# GTZAN_Clasificacion_musical

November 5, 2024

# 1 CLASIFICACIÓN MUSICAL - GTZAN

## 1.1 Introducción

A lo largo de este trabajo implementaremos diferentes métodos de Machine Learning con el objetivo de analizar pistas de audio musicales y sus espectrogramas para clasificarlos en sus correspondientes géneros.

Para ello utilizaremos el dataset GTZAN, que contiene 1000 pistas de audio de 30 segundos clasificadas en 10 géneros diferentes, junto con las 1000 imágenes de los respectivos espectrogramas. Además, en un archivo csv se recogen 60 variables para cada pista, donde se engloba información relativa a frecuencia, tono, contrastes espectrales, información armónica, etc.

Con el objetivo de aumentar el tamaño del dataset, cada pista de audio se divide en 10 pistas de 3 segundos, siendo este el dataframe que utilizaremos para la mayor parte del trabajo.

## 1.2 Preparación de datos

### 1.2.1 Instalación de librerías

[140]:
```
!jupyter notebook --version
```

```
7.0.8
```

[141]:
```
!conda --version
```

```
conda 24.9.2
```

[ ]:
```
pip install scikit-learn pandas matplotlib seaborn numpy librosa tensorflow
```

### 1.2.2 Data Cleansing

[144]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import librosa
```

Cargamos el dataframe de pistas de 3 segundos, comprobamos sus dimensiones y observamos las 5 primeras filas.

[146]:
```
df = pd.read_csv('Data/features_3_sec.csv')
```

[147]:
```
df.shape
```

[147]:
```
(9990, 60)
```

[148]:
```
df.head()
```

[148]:

| | filename | length | chroma_stft_mean | chroma_stft_var | rms_mean | \ |
|---|---|---|---|---|---|---|
| 0 | blues.00000.0.wav | 66149 | 0.335406 | 0.091048 | 0.130405 | |
| 1 | blues.00000.1.wav | 66149 | 0.343065 | 0.086147 | 0.112699 | |
| 2 | blues.00000.2.wav | 66149 | 0.346815 | 0.092243 | 0.132003 | |
| 3 | blues.00000.3.wav | 66149 | 0.363639 | 0.086856 | 0.132565 | |
| 4 | blues.00000.4.wav | 66149 | 0.335579 | 0.088129 | 0.143289 | |

| | rms_var | spectral_centroid_mean | spectral_centroid_var | \ |
|---|---|---|---|---|
| 0 | 0.003521 | 1773.065032 | 167541.630869 | |
| 1 | 0.001450 | 1816.693777 | 90525.690866 | |
| 2 | 0.004620 | 1788.539719 | 111407.437613 | |
| 3 | 0.002448 | 1655.289045 | 111952.284517 | |
| 4 | 0.001701 | 1630.656199 | 79667.267654 | |

| | spectral_bandwidth_mean | spectral_bandwidth_var | ... | mfcc16_var | \ |
|---|---|---|---|---|---|
| 0 | 1972.744388 | 117335.771563 | ... | 39.687145 | |
| 1 | 2010.051501 | 65671.875673 | ... | 64.748276 | |
| 2 | 2084.565132 | 75124.921716 | ... | 67.336563 | |
| 3 | 1960.039988 | 82913.639269 | ... | 47.739452 | |
| 4 | 1948.503884 | 60204.020268 | ... | 30.336359 | |

| | mfcc17_mean | mfcc17_var | mfcc18_mean | mfcc18_var | mfcc19_mean | mfcc19_var | \ |
|---|---|---|---|---|---|---|---|
| 0 | -3.241280 | 36.488243 | 0.722209 | 38.099152 | -5.050335 | 33.618073 | |
| 1 | -6.055294 | 40.677654 | 0.159015 | 51.264091 | -2.837699 | 97.030830 | |
| 2 | -1.768610 | 28.348579 | 2.378768 | 45.717648 | -1.938424 | 53.050835 | |
| 3 | -3.841155 | 28.337118 | 1.218588 | 34.770935 | -3.580352 | 50.836224 | |
| 4 | 0.664582 | 45.880913 | 1.689446 | 51.363583 | -3.392489 | 26.738789 | |

| | mfcc20_mean | mfcc20_var | label |
|---|---|---|---|
| 0 | -0.243027 | 43.771767 | blues |
| 1 | 5.784063 | 59.943081 | blues |
| 2 | 2.517375 | 33.105122 | blues |
| 3 | 3.630866 | 32.023678 | blues |
| 4 | 0.536961 | 29.146694 | blues |

[5 rows x 60 columns]

Eliminamos duplicados en caso de que existieran.

[150]:
```
df = df.drop_duplicates()
df.shape
```

[150]: (9990, 60)

Contamos el número de pistas que tenemos en cada género. Vemos que prácticamente tenemos el mismo número en todas ellas, por lo que podemos considerar que nuestra muestra es relativamente amplia y está repartida equitativamente.

[152]:
```
df.label.value_counts().reset_index()
```

[152]:

| | label | count |
|---|---|---|
| 0 | blues | 1000 |
| 1 | jazz | 1000 |
| 2 | metal | 1000 |
| 3 | pop | 1000 |
| 4 | reggae | 1000 |
| 5 | disco | 999 |
| 6 | classical | 998 |
| 7 | hiphop | 998 |
| 8 | rock | 998 |
| 9 | country | 997 |

A continuación, elegimos una pista de audio cualquiera para comprobar que funciona correctamente. La variable data es la señal de audio en forma de numpy array. Estas números representan las amplitudes de la señal a lo largo del tiempo. En nuestro caso el array es unidimensional ya que las pistas de audios son mono.

La variable sr es el 'sampling rate' o frecuencia de muestreo, es decir, el número de muestras por segundo que se toman de la pista, que por defecto es de 22050 Hz.

[154]:
```
audio_example = 'Data/genres_original/rock/rock.00017.wav'
audio_data , sr = librosa.load(audio_example)
print(type(audio_data), type(sr))
print('sr = ',sr,' Hz')
```

```
<class 'numpy.ndarray'> <class 'int'>
sr =  22050  Hz
```

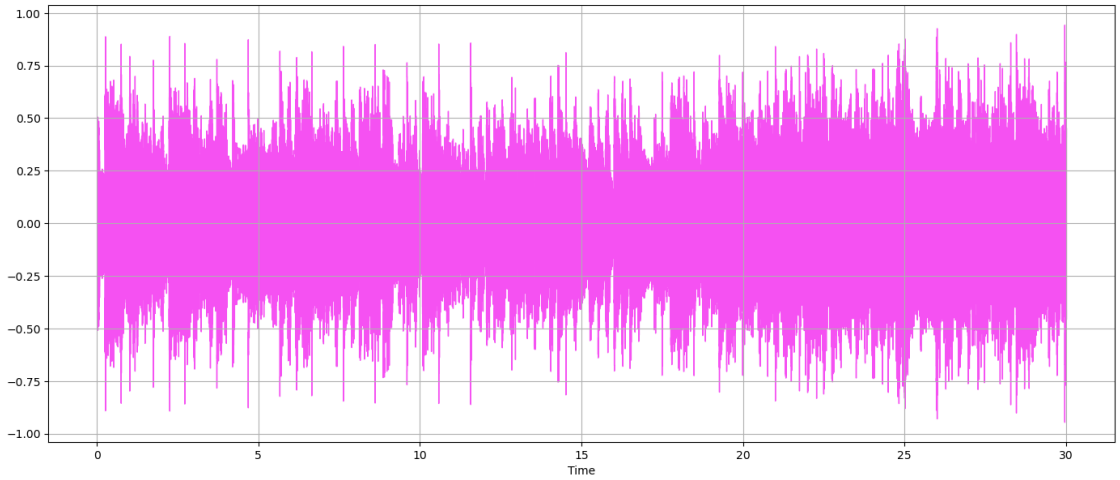
Usamos la librería IPython para crear un widget que nos permita visualizar la pista de audio que hemos elegido.

[156]:
```
import IPython
IPython.display.Audio(audio_data, rate=sr)
```

[156]: <IPython.lib.display.Audio object>

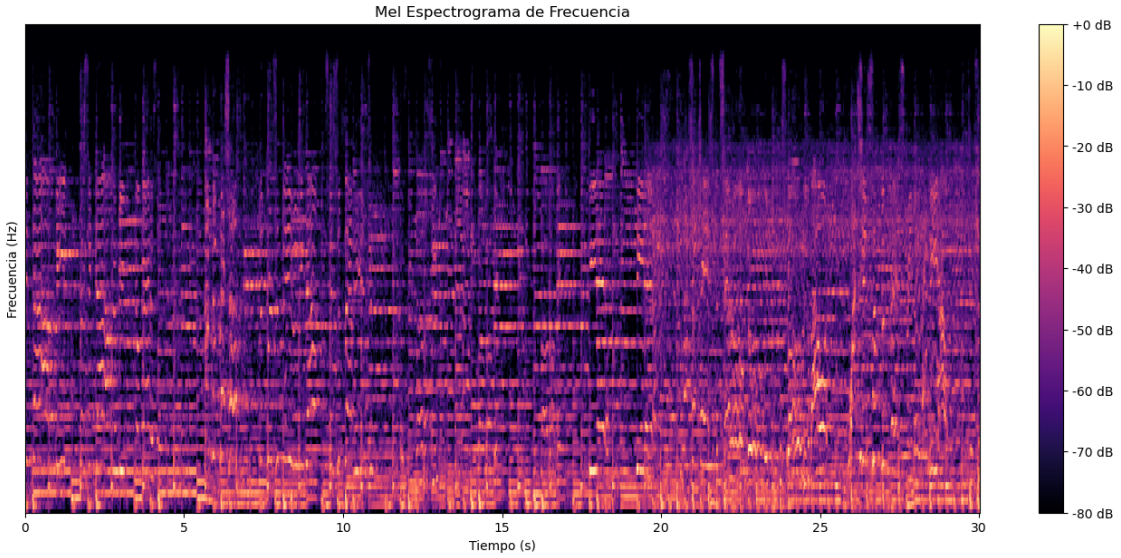
Visualizamos como varía la amplitud de la señal en función del tiempo.

[158]:
```
import librosa.display
plt.figure(figsize=(17,7))
plt.grid()
librosa.display.waveshow(audio_data, color='#f551f2')
plt.show()
```




A continuación, vamos a crear un Mel espectrograma de frecuencias. Para ello transforma la señal de audio al dominio de la frecuencia mediante una Short-time Fourier Transform (STFT) y lo convertimos a escala Mel, que agrupa las frecuencias más altas con menor resolución y las bajas con mayor detalle, para alinearse con cómo escuchamos los sonidos.

[160]:
```
# Creamos el Mel espectrograma
mel_spec_db = librosa.power_to_db(librosa.feature.melspectrogram(y = audio_data,␣
 ↪sr = sr, n_fft = 2048, hop_length = 512, n_mels = 128, power = 4.0), ref = np.
 ↪max)

# Representamos el Mel espectrograma
plt.figure(figsize=(17, 7))
librosa.display.specshow(mel_spec_db, sr=sr, x_axis='time', y_axis='off')
plt.title('Mel Espectrograma de Frecuencia')
plt.colorbar(format='%+2.0f dB')
plt.xlabel('Tiempo (s)')
plt.ylabel('Frecuencia (Hz)')
plt.show()
```




Con el espectrograma representamos visualmente cómo las frecuencias de la señal de audio, cambian a lo largo del tiempo. Es una herramienta muy usada en el análisis de sonido ya que nos permite observar la energía en diferentes bandas de frecuencia. El color indica la intensidad de la señal a una frecuencia e instante dados. Esto es especialmente útil para el reconocimiento de patrones en audio mediante métodos de Machine Learning, ya que con ello conseguimos asignar valores númericos a pistas de audio.

Una vez tenemos el dataframe cargado es necesario normalizar los datos. Los algoritmos de aprendizaje automático suelen funcionar mejor si las características tienen escalas similares, consiguiendo así mejorar el rendimiento y eficiencia del modelo.

En nuestro caso, hemos probado diferentes tipos de normalización, tal y como vemos en el código. La que mejores resultados nos proporciona es la normalización Min-Max, que escala los valores a un rango [0,1].

[163]:
```
from sklearn.preprocessing import MinMaxScaler, StandardScaler, MaxAbsScaler,␣
↪QuantileTransformer

# Excluimos las columnas 'filename' y 'label' para normalizar solo las␣
↪características numéricas
data = df.drop(columns=['filename', 'label'])

# Probamos diferentes tipos de normalización
#scaler = StandardScaler()
scaler = MinMaxScaler()
#scaler = MaxAbsScaler()
#scaler = QuantileTransformer()

data_norm = scaler.fit_transform(data)
```

```
# Convertimos las características normalizadas a un DataFrame
df_norm = pd.DataFrame(data_norm, columns=data.columns)

# Añadimos de nuevo las columnas 'filename' y 'label' al DataFrame normalizado
df_norm['filename'] = df['filename']
df_norm['label'] = df['label']

df_norm.head()
```

[163]:

| | length | chroma_stft_mean | chroma_stft_var | rms_mean | rms_var | \ |
|---|---|---|---|---|---|---|
| 0 | 0.0 | 0.355399 | 0.716757 | 0.293133 | 0.107955 | |
| 1 | 0.0 | 0.367322 | 0.670347 | 0.253040 | 0.044447 | |
| 2 | 0.0 | 0.373159 | 0.728067 | 0.296753 | 0.141663 | |
| 3 | 0.0 | 0.399349 | 0.677066 | 0.298024 | 0.075042 | |
| 4 | 0.0 | 0.355668 | 0.689113 | 0.322308 | 0.052149 | |

| | spectral_centroid_mean | spectral_centroid_var | spectral_bandwidth_mean | \ |
|---|---|---|---|---|
| 0 | 0.262173 | 0.034784 | 0.459205 | |
| 1 | 0.270969 | 0.018716 | 0.470831 | |
| 2 | 0.265293 | 0.023073 | 0.494051 | |
| 3 | 0.238427 | 0.023187 | 0.455246 | |
| 4 | 0.233460 | 0.016451 | 0.451651 | |

| | spectral_bandwidth_var | rolloff_mean | ... | mfcc17_mean | mfcc17_var | \ |
|---|---|---|---|---|---|---|
| 0 | 0.094130 | 0.346153 | ... | 0.397172 | 0.066062 | |
| 1 | 0.052261 | 0.363722 | ... | 0.351681 | 0.074001 | |
| 2 | 0.059922 | 0.378215 | ... | 0.420979 | 0.050639 | |
| 3 | 0.066234 | 0.329587 | ... | 0.387474 | 0.050617 | |
| 4 | 0.047830 | 0.318453 | ... | 0.460314 | 0.083860 | |

| | mfcc18_mean | mfcc18_var | mfcc19_mean | mfcc19_var | mfcc20_mean | mfcc20_var | \ |
|---|---|---|---|---|---|---|---|
| 0 | 0.371828 | 0.055344 | 0.380831 | 0.026797 | 0.506746 | 0.047781 | |
| 1 | 0.362068 | 0.076365 | 0.418452 | 0.082414 | 0.593029 | 0.065548 | |
| 2 | 0.400536 | 0.067509 | 0.433742 | 0.043841 | 0.546264 | 0.036062 | |
| 3 | 0.380430 | 0.050030 | 0.405824 | 0.041898 | 0.562204 | 0.034873 | |
| 4 | 0.388590 | 0.076524 | 0.409019 | 0.020763 | 0.517913 | 0.031713 | |

| | filename | label |
|---|---|---|
| 0 | blues.00000.0.wav | blues |
| 1 | blues.00000.1.wav | blues |
| 2 | blues.00000.2.wav | blues |
| 3 | blues.00000.3.wav | blues |
| 4 | blues.00000.4.wav | blues |

[5 rows x 60 columns]

### 1.2.3 Principal Component Analysis (PCA)

PCA es una técnica que nos permite reducir la dimensionalidad de los datos originales, transformándolo en un nuevo conjunto de variables (componentes principales), que se calculan teniendo en cuenta la varianza de la diferentes características. Esto nos permite reducir el ruido y redundancia de los datos y el tiempo de computación, manteniendo la mayor cantidad de información posible.

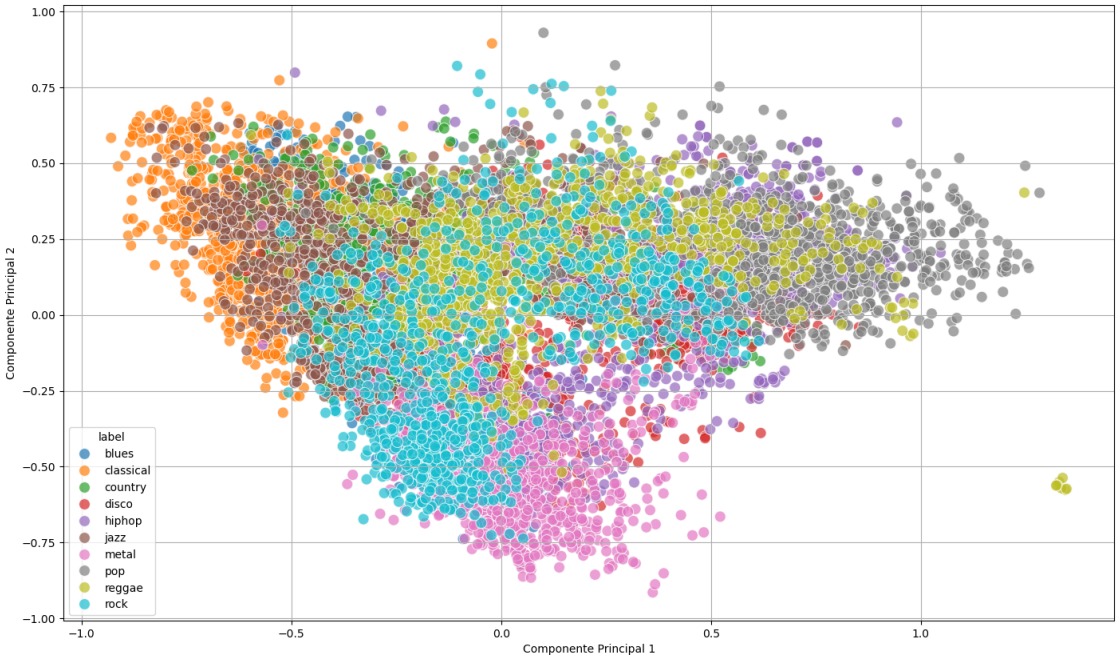
[166]:
```
from sklearn.decomposition import PCA

# Excluímos las columnas del dataframe normalizado que contienen NaNs
data = df_norm.drop(columns=['filename', 'label'])

#Fijamos el número de componentes principales que queremos
pca = PCA(n_components=3)
comp_principales = pca.fit_transform(data)
df_principal = pd.DataFrame(data = comp_principales, columns =␣
↪['cp1','cp2','cp3'])

# Reintroducimos las etiquetas en el dataframe reducido
df_principal['label'] = df['label']

df_principal_2D = df_principal.drop(columns=['cp3'])
df_principal_2D.head()
```

[166]:

| | cp1 | cp2 | label |
|---|---|---|---|
| 0 | -0.242167 | -0.121147 | blues |
| 1 | -0.276062 | -0.230291 | blues |
| 2 | -0.190024 | -0.119509 | blues |
| 3 | -0.278550 | -0.167689 | blues |
| 4 | -0.309771 | -0.187990 | blues |

En primer lugar, tomamos dos componentes principales y observamos la distribución espacial de los datos.

[168]:
```
import seaborn as sns

# Creamos el gráfico
fig = plt.figure(figsize = (17,10))
sns.scatterplot(x = 'cp1', y = 'cp2', data = df_principal, hue = 'label', alpha␣
↪= 0.7, s = 100 )
plt.xlabel('Componente Principal 1')
plt.ylabel('Componente Principal 2')
plt.grid()
```




Con la normalización escogida, vemos que la distribución de los datos no está demasiado diferenciada, aunque sí podemos apreciar cierta tendencia de agrupación de los géneros (especialmente en 'classical' y 'metal'). Por ello, añadimos una tercera componente principal y representamos los datos obtenidos en un gráfico 3D.

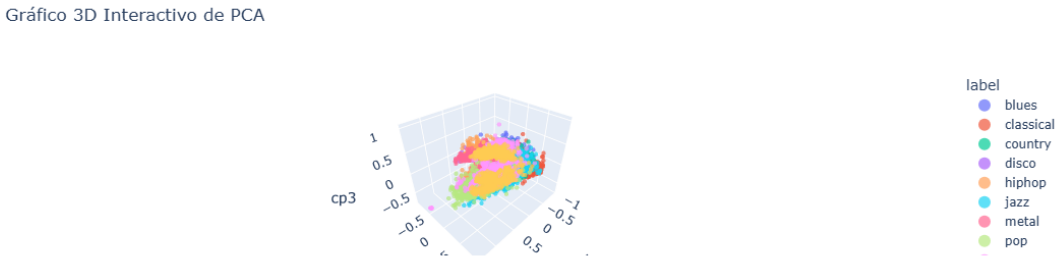
[170]:
```
df_principal.head()
```

[170]:

| | cp1 | cp2 | cp3 | label |
|---|---|---|---|---|
| 0 | -0.242167 | -0.121147 | 0.071058 | blues |
| 1 | -0.276062 | -0.230291 | 0.038986 | blues |
| 2 | -0.190024 | -0.119509 | 0.067502 | blues |
| 3 | -0.278550 | -0.167689 | 0.052075 | blues |
| 4 | -0.309771 | -0.187990 | 0.029072 | blues |

[171]:
```
import plotly.express as px

# Creamos el gráfico 3D interactivo con Plotly
fig = px.scatter_3d(
    df_principal, x='cp1', y='cp2', z='cp3',
    color='label',
    size=[0.1]*len(df_principal),
    size_max = 7,
    labels={'PC1': 'Componente Principal 1', 'PC2': 'Componente Principal 2',␣
 ↪'PC3': 'Componente Principal 3'},
    title="Gráfico 3D Interactivo de PCA")
```

```
fig.update_traces(marker=dict(line=dict(width=0)))

# Mostramos el gráfico
fig.show()
```




Se podría aumentar el número de componentes principales, aunque esto no necesariamente implica una mejora del modelo, ya que podría causar "overfitting". Por ello hemos restringido la dimensionalidad a 3 componentes principales.

La respuesta del modelo a los datos transformados con el PCA depende del tipo utilizado. En nuestro caso, hemos probado tanto con 2 componentes principales como con 3, aunque los resultados obtenidos han sido considerablemente peores en todos los casos. Todo ello, sumado a que el tiempo de cómputo es aceptable, hemos decidido descartar esta transformación y trabajar directamente con el dataframe.

### 1.2.4 Asignación de valores a las etiquetas

Antes de empezar a probar los diferentes modelos es necesario asignar un valor numérico a cada etiqueta. En nuestro caso hemos asignado a cada uno de los 10 géneros un número entre 0 y 9.

[175]:
```
from sklearn.preprocessing import LabelEncoder

generos=df_principal['label']
encoder = LabelEncoder()
generos_enc = encoder.fit_transform(generos)

# Visualizamos los valores para verificar que sea correcto
print(np.unique(generos_enc))
```

```
[0 1 2 3 4 5 6 7 8 9]
```

# 2 Entrenamiento y prueba de los diferentes modelos

## 2.1 División de datos

Comenzamos dividiendo nuestro conjunto de datos en dos subconjuntos, 70% para el de entrenamiento y 30% para el de test.

[179]:
```
from sklearn.model_selection import train_test_split

# Definimos las variables x e y
x = df_norm.drop(columns=['filename', 'label'])
y = generos_enc

# Dividimos los datos
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3,␣
 ↪stratify=y, random_state = 42)

# Comprobamos que la división sea correcta
print(len(x_train))
print(len(x_test))
```

```
6993
2997
```

## 2.2 k-Nearest Neighbors (k-NN)

Comenzamos probando el algoritmo k-Nearest Neighbors (k-NN). Con este método se clasifica un dato nuevo basándose en la cercanía a otros datos previamente clasificados. Primero se elige un valor para "k" (número de vecinos), luego se calcula la distancia entre el dato nuevo y todos los datos en el conjunto de entrenamiento. Los "k" vecinos más cercanos "votan" por su clase, y el nuevo dato se clasifica en la clase mayoritaria de sus vecinos.

Para intentar evitar que el método sobreentrene los datos (overfitting), realizaremos una validación cruzada. Esta es una técnica para evaluar el rendimiento de un modelo dividiendo el conjunto de datos en varios subconjuntos (folds). En cada iteración, se entrena el modelo con algunos de estos subconjuntos y se prueba con los restantes, rotando los subconjuntos hasta cubrir todas las combinaciones posibles. El rendimiento final se calcula como el promedio de las evaluaciones obtenidas en cada iteración, lo que permite medir la capacidad del modelo para generalizar en datos no vistos y reduce el riesgo de sobreajuste.

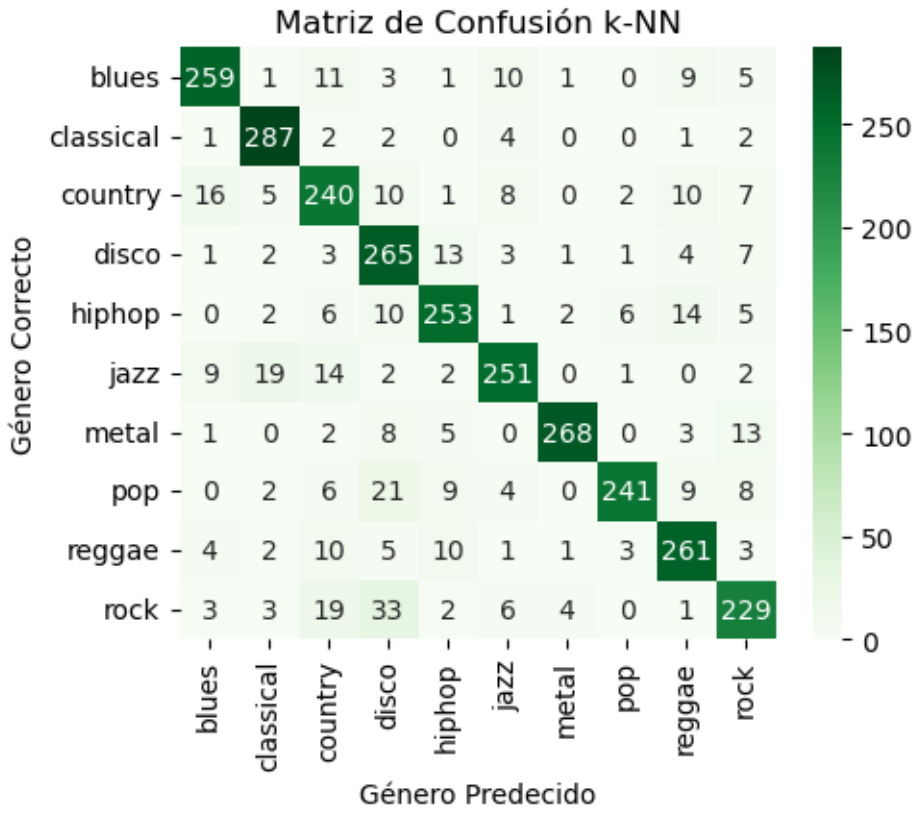
[183]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix

# Definimos el modelo k-NN
k = 10  # Número de vecinos próximos
KNN = KNeighborsClassifier(n_neighbors=k)
```

```
# Validación cruzada
folds = 10  # Número de folds
cv_scores = cross_val_score(KNN, x_train, y_train, cv=folds, scoring='accuracy')
print(f"Precisión promedio en la validación cruzada ({folds} folds): {np.
 ↪mean(cv_scores) * 100:.2f}%")

# Entrenamos el modelo
KNN.fit(x_train, y_train)

# Predicciones y reporte
y_pred_KNN = KNN.predict(x_test)
print("Reporte de clasificación:")
print(classification_report(y_test, y_pred_KNN, target_names=encoder.classes_))

# Calculamos la precisión del modelo
accuracy_KNN = accuracy_score(y_test, y_pred_KNN)
print(f"Precisión del modelo KNN con k={k}: {accuracy_KNN * 100:.2f}%")
```

```
Precisión promedio en la validación cruzada (10 folds): 84.94%
Reporte de clasificación:
              precision    recall  f1-score   support

       blues       0.88      0.86      0.87       300
   classical       0.89      0.96      0.92       299
     country       0.77      0.80      0.78       299
       disco       0.74      0.88      0.80       300
      hiphop       0.85      0.85      0.85       299
        jazz       0.87      0.84      0.85       300
       metal       0.97      0.89      0.93       300
         pop       0.95      0.80      0.87       300
      reggae       0.84      0.87      0.85       300
        rock       0.81      0.76      0.79       300

    accuracy                           0.85      2997
   macro avg       0.86      0.85      0.85      2997
weighted avg       0.86      0.85      0.85      2997

Precisión del modelo KNN con k=10: 85.22%
```

[184]:
```
# Obtenemos de la matriz de confusión
conf_matrix_KNN = confusion_matrix(y_test, y_pred_KNN)
plt.figure(figsize=(5, 4))
sns.heatmap(conf_matrix_KNN, annot=True, fmt='d', cmap='Greens',␣
 ↪xticklabels=encoder.classes_, yticklabels=encoder.classes_)
plt.xlabel('Género Predecido')
plt.ylabel('Género Correcto')
plt.title('Matriz de Confusión k-NN')
```

```
plt.show()
```




Observamos que el k-NN clasifica las pista de forma bastante satisfactoria, especialmente para 'blues', 'classical', 'metal' y 'reggae'.

### 2.3 Support Vector Machine (SVM)

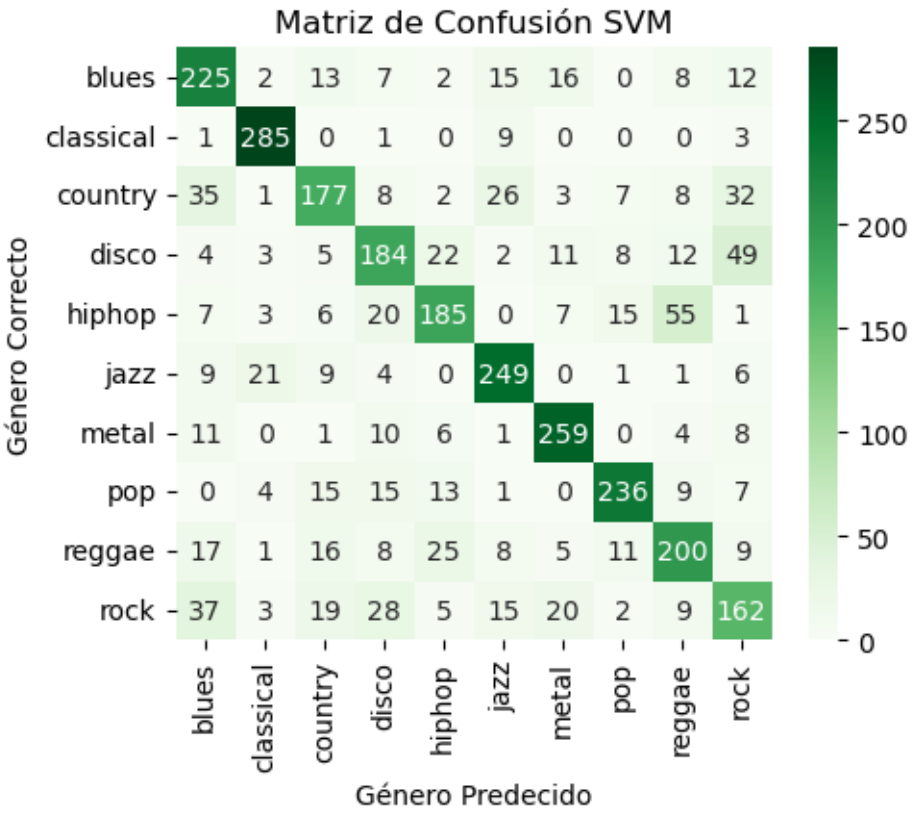
El algoritmo Support Vector Machine (SVM) clasifica datos separando las clases mediante un "hiperplano" óptimo en el espacio de características. Este hiperplano maximiza la distancia entre los puntos de datos de las diferentes clases, conocidos como "margen". SVM se enfoca en los puntos más cercanos al margen (vectores de soporte) para definir esta separación óptima.

[188]:
```
from sklearn.svm import SVC

# Definimos el modelo SVM
SVM = SVC(kernel = 'linear')

# Entrenamos el modelo
SVM.fit(x_train, y_train)

# Predicciones y reporte
```

```
y_pred_SVM = SVM.predict(x_test)
print("Reporte de clasificación:")
print(classification_report(y_test, y_pred_SVM, target_names=encoder.classes_))

# Evaluamos la precisión del modelo
accuracy_SVM = accuracy_score(y_test, y_pred_SVM)
print(f'Precisión del modelo SVM: {accuracy_SVM * 100:.2f}%')
```

```
Reporte de clasificación:
              precision    recall  f1-score   support

       blues       0.65      0.75      0.70       300
   classical       0.88      0.95      0.92       299
     country       0.68      0.59      0.63       299
       disco       0.65      0.61      0.63       300
      hiphop       0.71      0.62      0.66       299
        jazz       0.76      0.83      0.80       300
       metal       0.81      0.86      0.83       300
         pop       0.84      0.79      0.81       300
      reggae       0.65      0.67      0.66       300
        rock       0.56      0.54      0.55       300

    accuracy                           0.72      2997
   macro avg       0.72      0.72      0.72      2997
weighted avg       0.72      0.72      0.72      2997

Precisión del modelo SVM: 72.14%
```

[189]:
```
# Obtenemos de la matriz de confusión
conf_matrix_SVM = confusion_matrix(y_test, y_pred_SVM)
plt.figure(figsize=(5, 4))
sns.heatmap(conf_matrix_SVM, annot=True, fmt='d', cmap='Greens',␣
 ↪xticklabels=encoder.classes_, yticklabels=encoder.classes_)
plt.xlabel('Género Predecido')
plt.ylabel('Género Correcto')
plt.title('Matriz de Confusión SVM')
plt.show()
```




Podemos ver que la precisión del SVM es inferior a la del k-NN, clasificando mejor (otra vez) los géneros 'classical' y 'metal', mientras que 'rock' presenta el menor ratio de acierto.

### 2.4 Decision Tree

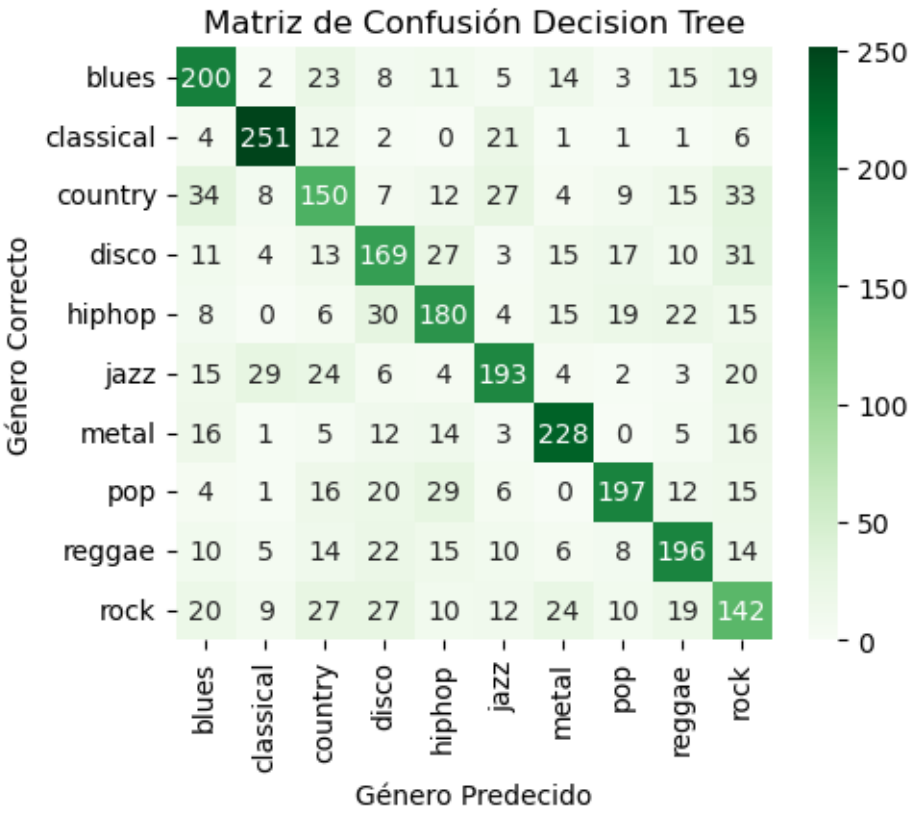
El algoritmo Decision Tree clasifica datos creando un modelo en forma de árbol, donde cada nodo representa una pregunta o condición sobre una característica específica del dato. Empezando desde la raíz, el árbol divide los datos en subconjuntos más pequeños en función de sus características, tomando decisiones en cada nodo hasta llegar a una clasificación en las hojas. Estas divisiones se eligen para maximizar la "pureza" de cada rama, de manera que los datos similares queden agrupados.

[193]:
```
from sklearn.tree import DecisionTreeClassifier

# Definimos el modelo de Decision Tree
DT = DecisionTreeClassifier(random_state=42)

# Entrenamos el modelo
DT.fit(x_train, y_train)

# Predicciones y reporte
```

```
y_pred_DT = DT.predict(x_test)
print("Reporte de clasificación:")
print(classification_report(y_test, y_pred_DT, target_names=encoder.classes_))

# Evaluamos la precisión del modelo
accuracy_DT = accuracy_score(y_test, y_pred_DT)
print(f"Precisión del modelo Decision Tree: {accuracy_DT * 100:.2f}%")
```

```
Reporte de clasificación:
              precision    recall  f1-score   support

       blues       0.62      0.67      0.64       300
   classical       0.81      0.84      0.82       299
     country       0.52      0.50      0.51       299
       disco       0.56      0.56      0.56       300
      hiphop       0.60      0.60      0.60       299
        jazz       0.68      0.64      0.66       300
       metal       0.73      0.76      0.75       300
         pop       0.74      0.66      0.70       300
      reggae       0.66      0.65      0.66       300
        rock       0.46      0.47      0.46       300

    accuracy                           0.64      2997
   macro avg       0.64      0.64      0.64      2997
weighted avg       0.64      0.64      0.64      2997

Precisión del modelo Decision Tree: 63.60%
```

[194]:
```
# Matriz de confusión
conf_matrix_DT = confusion_matrix(y_test, y_pred_DT)

# Visualización de la matriz de confusión
plt.figure(figsize=(5, 4))
sns.heatmap(conf_matrix_DT, annot=True, fmt='d', cmap='Greens',␣
 ↪xticklabels=encoder.classes_, yticklabels=encoder.classes_)
plt.xlabel('Género Predecido')
plt.ylabel('Género Correcto')
plt.title('Matriz de Confusión Decision Tree')
plt.show()
```




Con este algoritmo vemos que la precisión es bastante menor a los dos modelos anteriores, aunque de nuevo podemos ver que sigue clasificando correctamente los géneros 'classical' y 'metal'. En el resto de géneros se pierde precisión, especialmente en 'country' y 'rock'.

### 2.5 Regresión Logística

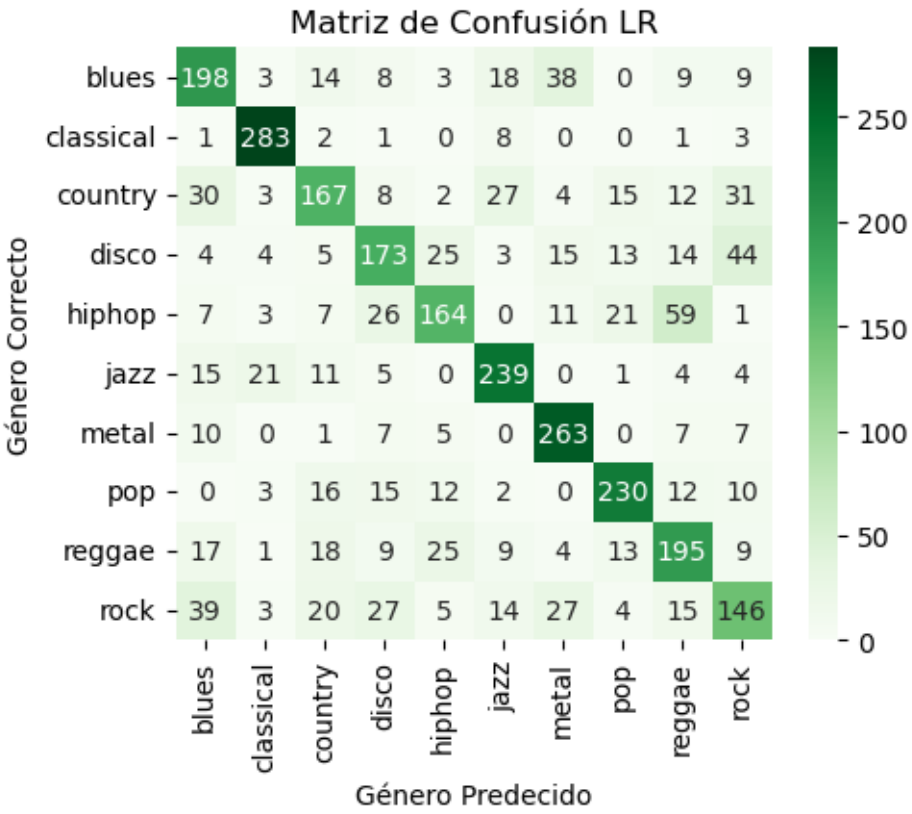
La regresión logística es un algoritmo de clasificación que estima la probabilidad de que un dato pertenezca a una clase específica. Utiliza una función sigmoide para transformar una combinación lineal de las características del dato en un valor entre 0 y 1, que representa esta probabilidad. Si la probabilidad es mayor que un umbral (habitualmente 0.5), el dato se clasifica en una clase; de lo contrario, en la otra.

[198]:
```
from sklearn.linear_model import LogisticRegression

# Definimos el modelo de Árbol de Decisión
LR = LogisticRegression(max_iter=10000,random_state=42)

# Entrenamos el modelo
LR.fit(x_train, y_train)

# Predicciones y reporte
```

```
y_pred_LR = LR.predict(x_test)
print("Reporte de clasificación:")
print(classification_report(y_test, y_pred_LR, target_names=encoder.classes_))

# Evaluamos la precisión del modelo
accuracy_LR = accuracy_score(y_test, y_pred_LR)
print(f"Precisión del modelo Regresión Logística: {accuracy_LR * 100:.2f}%")
```

```
Reporte de clasificación:
              precision    recall  f1-score   support

       blues       0.62      0.66      0.64       300
   classical       0.87      0.95      0.91       299
     country       0.64      0.56      0.60       299
       disco       0.62      0.58      0.60       300
      hiphop       0.68      0.55      0.61       299
        jazz       0.75      0.80      0.77       300
       metal       0.73      0.88      0.79       300
         pop       0.77      0.77      0.77       300
      reggae       0.59      0.65      0.62       300
        rock       0.55      0.49      0.52       300

    accuracy                           0.69      2997
   macro avg       0.68      0.69      0.68      2997
weighted avg       0.68      0.69      0.68      2997

Precisión del modelo Regresión Logística: 68.67%
```

[199]:
```
# Visualizamos de la matriz de confusión
conf_matrix_LR = confusion_matrix(y_test, y_pred_LR)
plt.figure(figsize=(5, 4))
sns.heatmap(conf_matrix_LR, annot=True, fmt='d', cmap='Greens',␣
 ↪xticklabels=encoder.classes_, yticklabels=encoder.classes_)
plt.xlabel('Género Predecido')
plt.ylabel('Género Correcto')
plt.title('Matriz de Confusión LR')
plt.show()
```




Observamos que este modelo predice relativamente bien los géneros (con una precisión de casi el 70%), pese a que este método tiende a funcionar mejor cuando el problema es binario, es decir, donde el número de conjuntos es dos.

### 2.6 Red Neuronal Artificial (ANN)

Las Redes Neuronales Artificiales (ANN) son modelos de machine learning inspirados en el funcionamiento del cerebro humano. Están compuestas por capas de "neuronas" artificiales que procesan y transforman datos a través de conexiones con "pesos" ajustables. La estructura básica consta de tres tipos de capas:

1. `Capa de entrada: reciben las características de los datos.`
2. `Capas ocultas: procesan la información mediante funciones de activación, aplicando transforma`
3. `Capa de salida: genera la predicción final.`

En nuestro caso, después de probar con diferentes capas y densidades de cada una, hemos elegido 6 capas. Entre ellas tenemos algunas densas, en las que hemos reducido la densidad paulatinamente, combinadas con capas de "Dropout", que apagan neuronas aleatoriamente ayudando así a reducir el overfitting. Finalmente, ajustamos la capa de salida para que tenga el mismo número de neuronas que géneros.

[203]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# Definimos de la red neuronal
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(x_train.shape[1],)))
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dense(len(np.unique(y)), activation='softmax'))  # Capa de salida

# Compilamos el modelo
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['accuracy'])

# Entrenamos el modelo
hist_ANN = model.fit(x_train, y_train, epochs=200, batch_size=64,␣
 ↪validation_split=0.3)
```

```
Epoch 1/200
```

```
C:\Users\USUARIO\AppData\Roaming\Python\Python312\site-
packages\keras\src\layers\core\dense.py:87: UserWarning:

Do not pass an `input_shape`/`input_dim` argument to a layer. When using
Sequential models, prefer using an `Input(shape)` object as the first layer in
the model instead.
```

```
77/77 ---------------------- 1s 4ms/step -
accuracy: 0.1767 - loss: 2.2087 - val_accuracy: 0.4209 - val_loss: 1.5735
Epoch 2/200
77/77 ---------------------- 0s 2ms/step -
accuracy: 0.3796 - loss: 1.6501 - val_accuracy: 0.5076 - val_loss: 1.3683
Epoch 3/200
77/77 ---------------------- 0s 2ms/step -
accuracy: 0.4558 - loss: 1.4522 - val_accuracy: 0.5019 - val_loss: 1.3001
Epoch 4/200
77/77 ---------------------- 0s 2ms/step -
accuracy: 0.4789 - loss: 1.3847 - val_accuracy: 0.5596 - val_loss: 1.2042
Epoch 5/200
77/77 ---------------------- 0s 2ms/step -
accuracy: 0.5052 - loss: 1.3102 - val_accuracy: 0.5963 - val_loss: 1.1212
Epoch 6/200
77/77 ---------------------- 0s 2ms/step -
accuracy: 0.5381 - loss: 1.2239 - val_accuracy: 0.6068 - val_loss: 1.0957
Epoch 7/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.5557 - loss: 1.1917 - val_accuracy: 0.6149 - val_loss: 1.0759
Epoch 8/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.5848 - loss: 1.1380 - val_accuracy: 0.6273 - val_loss: 1.0320
Epoch 9/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.5980 - loss: 1.1102 - val_accuracy: 0.6587 - val_loss: 0.9806
Epoch 10/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6056 - loss: 1.0719 - val_accuracy: 0.6368 - val_loss: 1.0145
Epoch 11/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6179 - loss: 1.0446 - val_accuracy: 0.6511 - val_loss: 0.9734
Epoch 12/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6155 - loss: 1.0462 - val_accuracy: 0.6778 - val_loss: 0.9225
Epoch 13/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6328 - loss: 1.0322 - val_accuracy: 0.6821 - val_loss: 0.9141
Epoch 14/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6411 - loss: 0.9990 - val_accuracy: 0.6773 - val_loss: 0.8957
Epoch 15/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6672 - loss: 0.9446 - val_accuracy: 0.6659 - val_loss: 0.9311
Epoch 16/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6377 - loss: 0.9663 - val_accuracy: 0.6611 - val_loss: 0.9410
Epoch 17/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6547 - loss: 0.9635 - val_accuracy: 0.6868 - val_loss: 0.8867
Epoch 18/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6582 - loss: 0.9641 - val_accuracy: 0.7007 - val_loss: 0.8578
Epoch 19/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6704 - loss: 0.9310 - val_accuracy: 0.6983 - val_loss: 0.8599
Epoch 20/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6886 - loss: 0.8731 - val_accuracy: 0.6935 - val_loss: 0.8563
Epoch 21/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6768 - loss: 0.8925 - val_accuracy: 0.7102 - val_loss: 0.8111
Epoch 22/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6882 - loss: 0.8723 - val_accuracy: 0.7088 - val_loss: 0.8254
Epoch 23/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6964 - loss: 0.8679 - val_accuracy: 0.7154 - val_loss: 0.8052
Epoch 24/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6999 - loss: 0.8371 - val_accuracy: 0.7336 - val_loss: 0.8041
Epoch 25/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7015 - loss: 0.8404 - val_accuracy: 0.7293 - val_loss: 0.7819
Epoch 26/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6985 - loss: 0.8494 - val_accuracy: 0.7235 - val_loss: 0.7913
Epoch 27/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.6967 - loss: 0.8612 - val_accuracy: 0.7212 - val_loss: 0.7896
Epoch 28/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7092 - loss: 0.8209 - val_accuracy: 0.7226 - val_loss: 0.7842
Epoch 29/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7070 - loss: 0.8092 - val_accuracy: 0.7302 - val_loss: 0.7654
Epoch 30/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7265 - loss: 0.7830 - val_accuracy: 0.7140 - val_loss: 0.7877
Epoch 31/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7080 - loss: 0.8113 - val_accuracy: 0.7374 - val_loss: 0.7439
Epoch 32/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7208 - loss: 0.7846 - val_accuracy: 0.7374 - val_loss: 0.7655
Epoch 33/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7193 - loss: 0.7791 - val_accuracy: 0.7469 - val_loss: 0.7398
Epoch 34/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7339 - loss: 0.7319 - val_accuracy: 0.7378 - val_loss: 0.7435
Epoch 35/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7262 - loss: 0.7617 - val_accuracy: 0.7407 - val_loss: 0.7407
Epoch 36/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7385 - loss: 0.7383 - val_accuracy: 0.7545 - val_loss: 0.7148
Epoch 37/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7302 - loss: 0.7299 - val_accuracy: 0.7417 - val_loss: 0.7321
Epoch 38/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7297 - loss: 0.7518 - val_accuracy: 0.7526 - val_loss: 0.7116
Epoch 39/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7370 - loss: 0.7568 - val_accuracy: 0.7345 - val_loss: 0.7516
Epoch 40/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7297 - loss: 0.7270 - val_accuracy: 0.7526 - val_loss: 0.7026
Epoch 41/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7437 - loss: 0.7215 - val_accuracy: 0.7612 - val_loss: 0.6936
Epoch 42/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7408 - loss: 0.7169 - val_accuracy: 0.7574 - val_loss: 0.6892
Epoch 43/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7439 - loss: 0.7214 - val_accuracy: 0.7579 - val_loss: 0.6906
Epoch 44/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7549 - loss: 0.6992 - val_accuracy: 0.7622 - val_loss: 0.6727
Epoch 45/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7504 - loss: 0.7021 - val_accuracy: 0.7593 - val_loss: 0.6781
Epoch 46/200
77/77 -------------------- 0s 3ms/step -
accuracy: 0.7564 - loss: 0.6761 - val_accuracy: 0.7650 - val_loss: 0.6563
Epoch 47/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7538 - loss: 0.6786 - val_accuracy: 0.7617 - val_loss: 0.6822
Epoch 48/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7436 - loss: 0.7009 - val_accuracy: 0.7655 - val_loss: 0.6711
Epoch 49/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7634 - loss: 0.6712 - val_accuracy: 0.7650 - val_loss: 0.6638
Epoch 50/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7446 - loss: 0.7121 - val_accuracy: 0.7731 - val_loss: 0.6575
Epoch 51/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7652 - loss: 0.6588 - val_accuracy: 0.7760 - val_loss: 0.6647
Epoch 52/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7662 - loss: 0.6459 - val_accuracy: 0.7798 - val_loss: 0.6377
Epoch 53/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7723 - loss: 0.6584 - val_accuracy: 0.7726 - val_loss: 0.6796
Epoch 54/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7625 - loss: 0.6587 - val_accuracy: 0.7707 - val_loss: 0.6628
Epoch 55/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7678 - loss: 0.6442 - val_accuracy: 0.7788 - val_loss: 0.6370
Epoch 56/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7580 - loss: 0.6624 - val_accuracy: 0.7836 - val_loss: 0.6374
Epoch 57/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7582 - loss: 0.6609 - val_accuracy: 0.7750 - val_loss: 0.6374
Epoch 58/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7760 - loss: 0.6312 - val_accuracy: 0.7917 - val_loss: 0.6242
Epoch 59/200
77/77 -------------------- 0s 3ms/step -
accuracy: 0.7842 - loss: 0.6170 - val_accuracy: 0.7822 - val_loss: 0.6319
Epoch 60/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7814 - loss: 0.6137 - val_accuracy: 0.7765 - val_loss: 0.6500
Epoch 61/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7780 - loss: 0.6129 - val_accuracy: 0.7784 - val_loss: 0.6545
Epoch 62/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7792 - loss: 0.6256 - val_accuracy: 0.7893 - val_loss: 0.6191
Epoch 63/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7735 - loss: 0.6287 - val_accuracy: 0.7960 - val_loss: 0.6148
Epoch 64/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7823 - loss: 0.5946 - val_accuracy: 0.7841 - val_loss: 0.6341
Epoch 65/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7847 - loss: 0.6152 - val_accuracy: 0.7765 - val_loss: 0.6527
Epoch 66/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7749 - loss: 0.6149 - val_accuracy: 0.7969 - val_loss: 0.6054
Epoch 67/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7718 - loss: 0.6087 - val_accuracy: 0.7912 - val_loss: 0.6104
Epoch 68/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7897 - loss: 0.5776 - val_accuracy: 0.7912 - val_loss: 0.6180
Epoch 69/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7850 - loss: 0.6199 - val_accuracy: 0.7865 - val_loss: 0.6052
Epoch 70/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7918 - loss: 0.5740 - val_accuracy: 0.7960 - val_loss: 0.5884
Epoch 71/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7804 - loss: 0.6037 - val_accuracy: 0.7984 - val_loss: 0.6115
Epoch 72/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7838 - loss: 0.5917 - val_accuracy: 0.7974 - val_loss: 0.5959
Epoch 73/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7915 - loss: 0.5950 - val_accuracy: 0.8055 - val_loss: 0.5862
Epoch 74/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7988 - loss: 0.5775 - val_accuracy: 0.7860 - val_loss: 0.6223
Epoch 75/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7974 - loss: 0.5646 - val_accuracy: 0.7950 - val_loss: 0.5881
Epoch 76/200
77/77 -------------------- 0s 3ms/step -
accuracy: 0.7917 - loss: 0.5767 - val_accuracy: 0.7960 - val_loss: 0.6032
Epoch 77/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7993 - loss: 0.5606 - val_accuracy: 0.7984 - val_loss: 0.5727
Epoch 78/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8034 - loss: 0.5568 - val_accuracy: 0.8132 - val_loss: 0.5736
Epoch 79/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8043 - loss: 0.5558 - val_accuracy: 0.8089 - val_loss: 0.5771
Epoch 80/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7948 - loss: 0.5707 - val_accuracy: 0.8098 - val_loss: 0.5722
Epoch 81/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8143 - loss: 0.5324 - val_accuracy: 0.7984 - val_loss: 0.5971
Epoch 82/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8000 - loss: 0.5554 - val_accuracy: 0.7898 - val_loss: 0.6121
Epoch 83/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8028 - loss: 0.5595 - val_accuracy: 0.8122 - val_loss: 0.5647
Epoch 84/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8061 - loss: 0.5424 - val_accuracy: 0.8051 - val_loss: 0.5771
Epoch 85/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.7964 - loss: 0.5486 - val_accuracy: 0.8079 - val_loss: 0.5778
Epoch 86/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8109 - loss: 0.5378 - val_accuracy: 0.8179 - val_loss: 0.5697
Epoch 87/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8169 - loss: 0.5211 - val_accuracy: 0.8017 - val_loss: 0.5793
Epoch 88/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8250 - loss: 0.4936 - val_accuracy: 0.8170 - val_loss: 0.5581
Epoch 89/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8127 - loss: 0.5334 - val_accuracy: 0.8155 - val_loss: 0.5475
Epoch 90/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8181 - loss: 0.5075 - val_accuracy: 0.8136 - val_loss: 0.5590
Epoch 91/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8066 - loss: 0.5454 - val_accuracy: 0.8255 - val_loss: 0.5336
Epoch 92/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8059 - loss: 0.5482 - val_accuracy: 0.8132 - val_loss: 0.5683
Epoch 93/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8172 - loss: 0.5096 - val_accuracy: 0.8112 - val_loss: 0.5758
Epoch 94/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8056 - loss: 0.5276 - val_accuracy: 0.8160 - val_loss: 0.5686
Epoch 95/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8198 - loss: 0.5117 - val_accuracy: 0.8136 - val_loss: 0.5449
Epoch 96/200
77/77 -------------------- 0s 3ms/step -
accuracy: 0.8129 - loss: 0.5211 - val_accuracy: 0.8284 - val_loss: 0.5458
Epoch 97/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8193 - loss: 0.4939 - val_accuracy: 0.8222 - val_loss: 0.5488
Epoch 98/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8366 - loss: 0.4804 - val_accuracy: 0.8203 - val_loss: 0.5422
Epoch 99/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8050 - loss: 0.5325 - val_accuracy: 0.8236 - val_loss: 0.5384
Epoch 100/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8282 - loss: 0.4863 - val_accuracy: 0.8298 - val_loss: 0.5228
Epoch 101/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8148 - loss: 0.5022 - val_accuracy: 0.8284 - val_loss: 0.5375
Epoch 102/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8194 - loss: 0.4861 - val_accuracy: 0.8160 - val_loss: 0.5600
Epoch 103/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8261 - loss: 0.4862 - val_accuracy: 0.8227 - val_loss: 0.5495
Epoch 104/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8122 - loss: 0.5042 - val_accuracy: 0.8208 - val_loss: 0.5637
Epoch 105/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8192 - loss: 0.4830 - val_accuracy: 0.8217 - val_loss: 0.5593
Epoch 106/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8188 - loss: 0.4889 - val_accuracy: 0.8232 - val_loss: 0.5334
Epoch 107/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8178 - loss: 0.4954 - val_accuracy: 0.8170 - val_loss: 0.5456
Epoch 108/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8307 - loss: 0.4930 - val_accuracy: 0.8356 - val_loss: 0.5259
Epoch 109/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8288 - loss: 0.4667 - val_accuracy: 0.8275 - val_loss: 0.5193
Epoch 110/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8252 - loss: 0.4698 - val_accuracy: 0.8370 - val_loss: 0.5116
Epoch 111/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8291 - loss: 0.4808 - val_accuracy: 0.8260 - val_loss: 0.5244
Epoch 112/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8299 - loss: 0.4800 - val_accuracy: 0.8279 - val_loss: 0.5221
Epoch 113/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8148 - loss: 0.5060 - val_accuracy: 0.8346 - val_loss: 0.5346
Epoch 114/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8351 - loss: 0.4631 - val_accuracy: 0.8427 - val_loss: 0.5094
Epoch 115/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8285 - loss: 0.4781 - val_accuracy: 0.8270 - val_loss: 0.5229
Epoch 116/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8374 - loss: 0.4737 - val_accuracy: 0.8232 - val_loss: 0.5290
Epoch 117/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8434 - loss: 0.4425 - val_accuracy: 0.8389 - val_loss: 0.5154
Epoch 118/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8442 - loss: 0.4383 - val_accuracy: 0.8308 - val_loss: 0.5157
Epoch 119/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8278 - loss: 0.4668 - val_accuracy: 0.8294 - val_loss: 0.5229
Epoch 120/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8248 - loss: 0.4873 - val_accuracy: 0.8360 - val_loss: 0.5035
Epoch 121/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8371 - loss: 0.4457 - val_accuracy: 0.8332 - val_loss: 0.5245
Epoch 122/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8220 - loss: 0.4894 - val_accuracy: 0.8346 - val_loss: 0.5099
Epoch 123/200
77/77 -------------------- 0s 3ms/step -
accuracy: 0.8379 - loss: 0.4469 - val_accuracy: 0.8432 - val_loss: 0.5013
Epoch 124/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8388 - loss: 0.4478 - val_accuracy: 0.8413 - val_loss: 0.5102
Epoch 125/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8404 - loss: 0.4461 - val_accuracy: 0.8389 - val_loss: 0.5114
Epoch 126/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8460 - loss: 0.4279 - val_accuracy: 0.8384 - val_loss: 0.5002
Epoch 127/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8402 - loss: 0.4372 - val_accuracy: 0.8494 - val_loss: 0.5032
Epoch 128/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8536 - loss: 0.4104 - val_accuracy: 0.8389 - val_loss: 0.5143
Epoch 129/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8322 - loss: 0.4500 - val_accuracy: 0.8475 - val_loss: 0.4951
Epoch 130/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8515 - loss: 0.4297 - val_accuracy: 0.8384 - val_loss: 0.5107
Epoch 131/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8470 - loss: 0.4007 - val_accuracy: 0.8356 - val_loss: 0.5101
Epoch 132/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8478 - loss: 0.4275 - val_accuracy: 0.8408 - val_loss: 0.5115
Epoch 133/200
77/77 -------------------- 0s 3ms/step -
accuracy: 0.8479 - loss: 0.4166 - val_accuracy: 0.8384 - val_loss: 0.5311
Epoch 134/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8445 - loss: 0.4250 - val_accuracy: 0.8418 - val_loss: 0.5010
Epoch 135/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8494 - loss: 0.4266 - val_accuracy: 0.8298 - val_loss: 0.5307
Epoch 136/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8420 - loss: 0.4364 - val_accuracy: 0.8422 - val_loss: 0.5037
Epoch 137/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8392 - loss: 0.4284 - val_accuracy: 0.8465 - val_loss: 0.4994
Epoch 138/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8386 - loss: 0.4473 - val_accuracy: 0.8418 - val_loss: 0.4938
Epoch 139/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8522 - loss: 0.4201 - val_accuracy: 0.8451 - val_loss: 0.4824
Epoch 140/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8475 - loss: 0.4217 - val_accuracy: 0.8413 - val_loss: 0.5088
Epoch 141/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8581 - loss: 0.3843 - val_accuracy: 0.8403 - val_loss: 0.5011
Epoch 142/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8577 - loss: 0.3937 - val_accuracy: 0.8484 - val_loss: 0.4867
Epoch 143/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8509 - loss: 0.4148 - val_accuracy: 0.8379 - val_loss: 0.4975
Epoch 144/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8437 - loss: 0.4472 - val_accuracy: 0.8456 - val_loss: 0.4991
Epoch 145/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8434 - loss: 0.4193 - val_accuracy: 0.8537 - val_loss: 0.4839
Epoch 146/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8467 - loss: 0.4318 - val_accuracy: 0.8379 - val_loss: 0.5116
Epoch 147/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8620 - loss: 0.3874 - val_accuracy: 0.8484 - val_loss: 0.4837
Epoch 148/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8653 - loss: 0.3719 - val_accuracy: 0.8546 - val_loss: 0.4928
Epoch 149/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8547 - loss: 0.3947 - val_accuracy: 0.8527 - val_loss: 0.4928
Epoch 150/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8451 - loss: 0.4258 - val_accuracy: 0.8317 - val_loss: 0.5218
Epoch 151/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8589 - loss: 0.3924 - val_accuracy: 0.8441 - val_loss: 0.4843
Epoch 152/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8611 - loss: 0.3826 - val_accuracy: 0.8489 - val_loss: 0.4983
Epoch 153/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8487 - loss: 0.4246 - val_accuracy: 0.8432 - val_loss: 0.5011
Epoch 154/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8574 - loss: 0.4044 - val_accuracy: 0.8437 - val_loss: 0.4984
Epoch 155/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8625 - loss: 0.3840 - val_accuracy: 0.8484 - val_loss: 0.4751
Epoch 156/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8387 - loss: 0.4363 - val_accuracy: 0.8503 - val_loss: 0.4818
Epoch 157/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8556 - loss: 0.3953 - val_accuracy: 0.8432 - val_loss: 0.5044
Epoch 158/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8605 - loss: 0.3948 - val_accuracy: 0.8508 - val_loss: 0.4794
Epoch 159/200
77/77 -------------------- 0s 3ms/step -
accuracy: 0.8563 - loss: 0.4042 - val_accuracy: 0.8551 - val_loss: 0.4704
Epoch 160/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8672 - loss: 0.3867 - val_accuracy: 0.8480 - val_loss: 0.4901
Epoch 161/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8620 - loss: 0.3698 - val_accuracy: 0.8518 - val_loss: 0.4726
Epoch 162/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8593 - loss: 0.3917 - val_accuracy: 0.8484 - val_loss: 0.4798
Epoch 163/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8603 - loss: 0.4001 - val_accuracy: 0.8446 - val_loss: 0.4981
Epoch 164/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8602 - loss: 0.3906 - val_accuracy: 0.8460 - val_loss: 0.4826
Epoch 165/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8650 - loss: 0.3668 - val_accuracy: 0.8456 - val_loss: 0.4838
Epoch 166/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8682 - loss: 0.3729 - val_accuracy: 0.8551 - val_loss: 0.4852
Epoch 167/200
```

```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8719 - loss: 0.3612 - val_accuracy: 0.8570 - val_loss: 0.4693
Epoch 168/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8787 - loss: 0.3542 - val_accuracy: 0.8484 - val_loss: 0.4813
Epoch 169/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8606 - loss: 0.3865 - val_accuracy: 0.8494 - val_loss: 0.5005
Epoch 170/200
77/77 -------------------- 0s 3ms/step -
accuracy: 0.8516 - loss: 0.3950 - val_accuracy: 0.8522 - val_loss: 0.4843
Epoch 171/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8615 - loss: 0.3734 - val_accuracy: 0.8480 - val_loss: 0.4821
Epoch 172/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8783 - loss: 0.3753 - val_accuracy: 0.8484 - val_loss: 0.4779
Epoch 173/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8687 - loss: 0.3673 - val_accuracy: 0.8389 - val_loss: 0.5177
Epoch 174/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8730 - loss: 0.3635 - val_accuracy: 0.8494 - val_loss: 0.4803
Epoch 175/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8594 - loss: 0.3896 - val_accuracy: 0.8489 - val_loss: 0.5110
Epoch 176/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8591 - loss: 0.3958 - val_accuracy: 0.8494 - val_loss: 0.5119
Epoch 177/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8486 - loss: 0.4068 - val_accuracy: 0.8570 - val_loss: 0.4662
Epoch 178/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8621 - loss: 0.3817 - val_accuracy: 0.8489 - val_loss: 0.4871
Epoch 179/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8673 - loss: 0.3706 - val_accuracy: 0.8489 - val_loss: 0.5051
Epoch 180/200
77/77 -------------------- 0s 3ms/step -
accuracy: 0.8765 - loss: 0.3531 - val_accuracy: 0.8580 - val_loss: 0.4730
Epoch 181/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8653 - loss: 0.3665 - val_accuracy: 0.8470 - val_loss: 0.5020
Epoch 182/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8613 - loss: 0.3858 - val_accuracy: 0.8599 - val_loss: 0.4701
Epoch 183/200
```

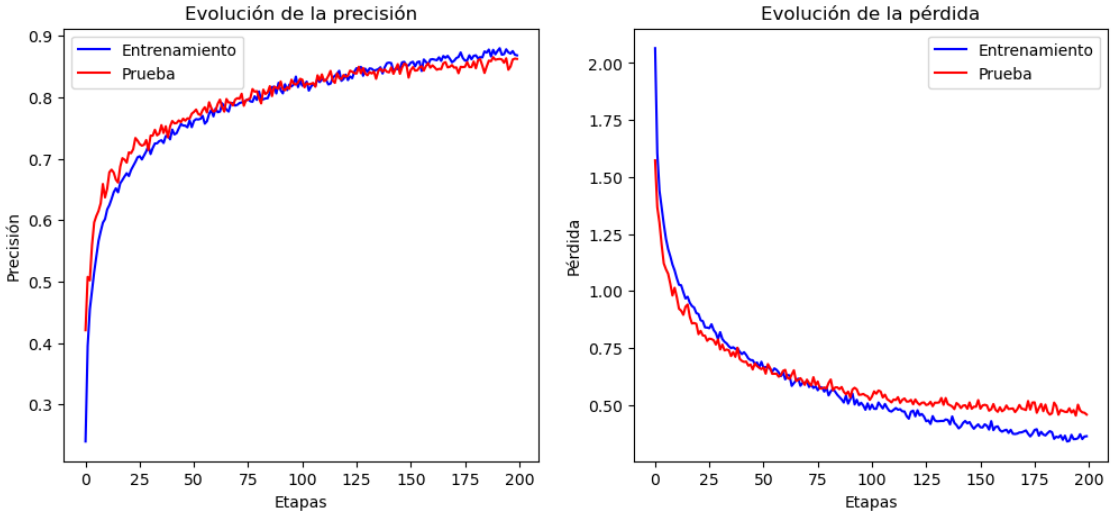
```
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8753 - loss: 0.3561 - val_accuracy: 0.8618 - val_loss: 0.4583
Epoch 184/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8591 - loss: 0.3946 - val_accuracy: 0.8513 - val_loss: 0.4638
Epoch 185/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8666 - loss: 0.3684 - val_accuracy: 0.8394 - val_loss: 0.5108
Epoch 186/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8682 - loss: 0.3777 - val_accuracy: 0.8499 - val_loss: 0.4961
Epoch 187/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8814 - loss: 0.3498 - val_accuracy: 0.8594 - val_loss: 0.4577
Epoch 188/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8635 - loss: 0.3647 - val_accuracy: 0.8584 - val_loss: 0.4786
Epoch 189/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8761 - loss: 0.3555 - val_accuracy: 0.8651 - val_loss: 0.4748
Epoch 190/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8672 - loss: 0.3759 - val_accuracy: 0.8608 - val_loss: 0.4704
Epoch 191/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8723 - loss: 0.3340 - val_accuracy: 0.8618 - val_loss: 0.4694
Epoch 192/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8847 - loss: 0.3331 - val_accuracy: 0.8622 - val_loss: 0.4858
Epoch 193/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8588 - loss: 0.3865 - val_accuracy: 0.8613 - val_loss: 0.4669
Epoch 194/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8665 - loss: 0.3561 - val_accuracy: 0.8561 - val_loss: 0.4777
Epoch 195/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8756 - loss: 0.3620 - val_accuracy: 0.8632 - val_loss: 0.4537
Epoch 196/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8754 - loss: 0.3528 - val_accuracy: 0.8446 - val_loss: 0.5017
Epoch 197/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8668 - loss: 0.3773 - val_accuracy: 0.8499 - val_loss: 0.4763
Epoch 198/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8738 - loss: 0.3538 - val_accuracy: 0.8613 - val_loss: 0.4681
Epoch 199/200
```

```
77/77 -------------------- 0s 3ms/step -
accuracy: 0.8638 - loss: 0.3726 - val_accuracy: 0.8627 - val_loss: 0.4662
Epoch 200/200
77/77 -------------------- 0s 2ms/step -
accuracy: 0.8743 - loss: 0.3484 - val_accuracy: 0.8622 - val_loss: 0.4583
```

Después de varias pruebas, variando la densidad de cada capa, el número de capas, el número de etapas (epochs) y el tamaño de la muestra de cada iteración (batch_size), concluimos los valores que mejores resultados predicen son los anteriormente mostrados. Para comprobar que la red funciona correctamente representamos la evolución de la precisión y la pérdida a lo largo de las etapas.

[205]:
```
# Graficamos la evolución de precisión y la pérdida
plt.figure()
plt.figure(figsize=(12, 5))

# Gráfica de Precisión
plt.subplot(1, 2, 1)
plt.plot(hist_ANN.history['accuracy'], label='Entrenamiento', color='blue')
plt.plot(hist_ANN.history['val_accuracy'], label='Prueba', color='red')
plt.xlabel('Etapas')
plt.ylabel('Precisión')
plt.legend()
plt.title('Evolución de la precisión')

# Gráfica de Pérdida
plt.subplot(1, 2, 2)
plt.plot(hist_ANN.history['loss'], label='Entrenamiento', color='blue')
plt.plot(hist_ANN.history['val_loss'], label='Prueba', color='red')
plt.xlabel('Etapas')
plt.ylabel('Pérdida')
plt.legend()
plt.title('Evolución de la pérdida')
plt.show()
plt.close()
```

```
<Figure size 640x480 with 0 Axes>
```



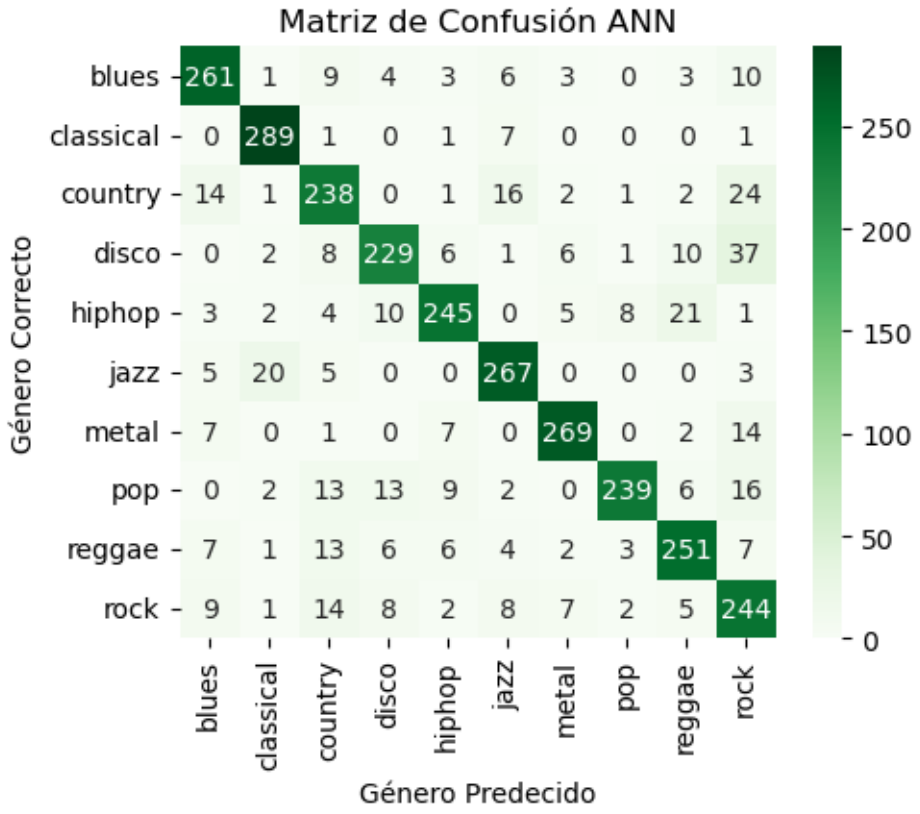

Como podemos observar en las figuras la evolución de los conjuntos de entrenamiento y prueba es muy similar, tanto para la precisión como para la pérdida. Cabe destacar que ambas curvas se empiezan a separar a medida que nos acercamos a las 200 etapas. Además, vemos que la precisión tampoco aumenta por lo que hemos decidido detener el entrenamiento en este punto.

Finalmente, al igual que con los modelos anteriores, presentamos el reporte de clasificación y la matriz de confusión.

[208]:
```
# Predicciones y reporte
y_pred_ANN = np.argmax(model.predict(x_test), axis=1)
print("Reporte de clasificación:")
print(classification_report(y_test, y_pred_ANN, target_names=encoder.classes_))

# Evaluamos la precisión del modelo
accuracy_ANN = accuracy_score(y_test, y_pred_ANN)
print(f"Precisión del modelo ANN: {accuracy_ANN * 100:.2f}%")

# Visualizamos de la matriz de confusión
conf_matrix_ANN = confusion_matrix(y_test, y_pred_ANN)
plt.figure(figsize=(5, 4))
sns.heatmap(conf_matrix_ANN, annot=True, fmt='d', cmap='Greens',␣
 ↪xticklabels=encoder.classes_, yticklabels=encoder.classes_)
plt.xlabel('Género Predecido')
plt.ylabel('Género Correcto')
plt.title('Matriz de Confusión ANN')
plt.show()
```

```
94/94 -------------------- 0s 988us/step
Reporte de clasificación:
              precision    recall  f1-score   support
```

```
              precision    recall  f1-score   support

       blues       0.85      0.87      0.86       300
   classical       0.91      0.97      0.94       299
     country       0.78      0.80      0.79       299
       disco       0.85      0.76      0.80       300
     hiphop        0.88      0.82      0.85       299
        jazz       0.86      0.89      0.87       300
       metal       0.91      0.90      0.91       300
         pop       0.94      0.80      0.86       300
      reggae       0.84      0.84      0.84       300
        rock       0.68      0.81      0.74       300

    accuracy                           0.84      2997
   macro avg       0.85      0.84      0.85      2997
weighted avg       0.85      0.84      0.85      2997

Precisión del modelo ANN: 84.48%
```




Vemos que esta red neuronal es uno de los modelos que mejor predicen los géneros (junto con el k-NN). Nuevamente los géneros mejor clasificados son “classical” y “metal”.

## 2.7 Red Neuronal Convolucional (CNN)

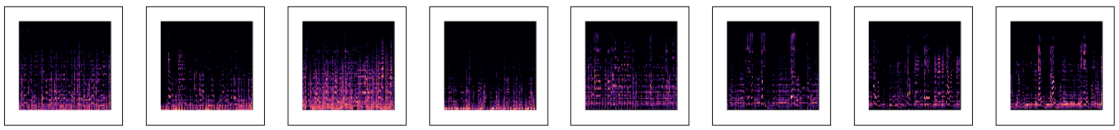
Las Redes Neuronales Convolucionales (CNN) son redes diseñadas para procesar datos con una estructura de tipo cuadrícula, como imágenes. Aprovechan las características espaciales de estos datos para aprender patrones locales mediante convoluciones. Al igual que en las ANN tenemos diferentes tipos de capas:

1. Capas de Convolución: Aplican filtros que recorren la imagen y extraen características locale
2. Capas Pooling: Reducen la dimensionalidad de las características extraídas, preservando la in
3. Capas Densas: Estas capas conectan todas las características extraídas para realizar la clasi

Como hemos visto anteriormente, podemos crear un Mel espectrograma a partir de las pistas de audio. Sin embargo, la base de datos incluye ya las imágenes de estos espectrogramas, por lo que, para acortar el tiempo de cómputo, las cargamos directamente de las carpetas.

[213]:
```
import os
from keras.preprocessing import image

# Creamos una función para cargar las imágenes y sus etiquetas
def load_images_from_path(path, label):
    images = []
    labels = []

    for file in os.listdir(path):
        img_path = os.path.join(path, file)
        # Cargamos la imagen y la redimensionamos
        img = image.load_img(img_path, target_size=(128, 128))  # Ajustamos el␣
↪tamaño de la imagen
        img_array = image.img_to_array(img)  # Convertimos la imagen a un array
        images.append(img_array)
        labels.append(label)  # Asignamos la etiqueta correspondiente
    return images, labels

# Creamos una función para mostrar las imágenes
def show_images(images):
    fig, axes = plt.subplots(1, 8, figsize=(20, 20), subplot_kw={'xticks': [],␣
↪'yticks': []})

    for i, ax in enumerate(axes.flat):
        if i < len(images):
            ax.imshow(images[i] / 255.0)  # Normalizamos la imagen para␣
↪visualizarla
        else:
            ax.axis('off')

    plt.show()
```

[214]:
```
# Lista de géneros musicales
genres =␣
 ↪['blues','classical','country','disco','hiphop','jazz','metal','pop','reggae','rock']

# Ruta base
base_path = 'Data/images_original'

# Inicializamos listas para almacenar imágenes y etiquetas
x = []
y = []

# Creamos un blucle para cargar imágenes de cada carpeta
for genre in genres:
    genre_path = os.path.join(base_path, genre)  # Construimos la ruta para cada␣
 ↪género
    if os.path.exists(genre_path):  # Verificamos si la ruta existe
        images, labels = load_images_from_path(genre_path, genre)  # Cargamos␣
 ↪imágenes y etiquetas
        x.extend(images)  # Agregamos las imágenes a la lista principal
        y.extend(labels)  # Agregamos las etiquetas a la lista principal
    else:
        print(f"Advertencia: La ruta '{genre_path}' no existe.")

# Convertimos a arrays de NumPy
x = np.array(x)
y = np.array(y)

# Asignamos valores numéricos a las etiquetas
y_enc = encoder.fit_transform(y)

# Mostramos algunas imágenes
show_images(x[:8])
```



Al igual que con los anteriores modelos, dividimos nuestro conjunto de imágenes en dos subconjuntos: entrenamiento y prueba.

[216]:
```
from tensorflow.keras.utils import to_categorical

x_train_CNN, x_test_CNN, y_train_CNN, y_test_CNN = train_test_split(x, y_enc,␣
 ↪stratify=y, test_size=0.3, random_state=42)
```

```
# Normalizamos
x_train_norm = np.array(x_train_CNN) / 255
x_test_norm = np.array(x_test_CNN) / 255

# Transformamos al formato One-Hot Encoding
y_train_encoded = to_categorical(y_train_CNN)
y_test_encoded = to_categorical(y_test_CNN)
```

Una vez hemos preparado los datos, definimos el modelo. Al igual que con el ANN hemos probado con diferente número de capas y densidades. También hemos probado a introducir capas dropout, para reducir el overfitting, pero los resultados no mejoraban, por lo que decidimos suprimirlas. Incluso intentamos redimensionar las imágenes (tanto a aumentarlas como a reducirlas) sin obtener resultados satisfactorios. Como intento final, probamos a aumentar el número de imágenes (duplicándolas, rotándolas,...), aunque en este caso el modelo no mejoraba, ya que la información del espectrograma se pierde al rotarlo.

Finalmente, los valores de batch_size y epochs que mejor predicen nuestros datos son los que se muestran a continuación.

[268]:
```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from keras.callbacks import EarlyStopping

# Definimos el modelo
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)))
model.add(MaxPooling2D(2, 2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(Dense(128, activation='relu'))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))  # Ajustamos a 10 el número de␣
 ↪neuronas de salida

# Compilamos el modelo
model.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])

# Entrenamos el modelo
hist_CNN = model.fit(x_train_norm, y_train_encoded,␣
 ↪validation_data=(x_test_norm, y_test_encoded), batch_size=64, epochs=20)
```

```
Epoch 1/20

C:\Users\USUARIO\AppData\Roaming\Python\Python312\site-
packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning:

Do not pass an `input_shape`/`input_dim` argument to a layer. When using
```

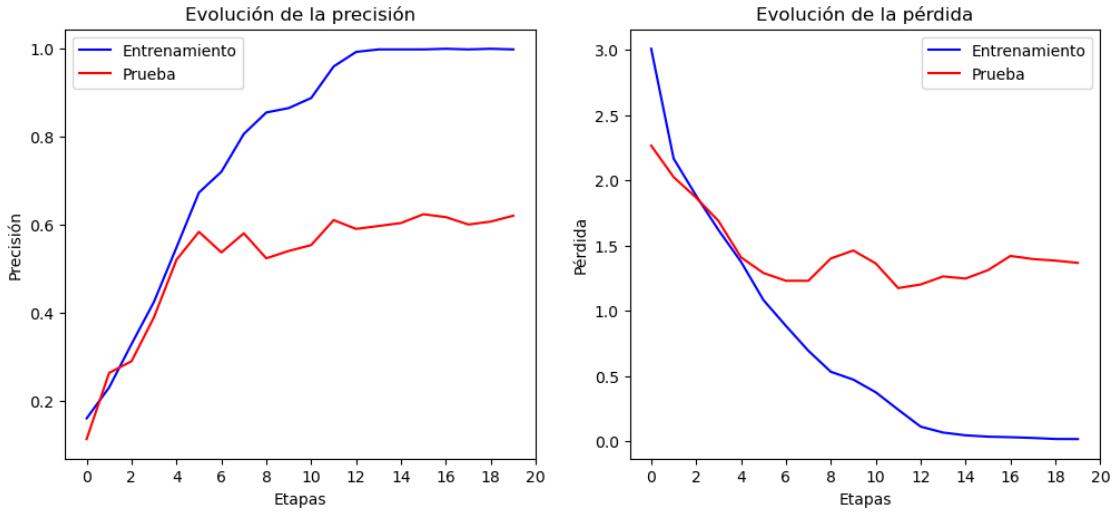
```
Sequential models, prefer using an `Input(shape)` object as the first layer in
the model instead.


11/11 ---------------------- 4s 249ms/step -
accuracy: 0.1423 - loss: 3.4219 - val_accuracy: 0.1133 - val_loss: 2.2665
Epoch 2/20
11/11 ---------------------- 3s 230ms/step -
accuracy: 0.1740 - loss: 2.2193 - val_accuracy: 0.2633 - val_loss: 2.0250
Epoch 3/20
11/11 ---------------------- 2s 223ms/step -
accuracy: 0.3058 - loss: 1.9137 - val_accuracy: 0.2900 - val_loss: 1.8680
Epoch 4/20
11/11 ---------------------- 2s 224ms/step -
accuracy: 0.3906 - loss: 1.6779 - val_accuracy: 0.3900 - val_loss: 1.6877
Epoch 5/20
11/11 ---------------------- 2s 222ms/step -
accuracy: 0.5445 - loss: 1.3796 - val_accuracy: 0.5200 - val_loss: 1.4092
Epoch 6/20
11/11 ---------------------- 2s 222ms/step -
accuracy: 0.6685 - loss: 1.1201 - val_accuracy: 0.5833 - val_loss: 1.2890
Epoch 7/20
11/11 ---------------------- 3s 233ms/step -
accuracy: 0.7318 - loss: 0.8712 - val_accuracy: 0.5367 - val_loss: 1.2296
Epoch 8/20
11/11 ---------------------- 3s 230ms/step -
accuracy: 0.8032 - loss: 0.7054 - val_accuracy: 0.5800 - val_loss: 1.2295
Epoch 9/20
11/11 ---------------------- 3s 232ms/step -
accuracy: 0.8565 - loss: 0.5482 - val_accuracy: 0.5233 - val_loss: 1.4007
Epoch 10/20
11/11 ---------------------- 3s 225ms/step -
accuracy: 0.8692 - loss: 0.4793 - val_accuracy: 0.5400 - val_loss: 1.4620
Epoch 11/20
11/11 ---------------------- 3s 244ms/step -
accuracy: 0.8575 - loss: 0.4171 - val_accuracy: 0.5533 - val_loss: 1.3628
Epoch 12/20
11/11 ---------------------- 3s 255ms/step -
accuracy: 0.9440 - loss: 0.2840 - val_accuracy: 0.6100 - val_loss: 1.1734
Epoch 13/20
11/11 ---------------------- 3s 248ms/step -
accuracy: 0.9884 - loss: 0.1186 - val_accuracy: 0.5900 - val_loss: 1.2012
Epoch 14/20
11/11 ---------------------- 3s 239ms/step -
accuracy: 0.9974 - loss: 0.0695 - val_accuracy: 0.5967 - val_loss: 1.2627
Epoch 15/20
11/11 ---------------------- 3s 238ms/step -
accuracy: 0.9946 - loss: 0.0510 - val_accuracy: 0.6033 - val_loss: 1.2467
```

```
Epoch 16/20
11/11 -------------------- 3s 231ms/step -
accuracy: 0.9978 - loss: 0.0313 - val_accuracy: 0.6233 - val_loss: 1.3114
Epoch 17/20
11/11 -------------------- 3s 229ms/step -
accuracy: 0.9983 - loss: 0.0291 - val_accuracy: 0.6167 - val_loss: 1.4204
Epoch 18/20
11/11 -------------------- 3s 231ms/step -
accuracy: 0.9966 - loss: 0.0264 - val_accuracy: 0.6000 - val_loss: 1.3964
Epoch 19/20
11/11 -------------------- 3s 231ms/step -
accuracy: 0.9993 - loss: 0.0148 - val_accuracy: 0.6067 - val_loss: 1.3843
Epoch 20/20
11/11 -------------------- 3s 234ms/step -
accuracy: 0.9980 - loss: 0.0138 - val_accuracy: 0.6200 - val_loss: 1.3670
```

Representamos la evolución del aprendizaje de la red.

[271]:
```python
# Graficamos la evolución de precisión y la pérdida
plt.figure()
plt.figure(figsize=(12, 5))

# Gráfica de Precisión
plt.subplot(1, 2, 1)
plt.plot(hist_CNN.history['accuracy'], label='Entrenamiento', color='blue')
plt.plot(hist_CNN.history['val_accuracy'], label='Prueba', color='red')
plt.xticks(ticks=range(0, 21, 2))
plt.xlabel('Etapas')
plt.ylabel('Precisión')
plt.legend()
plt.title('Evolución de la precisión')

# Gráfica de Pérdida
plt.subplot(1, 2, 2)
plt.plot(hist_CNN.history['loss'], label='Entrenamiento', color='blue')
plt.plot(hist_CNN.history['val_loss'], label='Prueba', color='red')
plt.xticks(ticks=range(0, 21, 2))
plt.xlabel('Etapas')
plt.ylabel('Pérdida')
plt.legend()
plt.title('Evolución de la pérdida')
plt.show()
plt.close()
```

```
<Figure size 640x480 with 0 Axes>
```



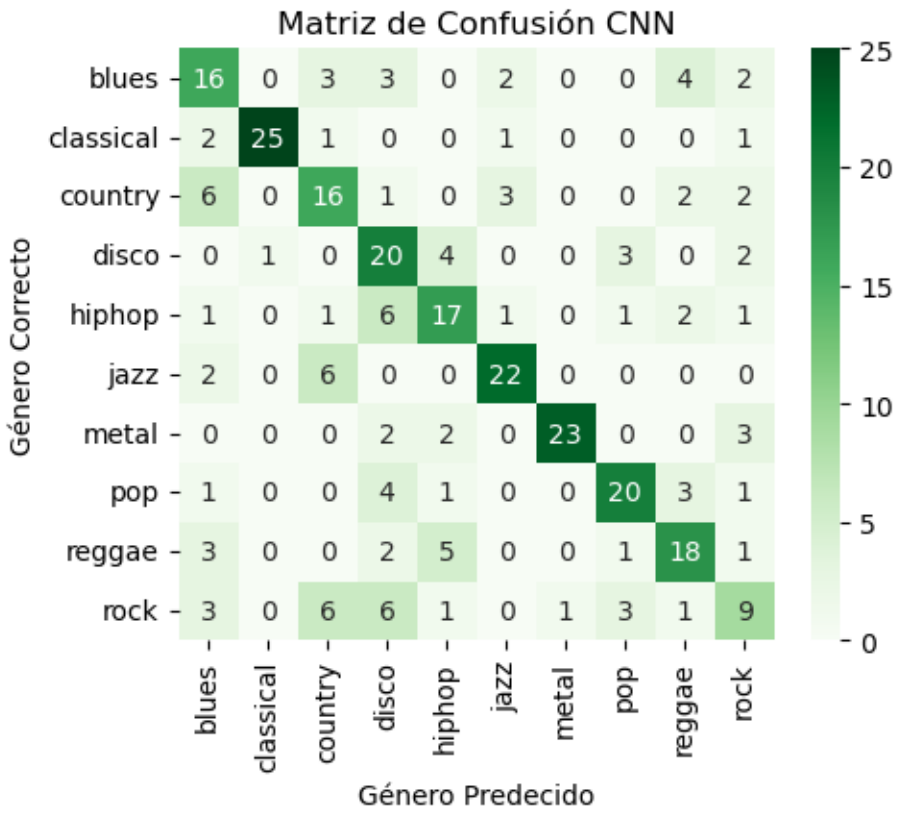

Vemos que, si bien la red aprende de forma aceptable en las primeras etapas, la precisión no aumenta ni la pérdida disminuye. Por lo tanto, aumentar el número de etapas tampoco haría que el modelo mejore. Lo que creemos que está pasando es que el bajo número de imágenes de las que disponemos están limitando el correcto aprendizaje de la red CNN. Esto nos genera overfitting, ajustando muy bien los puntos del conjunto de entrenamiento, pero haciendo que el modelo no tenga capacidad de generalizar.

Finalmente, realizamos el reporte de clasificación y representamos la matriz de confusión para poder comparar más fácilmente el modelo con el resto de métodos.

[275]:
```
# Predicciones y reporte
y_pred_CNN = np.argmax(model.predict(x_test_norm), axis=1)
y_test_classes = np.argmax(y_test_encoded, axis=1) # Convertimos de one-hot␣
 ↪encoding a etiquetas de clase
print("Reporte de clasificación:")
print(classification_report(y_test_classes, y_pred_CNN, target_names=encoder.
 ↪classes_))

# Evaluamos la precisión del modelo
accuracy_CNN = accuracy_score(y_test_classes, y_pred_CNN)
print(f"Precisión del modelo CNN: {accuracy_CNN * 100:.2f}%")

# Visualizamos de la matriz de confusión
conf_matrix_CNN = confusion_matrix(y_test_classes, y_pred_CNN)
plt.figure(figsize=(5, 4))
sns.heatmap(conf_matrix_CNN, annot=True, fmt='d', cmap='Greens',␣
 ↪xticklabels=encoder.classes_, yticklabels=encoder.classes_)
plt.xlabel('Género Predecido')
plt.ylabel('Género Correcto')
plt.title('Matriz de Confusión CNN')
```

```
plt.show()
```

```
10/10 ---------------------- 0s 38ms/step
Reporte de clasificación:
              precision    recall  f1-score   support

       blues       0.47      0.53      0.50        30
   classical       0.96      0.83      0.89        30
     country       0.48      0.53      0.51        30
       disco       0.45      0.67      0.54        30
      hiphop       0.57      0.57      0.57        30
        jazz       0.76      0.73      0.75        30
       metal       0.96      0.77      0.85        30
         pop       0.71      0.67      0.69        30
      reggae       0.60      0.60      0.60        30
        rock       0.41      0.30      0.35        30

    accuracy                           0.62       300
   macro avg       0.64      0.62      0.62       300
weighted avg       0.64      0.62      0.62       300

Precisión del modelo CNN: 62.00%
```



En este caso la precisión no es demasiado buena, especialmente en "rock" y "blues", aunque de nuevo el género "classical" lo clasifica bastante bien.

## 2.8 Conclusiones

A modo de resumen, recopilamos en la siguiente todos los modelos probados junto a su precisión.

[280]:
```
from IPython.display import display, HTML


resultados = {
    "Modelo":["k-Nearest Neighbors","Support Vector Machine","Decision Tree",
              "Regresión Logística","Red Neuronal Artificial","Red Neuronal␣
 ↪Convolucional"],
 "Precisión":[f"{accuracy_KNN * 100:.2f}%",f"{accuracy_SVM * 100:.
 ↪2f}%",f"{accuracy_DT * 100:.2f}%",
              f"{accuracy_LR * 100:.2f}%",f"{accuracy_ANN * 100:.
 ↪2f}%",f"{accuracy_CNN * 100:.2f}%"]}

tabla = pd.DataFrame(resultados)

display(HTML(tabla.to_html(index=False)))
```

| Modelo | Precisión |
|---|---|
| k-Nearest Neighbors | 85.22% |
| Support Vector Machine | 72.14% |
| Decision Tree | 63.60% |
| Regresión Logística | 68.67% |
| Red Neuronal Artificial | 84.48% |
| Red Neuronal Convolucional | 62.00% |

Como se recoge en la tabla y como hemos visto a lo largo del trabajo, los modelos que mejor clasifican el género musical son el k-Nearest Neighbors y el ANN, superando el 80% de precisión. Los que peor predicen son el Decision Tree y el CNN, mientras que el SVM y la Regresión Logística nos dan unos resultados intermedios, sin lograr alcanzar a los dos primeros.

Pese a todo, teniendo en cuenta que estamos clasificando las pistas en 10 géneros diferentes, vemos que todos los modelos predicen relativamente bien. En ningún caso la clasificación es aleatoria, puesto que en ese caso la precisión sería de $\sim$10%.