

Evidencia: Módulo 2 Uso de framework o biblioteca de aprendizaje máquina para la implementación de una solución.

Jorge Daniel Rea Prado - A01747327

TC3006C Inteligencia artificial avanzada para la ciencia de datos I

Grupo 101

Profesor:

Jorge Adolfo Ramírez Uresti

Campus Estado de México

Septiembre 2024

Implementación de un Modelo de Regresión Logística Usando scikit-learn para la Clasificación de Canciones

Introducción

En este proyecto, implementamos un modelo de Regresión Logística utilizando la biblioteca scikit-learn para resolver un problema de clasificación binaria. El objetivo principal es predecir si una canción será clasificada como "buena" o "mala" basándonos en un conjunto de características como popularidad, duración, energía, disponibilidad y una amplia variedad de géneros musicales.

Para este propósito, se ha utilizado un dataset que incluye diversas características relevantes de canciones, lo que nos permite entrenar un modelo de aprendizaje automático capaz de aprender patrones a partir de los datos y hacer predicciones en base a ellos. Además, se ha implementado un flujo completo que incluye la preparación de datos, el entrenamiento del modelo, la evaluación del rendimiento del modelo y la predicción en datos no vistos.

El modelo fue implementado utilizando el framework scikit-learn, que nos proporciona las herramientas necesarias para construir y entrenar modelos de aprendizaje automático de manera eficiente. Este enfoque nos permite aprovechar métodos de validación cruzada, optimización de hiperparámetros y técnicas para manejar clases desbalanceadas, asegurando que el modelo generalice correctamente y ofrezca predicciones confiables.

En este documento, presentaremos los detalles de la implementación, los resultados obtenidos, y las conclusiones derivadas del análisis de las métricas de evaluación, tales como la exactitud (accuracy), precisión (precision), recall y F1 Score, además de mostrar el comportamiento del modelo a través de la matriz de confusión.

Descripción del Modelo

Para abordar el problema de clasificación binaria, se decidió utilizar un modelo de Regresión Logística, este modelo es ampliamente utilizado en problemas donde la variable objetivo es categórica y binaria, como en nuestro caso, donde queremos clasificar canciones en dos categorías: "buena" o "mala". La Regresión Logística funciona modelando la relación entre las características (o variables predictoras) y la probabilidad de que una observación pertenezca a una clase específica. Utiliza la función logística para producir probabilidades, que luego se utilizan para asignar la clase final; este modelo es ideal para problemas como el nuestro, ya que es interpretativo, eficiente, y tiende a generalizar bien cuando se aplican técnicas adecuadas de regularización.

Justificación de la Selección del Modelo

La elección de la Regresión Logística se basó en los siguientes aspectos clave:

- **Naturaleza del problema:** Dado que nuestro problema es de clasificación binaria, la Regresión Logística es una opción natural.
- **Interpretabilidad:** Es fácil de interpretar, ya que los coeficientes estimados nos permiten entender la relación entre las características de la canción y la probabilidad de que una canción sea "buena".
- **Eficiencia:** La Regresión Logística es computacionalmente eficiente y puede manejar datasets de tamaño considerable sin requerir grandes recursos computacionales.
- **Regularización:** La regularización es una característica importante del modelo de Regresión Logística, lo que ayuda a controlar el sobreajuste y asegura que el modelo generalice mejor a datos no vistos.

Configuración del Modelo

El modelo fue configurado con los siguientes hiperparámetros clave:

- Solver: Se seleccionó el solver 'saga', ya que es eficiente para trabajar con datasets grandes y admite tanto la regularización L1 como L2; esto es importante para controlar el tamaño del dataset y la complejidad del modelo.
- Regularización (C): Se utilizó un valor de $C=0.01$, lo que implica una regularización L2 fuerte. Esto fue decidido para prevenir el sobreajuste del modelo y asegurar que no se ajuste demasiado a los datos de entrenamiento.
- Número de Iteraciones: El número máximo de iteraciones fue configurado en 500 para permitir que el solver 'saga' tenga suficiente tiempo para converger y encontrar una solución óptima.
- `class_weight='balanced'`: Debido a que el dataset tiene una distribución de clases desbalanceada (más ejemplos de canciones "buenas" que "malas"), se utilizó esta opción para ajustar automáticamente los pesos de las clases en función de su frecuencia en el conjunto de datos. Esto garantiza que el modelo no se vea sesgado hacia la clase mayoritaria.

Proceso de Entrenamiento

El modelo fue entrenado utilizando un conjunto de datos que fue dividido en tres subconjuntos:

- 70% para el entrenamiento: Se usó para ajustar los pesos del modelo.
- 15% para la validación: Nos permitió evaluar el rendimiento del modelo mientras ajustábamos los hiperparámetros, ayudando a detectar sobreajuste.

- 15% para la prueba: Fue reservado para evaluar el rendimiento final del modelo en datos completamente nuevos.

Validación Cruzada


Durante el proceso de entrenamiento, se implementó validación cruzada de 5 particiones (5-fold cross-validation) para evaluar la estabilidad del modelo. Este enfoque divide el conjunto de entrenamiento en 5 partes, entrena el modelo en 4 partes y lo evalúa en la parte restante, repitiendo este mismo proceso 5 veces. La validación cruzada nos ayudó a asegurar que el modelo generalizara correctamente y no estuviera sobreajustado a un subconjunto específico de los datos.

Implementación del Algoritmo de ML Usando Framework

El proyecto está organizado en tres archivos principales:

- `model.py`: Define la estructura del modelo de Regresión Logística.

Este archivo contiene la definición del modelo de **Regresión Logística** que se utilizará para la clasificación de canciones en "buenas" o "malas". El modelo fue creado utilizando la biblioteca `scikit-learn`, la cual permite la configuración de varios parámetros que afectan el comportamiento del modelo. A continuación, se describe el código en detalle:



```

1  # Importamos La Regresión Logística
2  from sklearn.linear_model import LogisticRegression
3
4  """
5      Esta función crea y devuelve un modelo de regresión logística aju
6      stado para resolver un problema de clasificación binaria.
7
8      Se utiliza el solver 'saga', que es eficiente para grandes datase
9      ts y admite regularización tanto L1 como L2.
10     Además, se aplica una regularización fuerte (C=0.01) para evitar
11     el sobreajuste y mejorar la capacidad de generalización del modelo.
12
13     Se establece un número de iteraciones elevado (500) para garantiz
14     ar que el solver tenga suficiente tiempo para converger.
15     El modelo utiliza la regularización L2 (Ridge) para penalizar los
16     coeficientes grandes y mejorar la estabilidad.
17
18     Se emplea 'class_weight=balanced' para ajustar los pesos automáti
19     camente en función de la frecuencia de las clases,
20     lo cual es útil cuando las clases están desbalanceadas (por ejemp
21     lo, si hay más canciones malas que buenas).
22
23     Finalmente, la semilla 'random_state=42' se establece para asegur
24     ar que los resultados sean reproducibles.
25 """
26 def create_model():
27     model = LogisticRegression(solver='saga', C=0.01, random_state=4
28     2, max_iter=500, penalty="l2", class_weight='balanced')
29     return model

```

Descripción del Código

1. Uso de la Regresión Logística:

- Qué se hace: Se importa LogisticRegression de sklearn.linear_model, que se utilizará para la clasificación binaria.
- Por qué se hace: La regresión logística es ideal para problemas de clasificación binaria como este, donde queremos predecir si una canción es "buena" o "mala".

- Para qué se hace: Esto permite entrenar un modelo que pueda predecir la probabilidad de que una canción pertenezca a la clase positiva ("buena").

2. Función `create_model()`:

- Qué se hace: Se define la función `create_model()` que devuelve un objeto de tipo `LogisticRegression` configurado con los parámetros óptimos.
- Por qué se hace: Al encapsular la configuración del modelo en una función, es más fácil mantener y reutilizar el código cuando sea necesario.
- Para qué se hace: Se crea un modelo optimizado para realizar predicciones sobre el dataset de canciones.

Explicación de los Parámetros

- `solver='saga':`
 - Qué es: El solver 'saga' es un algoritmo eficiente para manejar datasets grandes y admite tanto regularización L1 como L2.
 - Por qué se usa: Es especialmente útil cuando se trabaja con conjuntos de datos grandes y características escasas.
 - Para qué se usa: Mejora el rendimiento del modelo en términos de velocidad y capacidad de manejo de grandes volúmenes de datos.
- `C=0.01:`
 - Qué es: C controla la regularización, siendo 0.01 un valor que aplica una fuerte regularización.
 - Por qué se usa: Ayuda a prevenir el sobreajuste, evitando que el modelo se ajuste demasiado a los datos de entrenamiento.

- Para qué se usa: Mejorar la capacidad de generalización del modelo en datos no vistos.
- `max_iter=500`:
 - Qué es: El número máximo de iteraciones del solver.
 - Por qué se usa: Garantiza que el algoritmo tenga tiempo suficiente para converger y encontrar una solución óptima.
 - Para qué se usa: Evitar que el entrenamiento se detenga antes de tiempo, especialmente en datasets grandes o complejos.
- `penalty="l2"`:
 - Qué es: La regularización L2, también conocida como Ridge, penaliza grandes coeficientes del modelo.
 - Por qué se usa: Ayuda a estabilizar los coeficientes y evitar que un solo predictor domine el modelo.
 - Para qué se usa: Para reducir el riesgo de sobreajuste y mejorar la estabilidad del modelo.
- `class_weight="balanced"`:
 - Qué es: Ajusta automáticamente los pesos de las clases en función de su frecuencia.
 - Por qué se usa: En este problema, las clases están desbalanceadas (más canciones "buenas" que "malas"), por lo que este ajuste asegura que el modelo no favorezca a la clase mayoritaria.
 - Para qué se usa: Mejorar el rendimiento del modelo en la predicción de la clase minoritaria ("mala").

- `random_state=42:`
 - Qué es: Es una semilla aleatoria que asegura que los resultados sean reproducibles.
 - Por qué se usa: Asegura que los mismos datos y parámetros produzcan los mismos resultados en diferentes ejecuciones.
 - Para qué se usa: Facilitar la comparación de resultados entre ejecuciones y permitir la reproducibilidad del modelo.
- `train.py`: Ejecuta el proceso de entrenamiento, validación y evaluación del modelo.

Este archivo es responsable de todo el proceso de entrenamiento, validación, y evaluación del modelo de Regresión Logística. El código contiene múltiples secciones importantes que se explicarán a continuación.

```

1  import os
2  import pandas as pd
3  from sklearn.model_selection import train_test_split, cross_val_score
4  from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
5  from sklearn.preprocessing import StandardScaler
6  import pickle
7  from model import create_model
8
9
10 """
11     Sección: Cargar el dataset limpio (preprocesado)
12
13     Qué se hace: Se carga el archivo CSV que contiene los datos preprocesados para el modelo.
14     Por qué se hace: El dataset contiene las características que usaremos para entrenar y evaluar el modelo de regresión logística.
15     Para qué se hace: Este dataset es necesario para poder dividir los datos en conjuntos de entrenamiento, validación y prueba.
16 """
17 current_directory = os.path.dirname(__file__) # Obtiene el directorio donde está train.py
18 data_directory = os.path.join(current_directory, 'data') # Apunta a la carpeta 'data' donde está el CSV
19 file_path = os.path.join(data_directory, 'train.csv') # Define la ruta completa del archivo CSV
20 df = pd.read_csv(file_path) # Lee el dataset y lo almacena en un DataFrame

```

- Qué se hace: Aquí se cargan las bibliotecas necesarias y el dataset preprocesado desde un archivo CSV. El archivo está organizado en un directorio data.
- Por qué se hace: Necesitamos acceder a los datos preprocesados para realizar el entrenamiento del modelo.
- Para qué se hace: Sin el acceso a los datos, no se puede entrenar el modelo ni hacer predicciones.



```
1 X = df.drop(columns=['target']) # 'X' contiene todas las columnas menos 'target'
2 y = df['target'] # 'y' es la columna que contiene la variable objetivo (target)
```

NOTA: Se quitaron los comentarios, únicamente para mostrar un código más limpio en el documento

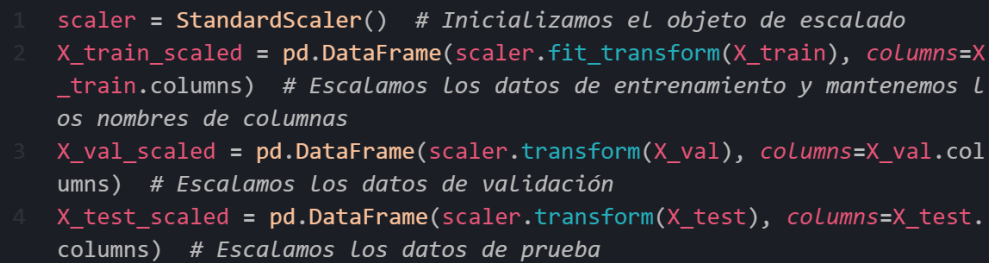
- Qué se hace: Se separan las características (X) de la variable objetivo (y), donde X representa todas las columnas que se utilizarán como predictores, y y es la columna target que queremos predecir.
- Por qué se hace: Necesitamos diferenciar entre las características que alimentarán el modelo y la variable que queremos predecir.
- Para qué se hace: Permitir al modelo aprender la relación entre las características y la variable objetivo.



```
1 X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42) # Dividimos en train (70%) y un conjunto temporal (30%)
2 X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42) # Dividimos el temporal en validación y prueba (15% cada uno)
```

- Qué se hace: Se divide el dataset en tres subconjuntos:
 - 70% para el conjunto de entrenamiento.
 - 15% para el conjunto de validación.

- 15% para el conjunto de prueba.
- Por qué se hace: Esto es crucial para evaluar el rendimiento del modelo tanto durante el entrenamiento como en datos completamente nuevos.
- Para qué se hace: Permitir ajustar el modelo (con el conjunto de validación) y evaluar su capacidad de generalización (con el conjunto de prueba).



```
1 scaler = StandardScaler() # Inicializamos el objeto de escalado
2 X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train), columns=X_train.columns) # Escalamos los datos de entrenamiento y mantenemos los nombres de columnas
3 X_val_scaled = pd.DataFrame(scaler.transform(X_val), columns=X_val.columns) # Escalamos los datos de validación
4 X_test_scaled = pd.DataFrame(scaler.transform(X_test), columns=X_test.columns) # Escalamos los datos de prueba
```

- Qué se hace: Se escalan las características usando StandardScaler, lo que ajusta los valores a una escala normalizada (media 0 y desviación estándar 1).
- Por qué se hace: Muchas veces, las características en el dataset tienen diferentes rangos (por ejemplo, la duración en milisegundos es mucho mayor que las proporciones como la bailabilidad). El escalado ayuda a mejorar el rendimiento del modelo y facilita la convergencia del algoritmo de optimización.
- Para qué se hace: Asegurar que el modelo de Regresión Logística no se vea afectado por características que tienen valores en diferentes escalas.



```
1 model = create_model() # Se crea el modelo utilizando los parámetros
    definidos en la función create_model
2 model.fit(X_train_scaled, y_train) # Entrenamos el modelo con los datos
    de entrenamiento
```

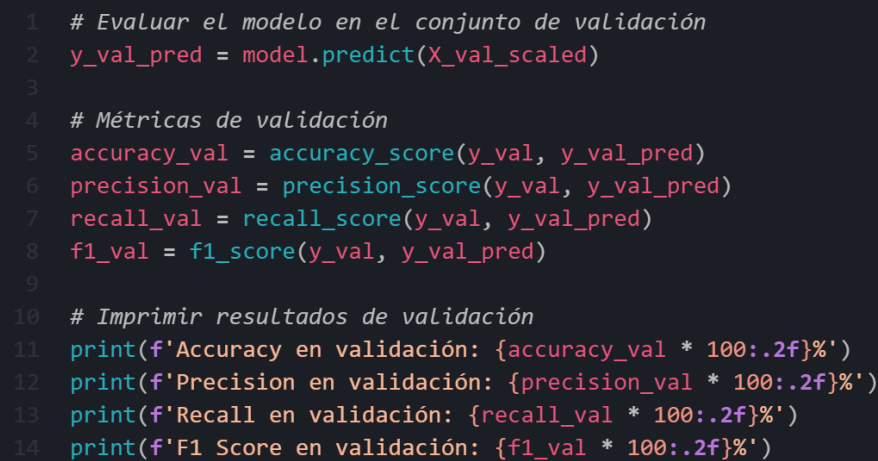
- Qué se hace: Se crea el modelo de Regresión Logística utilizando la función `create_model()` (definida en `model.py`) y luego se entrena el modelo con los datos escalados.
- Por qué se hace: El modelo necesita aprender las relaciones entre las características (X) y la variable objetivo (y) mediante el proceso de entrenamiento.
- Para qué se hace: Para que el modelo pueda hacer predicciones basadas en nuevos datos una vez entrenado.



```
1 cross_val_scores = cross_val_score(model, X_train_scaled, y_train, cv=
    5) # Realizamos 5-fold cross-validation
2 print(f'Cross-validation accuracy: {cross_val_scores.mean():.2f}%') #
    Imprimimos la exactitud promedio de la validación cruzada
```

- Qué se hace: Se realiza validación cruzada con 5 particiones (`cv=5`), lo que significa que el modelo se entrena y evalúa cinco veces en diferentes subconjuntos del conjunto de entrenamiento.

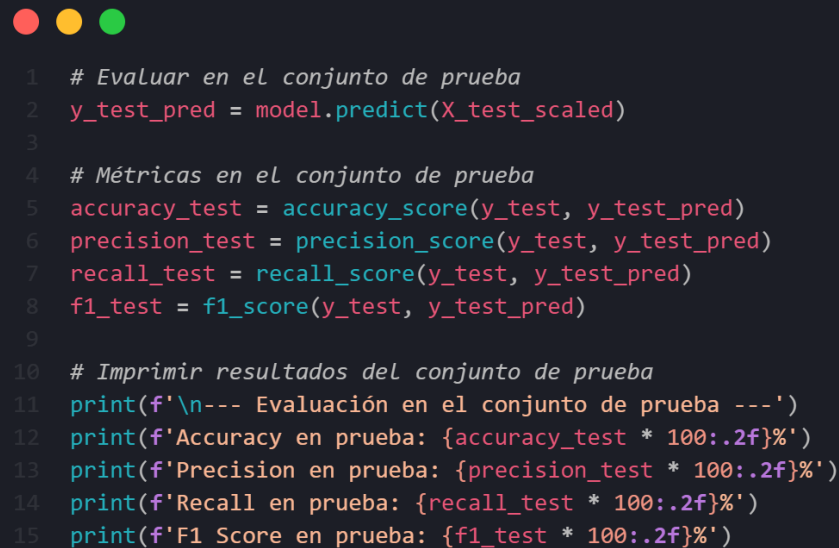
- Por qué se hace: La validación cruzada nos ayuda a asegurar que el modelo no está sobreajustado a un solo conjunto de datos y que generaliza bien en todos los subconjuntos de los datos.
- Para qué se hace: Obtener una estimación más robusta de la precisión del modelo en diferentes subconjuntos de datos.

A screenshot of a code editor with a dark background and light-colored text. The code is in Python and evaluates a model on a validation set. It includes comments in Spanish and uses standard machine learning metrics from the sklearn library. The code is numbered from 1 to 14.

```
1 # Evaluar el modelo en el conjunto de validación
2 y_val_pred = model.predict(X_val_scaled)
3
4 # Métricas de validación
5 accuracy_val = accuracy_score(y_val, y_val_pred)
6 precision_val = precision_score(y_val, y_val_pred)
7 recall_val = recall_score(y_val, y_val_pred)
8 f1_val = f1_score(y_val, y_val_pred)
9
10 # Imprimir resultados de validación
11 print(f'Accuracy en validación: {accuracy_val * 100:.2f}%')
12 print(f'Precision en validación: {precision_val * 100:.2f}%')
13 print(f'Recall en validación: {recall_val * 100:.2f}%')
14 print(f'F1 Score en validación: {f1_val * 100:.2f}%')
```

- Qué se hace: Se evalúa el modelo en el conjunto de validación, obteniendo varias métricas:
 - Exactitud (Accuracy): Proporción de predicciones correctas.
 - Precisión (Precision): Proporción de verdaderos positivos sobre todas las predicciones positivas.
 - Recall: Proporción de verdaderos positivos sobre todos los ejemplos reales positivos.
 - F1 Score: Media armónica entre la precisión y el recall.

- Por qué se hace: Estas métricas permiten evaluar el rendimiento del modelo de forma más completa.
- Para qué se hace: Asegurar que el modelo no solo sea preciso, sino que también sea capaz de predecir correctamente ambas clases.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in Python and evaluates a model's performance on a test set. It includes comments in Spanish and uses the sklearn metrics module for accuracy, precision, recall, and F1 score. The results are printed to the console with formatted strings.

```
1  # Evaluar en el conjunto de prueba
2  y_test_pred = model.predict(X_test_scaled)
3
4  # Métricas en el conjunto de prueba
5  accuracy_test = accuracy_score(y_test, y_test_pred)
6  precision_test = precision_score(y_test, y_test_pred)
7  recall_test = recall_score(y_test, y_test_pred)
8  f1_test = f1_score(y_test, y_test_pred)
9
10 # Imprimir resultados del conjunto de prueba
11 print(f'\n--- Evaluación en el conjunto de prueba ---')
12 print(f'Accuracy en prueba: {accuracy_test * 100:.2f}%')
13 print(f'Precision en prueba: {precision_test * 100:.2f}%')
14 print(f'Recall en prueba: {recall_test * 100:.2f}%')
15 print(f'F1 Score en prueba: {f1_test * 100:.2f}%')
```

- Qué se hace: Se evalúa el modelo en el conjunto de prueba para medir su rendimiento en datos completamente nuevos.
- Por qué se hace: Queremos asegurarnos de que el modelo generalice bien y no se sobreajuste a los datos de entrenamiento.
- Para qué se hace: Evaluar el rendimiento final del modelo y verificar que sea adecuado para producción.



```
1 conf_matrix = confusion_matrix(y_test, y_test_pred)
2 print("\nMatriz de confusión:")
3 print(conf_matrix)
```

- Qué se hace: Se genera la matriz de confusión usando la función `confusion_matrix` de `scikit-learn`, que compara las predicciones del modelo con las etiquetas reales en el conjunto de prueba.
- Por qué se hace: La matriz de confusión proporciona una representación detallada de los aciertos y errores del modelo, desglosando las predicciones en:
 - Verdaderos Positivos (TP): Canciones correctamente clasificadas como "buenas".
 - Verdaderos Negativos (TN): Canciones correctamente clasificadas como "malas".
 - Falsos Positivos (FP): Canciones clasificadas incorrectamente como "buenas".
 - Falsos Negativos (FN): Canciones clasificadas incorrectamente como "malas".
- Para qué se hace: Es útil para analizar en detalle los errores del modelo, especialmente si el conjunto de datos está desbalanceado. Por ejemplo, si hay muchos más ejemplos de canciones "buenas", es posible que el modelo esté prediciendo más falsos positivos.



```
1 model_directory = os.path.join(current_directory, 'model') # Apunta a
  la carpeta 'model'
2 model_path = os.path.join(model_directory, 'modeloRegresionFramework.p
  kl') # Define el nombre del archivo .pkl
3
4 # Guardar el modelo entrenado en formato pickle
5 with open(model_path, 'wb') as f:
6     pickle.dump(model, f) # Guarda el modelo entrenado en el archivo
```

- Qué se hace: El modelo entrenado se guarda en un archivo .pkl usando pickle.
- Por qué se hace: Guardar el modelo permite reutilizarlo sin necesidad de entrenarlo nuevamente.
- Para qué se hace: Esto permite cargar el modelo en el archivo predict.py para hacer predicciones con nuevos datos.
- predict.py: Realiza predicciones en nuevos datos utilizando el modelo ya entrenado. Este archivo se encarga de realizar predicciones utilizando el modelo de Regresión Logística previamente entrenado y guardado en el archivo modeloRegresionFramework.pkl. En esta sección, se genera un conjunto de datos aleatorios o manuales y se utiliza el modelo para predecir si una canción es "buena" o "mala". A continuación, se explica el código en detalle:

```
1 import os
2 import pandas as pd
3 import pickle
4 import numpy as np
5
6 # Cargar el modelo entrenado desde el archivo .pkl
7 current_directory = os.path.dirname(__file__) # Obtiene el directorio donde está predict.py
8 model_directory = os.path.join(current_directory, 'model') # Apunta a la carpeta 'model'
9 model_path = os.path.join(model_directory, 'modeloRegresionFramework.pkl')
10 with open(model_path, 'rb') as f:
11     model = pickle.load(f)
```

NOTA: Se quitaron los comentarios, únicamente para mostrar un código más limpio en el documento

- Qué se hace: Se carga el modelo previamente entrenado desde el archivo .pkl usando la función pickle.load().
- Por qué se hace: El modelo necesita ser cargado para realizar predicciones sin volver a entrenarlo.
- Para qué se hace: Esto permite reutilizar el modelo sin tener que pasar por el costoso proceso de entrenamiento nuevamente.

```

1  num_rows = 1 # Número de ejemplos a generar
2  test_data = {
3      'popularity': np.random.randint(0, 100, size=num_rows), # Popularidad aleatoria entre 0 y 100
4      'duration_ms': np.random.randint(120000, 300000, size=num_rows), # Duración entre 2 y 5 minutos
5      'explicit': np.random.choice([0, 1], size=num_rows), # Canción explícita (0 = no, 1 = sí)
6      'danceability': np.random.uniform(0, 1, size=num_rows), # Bailabilidad entre 0 y 1
7      'energy': np.random.uniform(0, 1, size=num_rows), # Energía entre 0 y 1
8      'key': np.random.randint(0, 12, size=num_rows), # Tonalidad (0-11)
9      'loudness': np.random.uniform(-60, 0, size=num_rows), # Volumen en dB (entre -60 y 0)
10     'mode': np.random.choice([0, 1], size=num_rows), # Modo (0 = menor, 1 = mayor)
11     'speechiness': np.random.uniform(0, 1, size=num_rows), # Cantidad de palabras en la canción
12     'acousticness': np.random.uniform(0, 1, size=num_rows), # Nivel de acústica
13     'instrumentalness': np.random.uniform(0, 1, size=num_rows), # Nivel de instrumentalidad
14     'liveness': np.random.uniform(0, 1, size=num_rows), # Presencia en vivo
15     'valence': np.random.uniform(0, 1, size=num_rows), # Emoción positiva
16     'tempo': np.random.uniform(60, 200, size=num_rows), # Tempo (beats por minuto)
17     'time_signature': np.random.choice([3, 4, 5], size=num_rows), # Compás de la canción
18 }

```

- Qué se hace: Se generan datos aleatorios que simulan las características de una canción (popularidad, duración, energía, etc.).
- Por qué se hace: Esto permite probar el modelo con datos nuevos sin necesidad de un dataset externo.
- Para qué se hace: Nos ayuda a validar el rendimiento del modelo en datos no vistos, simulando el proceso de predicción con datos aleatorios.



```
1 genres = [  
2     'track_genre_afrobeat', 'track_genre_alt-rock', 'track_genre_alternative', 'track_g  
   genre_ambient', 'track_genre_anime',  
3     'track_genre_black-metal', 'track_genre_bluegrass', 'track_genre_blues', 'track_gen  
   re_brazil', 'track_genre_breakbeat',  
4     'track_genre_british', 'track_genre_cantopop', 'track_genre_chicago-house', 'track_  
   genre_children', 'track_genre_chill',  
5     'track_genre_classical', 'track_genre_club', 'track_genre_comedy', 'track_genre_cou  
   ntry', 'track_genre_dance',  
6     'track_genre_dancehall', 'track_genre_death-metal', 'track_genre_deep-house', 'trac  
   k_genre_detroit-techno', 'track_genre_disco',  
7     'track_genre_disney', 'track_genre_drum-and-bass', 'track_genre_dub', 'track_genre_  
   dubstep', 'track_genre_edm',  
8     'track_genre_electro', 'track_genre_electronic', 'track_genre_emo', 'track_genre_fo  
   lk', 'track_genre_forro',  
9     'track_genre_french', 'track_genre_funk', 'track_genre_garage', 'track_genre_germa  
   n', 'track_genre_gospel',  
10    'track_genre_goth', 'track_genre_grindcore', 'track_genre_groove', 'track_genre_gru  
   nge', 'track_genre_guitar',  
11    'track_genre_happy', 'track_genre_hard-rock', 'track_genre_hardcore', 'track_genre_  
   hardstyle', 'track_genre_heavy-metal',  
12    'track_genre_hip-hop', 'track_genre_honky-tonk', 'track_genre_house', 'track_genre_  
   idm', 'track_genre_indian',  
13    'track_genre_indie', 'track_genre_indie-pop', 'track_genre_industrial', 'track_genr  
   e_iranian', 'track_genre_j-dance',  
14    'track_genre_j-idol', 'track_genre_j-pop', 'track_genre_j-rock', 'track_genre_jaz  
   z', 'track_genre_k-pop',  
15    'track_genre_kids', 'track_genre_latin', 'track_genre_latino', 'track_genre_malay',  
   'track_genre_mandopop',  
16    'track_genre_metal', 'track_genre_metalcore', 'track_genre_minimal-techno', 'track_  
   genre_mpb', 'track_genre_new-age',  
17    'track_genre_opera', 'track_genre_pagode', 'track_genre_party', 'track_genre_pian  
   o', 'track_genre_pop',  
18    'track_genre_pop-film', 'track_genre_power-pop', 'track_genre_progressive-house',  
   'track_genre_psych-rock', 'track_genre_punk',  
19    'track_genre_punk-rock', 'track_genre_r-n-b', 'track_genre_reggae', 'track_genre_re  
   ggaeton', 'track_genre_rock',  
20    'track_genre_rock-n-roll', 'track_genre_rockabilly', 'track_genre_romance', 'track_  
   genre_sad', 'track_genre_salsa',  
21    'track_genre_samba', 'track_genre_sertanejo', 'track_genre_show-tunes', 'track_genr  
   e_singer-songwriter', 'track_genre_ska',  
22    'track_genre_sleep', 'track_genre_songwriter', 'track_genre_soul', 'track_genre_spa  
   nish', 'track_genre_study',  
23    'track_genre_swedish', 'track_genre_synth-pop', 'track_genre_tango', 'track_genre_t  
   echno', 'track_genre_trance',  
24    'track_genre_trip-hop', 'track_genre_turkish', 'track_genre_world-music'  
25 ]  
26  
27  
28 # Inicializar todas las columnas de géneros a 0  
29 for genre in genres:  
30     test_data[genre] = np.zeros(num_rows)  
31  
32 # Para cada fila, seleccionar un género aleatorio y poner un 1 en esa columna  
33 for i in range(num_rows):  
34     random_genre = np.random.choice(genres) # Seleccionamos un género aleatorio  
35     test_data[random_genre][i] = 1 # Asignamos el género a la canción
```

- Qué se hace: Se asigna aleatoriamente un género a cada canción, estableciendo el valor 1 en la columna correspondiente, mientras que las demás columnas de género se inicializan en 0.
- Por qué se hace: Esto simula el hecho de que cada canción pertenece solo a un género musical.
- Para qué se hace: Permite representar correctamente la pertenencia a géneros musicales en el dataset de prueba.

```
1 test_df = pd.DataFrame(test_data) # Convertimos los datos generados en un DataFrame
```

- Qué se hace: Se crea un DataFrame de pandas a partir de los datos aleatorios generados en los pasos anteriores.
- Por qué se hace: Un DataFrame es el formato requerido por el modelo para hacer predicciones, ya que fue entrenado con este formato.
- Para qué se hace: Permitir que el modelo realice predicciones con los datos generados.

```
1 # Realizar predicciones
2 predictions = model.predict(test_df) # Usamos el modelo para predecir si las canciones son buenas o malas
```

- Qué se hace: Se utilizan los datos generados para realizar predicciones con el modelo entrenado.

- Por qué se hace: Queremos que el modelo clasifique si las canciones generadas aleatoriamente son "buenas" o "malas".
- Para qué se hace: Evaluar el comportamiento del modelo en nuevos datos y comprobar su capacidad para generalizar.

```

1 # Agregar las predicciones al DataFrame
2 test_df['prediction'] = predictions
3
4 # Definir qué es una "buena" canción (si la predicción es 1, es buena)
5 test_df['is_good_song'] = test_df['prediction'].apply(lambda x: 'Buena canción' if x ==
6 1 else 'Mala canción')
7
8 # Imprimir las características de cada canción junto con la predicción
9 for index, row in test_df.iterrows():
10     print(f"Canción {index + 1}:")
11     print(row)
12     print(f"Predicción: {row['is_good_song']}")
13     print("-" * 50)

```

- Qué se hace:
 - Se agregan las predicciones al DataFrame.
 - Se define si una canción es "buena" o "mala" en base a la predicción.
 - Se imprimen las características de cada canción junto con la predicción.
- Por qué se hace: Queremos interpretar los resultados de las predicciones y verificar si el modelo está clasificando correctamente las canciones generadas.
- Para qué se hace: Facilitar la comprensión de los resultados y analizar si el modelo está haciendo predicciones precisas basadas en las características de las canciones.

```

1  # Probar con un conjunto específico de datos (manual)
2  test_data_good_song = [...] # Datos de una buena canción
3
4  # Asegurarse de que test_data tenga el mismo número de columnas que X_train
5  test_data_full = np.zeros(128)
6  test_data_full[:len(test_data_good_song)] = test_data_good_song # Insertar Los valores
   conocidos
7
8  # Convertir a array y redimensionar para pasarlo al modelo
9  test_data_array = test_data_full.reshape(1, -1)
10
11 # Hacer La predicción
12 prediction = model.predict(test_data_array)
13 print(f'Predicción: {prediction[0]}') # Debería dar 1 si el modelo predice correctamen
   te.

```

- Qué se hace: Se pasa un conjunto de datos manual (conocido como una "buena canción") al modelo y se evalúa su predicción.
- Por qué se hace: Queremos verificar si el modelo clasifica correctamente ejemplos controlados, como una canción manualmente etiquetada como "buena".
- Para qué se hace: Asegurar que el modelo generaliza bien incluso con ejemplos manuales.

Evaluación del Modelo

Una parte crucial del proceso de construcción de un modelo de aprendizaje automático es su evaluación en diferentes conjuntos de datos para asegurar que generalice bien y no se sobreajuste. En este proyecto, el modelo de Regresión Logística fue evaluado utilizando varias métricas clave, tanto en el conjunto de validación como en el de prueba.

```
Cross-validation accuracy: 0.96%
Accuracy en validación: 95.96%
Precision en validación: 74.60%
Recall en validación: 99.75%
F1 Score en validación: 85.36%

--- Evaluación en el conjunto de prueba ---
Accuracy en prueba: 96.05%
Precision en prueba: 75.02%
Recall en prueba: 99.85%
F1 Score en prueba: 85.67%

Matriz de confusión:
[[14407  672]
 [    3 2018]]
```

- Conjunto de validación:

Después de entrenar el modelo, se evaluó su rendimiento en el conjunto de validación, obteniendo las siguientes métricas:

- Exactitud (Accuracy): 95.96%
- Precisión (Precision): 74.60%
- Recall: 99.75%
- F1 Score: 85.36%

Estos resultados muestran que el modelo tiene un rendimiento sólido en términos de exactitud y recall, lo que significa que es capaz de identificar correctamente la mayoría de las canciones "buenas". Sin embargo, la precisión es ligeramente más baja, lo que sugiere que el modelo clasifica algunas canciones "malas" como "buenas", lo que es reflejado en el F1 Score de 85.36%.

```
Cross-validation accuracy: 0.96%
Accuracy en validación: 95.96%
Precision en validación: 74.60%
Recall en validación: 99.75%
F1 Score en validación: 85.36%
```

- Conjunto de pruebas:

El modelo también fue evaluado en el conjunto de prueba (datos completamente nuevos que no se usaron en el entrenamiento), donde los resultados fueron similares:

- Exactitud (Accuracy): 96.05%
- Precisión (Precision): 75.02%
- Recall: 99.85%
- F1 Score: 85.67%

El modelo mostró un recall extremadamente alto (99.85%), lo que significa que casi todas las canciones "buenas" fueron correctamente identificadas. Sin embargo, la precisión relativamente baja (75.02%) indica que algunas canciones "malas" fueron clasificadas incorrectamente como "buenas".

```
--- Evaluación en el conjunto de prueba ---  
Accuracy en prueba: 96.05%  
Precision en prueba: 75.02%  
Recall en prueba: 99.85%  
F1 Score en prueba: 85.67%
```

- Matriz de Confusión

Para analizar con mayor detalle el rendimiento del modelo, se utilizó la matriz de confusión. Esta matriz nos muestra cuántas veces el modelo acertó y cuántas veces cometió errores al predecir canciones "buenas" o "malas":

```
Matriz de confusión:  
[[14407  672]  
 [    3 2018]]  
PS D:\Users\Jorge_Pae\Dev
```

- 14407 canciones fueron correctamente clasificadas como "malas".

- 672 canciones fueron incorrectamente clasificadas como "buenas" (falsos positivos).
- 2018 canciones fueron correctamente clasificadas como "buenas".
- 3 canciones fueron incorrectamente clasificadas como "malas" (falsos negativos).

El modelo demuestra un gran recall, lo que es valioso en casos donde es más importante identificar correctamente las canciones "buenas", aunque a costa de tener algunas falsas alarmas (canciones "malas" clasificadas como "buenas"). La alta exactitud en ambos conjuntos de validación y prueba sugiere que el modelo generaliza bien y no está sobreajustado a los datos de entrenamiento.

Conclusiones

Este proyecto demuestra cómo un modelo de Regresión Logística puede ser implementado para clasificar canciones en "buenas" o "malas" utilizando sus características. Los resultados obtenidos indican que el modelo es capaz de realizar predicciones sólidas y generaliza bien en datos no vistos. El modelo también está diseñado para ser reutilizado fácilmente, ya que se puede guardar y cargar mediante archivos pickle, lo que permite realizar predicciones en nuevos datos sin necesidad de volver a entrenarlo.