Improving Deezer's Music Recommendation Engine

(Deezer Recommender System – DSG17 Online Phase)

Gerber Andri

2025-07-10

Figure 1: Graph

Recommender Systems

Block Seminar
Spring Semester 2025

Lecturer: Dr. Guang Lu

February 2025

1. Introduction

Streaming music services like Deezer must deliver accurate, timely song suggestions to keep users engaged and satisfied. In this project, I build a music recommendation engine using data from the *DSG17 Online Phase* competition on Kaggle, leveraging a large dataset of user-listening histories [1]. The primary goal is to **predict whether a user will listen to a recommended track for more than 30 seconds** (`is_listened = 1`) or skip it early (`is_listened = 0`).

A key **challenge** in this task is the **imbalance** of the target: although millions of listening events are recorded, many of them are "skips" (0) versus "listened" (1) or vice versa, depending on the sample. To robustly evaluate our models on such an imbalanced dataset, I focus on the **ROC AUC** (Area Under the Receiver Operating Characteristic Curve) metric. This is also the official competition measure.

Why **ROC AUC**?

1. **Threshold Independence**: ROC AUC evaluates how well a model distinguishes positives from negatives across all possible thresholds. This helps us avoid picking an arbitrary cutoff for deciding "listen" vs. "skip," which might not be optimal given strong class imbalances or certain business constraints.
2. **Robust to Class Skew**: ROC AUC considers the model's True Positive Rate (TPR) and False Positive Rate (FPR) for different thresholds, offering a more reliable comparison when positives are relatively rare (or, in some datasets, especially frequent). While metrics like accuracy can be misleading when 70–80% of instances fall into one class, ROC AUC gives us a clearer sense of how effectively I separate listens from skips.
3. **Competition Requirement**: The **DSG17 competition** specifically states that the submissions are ranked by ROC AUC, reinforcing our choice.

Using **ROC AUC** as the **primary** measure, I still supplement further analysis—such as PR/ROC curves, threshold-dependent metrics, learning curves (Figures: 5–7 in the Appendix)-to better understand skip detection at specific thresholds. This ensures both alignment with the competition goals and a strong technical approach to imbalanced classification, which aligns with recent insights from studies [2].

To address this challenge, I explore **multiple recommender approaches**:

- **Baseline Methods (Group 1):**
  - Content-Based Filtering
  - Collaborative Filtering
  - Matrix Factorization (including a Probabilistic Matrix Factorization variant)
  - Restricted Boltzmann Machine (RBM)
- **Advanced Methods (Group 2):**
  - Factorization Machines (FM)
  - Neural Collaborative Filtering (NCF)
  - Neural Matrix Factorization (NeuMF)
  - GraphSAGE Recommender

I describe each step of the **data pipeline**—including EDA, feature engineering, model training, hyperparameter tuning, and final predictions—and then present results with a focus on ROC AUC plus other performance measures (precision, recall, F1) where feasible.

Finally, I answer three key questions from the assignment:

1. **How would I develop a recommendation algorithm to win this competition?**
2. **What do I propose to solve Deezer's general recommendation problems?**
3. **Do the two solutions above overlap?**

I conclude with a short reflection on our learning and technical insights.

2. Data & Evaluation

2.1 Dataset Overview

The Kaggle-provided training data (train.csv) logs user-song interactions, each row showing whether a user listened to a song (media_id) for over 30 seconds (is_listened=1) or skipped it (is_listened=0). The test data (test.csv) contains users and tracks without the is_listened label, for which I must predict the probability of listening.

Key fields include:

- **user_id, media_id, artist_id, album_id:** Identifiers for users, songs, artists, and albums.
- **genre_id:** The genre of each track.
- **ts_listen:** Unix timestamp for the listening event.
- **platform_name, platform_family:** The OS and device family used.
- **release_date:** Song release date (in YYYYMMDD format).
- **user_age, user_gender:** Demographics for users.
- **listen_type, context_type:** Whether a track was played via "Flow" or not, plus context (playlist, album, etc.).
- **media_duration:** Duration of the song (in seconds).
- **is_listened:** Target = 1 if the user listened >30 s, 0 if skipped (training data only).

2.2 Data Splitting & Evaluation Protocol

I partition the fully labeled training data (from the supplemented `train.csv`) into roughly 70% for training, 15% for validation, and 15% for evaluation using our helper function. This split is essential because `test.csv` lacks target values and is used only for final predictions. Our function supports several strategies: a **global (time-based) split**, which orders data by `ts_listen` so that the earliest 70% is training, the next 15% is validation, and the final 15% is evaluation; a **user-level split** that holds out each user's last one or two interactions; and a **random split** that shuffles data with a fixed seed. For baseline models, I used the global split because it best captured temporal dynamics and yielded higher ROC AUC, whereas for advanced models like NCF and NeuMF, a random split provided a slight boost by exposing the models to a more diverse set of interactions. Having defined our splitting and evaluation setup, I next perform an in-depth EDA and data cleaning in Section 3.1 to ensure consistent, high-quality input for our models.

3. Feature Engineering & Data Pipeline

In a real-world setting—especially with datasets of this size and complexity—a robust, modular data pipeline is critical for building reliable recommendation systems. Our approach for **feature engineering** and **data preparation** involved three main steps:

1. **EDA & Basic Cleaning** (*notebook 1*)
2. **Preprocessing to Create a Unified "Feature Store"** (*notebook 2*)
3. **Model-Specific Transformations** (*notebook 3*)

Below, I describe in detail how these steps interact, the rationale behind each decision, and how I leveraged them to streamline model development.

3.1 EDA & Basic Cleaning

Our initial exploratory analysis, documented in in *notebook 1*, confirmed that the dataset is robust. I found **no missing data**—except for a few edge cases in `release_date`—and **no duplicates**, which simplified our record-level checks. Given that most interactions occur **post-2015**, I filtered out older data to focus on more recent user behavior.

I also identified **right-skeId song durations** (extreme outliers up to 65,535 seconds), leading us to cap or remove improbable values. Furthermore, several categorical variables (such as `genre_id`, `artist_id`, `album_id`) displayed **long-tailed distributions**, motivating dimension reduction or pruning of extremely rare categories. The target variable is **moderately imbalanced** (~60% "listened" vs. ~40% "skipped"), and diverse user interactions further highlight the complexity of this dataset.

Building on these observations, I engineered features to capture **temporal** and **behavioral** signals. I derived time bins (hour, Iekday, weekend) from the UNIX timestamps and computed each track's "age" by subtracting its `release_date` from the `ts_listen`. I then introduced **user behavior metrics**—like skip ratios, average listening duration, and a composite behavior index—to encapsulate individual preferences. Additionally, I applied **frequency thresholds** to reduce sparsity in high-cardinality fields, and combined certain features (e.g., media duration with user demographics) for more nuanced interactions. While time and resource constraints limited the scope of the experiments, these enriched features form a **flexible foundation** for subsequent modeling approaches, allowing us to continuously refine performance.

To finalize our cleaning strategy, I established several "soft" constraints: capping outlier song durations, dropping rare categories, and restricting the dataset to valid, post-2015 observations. Collectively, these steps balanced data quality against coverage, ensuring that the processed data was both comprehensive and manageable for downstream model development.

3.2 Preprocessing & Unified Feature Store

Building on the **EDA observations**, I designed a robust, modular pipeline for feature engineering and data preparation resulting in our **unified feature store** (details in *notebook 2*). This ensures consistency, scalability, and reproducibility across models (see Figure: 3 in the Appendix) with the following steps:

1. **Chunk-wise Data Loading**
   - For large CSVs ($>$ 8 million rows), I used a chunk-based loader with `dtype` mappings (e.g., `int8` for `user_gender`, `int32` for `media_id`) to manage memory efficiently and merged them into one DataFrame.
2. **Filtering & Timestamps**
   - Timestamp (`ts_listen`) conversion to datetimes, dropping invalid entries.
   - Based on EDA I excluded listens before 2015 and extracted time features (hour, Iekday, is_weekend, day, week, month, year) to capture cyclical patterns.
3. **Basic Cleaning**
   - Replaced rare `genre_id` or `album_id` with -1 and removed outliers in `media_duration` using an IQR-based filter dropping extreme durations like 65k seconds.
4. **Rare Category Pruning / Dimension Reduction**
   - Dropped low-frequency categories (e.g. `genre_id`, `artist_id`) using a fixed threshold (e.g. frequency $<$50) or an **auto-threshold** at the 5% quantile.
5. **Derived Time Features**
   - Created `song_age_days` (difference between `ts_listen` and `release_date_parsed`).
   - Binned hours into broader categories (`binTime`), simplified weekdays into a weekend/weekday flag (`binwkd`), and grouped release years (`binRYear`).

6. **Behavioral & Aggregation Features**
    - Computed track-level metrics like `skip_score` (skip ratio) and `avg_song_listened` (fraction listened >30 seconds).
    - Derived user aggregates (e.g., average songs per session, session length) and time-based popularity (7d, 30d, 90d windows).
    - Formulated a composite `behavior_index` to capture user–genre affinities.
7. **User-Item Cross Features**
    - Combined features such as `md_mf_group` (merging media duration brackets with user gender) and identified each user's top genre (`user_top_genre`).
8. **Final Output**
    - Saved a consolidated DataFrame (CSV/Parquet) with advanced features and summary stats for consistent train–test merging.

As a result, I ended with a ~7.15M-row dataset containing 43 feature columns, providing a standardized "feature store" for all recommender algorithms.

3.3 Model-Specific Transformations

Despite the unified feature store, certain methods—like **Neural Collaborative Filtering (NCF)** or **GraphSAGE**—need extra steps (see : Figure: 4 in the Appendix and details in *notebook 3*):

- **NCF/NeuMF**: I remapped user/item IDs to contiguous integer indices (e.g., `[0..num_users - 1]`) to properly initialize embedding layers.
- **GraphSAGE**: I constructed a bipartite graph using positive interactions (is_listened=1) by offsetting item IDs (i.e., `item_id + num_users`) and building an edge index, with negative sampling applied during training.
- **FM**: Factorization Machines typically benefit from a sparse one-hot representation. I transformed features into a `csr_matrix` before fitting fastFM [3].

By placing these transformations after I build the feature store, all advanced features remain available; I simply pick or convert what each model needs.

3.4 Benefits of a Modular Pipeline

1. **Consistency Across Models:** Centralizing outlier handling, category pruning, and advanced feature creation ensures each algorithm (baseline or advanced) uses the same core data.
2. **Scalability:** With well-defined steps, I can refresh or partially re-run time-based features for new data, without rewriting the entire pipeline.
3. **Debugging & Iteration:** When anomalies appear—e.g., missing `release_date`—the pipeline's logs let us trace each step and fix issues quickly.
4. **Reuse & Collaboration:** Different teams can experiment with newly added features or try advanced architectures without duplicating data cleaning work, boosting overall productivity.

Overall, a **robust, modular pipeline** dramatically accelerates experimentation and deployment cycles, especially crucial for multi-million-row recommender systems. All subsequent models—whether baseline or advanced—benefit from these consistent, well-engineered features.

3.5 Hyperparameter Tuning

For baseline models, I used ParameterGrid (see **Table 1**) to tune parameters for content-based filtering, collaborative filtering, matrix factorization, and PMF. For advanced models, I grid searched FM parameters, used Keras Tuner's RandomSearch with early stopping to optimize NCF and NeuMF (see **Table 2**), and manually tuned GraphSAGE. The best configurations—selected based on validation ROC AUC—ensure consistency and reproducibility for final evaluation and test predictions (see implementation in *notebook 3*).

4. Group 1: Baseline Recommenders

In the baseline phase, I explore methods that are well-known for their computational efficiency and interpretability, using them as a comparative yardstick. These include Content-Based Filtering, Collaborative Filtering, Matrix Factorization, Probabilistic Matrix Factorization (PMF), and a Restricted Boltzmann Machine (RBM). Each approach targets a different blend of item attributes and user–item interactions, providing a comprehensive view of simpler yet foundational recommender strategies.

4.1 Content-Based Filtering

Content-based filtering focuses on item-level attributes such as genre, artist, album, and release date. I encode these features, transform them into appropriate numeric or categorical representations, and then train a logistic regression model to predict the probability that a user will listen to a recommended track for more than 30 seconds. Building on the principles of content-driven music recommendation outlined in Deldjoo et al. [4], this method is highly interpretable—allowing stakeholders to clearly see how each item attribute influences the recommendation. However, it may overlook more subtle user–item interaction patterns.

In our experiments, after tuning the logistic regression regularization parameter C, I achieved a validation ROC AUC of about 0.5237. On the **evaluation set**, the final **ROC AUC stood at 0.5274**. Although this demonstrates that content-based signals are not entirely negligible, it underscores that user behavior and collaborative insights are likely more influential for better ROC AUC.

4.2 Collaborative Filtering

Collaborative filtering leverages user–item interaction histories [5]. I include features such as `avg_song_listened` (average listen rate for the item) and `skip_score` (fraction of times a track is skipped) alongside `(user_id, media_id)` pairs, then train another logistic regression. Collaborative filtering shines in settings with abundant user feedback, as it directly encodes collective behavior patterns.

Tuning the same logistic regression hyperparameter C resulted in a validation ROC AUC around 0.5366, yielding an **evaluation ROC AUC of approximately 0.5383**. This surpasses content-based filtering by a reasonable margin, demonstrating the advantage gained by using aggregated user–item interaction signals.

4.3 Matrix Factorization

Matrix factorization transforms user–item interaction data into latent factors that capture underlying dimensions of preference. I incorporate contextual features (e.g., `context_type`, `binTime`) into the matrix, apply truncated SVD to reduce dimensionality, and then feed the latent factors into a logistic regression classifier. This hybrid approach can uncover non-trivial structures in user behavior while remaining relatively lightweight.

Our best matrix factorization model matched collaborative filtering in performance, with a validation ROC AUC of 0.5366 and an identical **evaluation ROC AUC of 0.5383**. The near tie suggests that either method captures similar structural signals in the user–item matrix when combined with basic context, which is consistent with findings in [6].

4.4 Probabilistic Matrix Factorization (PMF)

PMF focuses on minimal user–item features—just `(user_id, media_id)` plus the binary target—and learns latent factors along with per-user and per-item biases via gradient descent. Our implementation, inspired in part by the approach in [7], adapts this framework for binary outcomes by incorporating a stable sigmoid function and explicit bias updates. Nevertheless, our best PMF model (tuned over factor dimensions, learning rate, and regularization) yielded an **evaluation ROC AUC around 0.4850**, which fell below the other baselines. This indicates that purely factorizing user–item interactions, without contextual or item-level data, underperforms in our skip prediction scenario.

4.5 Restricted Boltzmann Machine (RBM)

An RBM can learn hidden binary representations from content-based features (e.g., `genre_id`, `artist_id`, `album_id`, and derived "song age"). I then feed these latent representations into a logistic regression layer for final classification. Although RBMs have been extensively used for modeling user ratings in collaborative filtering tasks (see, Salakhutdinov et al. [8]), in our pipeline I adapt the approach to extract latent features from item content. Our tuned RBM-based pipeline achieved a validation ROC AUC of 0.5174 and an **evaluation ROC AUC of 0.5187**—better than PMF, but still below collaborative filtering and matrix factorization.

4.6 Summary of Baseline Models

The following **Table 1** summarizes the ROC AUC metrics and best hyperparameters for each baseline recommender. Collaborative Filtering and Matrix Factorization both produce the **highest ROC AUC scores arround 0.5383 on our evaluation set**, highlighting the gains from user interaction data and/or latent factor embeddings, while PMF (0.4850) falls below random basline ROC AUC (0.5) and RBM remain below 0.52 in final performance.

Table 1: Performance of Baseline Models

Model	ValAUC	EvalAUC	BestHyperparameters
Content-Based Recommender	0.5237	0.5274	`C=1` (Logistic Regression)
Collaborative Filtering Recommender	0.5366	0.5383	`C=0.1` (Logistic Regression)
Matrix Factorization Recommender	0.5366	0.5383	`n_components=6` (Truncated SVD + LR)
PMF Recommender	0.4850	0.4850	`n_factors=10, lr=0.01, reg=0.01` (PMF)
RBM Recommender	0.5174	0.5187	`n_components=100` (RBM + LR)

Overall, these baseline results illustrate the relative strengths and limitations of simpler methods. Content-based approaches provide a basic signal but struggle to incorporate user-specific behavior. Pure user–item factorization is not sufficiently rich for skip prediction, and adding item-level or contextual signals (as in collaborative filtering and matrix factorization) boosts predictive power to around 0.5383 ROC AUC. This sets a benchmark for more advanced techniques in the next section.

5. Group 2: Advanced Recommenders

Moving beyond the baseline models, I seek to exploit an expanded feature set—including time-based, contextual, and user behavior variables—and to employ deeper architectures capable of learning complex interaction patterns. The four advanced methods tested are: Factorization Machines (FM), Neural Collaborative Filtering (NCF), Neural Matrix Factorization (NeuMF), and a GraphSAGE-based approach. Each integrates additional nuance, from pairwise feature interactions (FM) to graph-based embeddings (GraphSAGE). In the

following sections, I use **ROC AUC** on the **evaluation set** as our primary metric, since all models were optimized for this threshold-independent measure. In the Appendix, you'll find additional threshold-dependent metrics—**precision, recall, F1, and accuracy** (at a 0.5 threshold) with **confusion matrices**—for a more detailed view on the best performing models (NCF, NeuMF, GraphSAGE). I also include **PR** and **ROC AUC curves** to illustrate model behavior across thresholds, as well as a **"loss over epochs" plot** to show training convergence and potential overfitting. I also discuss factors like model complexity and training time (exact timings are unavailable since I realized too late that I omitted `%%time`).

5.1 Factorization Machines (FM)

Factorization Machines generalize linear models by representing features in a latent dimension and modeling pairwise interactions. I feed a large variety of features—identifiers (user, item, artist, album), time bins, user aggregates (like behavior index), and so on—into an FM classifier from fastFM [3]. Despite capturing many interactions automatically, the best FM configuration underperforms relative to other advanced models, finishing with a **~0.58 ROC AUC on the evaluation** set. That result highlights both the potential complexity of hyperparameter tuning for FMs and the value of specialized neural or graph-based approaches.

5.2 Neural Collaborative Filtering (NCF)

NCF uses embedding layers for both users and items, which are then concatenated and passed through a feed-forward neural network [9]. By tuning hyperparameters (e.g., embedding size, dropout rate) with Keras Tuner, I obtain a configuration that achieves an **evaluation ROC AUC of ~0.79**. The improvement shows that neural embeddings can capture user–item relationships more effectively than linear or purely factorized approaches, especially when sufficient data is available.

In the Appendix, Figure: 5 presents additional metrics—including precision, recall, F1, confusion matrix, PR/ROC curves, as well as ROC AUC and training and validation loss curves. Our training and validation loss drop rapidly in the first two epochs and plateau around epoch 3, indicating limited benefit from further training and notably good generalization. NCF's parameter count lies between FM (fewer) and NeuMF (more), and on a GPU (e.g., NVIDIA T4), training roughly 7 million interactions runs faster than GraphSAGE and is comparable to NeuMF.

5.3 Neural Matrix Factorization (NeuMF)

NeuMF builds on NCF by merging a Generalized Matrix Factorization (GMF) branch with an MLP branch [9]. The GMF branch models linear interactions (via element-wise multiplication of user and item embeddings), while the MLP branch captures non-linear relationships by concatenating embeddings and passing them through multiple dense layers. Our best NeuMF model yields an **evaluation ROC AUC of ~0.80**. Compared to standard NCF, this approach can exploit both linear and complex interactions, pushing performance slightly higher.

As shown in Figure: 6 in the Appendix, the training loss (blue) decreases from ~0.51 to ~0.44 over 3 epochs, while the validation loss (orange) dips from ~0.50 to ~0.49 then rises slightly, suggesting overfitting. Despite its dual-branch architecture (GMF + MLP) and higher parameter count compared to NCF, NeuMF still trains noticeably faster than GraphSAGE.

5.4 GraphSAGE Recommender

GraphSAGE, introduced by Hamilton et al. [10] and further advanced in scalable frameworks such as PyTorch Geometric [11] reframes the problem as embedding nodes in a bipartite user–item graph. Each positive interaction becomes an undirected edge, and I apply SAGEConv layers to iteratively aggregate neighbor information. Negative sampling complements training, distinguishing true edges from random user–item pairs.

As shown in Figure: 7 in the Appendix, the training loss (blue) decreases from ~0.95 to ~0.84 over nine epochs, while the validation loss (orange) moves from ~0.98 to ~0.96 and plateaus around epoch 7. This gap indicates steady learning yet suggests overfitting after epoch 7.

GraphSAGE emerges as the top-performing model in terms of **ROC AUC of ~0.82** on the **evaluation** set. This suggests that graph-based approaches can excel where user–item connectivity patterns are central, as adjacency relationships offer a structured way to capture user and item neighborhoods. This aligns with recent surveys on GNN-based recommender systems, which note that leveraging adjacency relationships boosts performance and scalability on large-scale graphs [12]. However, due to neighbor sampling at each training step, GraphSAGE typically takes longer to train than NCF or NeuMF when operating on millions of interactions.

5.5 Summary of Advanced Models

All four advanced models leverage richer (time-based, contextual, and behavioral) features and deeper architectures, surpassing the baseline ROC AUC of 0.5 for random guessing. **Factorization Machines (FM)** reach ~0.58 and can be preferable with moderate data or strict interpretability needs. **Neural Collaborative Filtering (NCF)** scores ~0.79, balancing complexity and speed, while **Neural Matrix Factorization (NeuMF)** improves to ~0.80 at the cost of overfitting. **GraphSAGE** leads with ~0.82, making it ideal when capturing user–item connectivity is central, albeit with a heavier training load due to neighbor sampling. **Table 2** presents each model's validation/evaluation ROC AUC and best hyperparameters.

Table 2: Performance of Advanced Models

Model	ValAUC	EvalAUC	BestHyperparameters
FM	~0.63	~0.58	`rank=30, n_iter=150, l2_reg_w=0.1, l2_reg_V=0.01`
NCF	~0.79	~0.79	`embedding_dim=16, fc_units=32, dropout_rate=0.4`
NeuMF	~0.80	~0.80	`gmf_embedding_dim=16, mlp_embedding_dim=24, num_mlp_layers=1, dropout_rate=0.2`
GraphSAGE	~0.89	~0.82	`embedding_dim=64, num_layers=2`

All three deep learning or graph-based methods offer significant gains over the best baseline approaches. While FM does not appear competitive under our current hyperparameter search, it may improve with a more extensive tuning protocol. In practice, I also see that deeper or graph-oriented models typically require more computational resources and careful sampling strategies. Nevertheless, their ability to fuse multiple feature types—including user behaviors, item properties, and temporal context—makes them especially relevant for large-scale music recommendation scenarios.

6. Key Questions & Discussion

6.1 How would you develop a recommendation algorithm to win this competition?

To maximize **ROC AUC** in a short-term, competition-focused setting, I would rely on a two-pronged strategy: **feature richness** and **ensemble methods**. First, I build a wide variety of contextual and behavioral features, including real-time popularity windows, skip ratios, user–genre affinity scores, and session-based signals. Second, I train several high-performing single models—like **GraphSAGE**, **NeuMF**, and a tuned **Collaborative Filtering** pipeline—and blend their outputs using either a **meta-learner** (e.g., a small logistic regression on their predicted scores) [13] or a **weighted average** scheme. This ensemble typically outperforms any single architecture by exploiting each model's unique strengths [14]. I would further refine hyperparameters (embedding sizes, learning rates, negative sampling strategies) with automated search tools such as **Keras Tuner**. Because the challenge emphasizes ROC AUC for a static test set, I need not worry as much about real-time latency; instead, I focus on carefully engineered features, robust ensembles, and extensive cross-validation to push ROC AUC to the limit.

6.2 What do you propose to solve Deezer's general recommendation problems?

For Deezer's **long-term, production-scale** needs, I propose a **modular pipeline** capable of handling massive volumes of streaming data and evolving user tastes in near real time. Key components would include:

1. **Scalable Data Ingestion & Processing:**
   Implement an incremental or streaming approach to update user–item interactions daily (or even hourly), re-computing time-based popularity metrics and partial user embeddings without rerunning the entire pipeline.
2. **Contextual & Session-Aware Methods:**
   Incorporate session-based neural models (e.g., RNNs or Transformers) to handle short session contexts and dynamic skipping behavior. This ensures up-to-date, context-sensitive recommendations rather than purely historical patterns.
3. **Hybrid Architectures & Diversity Control:**
   Combine collaborative signals with content features (genre, artist, or album attributes). Enforce track diversity in recommendations so that users discover new music and avoid "echo chamber" effects.
4. **Feedback Loop & Online Learning:**
   Continuously gather skip signals, partial plays, and user "likes." Integrate these into an online learning framework, ensuring that the system swiftly reacts to user drift or new content launches.
5. **Interpretability & Personalization:**
   Provide user-friendly explanations, e.g., "Recommended because you like [Artist X] and enjoy short tracks on weekday mornings." This builds trust, acceptance, and deeper engagement with Deezer's platform.

6.3 Do the two solutions overlap?

Yes—there is a significant **overlap** in foundational techniques. Both a Kaggle-style approach and a long-term Deezer production pipeline rely on robust feature engineering, factorization or embedding-based models, and consistent hyperparameter tuning. However, as **Figure: 2** illustrates, they differ in **scope and emphasis**:

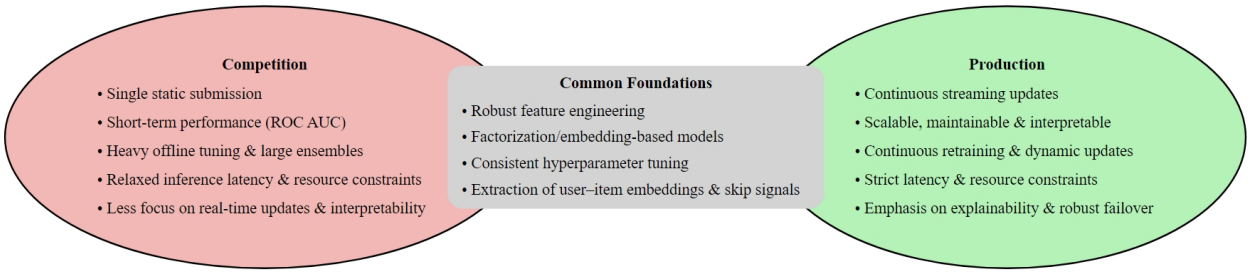

Figure 2: Comparison of Kaggle Competition and Real-World Production Needs

Competition:

- Targets a **single static submission** and **short-term performance (ROC AUC)**.
- Emphasizes **heavy offline tuning** and like in our case large ensembles without strict inference latency or memory constraints.
- Less concerned with real-time updates or interpretability.

Real-World Production:

- Requires **scalability, maintainability, and interpretability** to handle tens of millions of daily interactions.
- Incorporates **continuous feedback loops** and dynamic updates for new tracks or user changes.
- Demands **explainable results**, robust failover, and compliance with computational budgets and latency SLAs (Service-Level Agreements).

Thus, while the core modeling ideas are similar—extract user–item embeddings, tune models, handle skip signals—the **operational and product requirements** for Deezer's real business environment introduce additional layers: real-time streaming ingestion, interpretability needs, and continuous retraining constraints.

7. Conclusion

I developed a Deezer recommender system capable of predicting whether users will listen to newly recommended music tracks for over 30 seconds. Our **baseline** approaches (Content-Based, Collaborative Filtering, and Matrix Factorization) achieved **ROC AUC** of about **0.52–0.54**, while more specialized methods like **Probabilistic MF** and **RBM** fared slightly worse in our data setting. By expanding to **advanced models**—notably **Neural Collaborative Filtering (NCF)**, **NeuMF**, and **GraphSAGE**—I obtained substantially higher **ROC AUC** values, peaking at around **0.79–0.82** on our final evaluation split.

Key Takeaways:

1. **Advanced Feature Engineering**: Incorporating time-based popularity, user–genre behavior indices, and skip scores enhances model accuracy considerably.
2. **Deeper Architectures**: Neural methods and graph-based approaches can learn nuanced patterns, outperforming traditional baselines when sufficient data and computational resources are available.
3. **End-to-End Pipelines**: A clean, modular pipeline for data ingestion, outlier removal, feature construction, and final model training is essential—especially at the ~7M–row scale.
4. **Future Outlook**: For Deezer's live environment, I recommend a hybrid approach that continuously updates embeddings for new songs and integrates user session signals (e.g., sequence modeling) to handle skipping behavior in real time.

Overall, our results highlight that combining robust feature engineering with state-of-the-art modeling—like **GraphSAGE** or deep embedding methods—unlocks strong predictive performance for music skip prediction. Deezer can build on these insights, layering real-time data, interpretability modules, and user-centric design to maintain a market-leading recommendation engine.

References

1. DSG17 online phase, 2017 (2017). Available: https://kaggle.com/competitions/dsg17-online-phase.

2. Zangerle E, Bauer C (2022) Evaluating recommender systems: Survey and framework. ACM Computing Surveys 55: 1–38. Available: https://doi.org/10.1145/3556536.

3. Bayer I et al. (2014) fastFM: A library for factorization machines.

4. Deldjoo Y, Schedl M, Knees P (2024) Content-driven music recommendation: Evolution, state of the art, and challenges. Computer Science Review 51: 100618. Available: https://www.sciencedirect.com/science/article/pii/S1574013724000029.

5. Papadakis H, Papagrigoriou A, Panagiotakis C, Kosmas E, Fragopoulou P (2022) Collaborative filtering recommender systems taxonomy. Knowledge and Information Systems 64: 35–74. Available: https://api.semanticscholar.org/CorpusID:246476870.

6. Koren Y, Bell R, Volinsky C (2009) Matrix factorization techniques for recommender systems. Computer 42: 30–37. doi:10.1109/MC.2009.263.

7. ocontreras309 (2020) PMF for recommender systems.

8. Salakhutdinov R, Mnih A, Hinton G (2007) Restricted boltzmann machines for collaborative filtering. Proceedings of the 24th international conference on machine learning. ICML '07. New York, NY, USA: Association for Computing Machinery. pp. 791–798. Available: https://doi.org/10.1145/1273496.1273596.

9. He X, Liao L, Zhang H, Nie L, Hu X, et al. (2017) Neural collaborative filtering. arXiv preprint arXiv:170805031. Available: https://arxiv.org/abs/1708.05031.

10. Hamilton W, Ying Z, Leskovec J (2017) Inductive representation learning on large graphs (GraphSAGE). arXiv preprint arXiv:170602216. Available: https://arxiv.org/abs/1706.02216.

11. PyG Team (2023) PyTorch geometric.

12. Gao C, Zheng Y, Li N, Li Y, Qin Y, et al. (2023) A survey of graph neural networks for recommender systems: Challenges, methods, and directions. ACM Trans Recomm Syst 1. Available: https://doi.org/10.1145/3568022.

13. Wolpert DH (1992) Stacked generalization. Neural networks 5: 241–259.

14. Koren Y (2009) The BellKor solution to the netflix grand prize. Available: https://api.semanticscholar.org/CorpusID:6114578.

Appendix

A. Colab Notebooks

- **Notebook 1:** EDA.ipynb
- **Notebook 2:** Preprocess_Feature_store.ipynb
- **Notebook 3:** Specific_Preprocess_and_Modelling

B. Figures

C. Self-Reflection

Over the course of this recommender systems project, I have significantly expanded my understanding of:

- **Scientific Knowledge**
  - *Collaborative vs. Content Approaches:* I experienced how item similarity alone (content-based) can be outperformed by user–item interaction signals (collaborative).
  - *Matrix Factorization:* Hands-on with classical MF approaches showed the importance of factor-based user–item representations. I also learned about their limitations (e.g., cold start, negative sampling).
  - *Advanced Deep Learning:* Neural Collaborative Filtering (NCF), NeuMF, and GraphSAGE highlighted the power of deep networks—particularly in capturing non-linear interactions and adjacency-based patterns. I saw how performance can spike with dedicated architecture searching.
  - *Graph Methods:* Representing recommendations as a bipartite graph allowed us to leverage GraphSAGE for user–item node embeddings, offering high accuracy.
- **Technical Skills**
  - *Data Pipelines:* Building a robust, modular pipeline that can handle large data, from EDA to final merges with advanced features.
  - *Software Stacks:* Familiarity with frameworks such as fastFM, PyTorch Geometric, and Keras Tuner for hyperparameter searching.
  - *Performance Tuning:* Realizing that random splits vs. time-based splits can drastically affect measured ROC AUC. I also tested various negative-sampling strategies and discovered how crucial sampling is to final model performance.
- **Key Takeaways & Next Steps**
  - Complex models benefit from large, well-engineered feature spaces—but I must watch for overfitting.

  - Practical constraints (hardware, memory) limit hyperparameter searches; a systematic approach or Bayesian optimization might further improve results.
  - Explaining recommendations to end users is a growing area. Black-box neural methods demand interpretability solutions.

- **Feedback & Personal Growth**
  - I valued the chance to experiment with a real dataset, bridging theory from class with practical Kaggle-style competition tasks.
  - Deep recommender systems (NCF, NeuMF) were new territory for us but provided an excellent demonstration of how advanced methods often outperform simpler heuristics—yet require greater computational resources and thoughtful hyperparameter tuning.
  - The final step of building a pipeline that merges training stats (like skip ratios) into the test data gave us a sense of how real industry pipelines might function (though scaled up with stream-processing or near real-time computations).

In summary, the project honed my recommender system knowledge and taught us hands-on engineering skills in data preprocessing, feature store creation, advanced model training, and large-scale system design.

D. Additional Plots or Tables

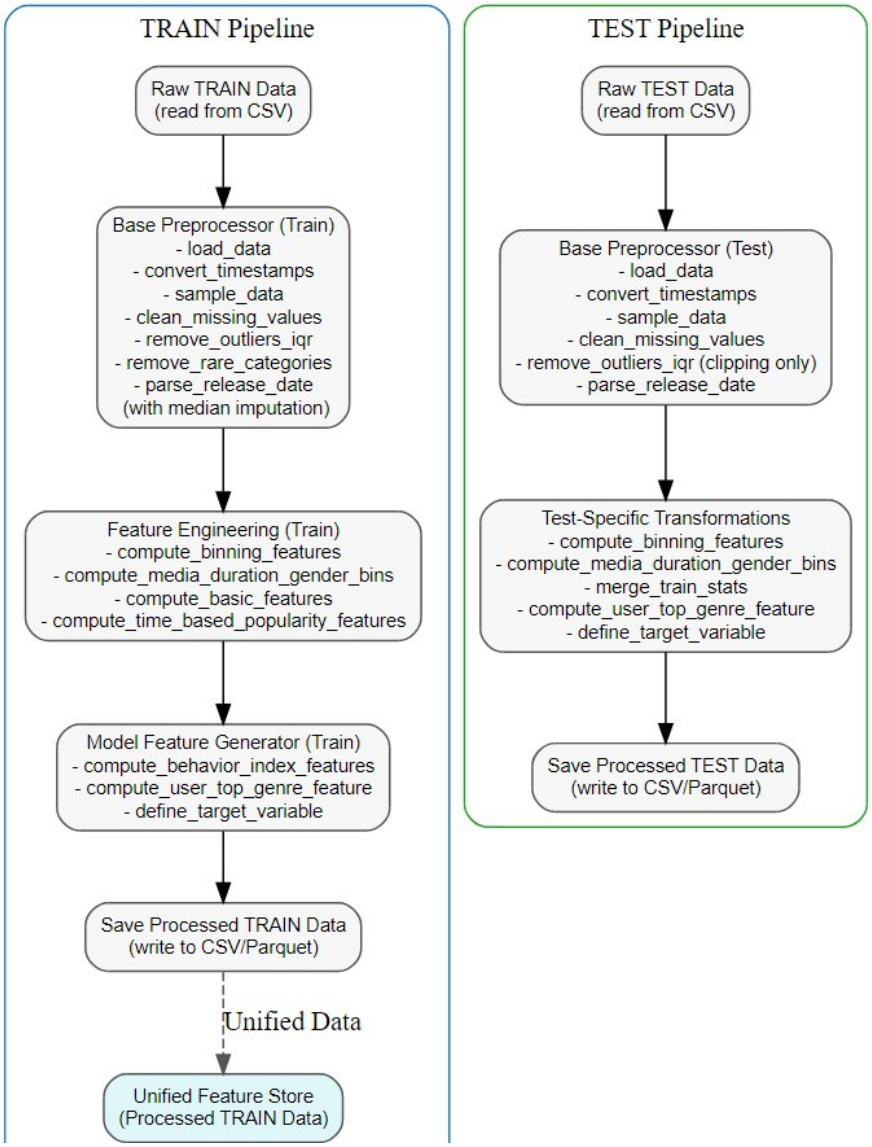

Figure 3: Pipeline

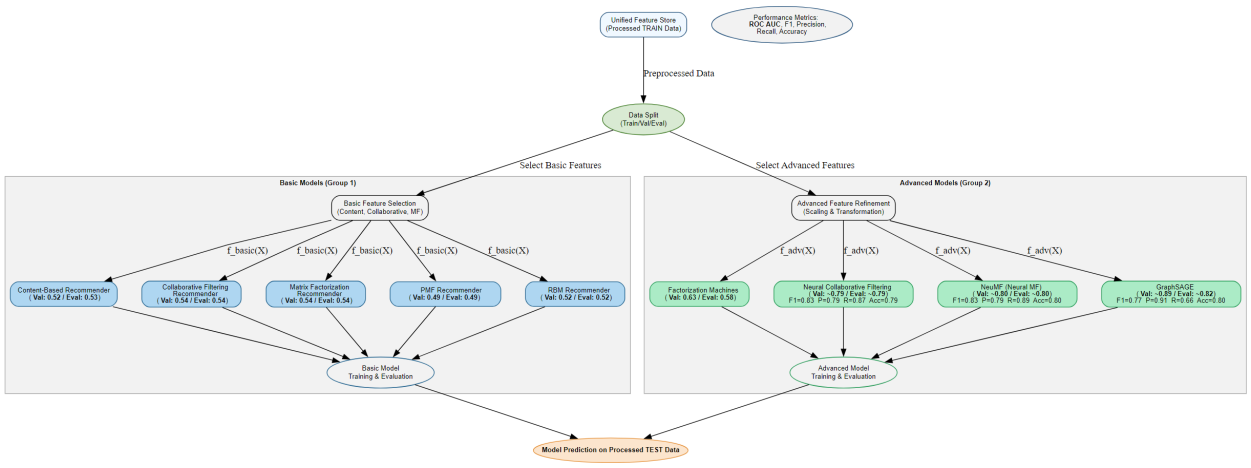

Figure 4: Modelling Pipline

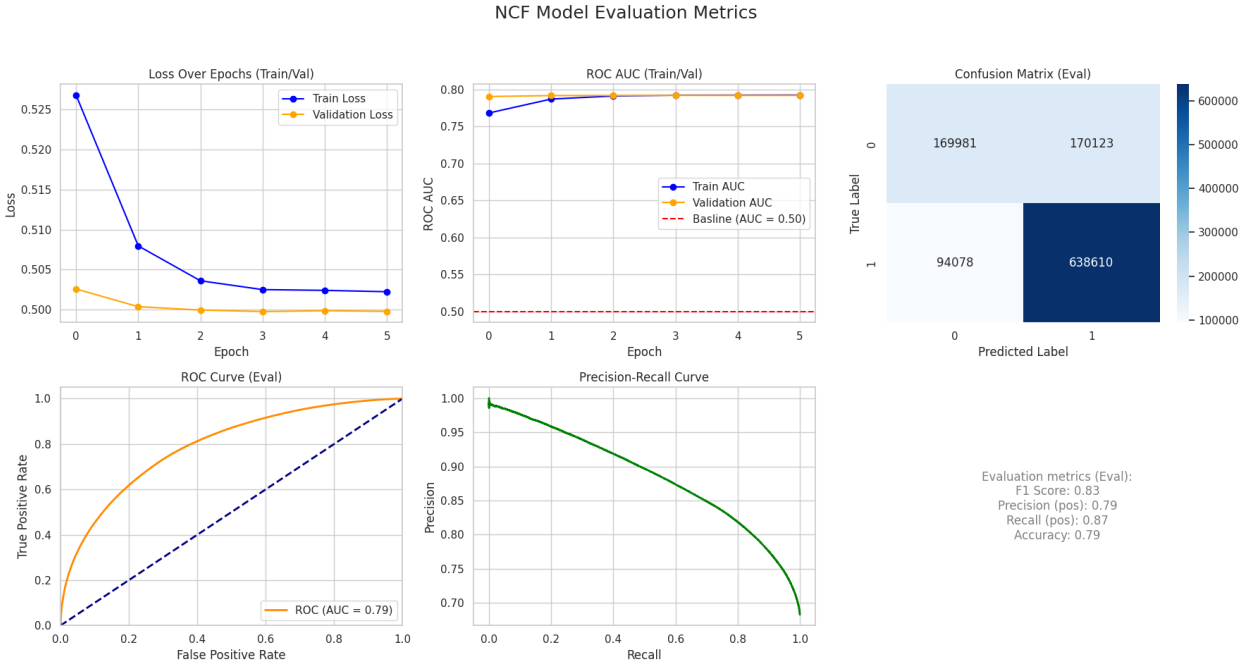

Figure 5: Model Evaluation Metrics for NCF

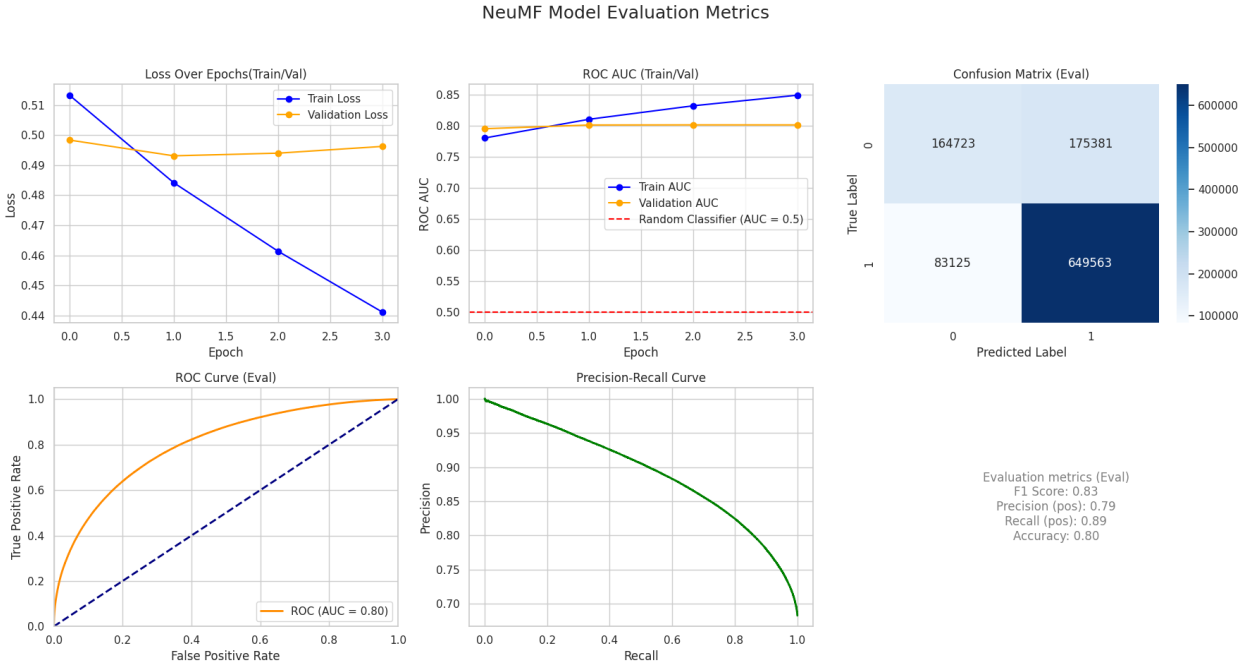

Figure 6: Model Evaluation Metrics for NeuMF

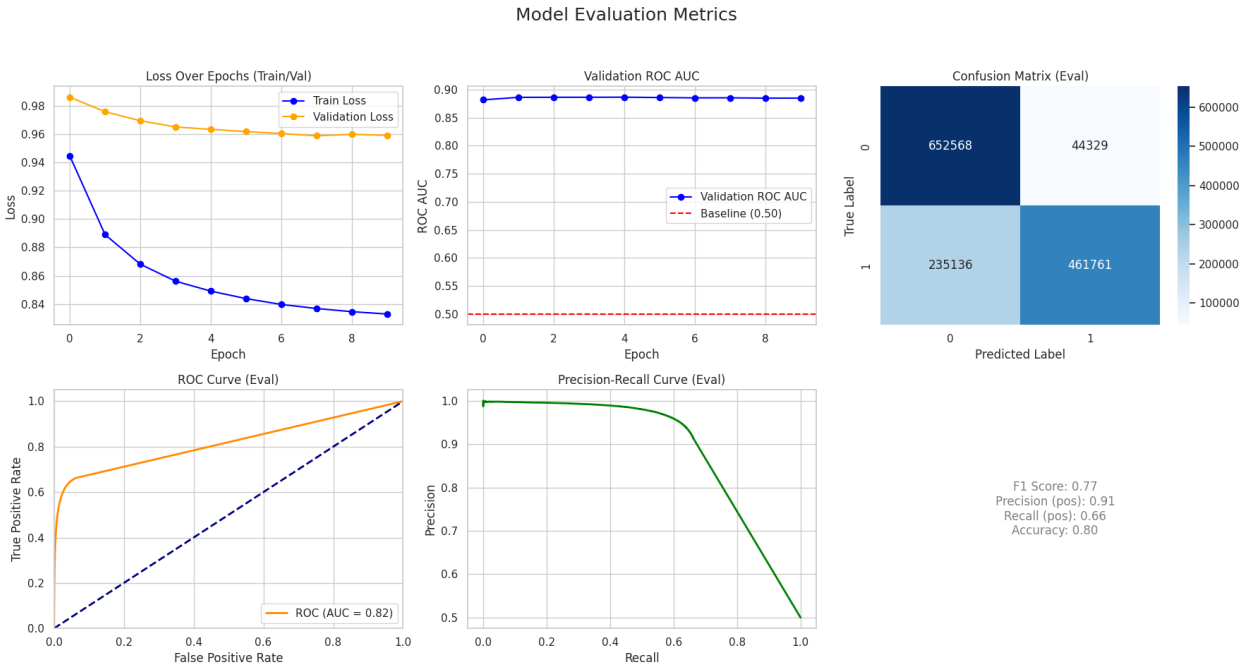

Figure 7: Model Evaluation Metrics for GraphSAGE