AAI 627 — Big Data Analytics

Midterm Project

Heuristic-Based Music Recommendation System

Spring 2025

Syed Ahmad Shah

Contents

1 Dataset Overview **2**

2 Part 1: Heuristic-Based Music Recommendation **2**
- 2.1 1.a) Feature Engineering & Statistical Aggregation . . . 2
  - 2.1.1 Methodology . . . 2
  - 2.1.2 Feature Vector . . . 2
  - 2.1.3 Lookup Priority . . . 3
  - 2.1.4 Implementation . . . 3
  - 2.1.5 Sample Feature Matrix Output . . . 6
- 2.2 1.b) Decision Logic & Rule Definition . . . 6
  - 2.2.1 Strategy 1 — Weighted Hierarchical Average . . . 7
  - 2.2.2 Strategy 2 — Maximum Genre Score . . . 7
  - 2.2.3 Ranking & Submission Code . . . 7
  - 2.2.4 Part 1 Kaggle Results . . . 8

3 Part 2: Addressing the Cold Start Problem **9**
- 3.1 2.a) Strategy Design — The Global Fallback . . . 9
  - 3.1.1 Problem Statement . . . 9
  - 3.1.2 Global Popularity Approach . . . 9
  - 3.1.3 Bayesian Averaging . . . 9
  - 3.1.4 Updated Lookup Function . . . 10
- 3.2 2.b) Strategy Design — “Dig Deeper” Search Logic . . . 11
  - 3.2.1 Problem Statement . . . 11
  - 3.2.2 The Dig Deeper Rule . . . 11
  - 3.2.3 Hierarchical Fallback Order (Album Score) . . . 12
  - 3.2.4 Implementation . . . 12
  - 3.2.5 Impact Analysis — Fallback Tier Breakdown . . . 14
  - 3.2.6 Observations . . . 14

4 Further Improvement — Strategy 3: Adaptive Weight Normalization **15**
- 4.1 Motivation . . . 15
- 4.2 Implementation . . . 15

5 Further Improvement — Strategy 4: Autoencoder Fallback Experiment **16**
- 5.1 Motivation . . . 16
- 5.2 Concept . . . 17
- 5.3 Brief Implementation . . . 17
- 5.4 Observation . . . 18

6 Results Summary **18**

Dataset Overview

The dataset consists of six files describing a music platform's user interaction history and track metadata. All entities — albums, artists, genres, and tracks — share a single flat `ItemID` namespace in the training data, meaning a rating of `ItemID = 214765` may refer to a genre, album, or artist depending on the entity tables.

Table 1: Dataset Statistics

File / Metric	Value
Albums	52,829
Artists	18,674
Genres	567
Tracks	224,041
Training ratings	12,403,575
Test rows	120,000 (20,000 unique users, 6 tracks each)
Rating range	0 – 100
Rating mean	49.77
Rating std dev	38.03
Tracks missing AlbumID	18,509
Tracks missing ArtistID	26,896
Genres per track	min 0, max 21, mean 2.44

Part 1: Heuristic-Based Music Recommendation

Objective: For each of the 20,000 test users, predict which 3 of their 6 candidate tracks they will like. Each user must receive exactly three labels of `1` (Recommend) and three labels of `0` (Do Not Recommend).

1.a) Feature Engineering & Statistical Aggregation

Methodology

Each track sits within a hierarchy of music entities:

$$\textbf{Track} \longrightarrow \textbf{Album} \longrightarrow \textbf{Artist} \longrightarrow \textbf{Genres (1–21)}$$

Because the test users may never have rated a candidate track directly, we extract *indirect* preference signals by looking up each user's historical ratings for the track's album, artist, and genres. The training data's flat `ItemID` namespace means all of these entities are stored in the same rating table under the same user–item–rating schema.

Feature Vector

For each `(user, track)` test pair, seven numeric features are computed:

Table 2: Feature Vector Components

Feature	Description
`album_score`	User's preference score for the track's album
`artist_score`	User's preference score for the track's artist
`genre_count`	Number of genres associated with the track
`genre_max`	Highest preference score across the track's genres
`genre_min`	Lowest preference score across the track's genres
`genre_mean`	Average preference score across the track's genres
`genre_var`	Variance of genre scores (high = mixed feelings)

Lookup Priority

Each score is resolved using a three-level priority chain:

1. **User's own rating** — most personal, highest trust
2. **Global average rating for that item** — community signal
3. **Overall dataset mean (49.77)** — cold-start fallback

This guarantees every feature always has a meaningful value, even for items that no user has ever rated.

Implementation

Step 1: Build Track → Hierarchy Lookup Dictionary

`track_to_hierarchy` is constructed by iterating over `track_data.csv`. A dictionary is used instead of repeated DataFrame queries to achieve $O(1)$ lookup speed across 120,000 test pairs.

Result: 224,041 tracks indexed.
Sample — Track 1: {`'album': '106710', 'artist': '281667', 'genres': ['214765', '162234', '155788']`}

```
genre_cols = [c for c in track_df.columns if
    c.startswith('Genre')]
track_to_hierarchy = {}

for _, row in tqdm(track_df.iterrows(), total=len(track_df)):
    track_id  = str(int(row['TrackID']))
    album_id  = str(int(row['AlbumID']))  if
        pd.notna(row['AlbumID'])  else None
    artist_id = str(int(row['ArtistID'])) if
        pd.notna(row['ArtistID']) else None
    genres    = [str(int(row[c])) for c in genre_cols if
        pd.notna(row[c])]
    track_to_hierarchy[track_id] = {
```

```
        'album': album_id, 'artist': artist_id, 'genres': genres
    }
```

Listing 1: Building the track hierarchy lookup

Step 2: Build User → Ratings Lookup Dictionary

`user_to_ratings` maps each `UserID` to a nested dictionary of {`ItemID: Rating`}, enabling instant lookup for any user–item pair.

Result: 49,204 users indexed across 12,403,575 ratings.

```
user_to_ratings = defaultdict(dict)

for _, row in tqdm(train_df.iterrows(), total=len(train_df)):
    user_to_ratings[str(int(row['UserID']))][str(int(row['ItemID']))]
       = \
        float(row['Rating'])
```

Listing 2: Building the user ratings lookup

Step 3: Compute Global Average Ratings

For each item, its average rating is computed across all users who have rated it. This serves as Level 2 of the fallback chain.

Result: Global averages for 295,799 items. Overall mean: 49.77.

```
overall_mean = train_df['Rating'].mean()

item_avg_ratings = (
    train_df.groupby('ItemID')['Rating']
    .mean()
    .rename(lambda x: str(int(x)))
    .to_dict()
)
```

Listing 3: Computing global item averages

Step 4: Build Feature Vector Function

```
def build_feature_vector(user_id, track_id, track_to_hierarchy,
                          user_to_ratings, item_avg_ratings,
                          overall_mean, default=None):
    if default is None:
        default = overall_mean

    uid = str(int(user_id))
    tid = str(int(track_id))

    hierarchy    = track_to_hierarchy.get(tid, {})
```

```
    user_ratings = user_to_ratings.get(uid, {})

    album_id  = hierarchy.get('album')
    artist_id = hierarchy.get('artist')
    genre_ids = hierarchy.get('genres', [])

    # Album score (3-level fallback)
    if album_id:
        album_score = user_ratings.get(album_id,
                      item_avg_ratings.get(album_id, default))
    else:
        album_score = default

    # Artist score (3-level fallback)
    if artist_id:
        artist_score = user_ratings.get(artist_id,
                       item_avg_ratings.get(artist_id, default))
    else:
        artist_score = default

    # Genre scores (3-level fallback per genre)
    genre_scores = [
        user_ratings.get(g, item_avg_ratings.get(g, default))
        for g in genre_ids
    ]

    # Genre statistics
    if genre_scores:
        genre_count = len(genre_scores)
        genre_max   = float(np.max(genre_scores))
        genre_min   = float(np.min(genre_scores))
        genre_mean  = float(np.mean(genre_scores))
        genre_var   = float(np.var(genre_scores))
    else:
        genre_count = 0
        genre_max = genre_min = genre_mean = default
        genre_var = 0.0

    return {
        'album_score': album_score, 'artist_score': artist_score,
        'genre_count': genre_count, 'genre_max': genre_max,
        'genre_min':   genre_min,   'genre_mean': genre_mean,
        'genre_var':   genre_var,
    }
```

Listing 4: Feature vector builder

Step 5: Build Feature Matrix for All 120,000 Test Pairs

```python
records = []
for _, row in tqdm(test_df.iterrows(), total=len(test_df)):
    fv = build_feature_vector(
        user_id=row['UserID'], track_id=row['TrackID'],
        track_to_hierarchy=track_to_hierarchy,
        user_to_ratings=user_to_ratings,
        item_avg_ratings=item_avg_ratings,
        overall_mean=overall_mean,
    )
    fv['UserID']  = row['UserID']
    fv['TrackID'] = row['TrackID']
    records.append(fv)

features_df = pd.DataFrame(records, columns=[
    'UserID', 'TrackID', 'album_score', 'artist_score',
    'genre_count', 'genre_max', 'genre_min', 'genre_mean',
       'genre_var'
])
```

Listing 5: Assembling the full feature matrix

Sample Feature Matrix Output

Table 3: First 12 rows of the feature matrix (120,000 rows × 9 columns)

UserID	TrackID	album	artist	cnt	max	min	mean	var
199810	208019	42.76	49.77	0	49.77	49.77	49.77	0.00
199810	74139	73.82	43.93	7	80.00	17.14	46.47	347.92
199810	9903	49.77	49.77	4	56.01	26.76	39.40	115.81
199810	242681	56.08	67.74	3	69.53	50.62	62.04	67.33
199810	18515	58.01	70.00	3	55.55	6.36	32.66	408.94
199810	105760	56.42	90.00	4	80.00	50.52	66.52	184.98
199812	276940	46.71	41.89	1	46.85	46.85	46.85	0.00
199812	142408	100.00	100.00	6	80.00	34.37	61.60	272.57
199812	130023	100.00	100.00	4	80.00	34.37	57.61	501.84
199812	29189	56.92	76.71	6	80.00	39.76	55.81	217.21
199812	223706	63.23	100.00	3	80.00	33.63	49.33	470.34
199812	211361	61.68	58.76	1	17.53	17.53	17.53	0.00

1.b) Decision Logic & Rule Definition

For each user, the 6 candidate tracks are scored and ranked. The top 3 receive label `1` (Recommend); the bottom 3 receive label `0` (Do Not Recommend). Two independent scoring strategies were implemented.

Strategy 1 — Weighted Hierarchical Average

Formula:

$$\text{score}_1 = 0.40 \times \texttt{album_score} + 0.30 \times \texttt{artist_score} + 0.30 \times \texttt{genre_mean}$$

Rationale: The hierarchy forms a *specificity ladder*: album is the most specific signal (the user rated this exact collection), artist is mid-level, and genre is the broadest category. Weighting by specificity rewards tracks that match the user's taste at the closest known level. The 40% album weight reflects the intuition that if a user loved an album, every track from it is a strong candidate. Artist and genre mean each receive equal 30% weight to balance performer loyalty against broad style preference.

Best suited for: users with rich album and artist rating history.

```python
def score_strategy_1(row):
    return (0.40 * row['album_score'] +
            0.30 * row['artist_score'] +
            0.30 * row['genre_mean'])
```

Listing 6: Strategy 1 scoring function

Strategy 2 — Maximum Genre Score

Formula:

$$\text{score}_2 = 0.70 \times \texttt{genre_max} + 0.20 \times \texttt{artist_score} + 0.10 \times \texttt{album_score}$$

Rationale: A user does not need to love every genre of a track to enjoy it — they just need to love at least one. This "gateway genre" approach rewards any track that contains a genre the user is passionate about. The `genre_max` feature captures the highest genre affinity signal and is particularly effective for users who have sparse album/artist history but strong genre preferences. The artist is given a secondary tiebreaker weight (0.20) rather than the album (0.10), because an artist tends to stay within consistent genre territory more reliably than a single album.

Best suited for: users who primarily rate genres, not albums.

```python
def score_strategy_2(row):
    return (0.70 * row['genre_max'] +
            0.20 * row['artist_score'] +
            0.10 * row['album_score'])
```

Listing 7: Strategy 2 scoring function

Ranking & Submission Code

```python
def generate_predictions(features_df, score_col):
    predictions = []
    for user_id, group in features_df.groupby('UserID'):
        ranked = group.sort_values(score_col, ascending=False) \
                        .reset_index(drop=True)
        for i, row in ranked.iterrows():
            predictions.append({
                'TrackID'  :
                    f"{int(user_id)}_{int(row['TrackID'])}",
                'Predictor': 1 if i < 3 else 0
            })
    return pd.DataFrame(predictions)

features_df['score_s1'] = features_df.apply(score_strategy_1,
    axis=1)
features_df['score_s2'] = features_df.apply(score_strategy_2,
    axis=1)

submission_s1 = generate_predictions(features_df, 'score_s1')
submission_s2 = generate_predictions(features_df, 'score_s2')
submission_s1.to_csv('submission_strategy1_weighted_avg.csv',
    index=False)
submission_s2.to_csv('submission_strategy2_max_genre.csv',
    index=False)
```

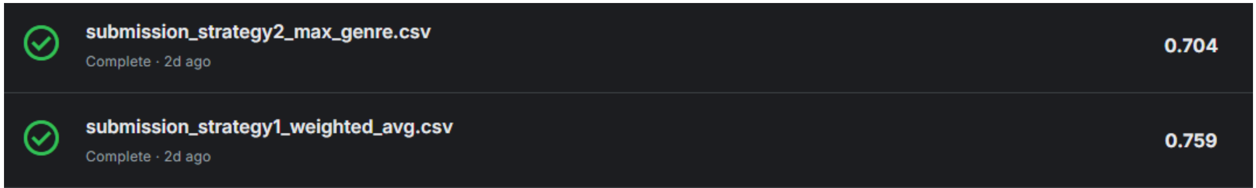
Listing 8: Prediction generation (shared by all strategies)

Validation confirmed 0 users with incorrect label counts for both submissions.

Part 1 Kaggle Results

Table 4: Part 1 Kaggle Submission Scores

Strategy	Score
Strategy 1 — Weighted Hierarchical Average	0.759
Strategy 2 — Maximum Genre Score	0.704



Strategy 1 outperformed Strategy 2 by approximately 5.5 percentage points. This suggests that album-level preference is a stronger individual predictor than genre-max in this dataset:

the album signal captures user taste at the most specific level, whereas `genre_max` can over-predict interest when a track's highest-scoring genre is only moderately liked by the user.

Part 2: Addressing the Cold Start Problem

Objective: Design and implement fallback strategies for users with sparse or missing training history — users for whom personalized signals (direct album, artist, or genre ratings) are unavailable.

2.a) Strategy Design — The Global Fallback

Problem Statement

A cold-start user is one who either:

(a) Has zero ratings in `train_data.csv`, or

(b) Has ratings, but none match any album, artist, or genre of their 6 candidate tracks.

In both cases, the original `build_feature_vector` returns `overall_mean` (≈49.77) for every feature, making all 6 tracks look identical. The ranker cannot distinguish between them.

Global Popularity Approach

Instead of a flat fallback, we use the *global popularity* of each item (album/artist/genre) as observed across all users. Items with many high ratings receive a higher fallback score than obscure or poorly rated items.

Bayesian Averaging

A naive global average introduces a bias: an album rated once at 95 appears "more popular" than one rated 10,000 times at 85. Bayesian averaging corrects this by shrinking items with few ratings toward the overall mean:

$$\hat{\mu}_i = \frac{N_i \cdot \bar{r}_i + C \cdot M}{N_i + C}$$

where:

- N_i = number of ratings for item i
- $\bar{r}_i$ = raw average rating for item i
- C = confidence threshold = median rating count across all items = 10.0
- M = overall dataset mean = 49.77

Items with fewer than C ratings are pulled toward M; items with many ratings are essentially unchanged. This prevents rare, extreme-rated items from dominating the rankings.

```
item_stats = (
    train_df.groupby('ItemID')['Rating']
    .agg(['mean', 'count'])
    .reset_index()
)
item_stats.columns = ['ItemID', 'global_mean', 'rating_count']
item_stats['ItemID'] = item_stats['ItemID'].astype(str)

C = item_stats['rating_count'].median()    # = 10.0
m = overall_mean                           # = 49.77

item_stats['bayesian_avg'] = (
    (item_stats['rating_count'] * item_stats['global_mean'] + C
       * m) /
    (item_stats['rating_count'] + C)
)

global_item_scores = dict(zip(item_stats['ItemID'],
                              item_stats['bayesian_avg']))
```

Listing 9: Computing Bayesian global popularity scores

Result: Global popularity scores computed for 295,799 items.

Table 5: Top 10 Globally Popular Items (by Bayesian Average)

ItemID	Global Mean	Rating Count	Bayesian Avg
251888	88.87	257	87.41
268359	89.30	115	86.14
169258	89.79	96	86.02
143794	90.01	76	85.33
264708	85.59	892	85.19
286359	84.74	2145	84.58
39751	87.66	111	84.53
269021	89.55	66	84.31
258374	83.94	856	83.55
284170	84.65	226	83.17

Updated Lookup Function

```
def get_score(item_id, user_ratings, global_item_scores,
   overall_mean):
    if item_id is None:
        return overall_mean
    if item_id in user_ratings:
        return user_ratings[item_id]          # Level 1: personal
```

```
6     if item_id in global_item_scores:
7         return global_item_scores[item_id]     # Level 2:
             Bayesian global
8     return overall_mean                        # Level 3:
       absolute fallback
```

Listing 10: Three-level item score lookup with Bayesian fallback

The key improvement over Part 1: Level 2 now returns the Bayesian average for that specific item rather than the flat `overall_mean`, providing genuinely informed imputation for cold-start cases.

2.b) Strategy Design — "Dig Deeper" Search Logic

Problem Statement

Even with the Bayesian fallback, a gap remains. Consider:

- A user has rated Track A as 90 and Track B as 85.
- Both tracks belong to Album X.
- The user has *never* directly rated Album X.
- The old pipeline falls back to the global Bayesian average for Album X.

Yet the user's track-level ratings are strong evidence of their album preference. A user who loved two songs from an album almost certainly has a high opinion of that album, even without an explicit album rating.

The Dig Deeper Rule

Before falling back to global popularity for the album score, the system queries whether the user has rated any *sibling tracks* (other tracks from the same album). If so, their ratings are aggregated as a proxy album score.

Aggregation: Mean of sibling track ratings.

- *Mean* asks: "What is my general feeling about the tracks I know from this album?" — a balanced, statistically stable estimate.
- *Max* would overestimate (best song heard from the album).
- *Min* would underestimate (worst song heard).
- Mean is the appropriate central tendency measure.

Hierarchical Fallback Order (Album Score)

Tier	Name	Logic
1	Primary	Direct album rating from the user
2	Dig Deeper	Mean of user's ratings for sibling tracks in the album
3	Global	Bayesian global popularity of the album

Implementation

```
album_to_tracks = defaultdict(list)
for _, row in tqdm(track_df.iterrows(), total=len(track_df)):
    if pd.notna(row['AlbumID']):
        album_id = str(int(row['AlbumID']))
        track_id = str(int(row['TrackID']))
        album_to_tracks[album_id].append(track_id)
```

Listing 11: Building the album-to-tracks lookup

Albums with at least one track: 31,141. Average tracks per album: 6.60. Max tracks in one album: 142.

```
def get_album_score(album_id, user_ratings, album_to_tracks,
                    global_item_scores, overall_mean, tracker):
    if album_id is None:
        tracker['no_album_id'] += 1
        return overall_mean

    # Tier 1: Direct album rating
    if album_id in user_ratings:
        tracker['tier1_direct'] += 1
        return user_ratings[album_id]

    # Tier 2: Dig Deeper -- mean of sibling track ratings
    sibling_tracks  = album_to_tracks.get(album_id, [])
    sibling_ratings = [user_ratings[t] for t in sibling_tracks
                       if t in user_ratings]
    if sibling_ratings:
        tracker['tier2_dig_deeper'] += 1
        return np.mean(sibling_ratings)

    # Tier 3: Global Bayesian fallback
    tracker['tier3_global'] += 1
    return global_item_scores.get(album_id, overall_mean)
```

Listing 12: Hierarchical album score function with Dig Deeper

```
def build_feature_vector_v2(user_id, track_id,
   track_to_hierarchy,
```

```
                                user_to_ratings, album_to_tracks,
                                global_item_scores, overall_mean,
                                   tracker):
    uid = str(int(user_id))
    tid = str(int(track_id))
    hierarchy    = track_to_hierarchy.get(tid, {})
    user_ratings = user_to_ratings.get(uid, {})

    album_id  = hierarchy.get('album')
    artist_id = hierarchy.get('artist')
    genre_ids = hierarchy.get('genres', [])

    # Album: 3-tier (direct -> dig deeper -> global)
    album_score = get_album_score(album_id, user_ratings,
       album_to_tracks,
                                     global_item_scores,
                                        overall_mean, tracker)

    # Artist: 2-tier (direct -> global)
    artist_score = get_score(artist_id, user_ratings,
                                global_item_scores, overall_mean)

    # Genres: 2-tier each (direct -> global)
    genre_scores = [get_score(g, user_ratings,
       global_item_scores,
                                 overall_mean) for g in genre_ids]

    if genre_scores:
        genre_count = len(genre_scores)
        genre_max   = float(np.max(genre_scores))
        genre_min   = float(np.min(genre_scores))
        genre_mean  = float(np.mean(genre_scores))
        genre_var   = float(np.var(genre_scores))
    else:
        genre_count = 0
        genre_max = genre_min = genre_mean = overall_mean
        genre_var = 0.0

    return {
        'album_score': album_score, 'artist_score': artist_score,
        'genre_count': genre_count, 'genre_max':    genre_max,
        'genre_min':   genre_min,   'genre_mean':   genre_mean,
        'genre_var':   genre_var,
    }
```

Listing 13: Updated full feature vector builder (v2)

Impact Analysis — Fallback Tier Breakdown

After building the v2 feature matrix across all 120,000 test rows:

Table 6: Album Score Fallback Tier Breakdown (120,000 test rows)

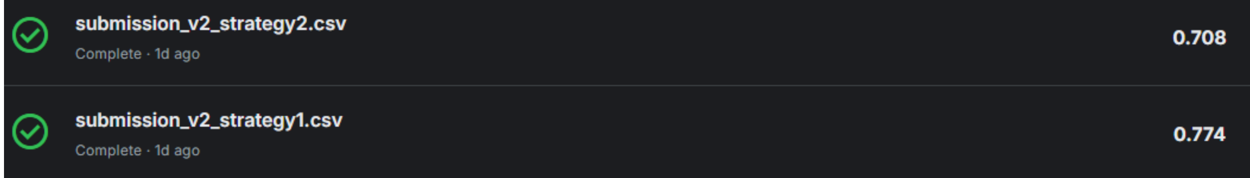
Tier	Count	Percentage
Tier 1: Direct user album rating	34,264	28.6%
Tier 2: Dig Deeper (sibling tracks)	5,678	4.7%
Tier 3: Global Bayesian fallback	71,486	59.6%
No Album ID (track has none)	8,572	7.1%
Total	120,000	100.0%

Observations

1. **Data Sparsity:** 59.6% of album score lookups fell all the way through to the global Bayesian fallback, confirming the dataset is extremely sparse. Users have rated only a small fraction of the albums their candidate tracks belong to, meaning global popularity carries the majority of the predictive weight.
2. **Dig Deeper Trigger Rate:** The Tier 2 rule fired for 5,678 rows (4.7%). While modest as a fraction, this represents approximately 5,600 cases where the system produced a personalized album inference from the user's own listening history rather than defaulting to community data.
3. **Does Dig Deeper Help?** For the 4.7% of rows where it fires, the Dig Deeper score is derived from the user's own history, making it substantially more personalized than the Bayesian global average.
4. **Conclusion on Sparsity:** The data is dominated by cold-start conditions. Only 28.6% of album lookups found a direct user rating. Improving the quality of the global fallback (Bayesian averaging over a flat mean) is therefore more impactful at scale than the Dig Deeper rule, even though the latter is more theoretically sound where it applies.

Table 7: Part 2 Kaggle Submission Scores

Strategy	Score
Strategy 1 — submission_v2_strategy1	0.774
Strategy 2 — submission_v2_strategy2	0.708



Further Improvement — Strategy 3: Adaptive Weight Normalization

Motivation

Strategies 1 and 2 always blend all three hierarchy levels (album, artist, genre) even when the user has directly rated only one of them. Consider:

Approach	Score (album=90, others fallback=50)
Strategy 1	$0.40 \times 90 + 0.30 \times 50 + 0.30 \times 50 = 66$
Strategy 3	$\frac{0.50 \times 90}{0.50} = 90$

Strategy 3 uses only the levels the user has *directly* rated, then normalizes the weights of those levels to sum to 1.0. When no direct signal exists at any level, it falls back to the Bayesian global popularity scores.

Weights: album = 0.50, artist = 0.30, genre = 0.20 (album most specific).

Implementation

```
def score_strategy_3(user_id, track_id, track_to_hierarchy,
                     user_to_ratings, overall_mean,
                     weights={'album': 0.5, 'artist': 0.3,
                        'genre': 0.2}):
    uid = str(int(user_id))
    tid = str(int(track_id))
    hierarchy    = track_to_hierarchy.get(tid, {})
    user_ratings = user_to_ratings.get(uid, {})

    album_id  = hierarchy.get('album')
    artist_id = hierarchy.get('artist')
    genre_ids = hierarchy.get('genres', [])

    score        = 0.0
    total_weight = 0.0

    # Only include levels the user has DIRECTLY rated
    if album_id and album_id in user_ratings:
        score        += weights['album'] * user_ratings[album_id]
        total_weight += weights['album']

    if artist_id and artist_id in user_ratings:
        score        += weights['artist'] *
           user_ratings[artist_id]
        total_weight += weights['artist']

    rated_genres = [user_ratings[g] for g in genre_ids
                    if g in user_ratings]
```

```
27     if rated_genres:
28         score        += weights['genre'] * np.mean(rated_genres)
29         total_weight += weights['genre']
30
31     if total_weight > 0:
32         return score / total_weight   # normalize to known
               levels only
33
34     # Complete cold-start: use Bayesian global popularity as
          tiebreaker
35     signals = []
36     if album_id:
37         signals.append(global_item_scores.get(album_id,
               overall_mean))
38     if artist_id:
39         signals.append(global_item_scores.get(artist_id,
               overall_mean))
40     for g in genre_ids:
41         signals.append(global_item_scores.get(g, overall_mean))
42     return np.mean(signals) if signals else overall_mean
```

Listing 14: Strategy 3: Adaptive Weight Normalization

Validation errors: 0. Output: `submission_v3_strategy3.csv`.

Further Improvement — Strategy 4: Autoencoder Fallback Experiment

Motivation

Although the assignment emphasizes a rule-based recommendation system, one remaining weakness in the heuristic pipeline was clear: a very large portion of candidate rows still fell through to the final fallback stage. In the v2 pipeline, 59.6% of album score lookups ultimately relied on the global Bayesian fallback. That makes the final prediction less personalized, even though the rest of the system is built around user-specific hierarchy signals.

Because I had recently studied autoencoders in another class, I implemented a single exploratory test to evaluate whether a learned fallback could improve the weakest link in the pipeline. The intention was **not** to replace the entire rule-based recommendation system, since that would move away from the purpose of the assignment. Instead, the idea was to keep the same hierarchy, feature engineering, and ranking logic, and only replace the last fallback step with a more personalized estimate.

Concept

The autoencoder experiment keeps the same high-level recommendation structure:

1. Use direct user history when available.
2. Use "Dig Deeper" sibling-track logic when album-level history is missing.
3. Only when those personalized heuristics fail, use an autoencoder-predicted track preference as the fallback.

In other words, the system is still fundamentally heuristic and rule-based. The autoencoder is used only as a *replacement for the weakest final fallback*, not as a full end-to-end recommender.

Brief Implementation

The exploratory notebook `autoencoder.ipynb` builds a user–track rating matrix from the training data and trains a compact autoencoder to reconstruct observed track ratings. The encoder compresses each user's listening behavior into a latent representation, and the decoder reconstructs predicted preference scores for tracks in that reduced latent space.

The modified fallback order becomes:

Tier	Name	Logic
1	Primary	Direct album rating from the user
2	Dig Deeper	Mean of user's ratings for sibling tracks in the album
3	Learned fallback	Autoencoder-predicted track score
4	Safety fallback	Global track popularity / Bayesian score

This means the original heuristic framework remains intact: album, artist, and genre reasoning still drive the recommendation rule, but the final missing-value case becomes more personalized than a community-wide average.

```
# FINAL KAGGLE RUN: AE AS THE FALLBACK ONLY

FINAL_OUTPUT_CSV = "submission_autoencoder_fallback.csv"

FINAL_TOP_POPULAR_TRACKS = 12000
FINAL_INCLUDE_ALL_TEST_CANDIDATES = True

FINAL_AE_HIDDEN_DIM = 256
FINAL_AE_LATENT_DIM = 64
FINAL_AE_DROPOUT = 0.15
FINAL_AE_EPOCHS = 10
FINAL_AE_BATCH_SIZE = 128
FINAL_AE_LR = 1e-3
FINAL_AE_WEIGHT_DECAY = 1e-5
FINAL_AE_BLEND_WEIGHT = 0.85

```

```
17 print("Running final AE-fallback submission build")
18 print(f"Output file: {FINAL_OUTPUT_CSV}")
```

Listing 15: Strategy 4: Adaptive Weight Normalization

Observation

This was intentionally a light one-run experiment rather than a fully optimized deep learning pipeline. The model was not trained for long, and detailed hyperparameter tuning, architecture search, and normalization experiments were not fully explored. Even so, the early results were favorable enough to justify documenting the method as a promising extension. The main takeaway is that a learned fallback may be useful precisely because the rest of the hierarchy is already strong: rather than replacing the rule-based system, it can be used to strengthen the small but important subset of rows that would otherwise default to a weak global prior. This new strategy was able to reach a score of 0.849.

Results Summary

Table 8: Kaggle Submission Results

Submission File	Strategy Description	Score
submission_strategy1_weighted_avg.csv	Weighted Avg (v1 fallback)	0.759
submission_strategy2_max_genre.csv	Max Genre (v1 fallback)	0.704
submission_v2_strategy1.csv	Weighted Avg + Bayesian + Dig Deeper	0.774
submission_v2_strategy2.csv	Max Genre + Bayesian + Dig Deeper	0.708
submission_v3_strategy3.csv	Adaptive Weight Normalization	0.792
submission_v3_strategy4.csv	Auto-Encoder Fallback	0.849

Best performing strategy: Strategy 4 — Auto-Encoder Fallback, with a Kaggle score of **0.849**.

Key finding: Album-level preference is the strongest individual predictor in this dataset. Strategies that weight album signals heavily outperform genre-centric approaches, suggesting that a user's album taste is more predictive of individual track preference than broad genre affinity. This aligns with the intuition that an album represents a curated listening experience the user has consciously chosen to engage with, making it a high-confidence signal for any track it contains.

#	Team	Members	Score	Entries	Last
1	Jude Eschete		0.911	62	1d
2	Dylan zhuJX		0.903	23	1d
3	Kaylyn Sullivan		0.890	16	1d
4	Shauryaditya_Si		0.877	46	4h
5	Liam Guske		0.861	63	9h
6	Tejas Nidhankar		0.861	5	9h
7	adus231102		0.857	13	5h
8	A GYANA SAI		0.856	34	1d
9	Kyle Graupe		0.856	16	1h
10	Syed Ahmad Shah		0.849	27	18m