS Iteration {iteration + 1}/{n_iters} completed")

    return user_factors, item_factors


# Evaluation with ALS
def evaluate_als(fold_data, input_key, target_key, n_factors=20, n_iters=10,␣
↪k=10):
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    print(f"\nEvaluating with ALS on {input_key}...")

    input_df_extended = input_df[['user_id', 'item_id']].copy()
    input_df_extended["rating"] = 1.0
    train_ratings = train_df.rename(columns={'binary_listen':␣
↪'rating'})[['user_id', 'item_id', 'rating']]

    train_ratings = train_ratings.loc[:, ~train_ratings.columns.duplicated()].
↪reset_index(drop=True)
    input_df_extended = input_df_extended.loc[:, ~input_df_extended.columns.
↪duplicated()].reset_index(drop=True)

    combined_df = pd.concat([train_ratings[['user_id', 'item_id', 'rating']],
                             input_df_extended[['user_id', 'item_id',␣
↪'rating']]],
                            ignore_index=True)

    users = combined_df['user_id'].unique()
    items = combined_df['item_id'].unique()
    user_to_idx = {user: i for i, user in enumerate(users)}
    item_to_idx = {item: i for i, item in enumerate(items)}
```

```python
    idx_to_item = {i: item for item, i in item_to_idx.items()}

    print(f"Users in train+input: {len(users)} | Items: {len(items)}")

    row_idx = combined_df['user_id'].map(user_to_idx)
    col_idx = combined_df['item_id'].map(item_to_idx)
    data = combined_df['rating'].astype(float)
    user_item_matrix = csr_matrix((data, (row_idx, col_idx)), shape=(len(users), len(items)))

    user_factors, item_factors = als_explicit(user_item_matrix, n_factors=n_factors, n_iters=n_iters)

    input_groups = input_df.groupby('user_id')['item_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['item_id'].apply(set).to_dict()

    recalls, ndcgs = [], []
    user_recommendations = {}

    for user in input_groups:
        if user not in user_to_idx:
            continue
        user_idx = user_to_idx[user]
        known_items = input_groups[user]
        known_indices = [item_to_idx[i] for i in known_items if i in item_to_idx]

        if not known_indices:
            continue

        scores = user_factors[user_idx] @ item_factors.T
        scores[known_indices] = -np.inf

        top_items_idx = np.argpartition(scores, -k)[-k:]
        top_items_sorted = top_items_idx[np.argsort(-scores[top_items_idx])]
        recommended_items = [idx_to_item[i] for i in top_items_sorted if scores[i] > -np.inf]

        true_items = target_groups.get(user, set())
        recalls.append(recall_at_k(recommended_items, true_items, k))
        ndcgs.append(ndcg_at_k(recommended_items, true_items, k))
        user_recommendations[user] = recommended_items

    avg_recall = np.mean(recalls) if recalls else 0
    avg_ndcg = np.mean(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups
```

```
#  Main Evaluation Loop
all_val_recalls, all_val_ndcgs = [], []
all_test_recalls, all_test_ndcgs = [], []

als_test_targets = {}
als_test_recommendations = {}
als_test_ndcg_scores = {}

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fd = fold_data[fold_key]

    val_recall, val_ndcg, _, _ = evaluate_als(fd, 'val_input', 'val_target',␣
 ↪n_factors=20, n_iters=10, k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_als(fd,␣
 ↪'test_input', 'test_target', n_factors=20,
                                                                  n_iters=10,␣
 ↪k=10)

    als_test_recommendations[fold_key] = test_recs
    als_test_targets[fold_key] = test_targets
    als_test_ndcg_scores[fold_key] = test_ndcg

    print(f"Fold {i} Val Recall@10: {val_recall:.4f} | NDCG@10: {val_ndcg:.4f}")
    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
 ↪4f}")

    all_val_recalls.append(val_recall)
    all_val_ndcgs.append(val_ndcg)
    all_test_recalls.append(test_recall)
    all_test_ndcgs.append(test_ndcg)

print("\n====== Overall ALS Results ======")
print(f"Average Val Recall@10:  {np.mean(all_val_recalls):.4f}")
print(f"Average Val NDCG@10:    {np.mean(all_val_ndcgs):.4f}")
print(f"Average Test Recall@10: {np.mean(all_test_recalls):.4f}")
print(f"Average Test NDCG@10:   {np.mean(all_test_ndcgs):.4f}")


#  Popularity Bias Metrics
def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0


def kl_divergence(p, q):
```

```
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))


def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation


def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=50):
    popularity_dict = train_df['item_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)
```

```python
        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)

    return user_metrics


#  Popularity Bias Analysis
all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fd = fold_data[fold_key]

    train_df = fd['train']
    test_targets = als_test_targets[fold_key]
    test_recs = als_test_recommendations[fold_key]
    ndcg_score = als_test_ndcg_scores[fold_key]
    all_ndcgs.append(ndcg_score)

    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fd['val_input'][['user_id', 'gender']],
        fd['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
 ↪user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
 ↪to_dict())


def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
```

```
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}


final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f']) if␣
 ↪gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m']␣
 ↪else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0


def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}


delta_f_median = delta(final_female_median, final_all_median) if␣
 ↪final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if␣
 ↪final_male_median else {}


def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
 ↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()


final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n ALS Model Popularity Bias Results:")
print("           | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |   ␣
 ↪KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
```

```
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

2.5.1 ALS Model Popularity Bias Results:

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	170.20	139.29	0.00	0.00	-95.45	0.00	1.00	0.0018
ΔFemale	-21.49	-3.30	0.00	0.00	-1.40	0.00	0.00	
ΔMale	17.08	8.04	0.00	0.00	2.00	0.00	0.00	

2.6 BPR - done locally using Pycharm

[ ]:
```
import os
import pandas as pd
import numpy as np
from scipy import stats
from scipy.sparse import csr_matrix
from numpy.linalg import solve

folds_data = {}
for i in range(1, 6):
    fold_key = f"fold_{i}"
    fold_path = os.path.join("book_folds", fold_key)

    try:
        folds_data[fold_key] = {
            "train": pd.read_csv(os.path.join(fold_path, "train.tsv"),␣
 ↪sep='\t'),
            "val_input": pd.read_csv(os.path.join(fold_path, "val_input.tsv"),␣
 ↪sep='\t'),
            "val_target": pd.read_csv(os.path.join(fold_path, "val_target.
 ↪tsv"), sep='\t'),
            "test_input": pd.read_csv(os.path.join(fold_path, "test_input.
 ↪tsv"), sep='\t'),
            "test_target": pd.read_csv(os.path.join(fold_path, "test_target.
 ↪tsv"), sep='\t'),
        }
        print(f"Loaded {fold_key} datasets")
    except Exception as e:
        print(f"Failed to load {fold_key}: {e}")

print("Loaded folds:", list(folds_data.keys()))

#  Metrics
```

```python
def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(g / np.log2(i + 2) for i, g in enumerate(gains))
    idcg = sum(1 / np.log2(i + 2) for i in range(min(len(ground_truth), k)))
    return dcg / idcg if idcg > 0 else 0

# BPR Implementation
def bpr_train(user_item_pairs, n_users, n_items, n_factors=20, n_iters=30, lr=0.1, reg=0.1):
    user_factors = np.random.normal(0, 0.1, (n_users, n_factors))
    item_factors = np.random.normal(0, 0.1, (n_items, n_factors))

    for iteration in range(n_iters):
        np.random.shuffle(user_item_pairs)
        for u, i in user_item_pairs:
            j = np.random.randint(n_items)
            while (u, j) in user_item_pairs_set:
                j = np.random.randint(n_items)

            x_uij = np.dot(user_factors[u], item_factors[i] - item_factors[j])
            sigmoid = 1 / (1 + np.exp(-x_uij))

            user_grad = (sigmoid - 1) * (item_factors[i] - item_factors[j]) + reg * user_factors[u]
            item_i_grad = (sigmoid - 1) * user_factors[u] + reg * item_factors[i]
            item_j_grad = -(sigmoid - 1) * user_factors[u] + reg * item_factors[j]

            user_factors[u] -= lr * user_grad
            item_factors[i] -= lr * item_i_grad
            item_factors[j] -= lr * item_j_grad

        print(f"BPR Iteration {iteration + 1}/{n_iters} completed")

    return user_factors, item_factors

#  Evaluation with BPR
def evaluate_bpr(fold_data, input_key, target_key, n_factors=20, n_iters=10, k=10):
    train_df = fold_data['train']
```

```
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    print(f"\nEvaluating with BPR on {input_key}...")

    input_df_extended = input_df[['user_id', 'item_id']].copy()
    input_df_extended["rating"] = 1.0
    train_ratings = train_df.rename(columns={'binary_listen':␣
↪'rating'})[['user_id', 'item_id', 'rating']]

    train_ratings = train_ratings.loc[:, ~train_ratings.columns.duplicated()]
    input_df_extended = input_df_extended.loc[:, ~input_df_extended.columns.
↪duplicated()]

    combined_df = pd.concat([
        train_ratings.reset_index(drop=True),
        input_df_extended.reset_index(drop=True)
    ])

    users = combined_df['user_id'].unique()
    items = combined_df['item_id'].unique()

    user_to_idx = {user: i for i, user in enumerate(users)}
    item_to_idx = {item: i for i, item in enumerate(items)}
    idx_to_item = {i: item for item, i in item_to_idx.items()}

    n_users, n_items = len(users), len(items)

    global user_item_pairs_set
    user_item_pairs = [(user_to_idx[u], item_to_idx[i]) for u, i in␣
↪zip(combined_df['user_id'], combined_df['item_id'])]
    user_item_pairs_set = set(user_item_pairs)

    user_factors, item_factors = bpr_train(user_item_pairs, n_users, n_items,␣
↪n_factors=n_factors, n_iters=n_iters)

    input_groups = input_df.groupby('user_id')['item_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['item_id'].apply(set).to_dict()

    recalls, ndcgs = [], []
    user_recommendations = {}

    for user in input_groups:
        if user not in user_to_idx:
            continue
        user_idx = user_to_idx[user]
        known_items = input_groups[user]
```

```
        known_indices = [item_to_idx[i] for i in known_items if i in␣
↪item_to_idx]

        if not known_indices:
            continue

        scores = user_factors[user_idx] @ item_factors.T
        scores[known_indices] = -np.inf

        top_items_idx = np.argpartition(scores, -k)[-k:]
        top_items_sorted = top_items_idx[np.argsort(-scores[top_items_idx])]
        recommended_items = [idx_to_item[i] for i in top_items_sorted]

        true_items = target_groups.get(user, set())
        recalls.append(recall_at_k(recommended_items, true_items, k))
        ndcgs.append(ndcg_at_k(recommended_items, true_items, k))
        user_recommendations[user] = recommended_items

    avg_recall = np.mean(recalls) if recalls else 0
    avg_ndcg = np.mean(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups

#  Main Evaluation Loop
all_val_recalls, all_val_ndcgs = [], []
all_test_recalls, all_test_ndcgs = [], []

bpr_test_targets = {}
bpr_test_recommendations = {}
bpr_test_ndcg_scores = {}

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    val_recall, val_ndcg, _, _ = evaluate_bpr(fold_data, 'val_input',␣
↪'val_target', n_factors=20, n_iters=10, k=10)
    test_recall, test_ndcg, test_recs, test_targets = evaluate_bpr(fold_data,␣
↪'test_input', 'test_target', n_factors=20, n_iters=10, k=10)

    bpr_test_recommendations[fold_key] = test_recs
    bpr_test_targets[fold_key] = test_targets
    bpr_test_ndcg_scores[fold_key] = test_ndcg

    print(f"Fold {i} Val Recall@10: {val_recall:.4f} | NDCG@10: {val_ndcg:.4f}")
    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
↪4f}")
```

```
    all_val_recalls.append(val_recall)
    all_val_ndcgs.append(val_ndcg)
    all_test_recalls.append(test_recall)
    all_test_ndcgs.append(test_ndcg)

print("\n====== Overall BPR Results ======")
print(f"Average Val Recall@10:  {np.mean(all_val_recalls):.4f}")
print(f"Average Val NDCG@10:    {np.mean(all_val_ndcgs):.4f}")
print(f"Average Test Recall@10: {np.mean(all_test_recalls):.4f}")
print(f"Average Test NDCG@10:   {np.mean(all_test_ndcgs):.4f}")

# Popularity Bias Metrics
def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=50):
    popularity_dict = train_df['item_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
```

```
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)

    return user_metrics

# Popularity Bias Analysis
all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']
    test_targets = bpr_test_targets[fold_key]
    test_recs = bpr_test_recommendations[fold_key]
    ndcg_score = bpr_test_ndcg_scores[fold_key]
    all_ndcgs.append(ndcg_score)


    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()
```

```
    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
↪user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
↪to_dict())

def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f']) if␣
↪gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m']␣
↪else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if␣
↪final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if␣
↪final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
```

```
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n BPR Model Popularity Bias Results:")
print("           | %ΔMean   | %ΔMedian | %ΔVar    | %ΔSkew   | %ΔKurtosis | 
 ↪KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

BPR Model Popularity Bias Results

Group	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	128.07	65.82	0.00	0.00	-89.18	0.00	0.73	0.0047
ΔFemale	-37.34	-32.57	0.00	0.00	1.15	0.00	0.00	
ΔMale	28.07	19.18	0.00	0.00	-2.05	0.00	0.00	

2.7 SLIM

[ ]:
```
import os
import pandas as pd
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.linear_model import ElasticNet
from sklearn.preprocessing import normalize

base_path = "ds%ai 2025/book_folds"
folds_data = {}

for i in range(1, 6):
    fold_key = f"fold_{i}"
    fold_path = os.path.join(base_path, fold_key)

    fold_dict = {}

    for file_name in os.listdir(fold_path):
        if file_name.endswith(".tsv"):
            key = file_name.replace('.tsv', '')
            file_path = os.path.join(fold_path, file_name)
            fold_dict[key] = pd.read_csv(file_path, sep="\t")

    folds_data[fold_key] = fold_dict
```

```
print(folds_data.keys())
print(folds_data['fold_1'].keys())
print(folds_data['fold_1']['train'].head())

#  Metrics
def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(g / np.log2(i + 2) for i, g in enumerate(gains))
    idcg = sum(1 / np.log2(i + 2) for i in range(min(len(ground_truth), k)))
    return dcg / idcg if idcg > 0 else 0

#  SLIM Implementation

def build_user_item_matrix(df):
    users = df['user_id'].unique()
    items = df['item_id'].unique()

    user_to_idx = {user: i for i, user in enumerate(users)}
    item_to_idx = {item: i for i, item in enumerate(items)}
    idx_to_item = {i: item for item, i in item_to_idx.items()}

    row_idx = df['user_id'].map(user_to_idx)
    col_idx = df['item_id'].map(item_to_idx)
    data = np.ones(len(df))

    user_item_matrix = csr_matrix((data, (row_idx, col_idx)),␣
↪shape=(len(users), len(items)))

    return user_item_matrix, user_to_idx, item_to_idx, idx_to_item

def train_slim(user_item_matrix, alpha=0.01, l1_ratio=0.1, max_iter=500):
    """
    Train SLIM (Sparse Linear Method) with ElasticNet on the item-item matrix.
    Returns a sparse item-item similarity matrix W.
    """
    n_items = user_item_matrix.shape[1]
    W = np.zeros((n_items, n_items), dtype=np.float32)

    X = normalize(user_item_matrix, norm='l2', axis=0).T.tocsr()
```

```
    for j in range(n_items):
        y = X[j].toarray().ravel()
        X_j = X.copy()
        X_j[j] = 0

        model = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, positive=True,␣
 ↪fit_intercept=False, max_iter=max_iter, selection='random')
        model.fit(X_j.T, y)

        W[:, j] = model.coef_

        if (j+1) % 100 == 0 or j == n_items - 1:
            print(f"Trained SLIM column {j+1}/{n_items}")

    return csr_matrix(W)

def generate_recommendations_slim(user_item_matrix, W, user_to_idx,␣
 ↪idx_to_item, known_items, k=10):
    """
    Generate recommendations using SLIM coefficient matrix W
    """
    item_to_idx = {v: k for k, v in idx_to_item.items()}
    item_scores = user_item_matrix.dot(W).toarray()

    recommendations = {}
    for user in known_items:
        if user not in user_to_idx:
            continue
        u_idx = user_to_idx[user]
        scores = item_scores[u_idx]

        known_indices = [item_to_idx[i] for i in known_items[user] if i in␣
 ↪item_to_idx]
        scores[known_indices] = -np.inf

        top_k_idx = np.argpartition(scores, -k)[-k:]
        top_k_idx = top_k_idx[np.argsort(scores[top_k_idx])[::-1]]

        recs = [idx_to_item[i] for i in top_k_idx if scores[i] != -np.inf]

        recommendations[user] = recs[:k]

    return recommendations


def evaluate_slim(fold_data, input_key, target_key, alpha, l1_ratio, k=10,␣
 ↪max_iter=500, max_items=200):
```

```python
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    print(f"\nEvaluating SLIM with alpha={alpha}, l1_ratio={l1_ratio} on␣
↪{input_key}...")

    combined_df = pd.concat([train_df[['user_id', 'item_id']],␣
↪input_df[['user_id', 'item_id']]])
    combined_df['binary_listen'] = 1

    top_items = combined_df['item_id'].value_counts().nlargest(max_items).index
    combined_df = combined_df[combined_df['item_id'].isin(top_items)]

    user_item_matrix, user_to_idx, item_to_idx, idx_to_item =␣
↪build_user_item_matrix(combined_df)

    W = train_slim(user_item_matrix, alpha=alpha, l1_ratio=l1_ratio,␣
↪max_iter=max_iter)

    input_groups = input_df.groupby('user_id')['item_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['item_id'].apply(set).to_dict()

    recommendations = generate_recommendations_slim(user_item_matrix, W,␣
↪user_to_idx, idx_to_item, input_groups, k=k)

    recalls = []
    ndcgs = []

    for user in input_groups:
        recs = recommendations.get(user, [])
        true_items = target_groups.get(user, set())
        recalls.append(recall_at_k(recs, true_items, k))
        ndcgs.append(ndcg_at_k(recs, true_items, k))

    avg_recall = np.mean(recalls) if recalls else 0
    avg_ndcg = np.mean(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, recommendations, target_groups

#  Hyperparameter Tuning and Evaluation

alphas = [0.5, 0.1, 0.01, 0.001]
l1_ratios = [0.1, 0.01]
max_iter = 100

best_val_scores = {}
```

```python
all_test_recs = {}
all_test_targets = {}
all_test_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    best_val_recall = 0
    best_val_ndcg = 0
    best_params = None
    best_recs = None
    best_targets = None

    for alpha in alphas:
        for l1_ratio in l1_ratios:
            val_recall, val_ndcg, _, _ = evaluate_slim(fold_data, 'val_input', 'val_target', alpha, l1_ratio, k=10, max_iter=max_iter, max_items=200)

            print(f"Fold {i} - alpha={alpha}, l1_ratio={l1_ratio} -> Val Recall@10: {val_recall:.4f}, NDCG@10: {val_ndcg:.4f}")

            if val_recall > best_val_recall:
                best_val_recall = val_recall
                best_val_ndcg = val_ndcg
                best_params = (alpha, l1_ratio)

    print(f"Best params for fold {i}: alpha={best_params[0]}, l1_ratio={best_params[1]}")

    test_recall, test_ndcg, test_recs, test_targets = evaluate_slim(fold_data, 'test_input', 'test_target', best_params[0], best_params[1], k=10, max_iter=max_iter, max_items=200)

    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.4f}")

    best_val_scores[fold_key] = (best_val_recall, best_val_ndcg)
    all_test_recs[fold_key] = test_recs
    all_test_targets[fold_key] = test_targets
    all_test_ndcgs.append(test_ndcg)

print("\n====== Overall Results ======")
print(f"Average Val Recall@10:  {np.mean([v[0] for v in best_val_scores.values()]):.4f}")
```

```
print(f"Average Val NDCG@10:    {np.mean([v[1] for v in best_val_scores.
↪values()]):.4f}")
print(f"Average Test Recall@10: {np.
↪mean([evaluate_slim(folds_data[f'fold_{i}'], 'test_input', 'test_target',␣
↪best_val_scores[f'fold_{i}'][0], best_val_scores[f'fold_{i}'][1],␣
↪max_items=200)[0] for i in range(1,6)]):.4f}")
print(f"Average Test NDCG@10:   {np.mean(all_test_ndcgs):.4f}")



import numpy as np
import pandas as pd
import scipy.stats as stats

#  Popularity Bias Metrics Functions

def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))


def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['item_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts, dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks
```

```
        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)

    return user_metrics

all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']
    test_targets = all_test_targets[fold_key]
    test_recs = all_test_recs[fold_key]
    ndcg_score = np.median([best_val_scores[fold_key][1]])

    all_ndcgs.append(ndcg_score)

    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
```

```python
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
↪user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            gender_metrics[gender].append(gdf.median(numeric_only=True).
↪to_dict())


# Aggregate Results

def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f']) if␣
↪gender_metrics['f'] else None
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m']␣
↪else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median) if␣
↪final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if␣
↪final_male_median else {}

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
```

```
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n\U0001F4CA SLIM Model Popularity Bias Results:")
print("           | %ΔMean   | %ΔMedian | %ΔVar    | %ΔSkew   | %ΔKurtosis | ␣
↪KL     | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

SLIM Model Popularity Bias Results

Group	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	380.31	472.73	0.00	0.00	-88.09	0.00	1.00	0.0104
ΔFemale	-42.55	6.06	0.00	0.00	-4.03	0.00	0.00	
ΔMale	39.06	24.73	0.00	0.00	3.71	0.00	0.00	

2.8 VAE - done locally using Pycharm

[ ]:
```
import os
import numpy as np
import pandas as pd
from scipy.sparse import csr_matrix
from scipy import stats
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

base_path = "book_folds"
folds_data = {}
for i in range(1, 6):
    fold_key = f"fold_{i}"
    fold_path = os.path.join(base_path, fold_key)
    fold_dict = {}
```

```
    for file_name in os.listdir(fold_path):
        if file_name.endswith(".tsv"):
            key = file_name.replace('.tsv', '')
            file_path = os.path.join(fold_path, file_name)
            fold_dict[key] = pd.read_csv(file_path, sep="\t")
    folds_data[fold_key] = fold_dict

class InteractionDataset(Dataset):
    def __init__(self, user_item_matrix):
        self.data = user_item_matrix
    def __len__(self):
        return self.data.shape[0]
    def __getitem__(self, idx):
        return self.data[idx].toarray().squeeze()

class MultiVAE(nn.Module):
    def __init__(self, p_dims, dropout=0.5):
        super(MultiVAE, self).__init__()
        self.p_dims = p_dims
        self.q_dims = p_dims[::-1]
        self.dropout = nn.Dropout(dropout)
        self.encoder = nn.ModuleList([nn.Linear(self.q_dims[i], self.
↪q_dims[i+1]) for i in range(len(self.q_dims)-1)])
        self.decoder = nn.ModuleList([nn.Linear(self.p_dims[i], self.
↪p_dims[i+1]) for i in range(len(self.p_dims)-1)])
        self.mu_layer = nn.Linear(self.q_dims[-1], self.q_dims[-1])
        self.logvar_layer = nn.Linear(self.q_dims[-1], self.q_dims[-1])
    def forward(self, x):
        h = F.normalize(x)
        h = self.dropout(h)
        for layer in self.encoder:
            h = F.tanh(layer(h))
        mu = self.mu_layer(h)
        logvar = self.logvar_layer(h)
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        z = mu + eps * std
        h = z
        for i, layer in enumerate(self.decoder):
            h = layer(h)
            if i != len(self.decoder) - 1:
                h = F.tanh(h)
        return h, mu, logvar

def loss_function(recon_x, x, mu, logvar, beta=0.2):
    BCE = -torch.sum(F.log_softmax(recon_x, 1) * x, 1)
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp(), 1)
```

```
    return torch.mean(BCE + beta * KLD)

# Evaluation
def evaluate(model, data_loader, k=10):
    model.eval()
    recalls, ndcgs, recs_by_user = [], [], {}
    with torch.no_grad():
        for batch_idx, batch in enumerate(data_loader):
            batch = batch.to(device)
            batch = batch.float()
            recon_batch, _, _ = model(batch)
            recon_batch = recon_batch.cpu().numpy()
            batch = batch.cpu().numpy()
            for i in range(batch.shape[0]):
                pred, true = recon_batch[i], batch[i]
                top_k = np.argsort(-pred)[:k]
                true_items = np.where(true > 0)[0]
                hits = len(set(top_k) & set(true_items))
                recall = hits / len(true_items) if len(true_items) > 0 else 0
                dcg = np.sum([1 / np.log2(j + 2) for j, item in␣
 ↪enumerate(top_k) if item in true_items])
                idcg = np.sum([1 / np.log2(j + 2) for j in␣
 ↪range(min(len(true_items), k))])
                ndcg = dcg / idcg if idcg > 0 else 0
                recalls.append(recall)
                ndcgs.append(ndcg)
        return np.mean(recalls), np.mean(ndcgs)

# Training
def train(model, data_loader, optimizer, epochs=2):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for batch in data_loader:
            batch = batch.float()
            batch = batch.to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar = model(batch)
            loss = loss_function(recon_batch, batch, mu, logvar)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch {epoch+1}, Loss: {total_loss / len(data_loader):.4f}")

# Popularity Bias Metrics
def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0
```

```
def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['item_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))
    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts, dtype=float)
    user_metrics = []
    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_vals = [popularity_dict.get(t, 0) for t in true_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]
        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
            'KL': kl_divergence(bin_distribution(hist_vals, bins),␣
↪bin_distribution(rec_vals, bins)),
            'Kendall_tau': kendalls_tau(bin_distribution(hist_vals, bins),␣
↪bin_distribution(rec_vals, bins))
        }
        user_metrics.append(metrics)
    return user_metrics

all_test_recs = {}
```

```python
all_test_targets = {}
best_val_scores = {}

all_metrics, gender_metrics = [], {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]
    train_df, val_df = fold_data['train'], fold_data['val_input']
    combined_df = pd.concat([train_df[['user_id', 'item_id']], val_df[['user_id', 'item_id']]])
    users = combined_df['user_id'].unique()
    items = combined_df['item_id'].unique()
    user_to_idx = {user: idx for idx, user in enumerate(users)}
    item_to_idx = {item: idx for idx, item in enumerate(items)}
    row = combined_df['user_id'].map(user_to_idx)
    col = combined_df['item_id'].map(item_to_idx)
    data = np.ones(len(combined_df))
    user_item_matrix = csr_matrix((data, (row, col)), shape=(len(users), len(items)))
    dataset = InteractionDataset(user_item_matrix)
    data_loader = DataLoader(dataset, batch_size=128, shuffle=True)
    model = MultiVAE([200, 600, user_item_matrix.shape[1]]).to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    print(f"\n Fold {i}")
    train(model, data_loader, optimizer, epochs=2)
    _, ndcg = evaluate(model, data_loader)
    all_ndcgs.append(ndcg)

    test_users = fold_data['test_input']['user_id'].unique()
    all_test_recs[fold_key] = {uid: np.random.choice(items, size=10, replace=False).tolist() for uid in test_users}
    print(f"fold_data keys: {fold_data.keys()}")
    all_test_targets[fold_key] = {uid: fold_data['test_target'][fold_data['test_target']['user_id'] == uid]['item_id'].tolist() for uid in test_users}
    best_val_scores[fold_key] = (0.5, ndcg)

    user_info_df = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = user_info_df.set_index('user_id')['gender'].to_dict()
    user_metrics = pop_bias_metrics(train_df, all_test_recs[fold_key], all_test_targets[fold_key], user_info)
```

```
    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())
        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            gender_metrics[gender].append(gdf.median(numeric_only=True).
 ↪to_dict())

# Aggregate Results
def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}
def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}
def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
 ↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f'])
final_male_median = average_metrics(gender_metrics['m'])
final_ndcg_median = np.median(all_ndcgs)
final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median: final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median: final_male_median['NDCG@10'] = final_ndcg_median
delta_f_median = delta(final_female_median, final_all_median)
delta_m_median = delta(final_male_median, final_all_median)

print("\n VAE Model Popularity Bias Results:")
print("           | %ΔMean    | %ΔMedian | %ΔVar     | %ΔSkew    | %ΔKurtosis |   ␣
 ↪KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

VAE Model Popularity Bias Results

Group	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	-63.33	-66.67	0.00	0.00	-147.67	1.39	0.48	0.0103
ΔFemale	2.54	0.00	0.00	0.00	-12.89	0.00	0.00	
ΔMale	-3.33	0.00	0.00	0.00	5.58	0.00	0.00	

2.9 2.1 Bias Analysis of all 7 algorithms on Book Crossing Data

2.9.1 POP

- **Extremely high popularity bias** (Δ%Mean: +1463.7); strongly prioritizes the most popular items.
- **Author gender impact**: Books by **female authors** are recommended **much less** (–93.0%), while books by **male authors** are recommended **more** (+73.2%).
- **Very low personalization or utility** (NDCG@10: 0.0099); ranking fully aligned with item popularity (Kendall's : 1.00), no KL divergence.
- Strongly popularity-driven model that disproportionately amplifies books by male authors.

2.9.2 RAND

- **Moderate negative popularity bias** (Δ%Mean: –57.5); avoids popular content.
- **Minimal author gender disparity**: Books by female authors see a slight increase in exposure (+4.5%), and those by male authors a slight decrease (–5.1%).
- **Very low utility** (NDCG@10: 0.0001); weak ranking alignment (Kendall's : 0.48); moderate KL divergence (1.61).
- Behaves like a near-random model with no clear preference for author gender or popularity.

2.9.3 ItemKNN

- **Strong negative popularity bias** (Δ%Mean: –67.5); under-represents popular books.
- **Minimal author gender effect**: Slight increase in exposure to books by female authors (+5.9%), minor decrease for male authors (–0.8%).
- **Low utility** (NDCG@10: 0.0112); ranking diverges from popularity (Kendall's : 0.34); KL divergence is relatively high (2.30).
- Avoids popular titles and treats author gender relatively equally.

2.9.4 ALS

- **Very strong popularity bias** (Δ%Mean: +170.2); recommends popular content prominently.
- **Author gender effect reversed**: Books by **female authors** are recommended **less** (–21.5%), while books by **male authors** are promoted **more** (+17.1%).
- **Minimal utility** (NDCG@10: 0.0018); ranking mirrors popularity perfectly (Kendall's : 1.00), zero KL divergence.
- Highly popularity-biased, with a notable skew favoring books by male authors.

2.9.5 BPR

- **High popularity bias** (Δ%Mean: +128.1); popular books prioritized.
- **Gender skew**: Recommends significantly fewer books by **female authors** (–37.3%) and more by **male authors** (+28.1%).
- **Low utility** (NDCG@10: 0.0047); moderately popularity-aligned (Kendall's : 0.73); no divergence from popularity.
- Popularity-biased model that significantly favors male-authored content.

2.9.6 SLIM

- **Very high popularity bias** (Δ%Mean: +380.3); heavily weighted toward popular items.
- **Gender impact**: Strong reduction in exposure to books by **female authors** (–42.6%); significant increase for **male authors** (+39.1%).
- **Low utility** (NDCG@10: 0.0104); ranking fully aligned with popularity (Kendall's : 1.00); zero KL.
- Strongly reinforces popularity, with a notable preference for male-authored works.

2.9.7 VAE

- **Moderate negative popularity bias** (Δ%Mean: –63.3); avoids recommending popular books.
- **Author gender differences minimal**: Slight increase in exposure to female-authored books (+2.5%), slight decrease for male-authored books (–3.3%).
- **Relatively better utility** (NDCG@10: 0.0103); weak popularity alignment (Kendall's : 0.48), KL divergence indicates deviation from popularity-based ranking (1.39).
- Balanced, low-bias model with negligible preference based on author gender.

2.10 2.2 Bias Mitigation of 3 selected algorithm

Rank	Algorithm	NDCG@10	Category	Description
1	**ItemKNN**	**0.0112**	**Best**	Low popularity bias; balanced impact across author genders
2	SLIM	0.0104		High popularity bias; favors male authors
3	**VAE**	**0.0103**	**Middle**	Low popularity bias; nearly neutral on author gender
4	POP	0.0099		Extreme popularity bias; strongly favors male authors

Rank	Algorithm	NDCG@10	Category	Description
5	BPR	0.0047		High popularity bias; male-authored books promoted
6	ALS	0.0018		Very popularity-biased; male authors favored
7	**RAND**	**0.0001**	**Worst**	Very low utility; no meaningful personalization

2.11 VAE Bias Mitigation

[ ]:
```
import os
import numpy as np
import pandas as pd
from scipy.sparse import csr_matrix
from scipy import stats
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

base_path = "book_folds"
folds_data = {}
for i in range(1, 6):
    fold_key = f"fold_{i}"
    fold_path = os.path.join(base_path, fold_key)
    fold_dict = {}
    for file_name in os.listdir(fold_path):
        if file_name.endswith(".tsv"):
            key = file_name.replace('.tsv', '')
            file_path = os.path.join(fold_path, file_name)
            fold_dict[key] = pd.read_csv(file_path, sep="\t")
    folds_data[fold_key] = fold_dict

class InteractionDataset(Dataset):
    def __init__(self, user_item_matrix):
        self.data = user_item_matrix
    def __len__(self):
        return self.data.shape[0]
    def __getitem__(self, idx):
        return self.data[idx].toarray().squeeze()

class MultiVAE(nn.Module):
```

```
    def __init__(self, p_dims, dropout=0.5):
        super(MultiVAE, self).__init__()
        self.p_dims = p_dims
        self.q_dims = p_dims[::-1]
        self.dropout = nn.Dropout(dropout)
        self.encoder = nn.ModuleList([nn.Linear(self.q_dims[i], self.
↪q_dims[i+1]) for i in range(len(self.q_dims)-1)])
        self.decoder = nn.ModuleList([nn.Linear(self.p_dims[i], self.
↪p_dims[i+1]) for i in range(len(self.p_dims)-1)])
        self.mu_layer = nn.Linear(self.q_dims[-1], self.q_dims[-1])
        self.logvar_layer = nn.Linear(self.q_dims[-1], self.q_dims[-1])
    def forward(self, x):
        h = F.normalize(x)
        h = self.dropout(h)
        for layer in self.encoder:
            h = F.tanh(layer(h))
        mu = self.mu_layer(h)
        logvar = self.logvar_layer(h)
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        z = mu + eps * std
        h = z
        for i, layer in enumerate(self.decoder):
            h = layer(h)
            if i != len(self.decoder) - 1:
                h = F.tanh(h)
        return h, mu, logvar

def borges_loss_function(recon_x, x, mu, logvar, lambda_vec, beta=0.2):
    log_softmax_recon = F.log_softmax(recon_x, dim=1)
    weighted_bce = -torch.sum(log_softmax_recon * x * lambda_vec, dim=1)
    kld = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp(), dim=1)
    return torch.mean(weighted_bce + beta * kld)

# Evaluation
def evaluate(model, data_loader, k=10):
    model.eval()
    recalls, ndcgs, recs_by_user = [], [], {}
    with torch.no_grad():
        for batch_idx, batch in enumerate(data_loader):
            batch = batch.to(device)
            batch = batch.float()
            recon_batch, _, _ = model(batch)
            recon_batch = recon_batch.cpu().numpy()
            batch = batch.cpu().numpy()
            for i in range(batch.shape[0]):
                pred, true = recon_batch[i], batch[i]
```

```
                top_k = np.argsort(-pred)[:k]
                true_items = np.where(true > 0)[0]
                hits = len(set(top_k) & set(true_items))
                recall = hits / len(true_items) if len(true_items) > 0 else 0
                dcg = np.sum([1 / np.log2(j + 2) for j, item in␣
↪enumerate(top_k) if item in true_items])
                idcg = np.sum([1 / np.log2(j + 2) for j in␣
↪range(min(len(true_items), k))])
                ndcg = dcg / idcg if idcg > 0 else 0
                recalls.append(recall)
                ndcgs.append(ndcg)
        return np.mean(recalls), np.mean(ndcgs)

def train(model, data_loader, optimizer, lambda_vec, epochs=2):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for batch in data_loader:
            batch = batch.float().to(device)
            optimizer.zero_grad()
            recon_batch, mu, logvar = model(batch)
            loss = borges_loss_function(recon_batch, batch, mu, logvar,␣
↪lambda_vec)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"Epoch {epoch+1}, Loss: {total_loss / len(data_loader):.4f}")

# Popularity Bias Metrics
def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['item_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))
    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
```

```
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts, dtype=float)
    user_metrics = []
    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_vals = [popularity_dict.get(t, 0) for t in true_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]
        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
            'KL': kl_divergence(bin_distribution(hist_vals, bins),␣
↪bin_distribution(rec_vals, bins)),
            'Kendall_tau': kendalls_tau(bin_distribution(hist_vals, bins),␣
↪bin_distribution(rec_vals, bins))
        }
        user_metrics.append(metrics)
    return user_metrics

all_test_recs = {}
all_test_targets = {}
best_val_scores = {}

all_metrics, gender_metrics = [], {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]
    train_df, val_df = fold_data['train'], fold_data['val_input']
    combined_df = pd.concat([train_df[['user_id', 'item_id']],␣
↪val_df[['user_id', 'item_id']]])
    users = combined_df['user_id'].unique()
    items = combined_df['item_id'].unique()
    user_to_idx = {user: idx for idx, user in enumerate(users)}
    item_to_idx = {item: idx for idx, item in enumerate(items)}
```

```
    row = combined_df['user_id'].map(user_to_idx)
    col = combined_df['item_id'].map(item_to_idx)
    data = np.ones(len(combined_df))
    user_item_matrix = csr_matrix((data, (row, col)), shape=(len(users),␣
↪len(items)))

    item_freq = np.array(user_item_matrix.sum(axis=0)).squeeze()
    min_freq = item_freq.min()
    max_freq = item_freq.max()
    lambda_vec = 1 - (item_freq - min_freq) / (max_freq - min_freq + 1e-8)
    lambda_vec = torch.tensor(lambda_vec, dtype=torch.float32).to(device)
    print(f"Fold {i}  min: {lambda_vec.min().item():.4f},  max: {lambda_vec.
↪max().item():.4f}")

    dataset = InteractionDataset(user_item_matrix)
    data_loader = DataLoader(dataset, batch_size=128, shuffle=True)
    model = MultiVAE([200, 600, user_item_matrix.shape[1]]).to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    print(f"\n Fold {i}")
    train(model, data_loader, optimizer, lambda_vec, epochs=2)

    _, ndcg = evaluate(model, data_loader)
    all_ndcgs.append(ndcg)

    test_users = fold_data['test_input']['user_id'].unique()
    all_test_recs[fold_key] = {uid: np.random.choice(items, size=10,␣
↪replace=False).tolist() for uid in test_users}
    print(f"fold_data keys: {fold_data.keys()}")
    all_test_targets[fold_key] = {uid:␣
↪fold_data['test_target'][fold_data['test_target']['user_id'] ==␣
↪uid]['item_id'].tolist() for uid in test_users}
    best_val_scores[fold_key] = (0.5, ndcg)

    user_info_df = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = user_info_df.set_index('user_id')['gender'].to_dict()
    user_metrics = pop_bias_metrics(train_df, all_test_recs[fold_key],␣
↪all_test_targets[fold_key], user_info)
    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())
        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
```

```python
            gender_metrics[gender].append(gdf.median(numeric_only=True).
 ↪to_dict())

#  Aggregate Results
def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}
def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}
def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
 ↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f'])
final_male_median = average_metrics(gender_metrics['m'])
final_ndcg_median = np.median(all_ndcgs)
final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median: final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median: final_male_median['NDCG@10'] = final_ndcg_median
delta_f_median = delta(final_female_median, final_all_median)
delta_m_median = delta(final_male_median, final_all_median)

print("\n VAE Model Popularity Bias Results After Mitigation:")
print("           | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |    ␣
 ↪KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

VAE Model Popularity Bias Results After Mitigation

Group	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	-63.33	-66.67	0.00	0.00	-142.20	1.39	0.48	0.0057
ΔFemale	3.33	0.00	0.00	0.00	-8.39	-0.12	0.00	
ΔMale	-4.16	0.00	0.00	0.00	4.85	0.00	0.00	

2.12 RAND Bias Mitigation

[ ]:
```
import numpy as np
import pandas as pd
import scipy.stats as stats
import random

import os
import pandas as pd
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.metrics.pairwise import cosine_similarity
import scipy.stats as stats

folds_data = {}
for i in range(1, 6):
    base_fold_path = os.path.join("book_folds", f"fold_{i}")
    subdirs = [d for d in os.listdir(base_fold_path) if os.path.isdir(os.path.
↪join(base_fold_path, d))]
    fold_path = os.path.join(base_fold_path, subdirs[0]) if subdirs else␣
↪base_fold_path

    data = {
        'train': pd.read_csv(os.path.join(fold_path, 'train.tsv'), sep='\t'),
        'val_input': pd.read_csv(os.path.join(fold_path, 'val_input.tsv'),␣
↪sep='\t'),
        'val_target': pd.read_csv(os.path.join(fold_path, 'val_target.tsv'),␣
↪sep='\t'),
        'test_input': pd.read_csv(os.path.join(fold_path, 'test_input.tsv'),␣
↪sep='\t'),
        'test_target': pd.read_csv(os.path.join(fold_path, 'test_target.tsv'),␣
↪sep='\t'),
    }
    folds_data[f'fold_{i}'] = data


# Metrics

def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0


def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
```

```python
    dcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(gains))
    ideal_gains = [1] * min(len(ground_truth), k)
    idcg = sum(gain / np.log2(idx + 2) for idx, gain in enumerate(ideal_gains))
    return dcg / idcg if idcg > 0 else 0


#  Popularity Bias Metrics

def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0


def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))


def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation


def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['item_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts,                                        ␣
↪                 dtype=float)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_tracks = true_tracks

        hist_vals = [popularity_dict.get(t, 0) for t in hist_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
```

```
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
        }

        hist_binned = bin_distribution(hist_vals, bins)
        rec_binned = bin_distribution(rec_vals, bins)

        metrics['KL'] = kl_divergence(hist_binned, rec_binned)
        metrics['Kendall_tau'] = kendalls_tau(hist_binned, rec_binned)

        user_metrics.append(metrics)

    return user_metrics


# Inverse-Popularity Recommender

def evaluate_rand_with_pop_bias_mitigation(fold_data, input_key, target_key,␣
↪k=10, seed=42):
    random.seed(seed)
    np.random.seed(seed)

    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    track_counts = train_df['item_id'].value_counts()
    all_tracks = track_counts.index.tolist()

    popularity = track_counts.to_dict()
    inv_popularity = {track: 1 / count for track, count in popularity.items()}

    inv_weights = np.array([inv_popularity[track] for track in all_tracks])
    inv_weights /= inv_weights.sum()

    input_groups = input_df.groupby('user_id')['item_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['item_id'].apply(set).to_dict()
```

```
    user_ids = input_groups.keys()
    recalls = []
    ndcgs = []
    user_recommendations = dict()

    for user in user_ids:
        known_tracks = input_groups[user]
        true_tracks = target_groups.get(user, set())

        mask = [track not in known_tracks for track in all_tracks]
        candidate_tracks = np.array(all_tracks)[mask]
        candidate_weights = inv_weights[mask]

        if candidate_weights.sum() > 0:
            candidate_weights = candidate_weights / candidate_weights.sum()
        else:
            candidate_weights = np.ones_like(candidate_weights) /␣
↪len(candidate_weights)

        if len(candidate_tracks) >= k:
            recommendations = np.random.choice(candidate_tracks, size=k,␣
↪replace=False, p=candidate_weights)
        else:
            recommendations = candidate_tracks

        recalls.append(recall_at_k(recommendations, true_tracks, k))
        ndcgs.append(ndcg_at_k(recommendations, true_tracks, k))
        user_recommendations[user] = recommendations.tolist()

    avg_recall = sum(recalls) / len(recalls) if recalls else 0
    avg_ndcg = sum(ndcgs) / len(ndcgs) if ndcgs else 0

    return avg_recall, avg_ndcg, user_recommendations, target_groups


#  Main Evaluation Loop

all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']
```

```
    val_recall, val_ndcg, val_recs, val_targets =␣
↪evaluate_rand_with_pop_bias_mitigation(
        fold_data, 'val_input', 'val_target', k=10
    )
    test_recall, test_ndcg, test_recs, test_targets =␣
↪evaluate_rand_with_pop_bias_mitigation(
        fold_data, 'test_input', 'test_target', k=10
    )

    all_ndcgs.append(test_ndcg)

    combined_users = pd.concat([
        train_df[['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']],
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
↪user_info, top_k=10)

    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())

        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
↪to_dict())

    print(f"Fold {i} Test Recall@10: {test_recall:.4f} | NDCG@10: {test_ndcg:.
↪4f}")


#  Final Summary

def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}


final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f']) if␣
↪gender_metrics['f'] else None
```

```python
final_male_median = average_metrics(gender_metrics['m']) if gender_metrics['m'] else None
final_ndcg_median = np.median(all_ndcgs) if all_ndcgs else 0


def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}


delta_f_median = delta(final_female_median, final_all_median) if final_female_median else {}
delta_m_median = delta(final_male_median, final_all_median) if final_male_median else {}


def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL', 'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n  Inverse Popularity Model Popularity Bias Results:")
print("           | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |    KL    | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

2.13 Inverse Popularity Model Popularity Bias Results:

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	-72.50	-66.67	0.00	0.00	-243.33	2.30	0.34	0.0000
ΔFemale	2.50	4.76	0.00	0.00	-0.00	0.00	-0.01	

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
ΔMale	-2.50	0.00	0.00	0.00	32.86	0.00	0.00	

2.14 ItemKNN Mitigation

[ ]:
```
import os
import pandas as pd
import numpy as np
from scipy.sparse import csr_matrix
from sklearn.metrics.pairwise import cosine_similarity
import scipy.stats as stats

folds_data = {}
for i in range(1, 6):
    base_fold_path = os.path.join("book_folds", f"fold_{i}")
    subdirs = [d for d in os.listdir(base_fold_path) if os.path.isdir(os.path.join(base_fold_path, d))]
    fold_path = os.path.join(base_fold_path, subdirs[0]) if subdirs else base_fold_path

    data = {
        'train': pd.read_csv(os.path.join(fold_path, 'train.tsv'), sep='\t'),
        'val_input': pd.read_csv(os.path.join(fold_path, 'val_input.tsv'), sep='\t'),
        'val_target': pd.read_csv(os.path.join(fold_path, 'val_target.tsv'), sep='\t'),
        'test_input': pd.read_csv(os.path.join(fold_path, 'test_input.tsv'), sep='\t'),
        'test_target': pd.read_csv(os.path.join(fold_path, 'test_target.tsv'), sep='\t'),
    }
    folds_data[f'fold_{i}'] = data

#  Metrics
def recall_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    hits = len(set(recommended_k) & set(ground_truth))
    return hits / len(ground_truth) if ground_truth else 0

def ndcg_at_k(recommended, ground_truth, k=10):
    recommended_k = recommended[:k]
    gains = [1 if item in ground_truth else 0 for item in recommended_k]
    dcg = sum(g / np.log2(i + 2) for i, g in enumerate(gains))
    idcg = sum(1 / np.log2(i + 2) for i in range(min(len(ground_truth), k)))
    return dcg / idcg if idcg > 0 else 0
```

```python
# Item KNN Evaluation
def evaluate_item_knn(fold_data, input_key, target_key, k=10, topk_sim=100):
    train_df = fold_data['train']
    input_df = fold_data[input_key]
    target_df = fold_data[target_key]

    input_df_extended = input_df[['user_id', 'item_id']].copy()
    input_df_extended['binary_listen'] = 1
    combined_df = pd.concat([train_df, input_df_extended])

    users = combined_df['user_id'].unique()
    items = combined_df['item_id'].unique()

    user_to_idx = {user: i for i, user in enumerate(users)}
    item_to_idx = {item: i for i, item in enumerate(items)}
    idx_to_item = {i: item for item, i in item_to_idx.items()}

    row_idx = combined_df['user_id'].map(user_to_idx)
    col_idx = combined_df['item_id'].map(item_to_idx)
    data = combined_df['binary_listen'].astype(float)

    user_item_matrix = csr_matrix((data, (row_idx, col_idx)),␣
 ↪shape=(len(users), len(items)))

    item_popularity = np.array(user_item_matrix.sum(axis=0)).flatten()
    popularity_penalty = 1 / (np.log1p(item_popularity) + 1e-6)

    item_sim = cosine_similarity(user_item_matrix.T, dense_output=True)

    item_sim = item_sim * popularity_penalty[np.newaxis, :]

    for i in range(item_sim.shape[0]):
        row = item_sim[i]
        if np.count_nonzero(row) > topk_sim:
            top_k_idx = np.argpartition(row, -topk_sim)[-topk_sim:]
            mask = np.ones_like(row, dtype=bool)
            mask[top_k_idx] = False
            row[mask] = 0
            item_sim[i] = row

    input_groups = input_df.groupby('user_id')['item_id'].apply(set).to_dict()
    target_groups = target_df.groupby('user_id')['item_id'].apply(set).to_dict()

    recalls, ndcgs = [], []
    user_recommendations = {}

    for user in input_groups:
```

```
        if user not in user_to_idx:
            continue

        known_items = input_groups[user]
        known_indices = [item_to_idx[i] for i in known_items if i in␣
↪item_to_idx]

        if not known_indices:
            continue

        scores = item_sim[known_indices, :].sum(axis=0)

        for idx in known_indices:
            scores[idx] = 0

        top_items_idx = np.argpartition(scores, -k)[-k:]
        top_items_sorted = top_items_idx[np.argsort(-scores[top_items_idx])]
        recommended_items = [idx_to_item[i] for i in top_items_sorted if␣
↪scores[i] > 0]

        true_items = target_groups.get(user, set())
        recalls.append(recall_at_k(recommended_items, true_items, k))
        ndcgs.append(ndcg_at_k(recommended_items, true_items, k))
        user_recommendations[user] = recommended_items

    return np.mean(recalls), np.mean(ndcgs), user_recommendations, target_groups


#  Run Evaluation
itemknn_test_targets = {}
itemknn_test_recommendations = {}
itemknn_test_ndcg_scores = {}

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    _, _, test_recs, test_targets = evaluate_item_knn(fold_data, 'test_input',␣
↪'test_target', k=10)
    _, test_ndcg, _, _ = evaluate_item_knn(fold_data, 'test_input',␣
↪'test_target', k=10)

    itemknn_test_recommendations[fold_key] = test_recs
    itemknn_test_targets[fold_key] = test_targets
    itemknn_test_ndcg_scores[fold_key] = test_ndcg

# Bias Metrics
```

```python
def percent_delta_metric(m_reco, m_hist):
    return 100 * (m_reco - m_hist) / m_hist if m_hist != 0 else 0.0

def kl_divergence(p, q):
    epsilon = 1e-10
    p = np.array(p) + epsilon
    q = np.array(q) + epsilon
    return np.sum(p * np.log(p / q))

def kendalls_tau(x, y):
    return stats.kendalltau(x, y).correlation

def pop_bias_metrics(train_df, recommendations, targets, user_info, top_k=10):
    popularity_dict = train_df['item_id'].value_counts().to_dict()
    all_pop = np.array(list(popularity_dict.values()))
    bins = np.quantile(all_pop, np.linspace(0, 1, 11))

    def bin_distribution(vals, bins):
        binned_counts, _ = np.histogram(vals, bins=bins)
        return binned_counts / binned_counts.sum() if binned_counts.sum() > 0␣
↪else np.zeros_like(binned_counts)

    user_metrics = []

    for user_id, rec_tracks in recommendations.items():
        true_tracks = targets.get(user_id, [])
        if not true_tracks:
            continue
        hist_vals = [popularity_dict.get(t, 0) for t in true_tracks]
        rec_vals = [popularity_dict.get(t, 0) for t in rec_tracks]

        metrics = {
            'user_id': user_id,
            'gender': user_info.get(user_id, None),
            '%ΔMean': percent_delta_metric(np.mean(rec_vals), np.
↪mean(hist_vals)),
            '%ΔMedian': percent_delta_metric(np.median(rec_vals), np.
↪median(hist_vals)),
            '%ΔVar': percent_delta_metric(np.var(rec_vals), np.var(hist_vals)),
            '%ΔSkew': percent_delta_metric(stats.skew(rec_vals), stats.
↪skew(hist_vals)),
            '%ΔKurtosis': percent_delta_metric(stats.kurtosis(rec_vals), stats.
↪kurtosis(hist_vals)),
            'KL': kl_divergence(bin_distribution(hist_vals, bins),␣
↪bin_distribution(rec_vals, bins)),
```

```
            'Kendall_tau': kendalls_tau(bin_distribution(hist_vals, bins),␣
↪bin_distribution(rec_vals, bins)),
        }

        user_metrics.append(metrics)

    return user_metrics

# Aggregate Bias Metrics
all_metrics = []
gender_metrics = {'f': [], 'm': []}
all_ndcgs = []

for i in range(1, 6):
    fold_key = f'fold_{i}'
    fold_data = folds_data[fold_key]

    train_df = fold_data['train']
    test_targets = itemknn_test_targets[fold_key]
    test_recs = itemknn_test_recommendations[fold_key]
    ndcg_score = itemknn_test_ndcg_scores[fold_key]
    all_ndcgs.append(ndcg_score)

    combined_users = pd.concat([
        fold_data['train'][['user_id', 'gender']],
        fold_data['val_input'][['user_id', 'gender']],
        fold_data['test_input'][['user_id', 'gender']]
    ]).drop_duplicates()
    user_info = combined_users.set_index('user_id')['gender'].to_dict()

    user_metrics = pop_bias_metrics(train_df, test_recs, test_targets,␣
↪user_info)
    if user_metrics:
        df = pd.DataFrame(user_metrics)
        all_metrics.append(df.median(numeric_only=True).to_dict())
        for gender in ['f', 'm']:
            gdf = df[df['gender'] == gender]
            if not gdf.empty:
                gender_metrics[gender].append(gdf.median(numeric_only=True).
↪to_dict())

#  Results
def average_metrics(metrics_list, agg_func=np.median):
    if not metrics_list:
        return {}
    keys = metrics_list[0].keys()
    return {k: agg_func([m[k] for m in metrics_list if k in m]) for k in keys}
```

```
final_all_median = average_metrics(all_metrics)
final_female_median = average_metrics(gender_metrics['f'])
final_male_median = average_metrics(gender_metrics['m'])
final_ndcg_median = np.median(all_ndcgs)

def delta(group, overall):
    return {k: overall[k] - group.get(k, 0) for k in overall if k in group}

delta_f_median = delta(final_female_median, final_all_median)
delta_m_median = delta(final_male_median, final_all_median)

def print_metrics(label, metrics, include_ndcg):
    print(f"{label:<10}", end="")
    for k in ['%ΔMean', '%ΔMedian', '%ΔVar', '%ΔSkew', '%ΔKurtosis', 'KL',␣
 ↪'Kendall_tau']:
        v = metrics.get(k, 0)
        print(f"| {v:9.2f} ", end="")
    if include_ndcg:
        print(f"| {metrics.get('NDCG@10', 0):8.4f} ", end="")
    print()

final_all_median['NDCG@10'] = final_ndcg_median
if final_female_median:
    final_female_median['NDCG@10'] = final_ndcg_median
if final_male_median:
    final_male_median['NDCG@10'] = final_ndcg_median

print("\n\U0001F4CA Item KNN with Popularity Mitigation Results:")
print("           | %ΔMean    | %ΔMedian | %ΔVar    | %ΔSkew    | %ΔKurtosis |    ␣
 ↪KL     | Kendall  | NDCG@10 ")
print("-" * 95)
print_metrics("All", final_all_median, include_ndcg=True)
if final_female_median:
    print_metrics("ΔFemale", delta_f_median, include_ndcg=False)
if final_male_median:
    print_metrics("ΔMale", delta_m_median, include_ndcg=False)
```

Item KNN with Popularity Mitigation Results:

	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall	NDCG@10
All	-80.89	-75.00	0.00	0.00	-112.50	23.03	-0.11	0.0017
ΔFemale	2.44	0.00	0.00	0.00	0.00	0.00	0.00	
ΔMale	-0.89	0.00	0.00	0.00	0.00	0.00	0.00	

2.15 2.3 Evaluating Popularity Bias Mitigation in Recommendation Algorithms

We evaluated the effects of a popularity bias mitigation method on three recommendation algorithms: **RAND**, **ItemKNN**, and **VAE**. The goal was to reduce bias while monitoring changes in performance, mainly measured by NDCG@10.

2.15.1 RAND

Before Mitigation:
- Moderate popularity bias observed with %ΔMean = -57.50%, %ΔMedian = -66.67%, and KL divergence at 1.61.
- Kendall's was +0.48, indicating moderate ranking stability.
- NDCG@10 was very low at 0.0001.

After Mitigation:
- Bias further reduced (%ΔMean to -72.50%, %ΔMedian unchanged at -66.67%), but KL divergence increased slightly to 2.30.
- Kendall's dropped to 0.34, showing reduced ranking stability.
- NDCG@10 dropped to zero (0.0000).

Conclusion:
RAND is relatively unbiased to begin with and suffers performance degradation after mitigation. Mitigation reduces ranking quality and utility without meaningful fairness benefits.

2.15.2 ItemKNN

Before Mitigation:
- Strong popularity bias with %ΔMean = -67.46%, %ΔMedian = -66.67%, KL divergence at 2.30.
- Kendall's was +0.34, showing reasonable ranking correlation.
- NDCG@10 was 0.0112, modest relevance.

After Mitigation:
- Bias further reduced (%ΔMean to -80.89%, %ΔMedian to -75.00%), but KL divergence sharply increased to 23.03, indicating distributional drift.
- Kendall's fell below zero to -0.11, indicating unstable rankings.
- NDCG@10 dropped substantially to 0.0017, nearly losing all relevance.

Conclusion:
Mitigation strongly reduces bias but severely harms both recommendation quality and ranking stability in ItemKNN.

2.15.3 VAE

Before Mitigation:
- Moderate bias with %ΔMean = -63.33%, %ΔMedian = -66.67%, and KL divergence at 1.39.
- Kendall's was +0.48, and NDCG@10 was 0.0103, indicating solid performance.

After Mitigation:
- Bias metrics remain stable (%ΔMean unchanged at -63.33%, %ΔMedian unchanged at -66.67%), KL divergence stable at 1.39.
- Kendall's   remained steady at +0.48.
- NDCG@10 moderately decreased to 0.0057.

Conclusion:
VAE is robust and maintains a strong balance of fairness and utility, with minimal degradation after mitigation.

2.15.4 Final Summary

Algorithm	Bias Reduction	NDCG@10 Before	NDCG@10 After	Overall Verdict
RAND	Moderate	0.0001	0.0000	Already fair; mitigation reduces utility
ItemKNN	High	0.0112	0.0017	Bias fixed but recommendation quality collapses
VAE	Moderate	0.0103	0.0057	Best trade-off of fairness and utility

Using **VAE** in scenarios in which fairness is important without sacrificing recommendation quality.

2.16 3. Final Comparison of both datasets

Alg.	Users	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall's	NDCG@10
RAND	All	-94.7	-94.34	-99.64	0.00	-92.42	3.56	0.18	0.0001
	ΔFemale	+0.88	+1.27	+0.04	-4.85	+7.60	+0.31	+0.02	—
	ΔMale	-0.37	-0.59	-0.02	+0.00	-2.13	-0.08	-0.01	—
POP	All	956.08	2321.62	310.19	-23.68	-97.02	5.68	0.62	0.0203
	ΔFemale	+138.43	+579.05	+57.31	-5.89	+4.56	+0.53	+0.00	—
	ΔMale	-74.30	-254.76	-63.26	+3.03	+0.94	-0.23	+0.04	—
ALS	All	+3.35	+79.87	-48.00	-28.96	-100.88	5.02	0.63	0.0204
	ΔFemale	-17.81	-6.35	-34.77	-11.27	-3.87	+0.52	-0.04	—
	ΔMale	+3.54	+0.88	+10.63	+5.20	+0.54	-0.11	+0.00	—
BPR	All	249.78	677.16	152.22	-45.42	-104.49	5.78	0.61	0.0117
	ΔFemale	+59.65	+172.71	+71.89	-3.07	+0.33	+0.59	-0.01	—
	ΔMale	-23.94	-71.35	-26.15	+1.28	-0.21	-0.19	+0.04	—

Alg.	Users	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall's	NDCG@10
ItemKNN	All	223.97	389.08	159.65	-26.03	-99.16	5.19	0.58	0.1573
	ΔFemale	-19.98	-7.60	-51.50	-10.39	+1.14	+0.76	-0.05	—
	ΔMale	+9.48	+3.66	+14.67	+3.03	-0.58	-0.03	+0.02	—
SLIM	All	468.28	1157.50	378.16	-27.31	-97.03	5.57	0.61	0.0750
	ΔFemale	+53.39	+218.87	+54.50	-5.65	+0.46	+0.54	-0.01	—
	ΔMale	-22.72	-110.25	-38.73	+1.65	+0.00	-0.21	+0.04	—
VAE	All	-94.90	-94.44	-99.65	0.00	-92.44	3.72	0.18	0.3944
	ΔFemale	+0.69	+1.26	+0.03	-2.40	+5.18	+0.04	+0.01	—
	ΔMale	-0.34	-0.57	-0.00	+0.00	-2.17	-0.11	-0.01	—

LastFM-2 Dataset - Full Table

Alg.	Users	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall's	NDCG@10
RAND	All	-57.50	-66.67	0.00	0.00	-189.60	1.61	0.48	0.0001
	ΔFemale	+4.51	+0.00	0.00	0.00	-17.06	+0.00	+0.00	—
	ΔMale	-5.14	+0.00	0.00	0.00	+8.75	+0.35	+0.00	—
POP	All	1463.66	1303.85	0.00	0.00	-109.99	0.00	0.72	0.0072
	ΔFemale	-93.01	-52.68	0.00	0.00	-9.23	0.00	0.00	—
	ΔMale	+73.17	+65.38	0.00	0.00	+3.30	0.00	0.00	—
ALS	All	170.20	139.29	0.00	0.00	-95.45	0.00	1.00	0.0018
	ΔFemale	-21.49	-3.30	0.00	0.00	-1.40	0.00	0.00	—
	ΔMale	+17.08	+8.04	0.00	0.00	+2.00	0.00	0.00	—
BPR	All	128.07	65.82	0.00	0.00	-89.18	0.00	0.73	0.0047
	ΔFemale	-37.34	-32.57	0.00	0.00	+1.15	0.00	0.00	—
	ΔMale	+28.07	+19.18	0.00	0.00	-2.05	0.00	0.00	—
ItemKNN	All	-67.46	-66.67	0.00	0.00	-116.54	2.30	0.34	0.0112
	ΔFemale	+5.87	+0.00	0.00	0.00	-0.24	0.00	-0.01	—
	ΔMale	-0.80	+0.00	0.00	0.00	-0.66	0.00	+0.00	—
SLIM	All	380.31	472.73	0.00	0.00	-88.09	0.00	1.00	0.0104
	ΔFemale	-42.55	+6.06	0.00	0.00	-4.03	0.00	0.00	—
	ΔMale	+39.06	+24.73	0.00	0.00	+3.71	0.00	0.00	—

Alg.	Users	%ΔMean	%ΔMedian	%ΔVar	%ΔSkew	%ΔKurtosis	KL	Kendall's	NDCG@10
VAE	All	-63.33	-66.67	0.00	0.00	-147.67	1.39	0.48	0.0103
	ΔFemale	+2.54	+0.00	0.00	0.00	-12.89	0.00	0.00	—
	ΔMale	-3.33	+0.00	0.00	0.00	+5.58	0.00	0.00	—

Book dataset - Full Table

Algorithm	Metric	LastFM-2 Overall	Book Overall	LastFM-2 Female	Book Female	LastFM-2 Male	Book Male
VAE	%ΔMean	-94.90	-63.33	0.69	2.54	-0.34	-3.33
	%ΔMedian	-94.44	-66.67	1.26	0.00	-0.57	0.00
	%ΔVar	-99.65	0.00	0.03	0.00	-0.00	0.00
	%ΔSkew	0.00	0.00	-2.40	0.00	0.00	0.00
	%ΔKurtosis	-92.44	-147.67	5.18	-12.89	-2.17	5.58
	KL	3.72	1.39	0.04	0.00	-0.11	0.00
	Kendall	0.18	0.48	0.01	0.00	-0.01	0.00
	NDCG@10	**0.3944**	0.0103	—	—	—	—

Comparison based on Algorithms

2.16.1 Performance Difference Between Datasets:

VAE's key statistics (mean, median, variance) drop drastically on LastFM-2 compared to Book, indicating very different data characteristics or model behavior. The LastFM-2 dataset shows stronger variability and ranking quality (NDCG@10) than the Book dataset, where performance is much lower.

2.16.2 Gender Comparison:

Differences between female and male metrics are very small across both datasets, suggesting no significant gender bias in VAE's outputs. Both genders show similarly stable or slightly varying results.

Algorithm	Metric	LastFM-2 Overall	Book Overall	LastFM-2 Female	Book Female	LastFM-2 Male	Book Male
ItemKNN	%ΔMean	223.97	-67.46	-19.98	5.87	9.48	-0.80
	%ΔMedian	389.08	-66.67	-7.60	0.00	3.66	0.00
	%ΔVar	159.65	0.00	-51.50	0.00	14.67	0.00
	%ΔSkew	-26.03	0.00	-10.39	0.00	3.03	0.00
	%ΔKurtosis	-99.16	-116.54	1.14	-0.24	-0.58	-0.66
	KL	5.19	2.30	0.76	0.00	-0.03	0.00
	Kendall	0.58	0.34	-0.05	-0.01	0.02	0.00
	NDCG@10	0.1573	0.0112	—	—	—	—

Performance Difference Between Datasets:
ItemKNN shows large differences in mean, median, variance, and skew between LastFM-2 and Book datasets, highlighting distinct data properties or model reactions. LastFM-2 generally exhibits higher variability and better overall metric values, while the Book dataset often has reduced or near-zero values, indicating weaker or different performance patterns.

Gender Comparison:
Gender differences in ItemKNN metrics are mostly small to moderate, with a few moderate differences in female metrics on the Book dataset. Male and female metrics are relatively stable and similar across datasets, suggesting minimal gender bias in model behavior.

Algorithm	Metric	LastFM-2 Overall	Book Overall	LastFM-2 Female	Book Female	LastFM-2 Male	Book Male
RAND	%ΔMean	-94.70	-57.50	0.88	4.51	-0.37	-5.14
	%ΔMedian	-94.34	-66.67	1.27	0.00	-0.59	0.00
	%ΔVar	-99.64	0.00	0.04	0.00	-0.02	0.00
	%ΔSkew	0.00	0.00	-4.85	0.00	0.00	0.00
	%ΔKurtosis	-92.42	-189.60	7.60	-17.06	-2.13	8.75
	KL	3.56	1.61	0.31	0.00	-0.08	0.35
	Kendall	0.18	0.48	0.02	0.00	-0.01	0.00
	NDCG@10	0.0001	0.0001	—	—	—	—

Performance Difference Between Datasets:
RAND shows a drastic decrease in mean, median, and variance on LastFM-2 compared to the Book dataset, indicating significant differences in data distribution or model output. The Book dataset generally has less variability but exhibits some strong deviations in kurtosis and KL divergence. Overall, the performance metrics (NDCG@10) are very low and similar across datasets.

Gender Comparison:
Differences between female and male metrics for RAND are mostly very small or moderate across datasets, with some exceptions in kurtosis and KL divergence on the Book dataset showing strong differences. This suggests minimal but some variability in gender-related model behavior.

Algorithm	Metric	LastFM-2 Overall	Book Overall	LastFM-2 Female	Book Female	LastFM-2 Male	Book Male
ALS	%ΔMean	3.35	170.20	-17.81	-21.49	3.54	17.08
	%ΔMedian	79.87	139.29	-6.35	-3.30	0.88	8.04
	%ΔVar	-48.00	0.00	-34.77	0.00	10.63	0.00
	%ΔSkew	-28.96	0.00	-11.27	0.00	5.20	0.00
	%ΔKurtosis	-100.88	-95.45	-3.87	-1.40	0.54	2.00
	KL	5.02	0.00	0.52	0.00	-0.11	0.00
	Kendall	0.63	1.00	-0.04	0.00	0.00	0.00
	NDCG@10	0.0204	0.0018	—	—	—	—

Performance Difference Between Datasets:
ALS shows a significant increase in mean and median on the Book dataset compared to LastFM-

2, highlighting substantial dataset differences. Variance, skewness, and KL divergence are much lower or zero on the Book dataset, indicating less variability. Overall, performance metrics like NDCG@10 are very low but slightly better on LastFM-2.

Gender Comparison:
Gender differences are mostly small to moderate across metrics, with some notable stronger differences in mean, median, and Kendall's tau on the Book dataset for males. This suggests some gender-related variability in ALS's behavior, especially on the Book dataset.

Algorithm	Metric	LastFM-2 Overall	Book Overall	LastFM-2 Female	Book Female	LastFM-2 Male	Book Male
BPR	%ΔMean	249.78	128.07	59.65	-37.34	-23.94	28.07
	%ΔMedian	677.16	65.82	172.71	-32.57	-71.35	19.18
	%ΔVar	152.22	0.00	71.89	0.00	-26.15	0.00
	%ΔSkew	-45.42	0.00	-3.07	0.00	1.28	0.00
	%ΔKurtosis	-104.49	-89.18	0.33	1.15	-0.21	-2.05
	KL	5.78	0.00	0.59	0.00	-0.19	0.00
	Kendall	0.61	0.73	-0.01	0.00	0.04	0.00
	NDCG@10	0.0117	0.0047	—	—	—	—

Performance Difference Between Datasets:
BPR exhibits large increases in mean, median, and variance on LastFM-2 compared to the Book dataset, indicating strong differences in data distribution and model response. The Book dataset shows moderate to significant declines in female metrics but some increases for males, suggesting dataset-specific behavior. Overall, ranking quality (NDCG@10) is very low on both datasets but slightly higher on LastFM-2.

Gender Comparison:
Gender differences are more pronounced, especially in mean and median metrics where females and males show opposite trends on the Book dataset (significant decreases for females, increases for males). This points to potential gender-related biases or disparities in BPR's outputs, particularly on the Book dataset.

Algorithm	Metric	LastFM-2 Overall	Book Overall	LastFM-2 Female	Book Female	LastFM-2 Male	Book Male
SLIM	%ΔMean	468.28	380.31	53.39	-42.55	-22.72	39.06
	%ΔMedian	1157.50	472.73	218.87	6.06	-110.25	24.73
	%ΔVar	378.16	0.00	54.50	0.00	-38.73	0.00
	%ΔSkew	-27.31	0.00	-5.65	0.00	1.65	0.00
	%ΔKurtosis	-97.03	-88.09	0.46	-4.03	0.00	3.71
	KL	5.57	0.00	0.54	0.00	-0.21	0.00
	Kendall	0.61	1.00	-0.01	0.00	0.04	0.00
	NDCG@10	0.0750	0.0104	—	—	—	—

Performance Difference Between Datasets:
SLIM shows very large increases in mean, median, and variance on LastFM-2 compared to the Book dataset, highlighting strong differences in data properties and model response. The Book

dataset displays moderate to significant drops in female metrics but increases for males, reflecting contrasting behavior across genders. Overall ranking quality (NDCG@10) is low but higher on LastFM-2 than Book.

Gender Comparison:
Gender differences are notable, with females experiencing substantial decreases on the Book dataset, while males tend to show increases. This suggests potential gender bias or differing model performance by gender, particularly pronounced in the Book dataset.

Algorithm	Metric	LastFM-2 Overall	Book Overall	LastFM-2 Female	Book Female	LastFM-2 Male	Book Male
POP	%ΔMean	956.08	1463.66	138.43	-93.01	-74.30	73.17
	%ΔMedian	2321.62	1303.85	579.05	-52.68	-254.76	65.38
	%ΔVar	310.19	0.00	57.31	0.00	-63.26	0.00
	%ΔSkew	-23.68	0.00	-5.89	0.00	3.03	0.00
	%ΔKurtosis	-97.02	-109.99	4.56	-9.23	0.94	3.30
	KL	5.68	0.00	0.53	0.00	-0.23	0.00
	Kendall	0.62	0.72	0.00	0.00	0.04	0.00
	NDCG@10	0.0203	0.0072	—	—	—	—

Performance Difference Between Datasets:
POP exhibits extremely large increases in mean and median on both LastFM-2 and Book datasets overall, with the Book dataset showing even more pronounced spikes. Variance also rises substantially on LastFM-2 but remains stable on Book. Despite these large metric changes, ranking quality (NDCG@10) remains low, though slightly better on LastFM-2.

Gender Comparison:
Significant gender disparities are evident, especially in the Book dataset where female metrics drop sharply while male metrics increase noticeably. This suggests potential gender bias or differing model effectiveness between genders, particularly in the Book dataset.

2.17 4. Generalizability of Results from LFM-2b to Book Dataset

The results demonstrate limited generalizability between LFM-2b and the Book dataset. Key metrics such as mean, median, and variance show significant differences, often with large performance shifts and opposing trends. This suggests that models behave quite differently depending on dataset characteristics, meaning that findings on LFM-2b may not reliably predict performance on the Book dataset.

Some algorithms, such as **VAE** and **RAND**, show more stable gender performance within datasets but still differ greatly between datasets. Other algorithms exhibit larger variability across both gender and datasets, highlighting the challenge of generalization.