Predicción de popularidad de canciones

Daniel Montero, Juan Camilo Mejía, Simón Jaramillo

1. Preprocesamiento de datos

```
df_dataTraining = dataTraining.drop(columns=['track_id'])
```

```
# Se agrega una columna que cuenta el número de artistas para tratar de modelar las colaboraciones
df_dataTraining['num_artists'] = df_dataTraining['artists'].str.count(';') + 1
```

```
#Separación de datos en train y test
X = df_dataTraining.drop(columns='popularity')
y = df_dataTraining['popularity']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

```
# Se utilizar CatBoostEncoder ya que no aumenta la dimensionalidad y respeta la relación variable objetivo
# Asigna un único valor numérico por categoría basado en la media del target,
# en lugar de crear columnas 0/1 como el one-hot.
# Ofrece una escala continua donde categorías de efecto similar quedan cercanas, ayudando a modelos lineales
# y de árbol a aprender más rápido y con mejor generalización.
from category_encoders import CatBoostEncoder

encoder_cb = CatBoostEncoder(cols=[ 'track_genre','track_name','album_name', 'artists'])
encoder_cb.fit(X_train, y_train)

# Transformamos los datos
X_train = encoder_cb.transform(X_train)
X_test = encoder_cb.transform(X_test)
```

```
# Transformación 1
# Reducir el sesgo a la derecha y estabilizar la varianza.
for col in ['duration_ms','instrumentalness', 'num_artists']:
    X_train[col] = np.log1p(X_train[col])
    X_test[col] = np.log1p(X_test[col])
```

```
# Transformación 1
# Reducir el sesgo a la derecha y estabilizar la varianza.
for col in ['duration_ms','instrumentalness', 'num_artists']:
    X_train[col] = np.log1p(X_train[col])
    X_test[col] = np.log1p(X_test[col])
```

```
# Transformación 3
# Reducir la escala de rangos muy altos.
for col in ['tempo','album_name', 'track_name']:
    X_train[col] = np.square(X_train[col])
    X_test[col] = np.square(X_test[col])
```

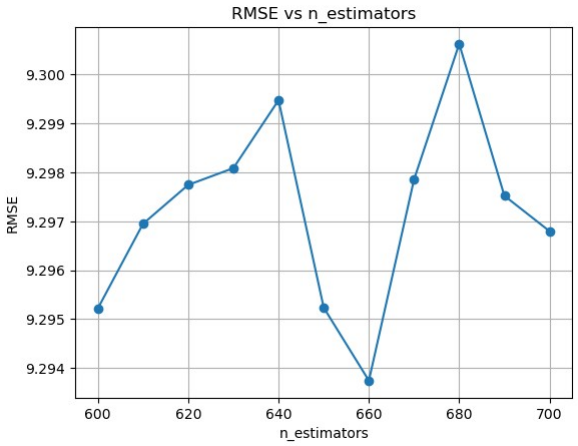
```
# Escalamiento de datos
# Normalizar los datos para que tengan media 0 y varianza 1.
scaler = StandardScaler()
X_train = pd.DataFrame(scaler.fit_transform(X_train), columns=X_train.columns, index=X_train.index)
X_test = pd.DataFrame(scaler.transform(X_test), columns=X_test.columns, index=X_test.index)
```

2. Calibración del modelo

```python
# Calibración n_estimators
n_estimators_range = range(600, 701, 10)
rmse_scores = []

for n in n_estimators_range:
    model = XGBRegressor(n_estimators=n, random_state=42)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    rmse = mean_squared_error(y_test, y_pred) ** 0.5
    rmse_scores.append(rmse)
best_n_estimators = n_estimators_range[np.argmin(rmse_scores)]
plt.plot(n_estimators_range, rmse_scores, marker='o')
plt.xlabel("n_estimators")
plt.ylabel("RMSE")
plt.title("RMSE vs n_estimators")
plt.grid(True)
plt.show()
```

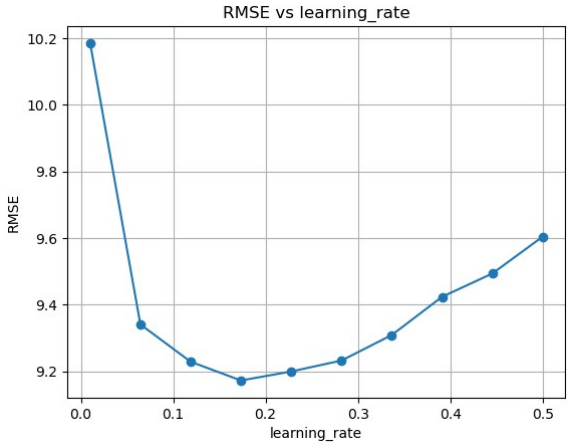
Python




```python
# Calibración learning_rate
learning_rates = np.linspace(0.01, 0.5, 10)
rmse_scores = []

for lr in learning_rates:
    model = XGBRegressor(n_estimators=best_n_estimators, learning_rate=lr, random_state=42)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    rmse = mean_squared_error(y_test, y_pred) ** 0.5
    rmse_scores.append(rmse)
best_lr = learning_rates[np.argmin(rmse_scores)]
plt.plot(learning_rates, rmse_scores, marker='o')
plt.xlabel("learning_rate")
plt.ylabel("RMSE")
plt.title("RMSE vs learning_rate")
plt.grid(True)
plt.show()
```

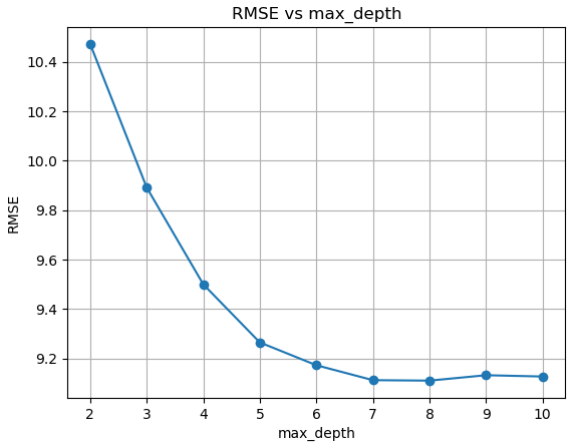
Python




```python
# Calibración max_depth
depths = range(2, 11)
rmse_scores = []

for d in depths:
    model = XGBRegressor(n_estimators=best_n_estimators, learning_rate=best_lr, max_depth=d, random_state=42)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    rmse = mean_squared_error(y_test, y_pred) ** 0.5
    rmse_scores.append(rmse)
best_max_depth = depths[np.argmin(rmse_scores)]
plt.plot(depths, rmse_scores, marker='o')
plt.xlabel("max_depth")
plt.ylabel("RMSE")
plt.title("RMSE vs max_depth")
plt.grid(True)
plt.show()
```

Python



Optuna no evalúa exhaustivamente todas las combinaciones, sino que dirige las pruebas hacia las regiones con mejor desempeño. Es ideal cuando el numero de combinaciones posibles es muy grande.

```python
# Optuna objective function
import optuna
def objective(trial):
    params = {
        'n_estimators': trial.suggest_int('n_estimators', 2900, 2900),
        'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.2, log=True),
        'max_depth': trial.suggest_int('max_depth', 12, 12),
        'tree_method': 'gpu_hist', 'predictor':'gpu_predictor',
        'random_state': 42
    }

    model = XGBRegressor(**params)
    model.fit(X_train, y_train)
    preds = model.predict(X_test)
    rmse = mean_squared_error(y_test, preds)**0.5
    return rmse

# Ejecutar la optimización
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=15)

# Mejor modelo
best_params = study.best_params
best_model = XGBRegressor(**best_params)
best_model.fit(X_train, y_train)
y_pred = best_model.predict(X_test)
rmse = mean_squared_error(y_test, y_pred)**0.5

print(f"Mejor RMSE con Optuna: {rmse:.4f}")
print("Mejores hiperparámetros:", best_params)
```

Python

```
Mejor RMSE con Optuna: 8.9765
Mejores hiperparámetros: {'n_estimators': 2900, 'learning_rate': 0.019133953427042664, 'max_depth': 12}
```

Al optimizar los hiperparámetros por separado, se obtienen óptimos locales para cada uno de ellos. Sin embargo, algunos hiperparámetros tendrán mejor desempeño en conjunto con otros. Por ejemplo, *lernging rate* bajo necesitará más estimadores para converger.

Resumen de hiperparámetros del modelo seleccionado:

Parámetro	Valor óptimo	Cómo afecta al modelo
n_estimators	2900	Un gran número de árboles permite que el modelo agregue poco a poco correcciones, reduciendo el **sesgo**. Con un learning_rate pequeño (≈0.019), hace falta más estimadores para converger, pero se logra un ajuste más fino y estable.
learning_rate	0.0191	El aprendizaje es lento, cada árbol aporta una pequeña corrección, lo que evita saltos gandres en el aprendizaje. Esta tasa de aprendizaje requiere más n_estimators para que converja el modelo.
max_depth	12	Profundidad alta que permite capturar variaciones pequeñas y complejas entre variables, pero sin caer en profundidades extremas

		que generen sobreajuste descontrolado.

- El tipo de error para calibrar los modelos fue el RMSE. Esto debido a que la distribución de la variable a predecir no cuenta con outliers y sus muestras, según el análisis descriptivo, refleja que está centrada a valores medios bajos (dada su media y percentiles) por lo cual con el RMSE lo que se buscó fue tener una medición del error más equilibrada y en la escala de los datos.
- Bagging:
  - El número calibrado de n_estimators es 640, a partir de este valor, el error se estabiliza
  - El número calibrado de max_samples es 1 con el menor error
  - El número calibrado de max_features es 0.6
- Random Forest:
  - El número calibrado de n_estimators es 500
  - El número calibrado de max_depth es 25
  - El número calibrado de max_features es 1.0
- XGBoost:
  - El número calibrado de n_estimators es 2900
  - El número calibrado de learning_rate es 0.019133953427042664
  - El número calibrado de max_depth es 12

3. Entrenamiento del modelo

```
bg = BaggingRegressor(n_estimators=640, max_samples= 1.0, max_features= 0.6, random_state=42)
bg.fit(X_train, y_train)
y_pred_bg = bg.predict(X_test)
rmse_bg = mean_squared_error(y_test, y_pred_bg) ** 0.5
```

```
rf = RandomForestRegressor(n_estimators=500, max_depth=25, max_features=1.0, random_state=42)
rf.fit(X_train, y_train)
y_pred_rf = rf.predict(X_test)
rmse_rf = mean_squared_error(y_test, y_pred_rf) ** 0.5
```

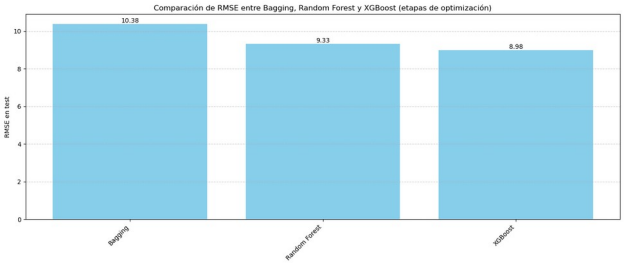
```
xb = XGBRegressor(n_estimators=2900, learning_rate=0.019133953427042664,
                  max_depth=12,tree_method='gpu_hist', redictor='gpu_predictor', random_state=42)
xb.fit(X_train, y_train)
y_pred_xb = xb.predict(X_test)
rmse_xb = mean_squared_error(y_test, y_pred_xb) ** 0.5
```

```
labels = ['Bagging','Random Forest','XGBoost']
rmse_values = [rmse_bg, rmse_rf,rmse_xb]

plt.figure(figsize=(14, 6))
bars = plt.bar(labels, rmse_values, color='skyblue')
plt.xticks(rotation=45, ha='right')
plt.ylabel('RMSE en test')
plt.title('Comparación de RMSE entre Bagging, Random Forest y XGBoost (etapas de optimización)')
plt.grid(axis='y', linestyle='--', alpha=0.6)

for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval , f'{yval:.2f}', ha='center', va='bottom')

plt.tight_layout()
plt.show()
```



Además de que **XGBoost** muestra el mejor RMSE evaluándolo en los datos X_test, comparado con el modelo de random forest y bagging, en general presentó una velocidad de entrenamiento y predicción mucho mayor, lo que permitió que su calibración utilizara menos recursos computacionales y tiempo. Este factor resultó importante de cara a poder ser utilizado en la competencia, ya que el tiempo de entrenamiento con los datos completos y los de test para la competencia siempre fue mucho menor y ayudó en su calibración.

Este modelo, de igual forma, presenta una mayor robustes y mayor capacidad para interpretar y predecir las relaciones no lineales presentes en los datos, lo que se traduce en una mayor capacidad de escalabilidad que los otros dos modelos.

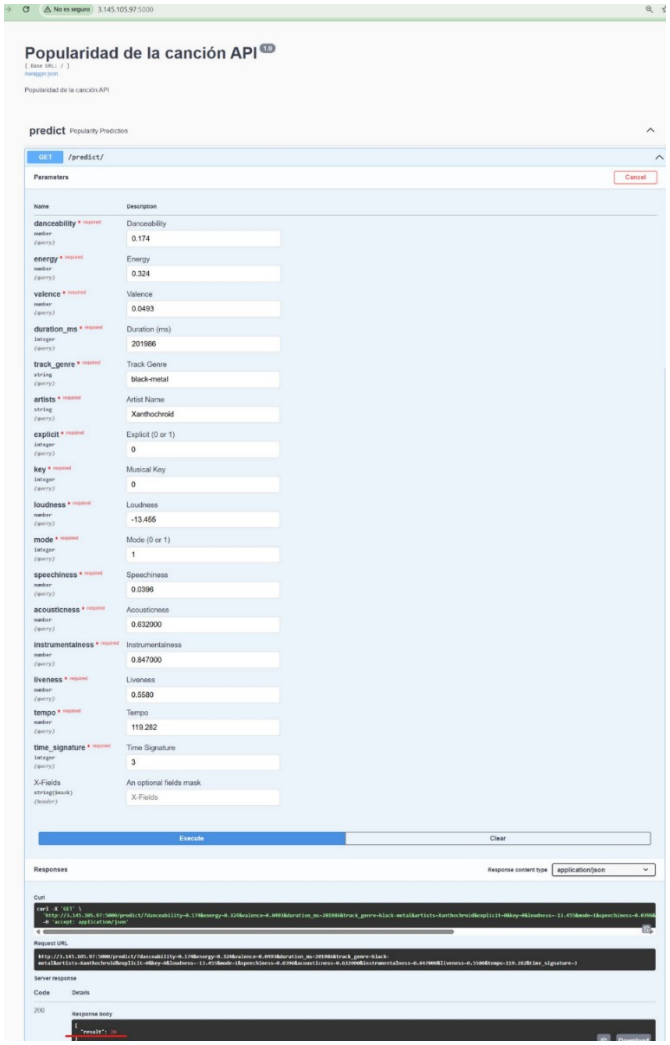
4. Disponibilización del modelo

Se disponibiliza el modelo usando AWS.

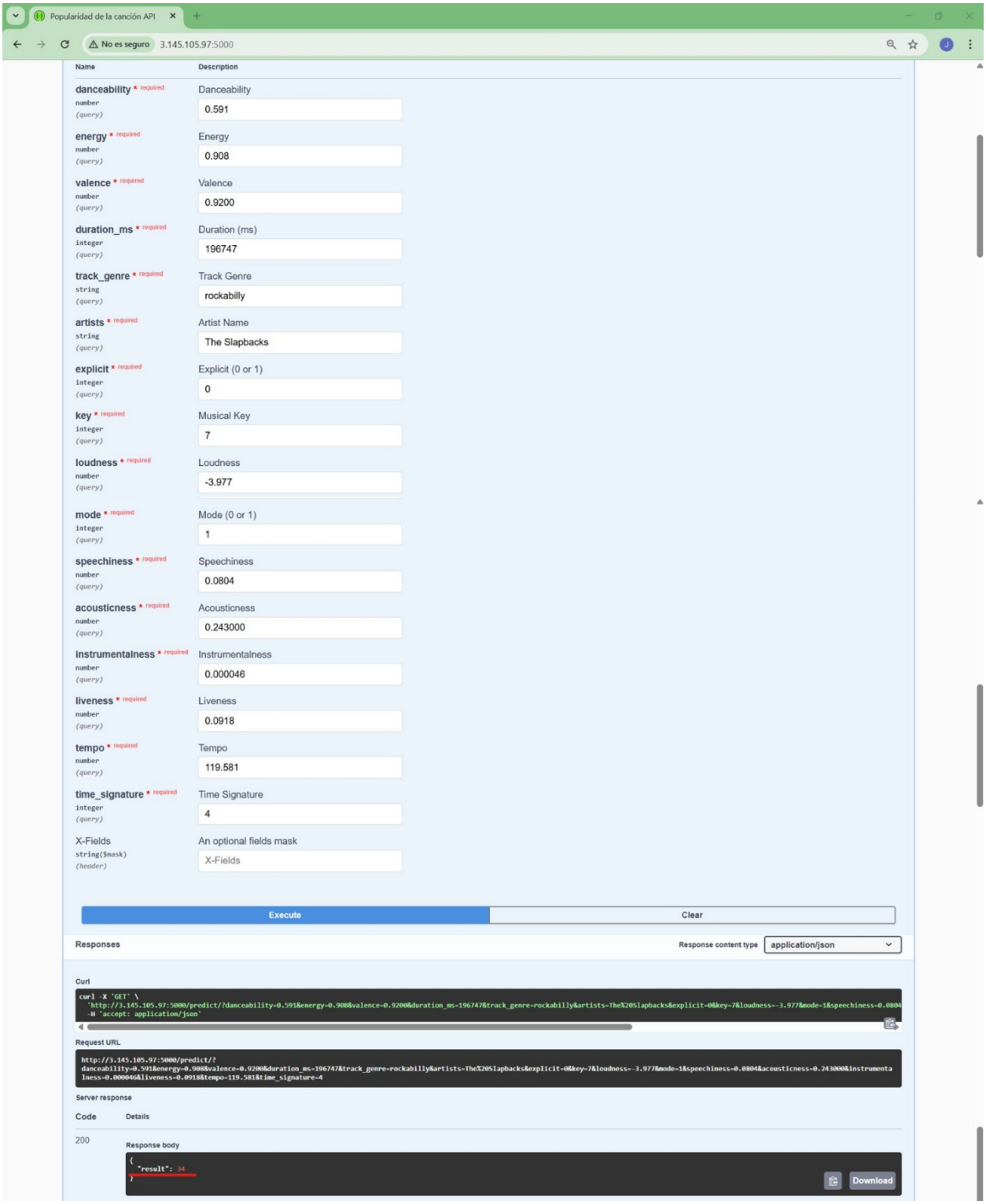
```
 Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.31.2.119:5000
Press CTRL+C to quit
```

Predicciones de 2 registros del *data set* test:

	artists	duration_ms	explicit	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	valence	tempo	time_signature	track_genre
79644	Xanthochroid	201986	False	0.174	0.324	0	-13.455	1	0.0396	0.632000	0.847000	0.5580	0.0493	119.282	3	black-metal



	artists	duration_ms	explicit	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	valence	tempo	time_signature	track_genre
54259	The Slapbacks	196747	False	0.591	0.908	7	-3.977	1	0.0804	0.243000	0.000046	0.0918	0.9200	119.581	4	rockabilly



5. Conclusiones

- Variables numéricas presentan distribuciones muy sesgadas y colas largas (ej: duration_ms, popularity).
- La matriz de correlaciones mostró relativamente baja multicolinealidad, lo que permite usar varias variables sin excesiva redundancia.

- track_genre y artists tienen cardinalidad moderada; track_name y album_name son casi únicas.
- Para estas últimas, un one-hot sería inviable (dimensionalidad excesiva), por lo que se opta por codificación basada en target.
- Separación 67 %/33 % **antes** de cualquier transformación para evitar fugas de información.
- Uso de CatBoostEncoder en variables categóricas. La ventaja de este categorizador es que se obtiene una sola columna numérica por categoría y con regularización hacia la media global.
- **Log(1+x)** en duration_ms, instrumentalness, num_artists: comprime colas y estabiliza la varianza.
- **Exp(x)** en speechiness, liveness: amplifica diferencias sutiles en rangos [0–1].
- **Raíz cuadrada** en tempo y en las codificaciones del CatBooster, para moderar valores extremos.
- StandardScaler sobre todas las variables: media ≈ 0, desviación ≈ 1, mejor convergencia y comparabilidad de coeficientes.
- **BaggingRegressor**: Óptimo local de RMSE en ~660 árboles, max_samples=1.0, max_features=0.6.
- **RandomForestRegressor**: Mejor desempeño a n_estimators=500, max_depth=25, max_features=1.0.
- **XGBoostRegressor**: mínimo RMSE cerca de learning_rate≈0.15–0.2 y max_depth≈6–7.
- **Optimización con Optuna** RMSE = 8.9765 n_estimators = 2900 learning_rate =0.01913 max_depth = 12. La mayor ventaja de Optuna es que captura interacciones entre parámetros y dirige la búsqueda hacia combinaciones globalmente óptimas sin hacer búsqueda exhaustiva.
- XGBoost gana con la menor métrica de error, compensando el mayor coste computacional con mejor precisión.