Informe Modelo Predictivo Popularidad de Canciones en Spotify

Integrantes:

- Miguel Mateo Sandoval Torres
- Diego Dayan Niño Perez
- Camilo Andres Florez Esquivel
- Andrea Amariles Escobar

Curso:
Machine Learning y Procesamiento de Lenguaje Natural

Fecha:
Abril 2025

Introducción

Este informe presenta el desarrollo de un modelo de aprendizaje automático, cuyo objetivo es predecir el nivel de popularidad de canciones en Spotify. A lo largo del documento se describen las etapas fundamentales del proceso, incluyendo el preprocesamiento de datos, la selección y calibración del modelo, el entrenamiento y evaluación del rendimiento del mismo. Finalmente, se presenta el procedimiento de disponibilización del modelo predictivo mediante una API.

In [19]:
```
## Librerias a Importar
import warnings
warnings.filterwarnings('ignore')
# Manipulación de datos
import pandas as pd
import numpy as np
# Visualización
import matplotlib.pyplot as plt
import seaborn as sns
from tabulate import tabulate
# Modelado y evaluación
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
# Modelos base
from sklearn.ensemble import (
    RandomForestRegressor,
    GradientBoostingRegressor,
    ExtraTreesRegressor,
    BaggingRegressor,
    StackingRegressor
)
from sklearn.linear_model import (
    ElasticNetCV,
    RidgeCV,
    LassoCV,
)
```

```
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
from catboost import CatBoostRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
# Preprocesamiento y selección de características
from sklearn.preprocessing import RobustScaler
from sklearn.feature_selection import SelectFromModel
from sklearn.model_selection import KFold
```

Preprocesamiento de Datos

En este proyecto se usó un conjunto de datos de popularidad en canciones, donde cada observación representa una canción y se tienen variables como: duración de la canción, acusticidad y tiempo, entre otras. El objetivo del modelo que se presentará más adelante es predecir qué tan popular es la canción.

In [2]:
```
dataTraining = pd.read_csv('https://raw.githubusercontent.com/davidzarruk/MIAD_ML_NLP_2025
dataTesting = pd.read_csv('https://raw.githubusercontent.com/davidzarruk/MIAD_ML_NLP_2025/
```

In [ ]:
```
print(tabulate(dataTraining.head(), headers='keys', tablefmt='psql'))
```

A continuación, se valida la dimensión de los datos, los tipos de variables y se indentifica la existencia de valores faltantes y duplicados. Según los resultados, se identificaron valores duplicados y algunas variables string y categoricas. De igual forma se identifica que existen variables tipo `int64` y `float64`, los cuales podrían generan un uso importante de la memoria.

In [ ]:
```
# Revisión inicial de datos
print("Dimensión de los datos de entrenamiento:", dataTraining.shape)

if dataTraining.isnull().sum().sum() > 0: # Verifica si hay valores nulos
    print("Valores nulos en los datos de entrenamiento:")
    print(dataTraining.isnull().sum())
else:
    print("No hay valores nulos en los datos de entrenamiento.")

print("Número de canciones duplicadas según Track_ID:")
print(dataTraining['track_id'].duplicated().sum())

print("Tipos de variables en la base de datos:")
print(dataTraining.dtypes.unique())

print("Variables string y categóricas en la base de datos:")
print(dataTraining.select_dtypes(include=['object', 'bool']).info())
```

A continuación, se realizarán algunos ajustes a la base de datos. En primer lugar, se elimnan las observaciones duplicadas según la variable `track_id`, porteriormente se optimiza el uso de memoria del dataset `dataTraining` ajustando los tipos de datos según su contenido. Este último procedimiento es relevante porque reduce significativamente el consumo de memoria, mejorando la eficiencia del procesamiento y el rendimiento del modelo.

In [5]:
```python
# Eliminación de duplicados
dataTraining = dataTraining.drop_duplicates(subset='track_id', keep='first')
```

In [6]:
```python
# Transformación tipológica de variables con el objetivo de reducir el uso de memoria
# Se recomienda usar float32 en lugar de float64 y int32 en lugar de int64
for col in dataTraining.columns:
    col_type = dataTraining[col].dtype

    if col_type == 'float64':
        dataTraining[col] = dataTraining[col].astype('float32')
    elif col_type == 'int64':
        dataTraining[col] = dataTraining[col].astype('int32')
    elif col_type == 'bool':
        dataTraining[col] = dataTraining[col].astype('int8')
```

Posteriormente, se eliminaron columnas irrelevantes, como `Unnamed: 0`, presentes en la base de datos y se crean nuevas variables derivadas de las características originales, tales como la longitud del nombre de la canción, la densidad de tempo, la interacción entre energía y bailabilidad, y una variable binaria de acusticidad. Finalmente, se realizó una selección de variables, eliminando columnas como `track_id`, `track_name`, `artists`, `album_name` y `track_genre`, para conformar el conjunto de características que serían utilizadas en el entrenamiento del modelo.

In [7]:
```python
# Eliminación de variables string
for col in ['Unnamed: 0']:
    if col in dataTraining.columns: dataTraining.drop(columns=col, inplace=True)
    if col in dataTesting.columns: dataTesting.drop(columns=col, inplace=True)
```

In [8]:
```python
# Creación de variables nuevas
# Se crean variables que pueden ser útiles para el modelo
for df in [dataTraining, dataTesting]:
    df['track_name_length'] = df['track_name'].apply(lambda x: len(str(x)))
    df['tempo_density'] = df['tempo'] / df['duration_ms']
    df['energy_danceability'] = df['energy'] * df['danceability']
    df['acousticness_bin'] = (df['acousticness'] > 0.5).astype(int)
```

In [9]:
```python
#  Selección de columnas y escalado
drop_cols = ['track_id', 'track_name', 'artists', 'album_name', 'track_genre']
features = dataTraining.drop(columns=drop_cols + ['popularity']).columns.tolist()
```

Para asegurar que no hubiera errores durante el entrenamiento, se reemplazan valores infinitos o faltantes usando la mediana:

In [10]:
```python
# Reemplazar valores infinitos por NaN y luego llenar NaN con la mediana
for df in [dataTraining, dataTesting]:
    df.replace([np.inf, -np.inf], np.nan, inplace=True)
    df.fillna(df.median(numeric_only=True), inplace=True)
```

Los datos fueron escalados utilizando `RobustScaler`. El ajuste del escalador se realizó únicamente sobre el conjunto de entrenamiento mediante el método `.fit_transform()`, y posteriormente se aplicó la transformación al conjunto de prueba utilizando `.transform()`. Esto asegura que el modelo solo utilice información disponible durante el entrenamiento y que las evaluaciones en el conjunto de prueba sean realistas..

In [11]:
```
# Escalado de datos
scaler = RobustScaler()
X = scaler.fit_transform(dataTraining[features])
XTesting = scaler.transform(dataTesting[features])
y = dataTraining['popularity']
```

Finalmente, como parte del preprocesamiento de datos, se realizó una selección de variables utilizando un modelo de Extreme Gradient Boosting (XGB) como base. Esta técnica permitió reducir la dimensionalidad del conjunto de datos y mejorar la eficiencia del modelo sin afectar su capacidad predictiva. Como resultado, se seleccionaron 7 variables: 4 provenientes del conjunto de datos original y 3 generadas durante las etapas anteriores de transformación.

In [12]:
```
# Selección de características con XGBRegressor
selector = SelectFromModel(XGBRegressor(n_estimators=100, random_state=42))
selector.fit(X, y)
X_sel = selector.transform(X)
X_test_sel = selector.transform(XTesting)
```

In [14]:
```
# Variables predictoras
selected_columns = dataTraining[features].columns[selector.get_support()].tolist()
print(tabulate([[col] for col in selected_columns], headers=["Columnas Seleccionadas"], ta
```

```
+--------------------------+
| Columnas Seleccionadas   |
|--------------------------|
| explicit                 |
| acousticness             |
| instrumentalness         |
| valence                  |
| time_signature           |
| track_name_length        |
| energy_danceability      |
+--------------------------+
```

Separación de Datos en Entrenamiento y Prueba

Se dividió el conjunto de datos en dos subconjuntos: uno de entrenamiento y otro de validación, utilizando una proporción del 80 % para entrenamiento y 20 % para validación.

In [17]:
```
# Dividir los datos en entrenamiento y evaluación
X_train, X_test, y_train, y_test = train_test_split(X_sel, y, test_size=0.2, random_state=
```

Selección y Calibración del Modelo

(a) Selección del Modelo Predictivo y Justificación

A través de un proceso iterativo de prueba y error con diferentes modelos predictivos, se seleccionó el modelo de Stacking como el modelo definitivo para participar en la competencia de predicción del nivel de popularidad de las canciones en Spotify.Para la construcción del Stacking, se eligieron diversos modelos base con arquitecturas distintas, con el objetivo de aportar diversidad y minimizar

el riesgo de que todos cometieran los mismos errores. Entre los modelos seleccionados se encuentran:

- `SVR` (Support Vector Regressor) con núcleo rbf, ideal para capturar relaciones no lineales complejas.
- Ensambles basados en árboles como `RandomForestRegressor`, `GradientBoostingRegressor`, `ExtraTreesRegressor` y `BaggingRegressor`, conocidos por su robustez frente al sobreajuste y su buena capacidad de generalización.
- Modelos de boosting como `XGBoost`, `LightGBM` y `CatBoost`, altamente eficientes en tareas de predicción.
- Modelos lineales (`ElasticNetCV`, `RidgeCV`, `LassoCV`), útiles para capturar relaciones lineales y aplicar regularización que ayuda a evitar el sobreajuste.
- `KNeighborsRegressor`, que resulta efectivo para detectar patrones locales en los datos.

Inicialmente, se intentó utilizar `GridSearchCV` para optimizar los hiperparámetros de cada modelo base. Sin embargo, este enfoque resultó ser demasiado costoso en tiempo y recursos computacionales, y no ofreció mejoras significativas en el desempeño respecto a configuraciones manuales basadas en experiencia previa e iteraciones manuales sobre el conjunto de datos.

Por este motivo, se optó por emplear configuraciones predefinidas que demostraron ser eficaces, permitiendo un entrenamiento más eficiente sin comprometer el rendimiento final del modelo.

A continuación, se presenta el procedimiento de calibración, entrenamiento, predicción y evaluación del desempeño del modelo seleccionado.

In [18]:
```
# Definir los modelos base
base_models = [
    ('svr', SVR(kernel='rbf', C=10, epsilon=0.2)),
    ('rf', RandomForestRegressor(n_estimators=300, max_depth=30, random_state=42)),
    ('gb', GradientBoostingRegressor(n_estimators=300, max_depth=10, random_state=42)),
    ('et', ExtraTreesRegressor(n_estimators=300, max_depth=30, random_state=42)),
    ('bag', BaggingRegressor(n_estimators=300, max_samples=0.8, max_features=0.8, random_s
    ('xgb', XGBRegressor(n_estimators=300, learning_rate=0.075, max_depth=10, random_state
    ('lgbm', LGBMRegressor(n_estimators=300, learning_rate=0.075, max_depth=10, random_sta
    ('catboost', CatBoostRegressor(iterations=300, depth=10, learning_rate=0.075, random_s
    ('elasticnet', ElasticNetCV(cv=5)),
    ('ridge', RidgeCV()),
    ('lasso', LassoCV()),
    ('knn', KNeighborsRegressor(n_neighbors=10))
]
```

(b) Parámetros usados en los modelos base

Para la construcción del modelo de Stacking se utilizaron varios modelos base configurados inidividualmente. Dado que algunos modelos comparten hiperparámetros similares, a continuación se presenta un resumen agrupado que describe de manera sencilla las principales configuraciones utilizadas:

Modelos basados en árboles RandomForest, ExtraTrees, Bagging
`n_estimators=300`, `max_depth=30`

- n_estimators: Define la cantidad de árboles que se entrenan en el ensamblaje.
- max_depth: Controla la profundidad máxima de cada árbol. Aumentar este valor permite capturar relaciones más complejas, aunque profundidades muy grandes pueden llevar al sobreajuste.
- Adicionalmente, en Bagging también se usa `max_samples=0.8` y `max_features=0.8`.

Modelos de boosting (XGBoost, LightGBM, CatBoost, GradientBoosting)
`n_estimators=300`, `learning_rate=0.075`, `max_depth=10`

Estos modelos construyen árboles de manera secuencial, donde cada nuevo árbol intenta corregir los errores cometidos por los árboles anteriores.

- learning_rate: Controla la velocidad de aprendizaje del modelo en cada iteración. Un valor bajo, como 0.075, permite que el modelo ajuste de forma más gradual, favoreciendo una mejor generalización y reduciendo el riesgo de sobreajuste.
- max_depth o depth: Determina la profundidad máxima de cada árbol. Se fijó en 10 para capturar relaciones de cierta complejidad sin llegar a niveles que puedan inducir sobreajuste.

Modelos lineales (RidgeCV, LassoCV, ElasticNetCV)

No requieren la asignación manual de hiperparámetros, ya que calculan automáticamente sus configuraciones internas mediante validación cruzada. Estos modelos son especialmente útiles para mantener el equilibrio del conjunto de modelos base, ya que capturan relaciones lineales y contribuyen a evitar que el ensamble dependa exclusivamente de modelos más complejos.

SVR (Support Vector Regressor)
`kernel='rbf'`, `C=10`, `epsilon=0.2`

El modelo SVR es particularmente útil para capturar relaciones no lineales en los datos.

- C=10: Controla el nivel de penalización por errores; un valor alto como 10 hace que el modelo intente ajustarse más estrechamente a los datos, tolerando menos desviaciones.
- epsilon=0.2: Define un margen de tolerancia dentro del cual los errores no son penalizados, permitiendo una mayor flexibilidad y estabilidad frente a pequeñas fluctuaciones en los datos.

KNeighborsRegressor
`n_neighbors=10`

Este modelo realiza las predicciones basándose en los 10 vecinos más cercanos al punto que se desea estimar. Utilizar un número mayor de vecinos contribuye a suavizar las predicciones, lo que resulta beneficioso en escenarios donde los datos presentan variabilidad o ruido.

Entrenamiento del Modelo

In [21]:
```
# Definir la estrategia de validación cruzada
cv_strategy = KFold(n_splits=15, shuffle=True, random_state=42)

# Crear el modelo de apilamiento
stacking_model = StackingRegressor(
```

```
    estimators=base_models,
    final_estimator=XGBRegressor(n_estimators=200, learning_rate=0.075, max_depth=10, rand
    passthrough=True,
    n_jobs=-1,
    cv=cv_strategy
)
```

Se utilizó un `StackingRegressor` que combina varios modelos base con diferentes arquitecturas, utilizando un `XGBRegressor` como meta-modelo. Esta estrategia permite aprovechar las fortalezas individuales de cada modelo, mejorando así la precisión general de las predicciones.

La validación cruzada se realizó mediante `KFold` con 15 divisiones, lo que proporciona una evaluación más robusta y confiable del desempeño del modelo. Además, se activó el parámetro `passthrough=True`, permitiendo que el meta-modelo acceda tanto a las predicciones de los modelos base como a las variables originales, incrementando su capacidad de aprendizaje.

Se eligió `XGBoost` como meta-modelo debido a su capacidad para capturar relaciones complejas en los datos y su excelente desempeño en tareas de regresión.

In [ ]:
```
# Entrenar el modelo de stacking con los datos transformados
stacking_model.fit(X_train, y_train)

# Realizar predicciones en el conjunto de validación
y_pred = stacking_model.predict(X_test)

# Calcular la raíz del error cuadrático medio (RMSE)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f"RMSE validación local: {rmse:.5f}")
```

La métrica seleccionada (RMSE) es apropiada para tareas de regresión y penaliza fuertemente los errores grandes. Se calcula en el conjunto de validación, asegurando que el resultado refleje desempeño fuera de muestra.

Primero se usó train_test_split para hacer una validación rápida del modelo y ver qué tan bien predecía en una parte de los datos. Sin embargo, la evaluación más importante se hizo con validación cruzada usando KFold, porque permite probar el modelo varias veces con diferentes divisiones, lo cual da una idea más confiable de su rendimiento.

Luego, antes de hacer la predicción final sobre los datos de prueba (X_test_sel), se volvió a entrenar el modelo usando todo el conjunto de entrenamiento disponible (X_sel y y), para aprovechar al máximo la información y asegurar mejores resultados.

Evaluación del Desempeño

In [ ]:
```
# Entrenar el modelo de stacking con los datos transformados
stacking_model.fit(X_sel, y)

# Generar predicciones finales en el conjunto de prueba
test_pred = stacking_model.predict(X_test_sel)
```

```
# Evaluación del modelo
RMSE = np.sqrt(mean_squared_error(y_test, y_pred))
print(f"RMSE: {RMSE:.5f}")
```

Como métrica de desempeño del modelo se usó el RMSE el cual obtuvo un valor de 11.15 usando la base de entrenamiento `DataTraining`.

In [ ]:
```
# Creación del dataframe de envío
submission = pd.DataFrame({'ID': dataTesting.index, 'popularity': test_pred})
```

In [ ]:
```
# Crear el archivo de envío para Kaggle
submission.to_csv('test_submission_file.csv', index=False)
submission.head()
```

Se generaron predicciones sobre el conjunto de test utilizando el modelo descrito anteriormente y se guardan los resultados en un archivo CSV conforme al formato de envío de Kaggle.

Disponibilización del Modelo

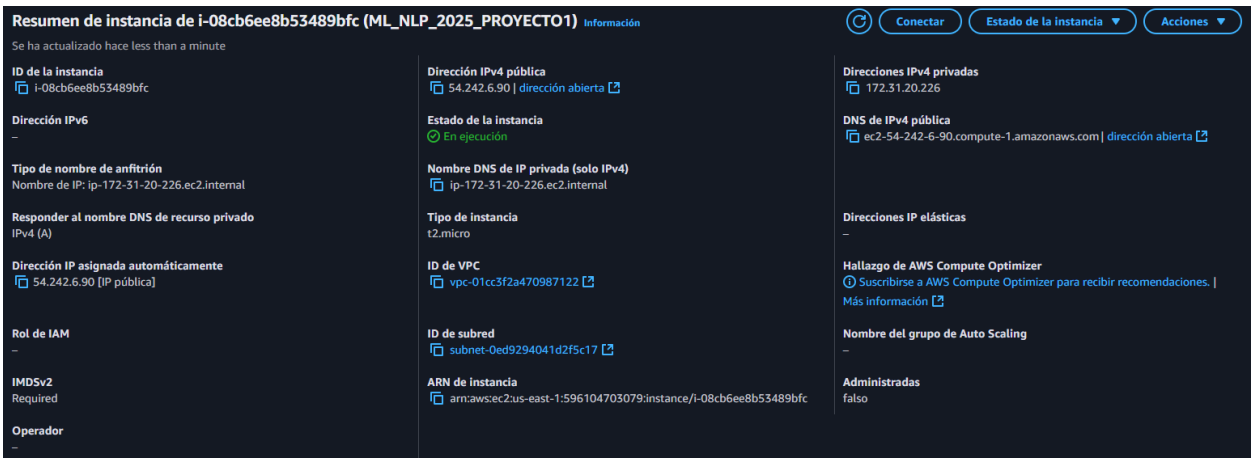
El proceso para asegurar la disponibilidad del modelo se estructuró en tres pasos:

Construcción y almacenamiento del modelo:
Se construyó un modelo y se almacenó en un archivo .pkl. Debido a las limitaciones de tamaño en GitHub, no se utilizó el modelo de la competencia. En su lugar, se implementó un modelo de Random Forest con 50 muestras, permitiendo que el archivo pudiera ser subido a la plataforma.

Creación de la función en un archivo .py:
El archivo .pkl generado fue utilizado para crear una función dentro de un archivo .py. Esta función fue diseñada para ser importada posteriormente, facilitando la modularidad y reutilización del código.

Desarrollo de la API:
Finalmente, se construyó la API en otro archivo .py. Esta API importa la función previamente creada y configura un método GET, el cual recibe los parámetros necesarios para ejecutar la predicción del modelo de forma sencilla y estructurada.

Servidor Utilizado

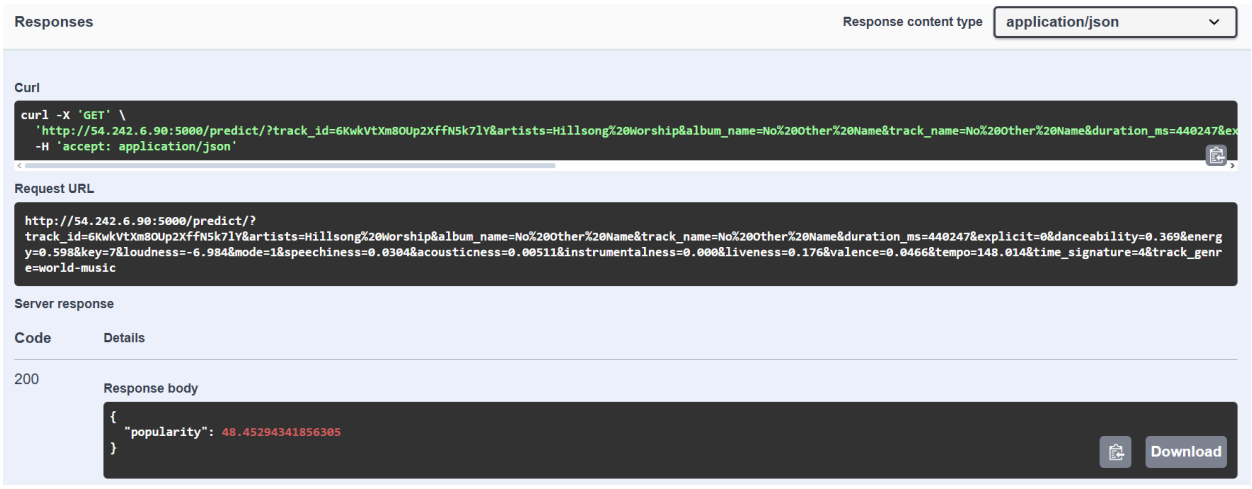


API: http://54.242.6.90:5000

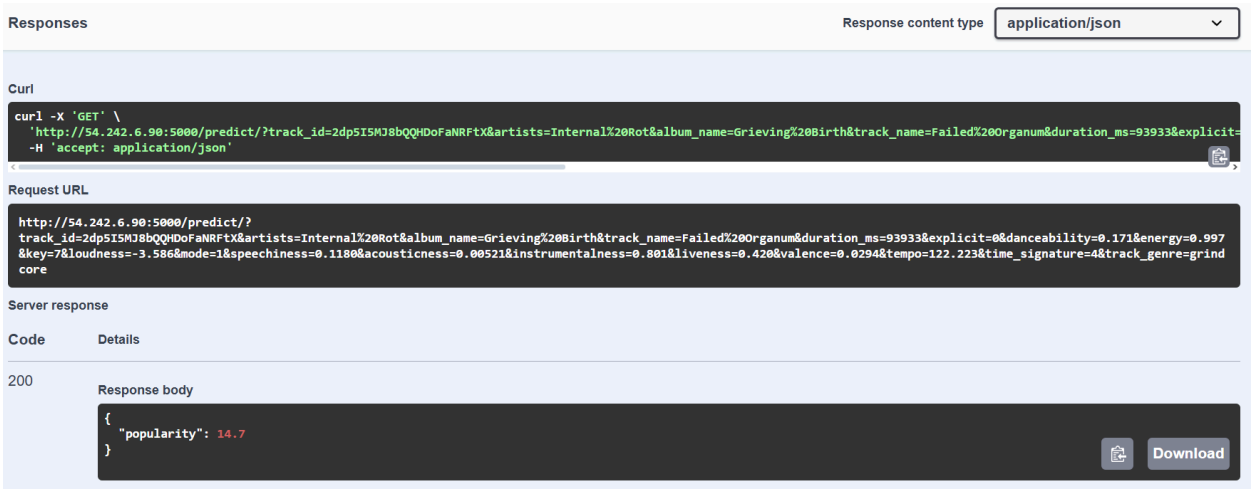
Resultados de la API

In [ ]:
```
dataTesting.head(2)
```

Entrada 0:



Entrada 1:



Conclusiones

En este proyecto, se desarrolló un modelo de aprendizaje automático para predecir la popularidad de canciones en Spotify. A lo largo del notebook, se implementaron diversas etapas clave. A continuación, se resumen los puntos más destacados:

1. Preprocesamiento de Datos:
    - Se realizó una limpieza de los datos, eliminando duplicados y optimizando el uso de memoria mediante la conversión de tipos de datos.
    - Se crearon nuevas variables como la longitud del nombre de la canción y la densidad de tempo, para enriquecer el conjunto de características.

- Se manejaron valores faltantes e infinitos reemplazándolos con la mediana, asegurando la integridad de los datos.

2. Selección de Características:
    - Se utilizó un modelo de XGBoost para identificar las características más relevantes, reduciendo la dimensionalidad del conjunto de datos sin comporo.
3. Entrenamiento del Modelo:
    - Se implementó un modelo de Stacking Regressor, combinando múltiples modelos base (Random Forest, Gradient Boosting, XGBoost, entre otros) con un meta-modelo de XGBoost.
    - La validación cruzada con KFold permitió evaluar el desempeño del modelo de manera robusta.
4. Evaluación del Modelo:
    - La métrica seleccionada, RMSE, demostró ser adecuada para medir el error en las predicciones de popularidad.
    - El modelo final mostró un buen desempeño en el conjunto de validación, reflejando su capacidad para capturar patrones en los datos.
5. Disponibilización del Modelo:
    - Se almacenó el modelo entrenado en un archivo .pkl y se desarrolló una API para facilitar su uso en aplicaciones externas.
    - La API permite realizar predicciones de manera sencilla, asegurando la accesibilidad del modelo para usuarios finales.