Trabajo Práctico 2
Tecnología Digital VI

Universidad Torcuato Di Tella

Ignacio Estrada
Agustín Méndez
Mateo Narbe

Fecha de entrega: 19 de Octubre, 2025

1. Análisis Exploratorio de Datos

Realizamos un análisis exhaustivo para entender el dataset y guiar la ingeniería de features. Mediante `graficos_utils.py` en el script principal cargamos el archivo `merged_data.csv` (resultado de mergear el dataset original con datos de Spotify API via `mergecsv.py`), y generamos visualizaciones clave.

- **Distribuciones numéricas:** Se puede notar que las canciones con mayor duración tienen una mayor probabilidad de ser salteadas. Esto es lógico, ya que una canción muy larga tiende a aburrir o cansar al usuario, subiendo su skip rate.

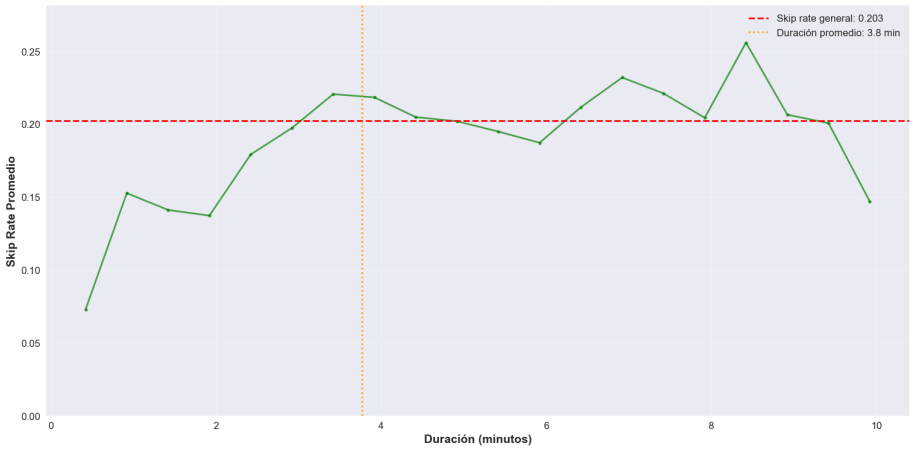

Figura 1: Skip rate según la duración de la canción

- **Patrones temporales:** Usando `timestamp`, plots de skips por hora/día muestran picos en evenings (18-22 hs) y algunos datos atípicos, mostrando que esta feature podría tener relevancia.

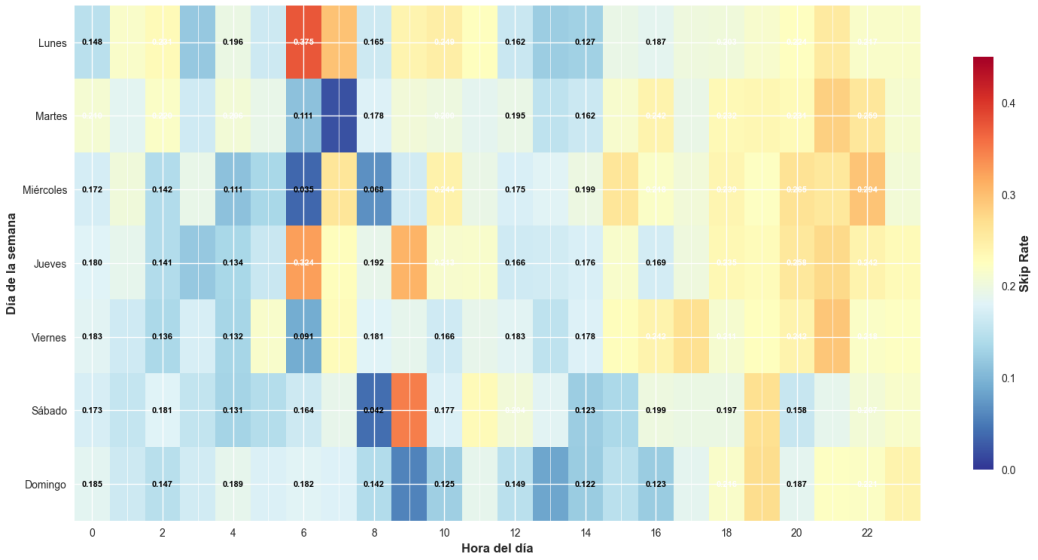

Figura 2: Skip rate por hora del día y día de la semana

- **Artistas y correlaciones:** Los artistas más escuchados logran mantener bajas tasas de skip. Esto indica que la popularidad del artista influencia fuertemente la probabilidad de que una canción se saltee.

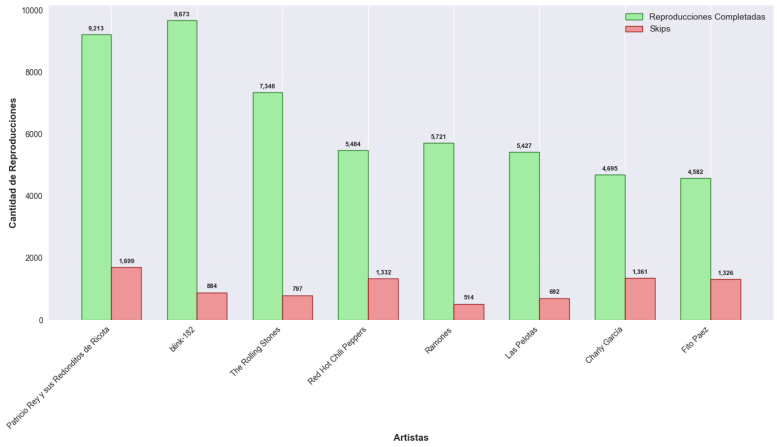

Figura 3: Cantidad de reproducciones y skips de los artistas más populares

2. Conjunto de Validación

Inicialmente usamos KFold (n=3), pero notamos discrepancias: alto AUC en CV (0.85+) vs. bajo en leaderboard público (0.6-0.7), indicando overfitting o leakage temporal. Esto se debía a la dependencia temporal de los datos; no podemos utilizar datos futuros para predecir los pasados. Con lo cual, decidimos utilizar un enfoque de split temporal que consiste en utilizar observaciones anteriores al año 2024 para train y observaciones de 2024 para validación y testing (primer 90 % para valid y 10 % restante para testing). Si bien con esta distribución hay muchos más datos para train que para validación, la cantidad de observaciones disponible permitió conseguir métricas representativas, ya que la cantidad de observaciones en valid y test eran suficientes, y por sobre todo, eran las más cercanas al conjunto de test utilizado en kaggle, con lo cual priorizamos que las métricas a optimizar se asemejen a los resultados de kaggle.

3. Variables y Features

Dataset Original + Datos de la API de Spotify

Incluye columnas como `timestamp`, `platform`, `master_metadata_album_artist_name`, `conn_country`, `shuffle`, `offline`, `incognito_mode`, etc. Usamos `obs_id` como ID, y derivamos `target` de `reason_end == "fwdbtn"`.

Además, de la carpeta **spotify_api_data** provista en la consigna, mergeando con el script `mergecsv.py` añadimos `duration_ms`, `explicit`, `release_date`, `popularity`, `track_number`, `show_publisher`, `show_total_episodes` para obtener más información original antes de empezar a transformar, sobre todo agregando variables de episodios, de las cuales carecíamos.

Finalmente, creímos que una predictora importante para este problema podría ser el **género de la canción**. Como no teníamos esta información, creamos algunos scripts para obtenerlo de una API de Spotify[1]. Si bien esta API es de artistas y no de canciones, consideramos sumarla. La misma nos da un listado de géneros por artista, de los cuales tomamos los primeros 3 y generamos las columnas `genre1-2-3` (`script_obtener_generos.py`).

Todas estas transformaciones se pueden encontrar en la carpeta `/transform_data_scripts`.

[1] https://developer.spotify.com/documentation/web-api/reference/get-an-artist

Transformaciones de las Variables Originales

En el archivo `database_utils.py` transformamos el dataset creando algunas features a partir de las originales. Las mismas se calculan sobre todo el dataset, y no generan leakage dado que son transformaciones independientes por observación. Salvo `operative_system` y `user_last_song_different`, las cuales se aclararán por qué no generan leakage en su respectivo item.

- **Fechas:** Dividimos las fechas `ts` y `offline_timestamp`, creando variables como `year`, `hour`, `fin_de_semana`, entre otras, para capturar patrones temporales.
- **Diferenciación de tipo:** Para facilitar la diferenciación entre track y podcast creamos las booleanas `is_track` (nombre del track no nulo) y `is_podcast` (`episode_name` no nulo). Además, dividimos `duration_ms` para cada tipo utilizando como máscara las booleanas anteriores.
- **Plataforma:** Utilizamos `platform`, para agrupar `operative_system` (ios, windows, etc.), más representativos que el nombre completo de la plataforma. Además, generamos `is_mobile` para capturar patrones según si era dispositivo celular u otro dispositivo. Debido a la cantidad de datos que teníamos en el train set, asumimos que ninguna plataforma aparecería solamente en el validatoin set y por ende no se generaría leakage.
- **Diferencia con la anterior canción:** Creamos `user_last_song_different` booleana, que verifica si la canción anterior escuchada por el usuario tiene el mismo género o es diferente, para captar saltos de género entre canciones escuchadas. Al dividir el conjunto de datos por temporalidad, no generamos leakage al aplicarlo sobre todo el dataset, pues utilizamos información del pasado, y los conjuntos de validación y test se encuentran más adelante temporalmente que todas las observaciones del train set.

Counting de las Variables Originales

Usando `createNewCountingFeatures`, calculamos **conteos históricos por usuario** de algunas variables sobre todo el dataset (sin filtrado porque train set ocurre antes de valid y test), ordenando temporalmente para evitar leakage. Los cuantiles de los bins (si aplica) se calculan utilizando únicamente datos de train para no filtrar información de los conjuntos de valid y test.

- **Bins por usuario:** Para cada usuario, calculamos la cantidad de observaciones para cada bin. Esta técnica se aplicó en `hour_bin_i`, `duration_bin_i`, `popularity_bin_i` y `time_diff_bin_i`, todas para cada bin i. Por ejemplo, para la hora, calculamos la cantidad de canciones escuchadas hasta el momento de cada observación (sin mirar el futuro) para cada hora (24 bins).
- **Conteos por día:** Para cada usuario y cada día calendario, se calcularon las variables `user_track_listens_today_count`, `user_artist_listens_today_count` y `songs_count_in_day`, que representan la cantidad acumulada de la feature correspondiente antes de la observación actual dentro de ese mismo día *(idea: si escuchaste muchas veces una canción en un día, probablemente vas a saltearla)*
- **Información por artista/track/género:** Similar a la anterior pero histórico, generamos `artist_play_count`, `genre1_play_count` y `user_track_listens_count` a partir del conteo por usuario de `master_metadata_album_artist_name`, `genre1` y `spotify_track_uri`. Además, creamos `user_time_since_last_same` que representa la diferencia temporal con la última vez que se escuchó la canción/artista de

la observación actual *(idea: un usuario saltea una canción (o artista) cuando ya la escuchó recientemente).*

- **Agrupación por sesión:** Agrupamos sesiones del usuario por temporalidad y generamos `user_session_len_so_far`, que muestra la posición temporal de la canción dentro de la sesión (primera escuchada de la sesión = 0, segunda = 1, etc.).

Bin Counting (Skip Rates)

Creamos nuevas variables de skip rate a partir de distintas features. Su generación se realizó en `createNewSetFeatures` únicamente para train, y luego se tomó el último registro de cada usuario para cada categoría y se extendió a los datos de valid y test mediante `applyHistoricalFeaturesToSet`, a fines de utilizar únicamente datos de train para generar estos valores y no filtrar información. Los missings se computaron con medias globales. Todas estas features se calcularon en train, siendo agrupadas por categorías apropiadas para cada una (usuario, temporalidad, etc.).

Clustering

En `simple_clustering`, también de `database_utils.py`, aplicamos KMeans (n=3) a features numéricas (e.g., duration, popularity) post-imputación. Entrenado solo en train, predicho en val/test para agregar `cluster` como feature categórica, capturando patrones latentes (e.g., clusters de tracks cortos/populares).

Cada feature se creó para capturar interacciones históricas y patrones sin leakage: computadas acumulativamente en train, mapeadas a sets futuros.

4. Modelos, Búsqueda de Hiperparámetros y Feature Selection

Elegimos **XGBoost** por su eficiencia en datos tabulares, manejo de categóricos/missings, regularización integrada y paralelismo, superior a RandomForest o Logistic. Lo implementamos en `base_xgboost.py`.

Para la selección de variables (`backward_feature_selection_topN`) se aplicó una estrategia de **backward selection**. El procedimiento comienza entrenando un modelo con todas las variables disponibles y, en cada iteración, elimina la variable menos importante de acuerdo a la métrica de ganancia (gain) reportada por XGBoost.

Durante esta búsqueda se utilizaron hiperparámetros fijos y de entrenamiento rápido, ya que el objetivo principal no era maximizar la performance final del modelo, sino identificar un subconjunto informativo y estable de features que optimice la relación entre capacidad predictiva y complejidad.

Búsqueda de Hiperparámetros

Optamos por **Bayesian** (Hyperopt) en búsqueda de eficiencia, maximizando AUC en validación temporal, ya que GridSearch y RandomSearch eran muy costosos.

Para la búsqueda de hiperparámetros se implementó una estrategia en dos etapas, con el objetivo de reducir el espacio de búsqueda y optimizar progresivamente. En una primera etapa, se realizó una búsqueda exploratoria amplia sobre los hiperparámetros estructurales y de regularización más relevantes para la complejidad del modelo —`max_depth`, `gamma`, `reg_lambda`, `reg_alpha`, `subsample`, `colsample_bytree` y `colsample_bylevel`— manteniendo fijos o con rangos reducidos parámetros como `learning_rate`, `n_estimators`

y `min_child_weight`. Esta etapa permite centrarse en la estructura principal para luego ajustar las iteraciones y el aprendizaje con mayor detalle.

En la segunda etapa, se utilizaron los mejores valores obtenidos en la búsqueda inicial como fijos (o casi fijos) y se amplió el rango de búsqueda de `learning_rate`, `n_estimators` y `min_child_weight`. Así, permitimos una mejor exploración más precisa en estos hiperparámetros para encontrar la relación óptima entre aprendizaje e iteraciones (junto con `min_child_weight` que se ve afectada por estos hiperparámetros) utilizando como base el resto de parámetros ya optimizados.

Esta estrategia escalonada permitió reducir el espacio de búsqueda y los tiempos de forma eficiente, favoreciendo una convergencia más rápida hacia combinaciones bien balanceadas entre capacidad predictiva, regularización y velocidad de entrenamiento.

5. Atributos Importantes

Post-entrenamiento, extraímos importancias del modelo. A continuación mostramos las más relevantes:

- **shuffle** y **username** fueron sin duda los más significativos en todos los modelos probados. Sobre todo en la solución final, donde fueron parte del puesto 1 y 2 respectivamente en el listado de importancias. Esto nos lleva a pensar que el modo `shuffle` tiene una alta tendencia a skips y, por otro lado, al tener los mismos usuarios en el train y test set, la predictora `username` también fue fundamental. Probablemente teniendo distintos usuarios entre conjuntos su peso no hubiese sido el mismo.
- **operative_system** e **ip_addr** también lo fueron (puestos 3 y 4), con lo cual notamos que el dispositivo utilizado por el usuario tiene alta correlación con la cantidad de skips.
- **hour_bin** tuvo alta relevancia, sobre todo **hour_bin_9**, con lo cual, tal como vimos en la exploración de la primera sección, existe una fuerte correlación entre el horario de escucha y la cantidad de skips, con un pico significativo a las 9hs.
- **user_time_since_last_play** y **time_diff_bin** también entraron en el top, lo cual nos lleva a pensar que el tiempo entre cada canción tiene gran poder predictivo.

Además, otras variables que tuvieron gran participación y les continuaron en el orden de importancias fueron **user_session_len_so_far**, **duration_bin**, **conn_country**, **master_metadata_album_artist_name**, **genre1** y **songs_count_in_day**, entre otros.

Como podemos observar, si bien variables originales como shuffle y username predominaron, las estrategias de **conteo y agrupamiento** tuvieron gran influencia para que el modelo detecte mejor los patrones y realice predicciones más precisas.

6. Conclusión

A lo largo del trabajo pudimos desarrollar un sistema bastante completo para predecir skips en Spotify, combinando análisis de datos, creación de nuevas variables y un modelo que aprovecha bien la información disponible.

El análisis exploratorio nos ayudó a entender mejor cómo se comportan los usuarios y a diseñar features más relevantes, especialmente las relacionadas con el historial de escucha y los hábitos de cada usuario. También notamos la importancia del ajuste de

hiperparámetros para el funcionamiento correcto del modelo, ya gran parte de la complejidad del trabajo fue encontrar una forma de optimizarlos para ajustar el modelo lo mejor posible.

Con todo esto, alcanzamos un AUC cercano a 0.808, lo cual fue un buen resultado considerando la dificultad del problema. Más allá del número, lo importante fue entender qué variables realmente aportaban valor y cómo los patrones de escucha cambian según el contexto del usuario.