Music Genre and Composer Classification Using Deep Learning

August 10, 2024

Authors: Zain Ali, Angel Benitez, and Outhai Xayavongsa

Contents

0.1 Introduction

In this project, we employ deep learning to classify classical music compositions by their composers. Leveraging a dataset of 3,929 MIDI files from 175 composers—including Bach, Beethoven, Chopin, and Mozart—we develop Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN) models to identify the Composer of a given piece. Initially, we concentrate on the four mentioned composers to fine-tune our approach. In the end, we created a model encompassing all 145 composers in the dataset, assessing its generalization capabilities across diverse musical styles. We also performed optimizations and many other techniques to get the best models within the last few weeks.

If you would like more information about the files or need access to the full project, please go to our GitHub repository: https://github.com/zainnobody/AAI-511-Final-Project. Feel free to fork or clone it. The README file also contains more information.

Note: Artist and Composer are two words used to describe the creators of classical music, which is now in MIDI format. In this notebook, we use Artist, and the paper and other content use Composer.

0.1.1 Libraries Import

Following are all the libraries and packages used within our project.

[ ]:
```
import os
import shutil
import zipfile
import random
import time
from collections import Counter

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import mido
from mido import MidiFile, bpm2tempo, tick2second
import pretty_midi
import pygame

from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.utils import shuffle
from sklearn.metrics import classification_report, confusion_matrix

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,␣
↪Dropout, LSTM
```

```python
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.initializers import glorot_uniform
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from tensorflow.keras.utils import to_categorical

from keras import backend
```

0.1.2 Global Variables

The following variables were used throughout the project. Although the variables are used globally, they were not used as constants, so they are not all capitalized. If you are cloning the GitHub, feel free to change the values.

[ ]:
```python
# Directory where the raw data will be extracted
raw_data_zip = "raw_data/midi_classic_music_data.zip"  # Location of the zip␣
↪file
raw_data_extracted = "raw_data_unzipped"  # Location where you would like the␣
↪zip file to extract everything
specific_artists = [
    "Bach",
    "Beethoven",
    "Chopin",
    "Mozart",
]  # These are used for the initial LSTM and CNN analysis
```

0.2 Data Collection

The data was quite unorganized and downloaded in a zip format. Several steps were taken to make the data useful and well organized. Get more information about the data within Kaggle at: https://www.kaggle.com/datasets/blanderbuss/midi-classic-music.

[ ]:
```python
# Function to delete a directory and its contents
def delete_dir(dir_to_delete):
    try:
        file_count = sum([len(files) for r, d, files in os.walk(dir_to_delete)])
        shutil.rmtree(dir_to_delete)
        print(f"Directory {dir_to_delete} and all its contents ({file_count}␣
↪files) have been successfully deleted.")
    except Exception as e:
        print(f"An error occurred while trying to delete the directory: {e}")
```

[ ]:
```python
# Function to unzip
def unzip_file(zip_path, extract_to):
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(extract_to)
    return extract_to
```

[ ]:
```python
# Function to move contents of a directory up one level
def move_contents_up_one_dir(path):
    path = os.path.abspath(path)
    parent_dir = os.path.dirname(path)
    if path == parent_dir or not os.path.exists(path):
        print("Operation not allowed or path does not exist.")
        return
    for item in os.listdir(path):
        shutil.move(os.path.join(path, item), os.path.join(parent_dir, item))
    os.rmdir(path)
    print(f"All contents moved from {path} to {parent_dir} and directory␣
 ↪removed.")
```

[ ]:
```python
# Function to rename .MID files to .mid for consistency
def rename_mid_files(directory):
    rename_count = 0
    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith('.MID'):
                old_file_path = os.path.join(root, file)
                new_file_path = os.path.join(root, file[:-4] + '.mid')
                os.rename(old_file_path, new_file_path)
                rename_count += 1
    return rename_count
```

[ ]:
```python
# Function to delete .zip files
def delete_zip_files(directory):
    delete_count = 0
    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith('.zip'):
                file_path = os.path.join(root, file)
                os.remove(file_path)
                delete_count += 1
    return delete_count
```

[ ]:
```python
# Function to Move Folder Contents
def move_folder_contents(src_folder, dest_folder):
    if not os.path.exists(dest_folder):
        os.makedirs(dest_folder)

    for item in os.listdir(src_folder):
        src_item = os.path.join(src_folder, item)
        dest_item = os.path.join(dest_folder, item)

        if os.path.isdir(src_item):
            shutil.move(src_item, dest_folder)
```

```python
        else:
            shutil.move(src_item, dest_item)

    delete_dir(src_folder)
```

[ ]:
```python
# Funtion to move the content of the corrected directory
def directory_name_corrections(name_corrections_dirs):
    for src_folder, dest_folder in name_corrections_dirs.items():
        src_path = os.path.join(raw_data_extracted, src_folder)
        dest_path = os.path.join(raw_data_extracted, dest_folder)
        print(f"Moving contents from {src_path} to {dest_path}...")
        move_folder_contents(src_path, dest_path)

    print("Folder contents moved and directories deleted successfully.")
```

[ ]:
```python
# Function to Categorize Files by Directory
def categorize_files_by_dir(path):
    files_and_dirs = os.listdir(path)
    directories = {name for name in files_and_dirs if os.path.isdir(os.path.
 ↪join(path, name))}
    categorized_files = {}
    unassigned_files = {}

    for file_name in files_and_dirs:
        file_path = os.path.join(path, file_name)
        if os.path.isfile(file_path) and file_name.endswith('.mid'):
            first_word = file_name.split()[0]
            if first_word in directories:
                if first_word not in categorized_files:
                    categorized_files[first_word] = []
                categorized_files[first_word].append(file_name)
            else:
                if first_word not in unassigned_files:
                    unassigned_files[first_word] = []
                unassigned_files[first_word].append(file_name)

    print("Categorized Files Summary:")
    for key, files in categorized_files.items():
        print(f"Artist {key}: {len(files)} files")

    print("\nUnassigned Files Summary:")
    for key, files in unassigned_files.items():
        print(f"Artist {key}: {len(files)} files")

    return categorized_files, unassigned_files, sorted(directories)
```

[ ]:
```python
# Function to Display Information about Categorized and Unassigned Files
def display_info(categorized_files, unassigned_files):
    print("Categorized Files Summary:")
    for key, files in categorized_files.items():
        print(f"Artist '{key}': {len(files)} files")

    print("\nUnassigned Files Summary:")
    if unassigned_files:
        for key, files in unassigned_files.items():
            print(f"Artist '{key}': {len(files)} files")
    else:
        print("No unassigned files found.")
```

[ ]:
```python
# Correcting placement of files.
def corrections_to_file_placements(unassigned_files,␣
 ↪corrections_to_file_placement):
    for old_key, new_key in corrections_to_file_placement.items():
        if old_key in unassigned_files:
            unassigned_files[new_key] = unassigned_files.pop(old_key)
```

[ ]:
```python
# Function to move files to their respective directories
def move_files_to_directories(base_path, files_to_move):
    for directory, files in files_to_move.items():
        dir_path = os.path.join(base_path, directory)
        # Create directory if it doesn't exist
        if not os.path.exists(dir_path):
            os.makedirs(dir_path)
        # Move each file to the new directory
        for file_name in files:
            shutil.move(
                os.path.join(base_path, file_name), os.path.join(dir_path,␣
 ↪file_name)
            )
```

[ ]:
```python
# We wanted to have own exception
class ArtistNotFoundError(Exception):
    def __init__(self, missing_artists):
        self.missing_artists = missing_artists
        super().__init__(
            f"The following specific artists are not in the all artists list:␣
 ↪{', '.join(missing_artists)}"
        )
```

[ ]:
```python
# Get list of all the artist dirs
def get_all_artists(raw_data_extracted):
    all_artists = {
        name
```

```
        for name in os.listdir(raw_data_extracted)
        if os.path.isdir(os.path.join(raw_data_extracted, name))
    }
    all_artists.remove("augmented_pitch")
    return all_artists
```

```
[ ]: # Correct misnamed folders and move contents accordingly
name_corrections_dirs = {
    "Albe'niz": "Albeniz",
    "Albe üniz": "Albeniz",
    "Mendelsonn": "Mendelssohn",
    "Tchakoff": "Tchaikovsky",
    "Handel": "Handel",
    "Haendel": "Handel",
    "Straus": "Strauss",
    "Strauss, J": "Strauss",
}

corrections_to_file_placement = {"Pachebel": "Pachelbel", "Lizt": "Liszt"}
```

0.2.1 Full Steps

Above are the functions, and all are used within `initial_start`.

```
[ ]: def initial_start(raw_data_zip, raw_data_extracted, specific_artists):
    # This is in case of testing and if the initial raw files need to be␣
 ↪deleted.
    if os.path.exists(raw_data_extracted):
        delete_dir(raw_data_extracted)
    raw_data_extracted = unzip_file(raw_data_zip, raw_data_extracted)

    display(f"Extracted to: {raw_data_extracted}")

    # There is a directory 'midiclassics' that needs to be moved one directory␣
 ↪up to make all the structure similar.
    move_contents_up_one_dir(os.path.join(raw_data_extracted, "midiclassics"))

    # Rename .MID files to .mid for consistency
    renamed_files_count = rename_mid_files(raw_data_extracted)
    print(f"Total .MID files renamed: {renamed_files_count}")

    # Delete .zip files
    deleted_files_count = delete_zip_files(raw_data_extracted)
    print(f"Total .zip files deleted: {deleted_files_count}")

    # Categorize files and corrections to dir and files
    categorized_files, unassigned_files, all_artists = categorize_files_by_dir(
```

```
        raw_data_extracted
    )
    corrections_to_file_placements(unassigned_files,␣
↪corrections_to_file_placement)
    directory_name_corrections(name_corrections_dirs)

    # Move categorized and unassigned files to their respective directories
    move_files_to_directories(raw_data_extracted, categorized_files)
    move_files_to_directories(raw_data_extracted, unassigned_files)

    # Final check to see the artists from the project are present within list␣
↪of artists
    all_artists = get_all_artists(raw_data_extracted)

    missing_artists = [
        artist for artist in specific_artists if artist not in all_artists
    ]

    if not missing_artists:
        print("\n\nAll specific artists are in the all artists list.")
    else:
        raise ArtistNotFoundError(missing_artists)


# Processes data, only need once in the beginning.
initial_start(raw_data_zip, raw_data_extracted, specific_artists)
```

0.3 Data Pre-Processing

[ ]:
```
# Function to calculate the length of a MIDI file
def calculate_midi_length(file_path, debug=True):
    try:
        midi_file = MidiFile(file_path)
        total_time = 0.0

        for track in midi_file.tracks:
            current_time = 0.0
            tempo = bpm2tempo(120)  # Default tempo is 120 BPM
            for msg in track:
                if msg.is_meta and msg.type == "set_tempo":
                    tempo = msg.tempo
                current_time += tick2second(msg.time, midi_file.ticks_per_beat,␣
↪tempo)
            if current_time > total_time:
                total_time = current_time

        return total_time
```

```
    except Exception as e:
        if debug:
            print(f"Error processing {file_path}: {e}")
        return None
```

[ ]:
```
# Function to walk through directories and calculate MIDI lengths for a␣
 ↪specific artist
def get_midi_lengths_for_artist(raw_data_extracted, artist, debug = True):
    artist_directory = os.path.join(raw_data_extracted, artist)
    midi_lengths = {}
    file_count = 0

    for root, dirs, files in os.walk(artist_directory):
        for file in files:
            if file.endswith('.mid'):
                file_path = os.path.join(root, file)
                relative_path = os.path.relpath(file_path, raw_data_extracted)
                midi_length = calculate_midi_length(file_path, debug = debug)
                if midi_length is not None:
                    midi_lengths[relative_path] = midi_length
                    file_count += 1

    return midi_lengths, file_count
```

0.3.1 Understanding length

[ ]:
```
def get_midi_lengths_for_artists(
    raw_data_extracted, specific_artists, graph=True, debug=True
):

    # Dictionary to hold all results
    all_midi_lengths = {}
    artist_file_counts = {}

    # Get the MIDI lengths and file counts for each artist
    for artist in specific_artists:
        midi_lengths, file_count = get_midi_lengths_for_artist(
            raw_data_extracted, artist, debug=debug
        )
        all_midi_lengths.update(midi_lengths)
        artist_file_counts[artist] = file_count

    if debug:
        # Print the count of MIDI files for each artist
        for artist, count in artist_file_counts.items():
            print(f"{artist}: {count} MIDI files")
```

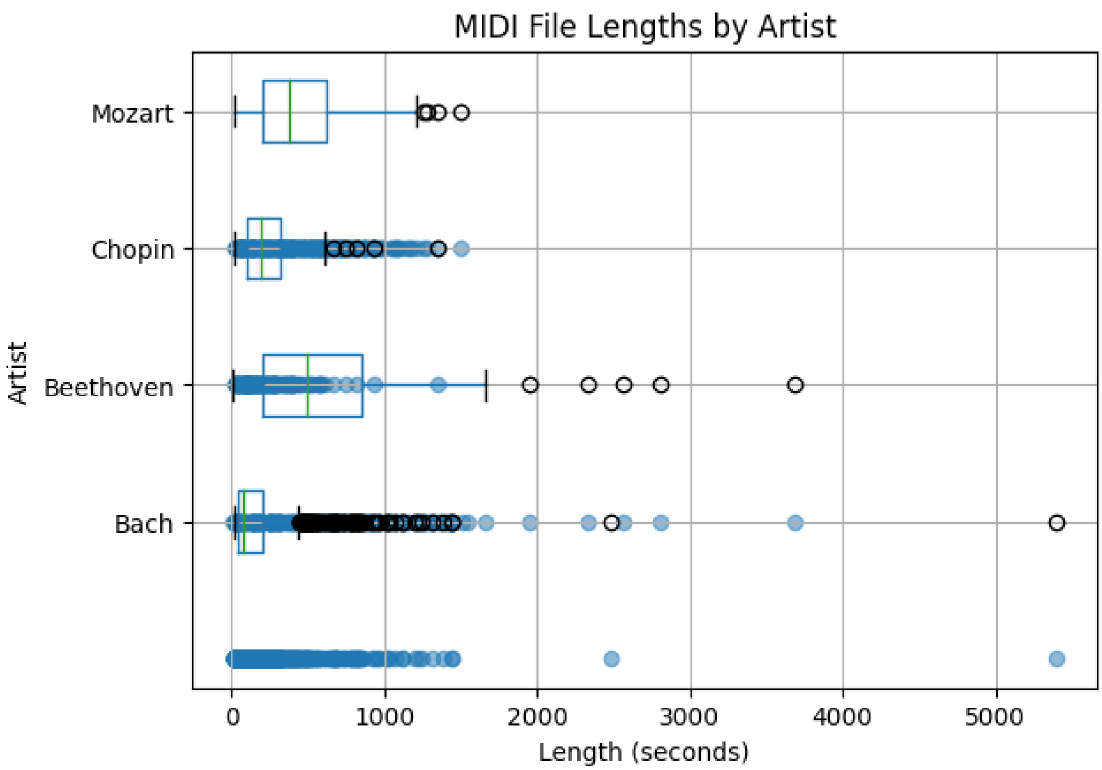
```
    # Create the initial DataFrame directly from the dictionary
    midi_file_lengths_df = pd.DataFrame(
        list(all_midi_lengths.items()), columns=["Path", "Length"]
    )
    midi_file_lengths_df["Artist"] = midi_file_lengths_df["Path"].apply(
        lambda x: next(
            (artist for artist in specific_artists if artist in x), "Unknown"
        )
    )

    if graph:
        # Create horizontal box plots
        plt.figure(figsize=(12, 8))
        midi_file_lengths_df.boxplot(by="Artist", column=["Length"], vert=False)
        plt.scatter(
            midi_file_lengths_df["Length"], midi_file_lengths_df["Artist"],␣
↪alpha=0.5
        )
        plt.title("MIDI File Lengths by Artist")
        plt.suptitle("")
        plt.xlabel("Length (seconds)")
        plt.ylabel("Artist")
        plt.yticks(rotation=0)
        plt.show()

    return midi_file_lengths_df
```

[ ]:

```
paths_artist_length_data = get_midi_lengths_for_artists(
    raw_data_extracted, specific_artists
)
paths_artist_length_data.to_pickle("paths_artist_length_data.pkl")
```

```
Error processing raw_data_unzipped/Beethoven/Anhang 14-3.mid: Could not decode
key with 3 flats and mode 255
Error processing raw_data_unzipped/Mozart/Piano Sonatas/Nueva carpeta/K281 Piano
Sonata n03 3mov.mid: Could not decode key with 2 flats and mode 2
Bach: 1024 MIDI files
Beethoven: 212 MIDI files
Chopin: 136 MIDI files
Mozart: 256 MIDI files
```

```
<Figure size 1200x800 with 0 Axes>
```



0.3.2 Temple Change Augmentation to handle class imbalance.

[ ]:

```
# Data Augmentation (Pitch Shifting)
def augment_midi_pitch_shift(file_path, output_dir, shift=2):
    try:
        midi_file = MidiFile(file_path)
        new_midi_file = MidiFile()

        for track in midi_file.tracks:
            new_track = mido.MidiTrack()
            new_midi_file.tracks.append(new_track)
            for msg in track:
                if msg.type == "note_on" or msg.type == "note_off":
                    msg.note = min(max(msg.note + shift, 0), 127)
                new_track.append(msg)

        output_path = os.path.join(
            output_dir,
            os.path.basename(file_path).replace(".mid", f"_pitch_{shift}.mid"),
        )
        new_midi_file.save(output_path)
```

```python
    except mido.KeySignatureError as e:
        print(f"Error processing {file_path}: {e}")
    except KeyError as e:
        print(f"KeyError processing {file_path}: {e}")
    except Exception as e:
        print(f"Unexpected error processing {file_path}: {e}")


def process_and_augment_midi_files(
    raw_data_extracted,
    specific_artists,
    output_subdir="augmented_pitch",
    shifts=[2, -2],
):
    # Create the output directory
    augmented_pitch_dir = os.path.join(raw_data_extracted, output_subdir)
    os.makedirs(augmented_pitch_dir, exist_ok=True)

    # Walk through the directory and process MIDI files
    for root, dirs, files in os.walk(raw_data_extracted):
        for file in files:
            if file.endswith(".mid"):
                # Check if any artist name in specific_artists is in the file␣
 ↪path
                if any(
                    artist in os.path.join(root, file) for artist in␣
 ↪specific_artists
                ):
                    file_path = os.path.join(root, file)
                    for shift in shifts:
                        augment_midi_pitch_shift(
                            file_path, augmented_pitch_dir, shift=shift
                        )


process_and_augment_midi_files(raw_data_extracted, specific_artists)
```

```
Error processing raw_data_unzipped/Beethoven/Anhang 14-3.mid: Could not decode
key with 3 flats and mode 255
Error processing raw_data_unzipped/Beethoven/Anhang 14-3.mid: Could not decode
key with 3 flats and mode 255
Error processing raw_data_unzipped/Mozart/Piano Sonatas/Nueva carpeta/K281 Piano
Sonata n03 3mov.mid: Could not decode key with 2 flats and mode 2
Error processing raw_data_unzipped/Mozart/Piano Sonatas/Nueva carpeta/K281 Piano
Sonata n03 3mov.mid: Could not decode key with 2 flats and mode 2
```

Ignoring the few files that are not working, as we have a good amount of data.

0.4 Long Short-Term Memory (LSTM)

From here and down, the content is divided into types of models tried within the project: Long short-term memory (LSTM) and Convolutional Neural Network (CNN).

0.4.1 Feature Extraction

Extracting Features Function

[ ]:
```
# Feature Extraction
def extract_features(file_path):
    try:
        midi_file = MidiFile(file_path)
        features = {
            "length": 0,
            "num_notes": 0,
            "note_freq": Counter(),
            "tempo_changes": [],
            "velocities": [],
            "time_sigs": Counter(),
            "key_sigs": Counter(),
            "polyphony": [],
        }

        note_on_times = {}
        polyphony_count = Counter()

        for track in midi_file.tracks:
            current_time = 0.0
            for msg in track:
                current_time += tick2second(
                    msg.time, midi_file.ticks_per_beat, bpm2tempo(120)
                )

                if msg.type == "note_on" and msg.velocity > 0:
                    features["num_notes"] += 1
                    features["note_freq"][msg.note] += 1
                    features["velocities"].append(msg.velocity)
                    if current_time in note_on_times:
                        note_on_times[current_time].append(msg.note)
                    else:
                        note_on_times[current_time] = [msg.note]
                elif msg.type == "set_tempo":
                    features["tempo_changes"].append(mido.tempo2bpm(msg.tempo))
                elif msg.type == "time_signature":
                    features["time_sigs"][(msg.numerator, msg.denominator)] += 1
                elif msg.type == "key_signature":
                    features["key_sigs"][msg.key] += 1
```

```python
        features["length"] = current_time

        for time, notes in note_on_times.items():
            polyphony_count[len(notes)] += 1
        features["polyphony"] = polyphony_count

    except mido.KeySignatureError as e:
        print(f"Error processing {file_path}: {e}")
        return None
    except KeyError as e:
        print(f"KeyError processing {file_path}: {e}")
        return None
    except Exception as e:
        print(f"Unexpected error processing {file_path}: {e}")
        return None

    return features
```

[ ]:
```python
# Extract features from all MIDI files, including augmented files
features_list = []

for root, dirs, files in os.walk(raw_data_extracted):
    for file in files:
        if file.endswith(".mid"):
            for artist in specific_artists:
                if artist in os.path.join(root, file):
                    features = extract_features(os.path.join(root, file))
                    if features:
                        features["path"] = os.path.join(root, file)
                        features["artist"] = artist
                        features_list.append(features)

# Also include features from the augmented directory
for root, dirs, files in os.walk(augmented_pitch_dir):
    for file in files:
        if file.endswith(".mid"):
            for artist in specific_artists:
                if artist in os.path.join(root, file):
                    features = extract_features(os.path.join(root, file))
                    if features:
                        features["path"] = os.path.join(root, file)
                        features["artist"] = artist
                        features_list.append(features)

# Convert to DataFrame for analysis
features_list_df = pd.DataFrame(features_list)
```

```python
# Print extracted features
print("Extracted features:")
features_list_df.head()
```

```
Error processing raw_data_unzipped/Beethoven/Anhang 14-3.mid: Could not decode
key with 3 flats and mode 255
Error processing raw_data_unzipped/Mozart/Piano Sonatas/Nueva carpeta/K281 Piano
Sonata n03 3mov.mid: Could not decode key with 2 flats and mode 2
Extracted features:
```

[ ]:

```
       length  num_notes                                            note_freq  \
0    0.000000        567  {51: 4, 48: 7, 55: 16, 60: 24, 63: 31, 62: 22,…
1    0.000000       4301  {56: 190, 68: 134, 67: 161, 61: 108, 64: 88, 6…
2  539.194792       6517  {37: 131, 44: 441, 49: 265, 52: 141, 56: 416, …
3  136.916667       1019  {72: 80, 77: 13, 69: 41, 65: 79, 67: 41, 74: 4…
4    1.500000        708  {67: 69, 72: 42, 64: 26, 60: 29, 76: 21, 74: 1…

                                      tempo_changes  \
0                                  [140.00014000014]
1  [55.99997760000896, 50.0, 44.000011733336464, …
2  [160.0, 89.9999550000225, 160.0, 140.000140000…
3  [165.000165000165, 165.000165000165, 165.00016…
4  [89.9999550000225, 69.00001725000432, 89.99995…

                                         velocities  \
0  [88, 92, 81, 82, 84, 89, 103, 94, 91, 93, 80, …
1  [70, 70, 60, 60, 60, 70, 70, 60, 60, 60, 70, 7…
2  [58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 5…
3  [60, 60, 92, 60, 60, 92, 60, 60, 60, 60, 60, 6…
4  [91, 94, 96, 97, 100, 98, 94, 95, 88, 70, 55, …

                                          time_sigs           key_sigs  \
0                                       {(4, 8): 1}          {'Bb': 1}
1  {(4, 4): 15, (8, 4): 5, (3, 4): 1, (5, 4): 5, …  {'Ab': 3, 'F': 2}
2                                       {(4, 4): 1}           {'C': 1}
3                                       {(6, 8): 1}           {'F': 1}
4                                       {(4, 4): 1}           {'C': 1}

                                          polyphony  \
0                              {1: 523, 3: 2, 2: 19}
1  {2: 508, 1: 2356, 3: 142, 4: 49, 5: 2, 7: 19, …
2  {1: 4167, 6: 2, 5: 2, 2: 802, 3: 232, 4: 3, 8: 2}
3                      {3: 80, 2: 279, 1: 217, 4: 1}
4  {4: 13, 5: 30, 6: 19, 2: 1, 3: 4, 7: 28, 8: 15…

                                               path     artist
0  raw_data_unzipped/C.P.E.Bach/C.P.E.Bach Solfeg…       Bach
```

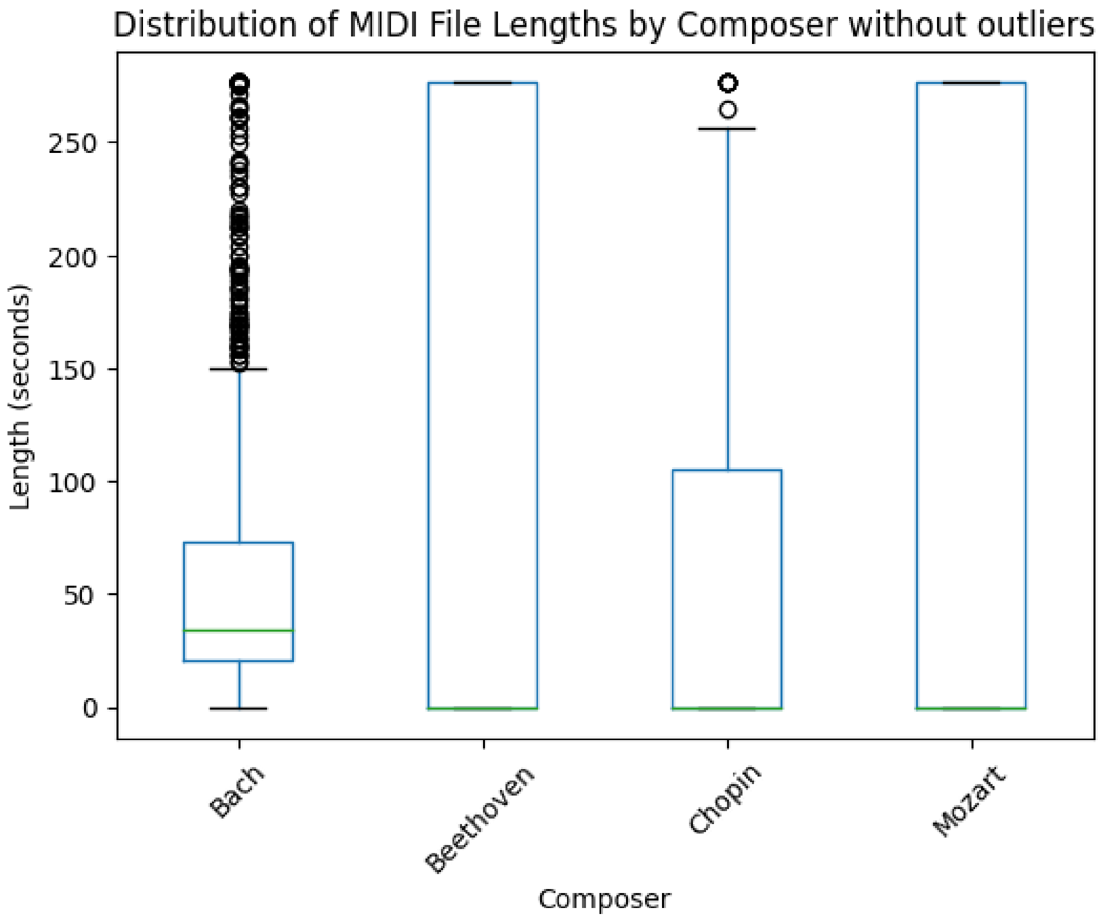
```
1  raw_data_unzipped/Busoni/Fantasia Nach J. S. B…       Bach
2  raw_data_unzipped/Beethoven/Piano Sonata No.27…  Beethoven
3  raw_data_unzipped/Beethoven/Sieben Bagatellen,…  Beethoven
4  raw_data_unzipped/Beethoven/Lieder op48 n4 ''D…  Beethoven
```

[ ]:
```
# Handling Outliers
def handle_outliers(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    df[column] = np.where(df[column] > upper_bound, upper_bound, df[column])
    df[column] = np.where(df[column] < lower_bound, lower_bound, df[column])


for col in ["length", "num_notes"]:
    handle_outliers(features_list_df, col)
```

EDA Visuals

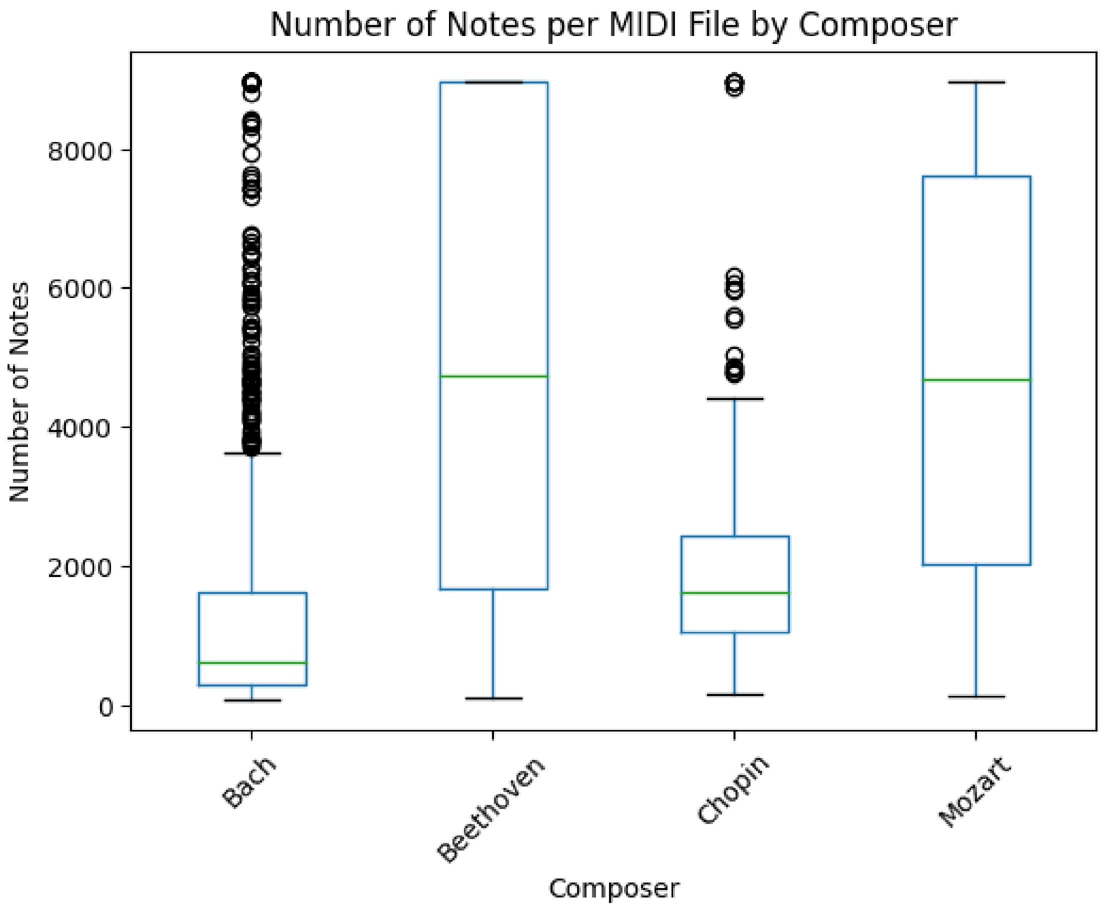
[ ]:
```
# Distribution of MIDI file lengths by composer after outlier removal
plt.figure(figsize=(12, 6))
features_list_df.boxplot(by="artist", column=["length"], grid=False)
plt.title("Distribution of MIDI File Lengths by Composer without outliers")
plt.suptitle("")
plt.xlabel("Composer")
plt.ylabel("Length (seconds)")
plt.xticks(rotation=45)
plt.show()
```

```
<Figure size 1200x600 with 0 Axes>
```

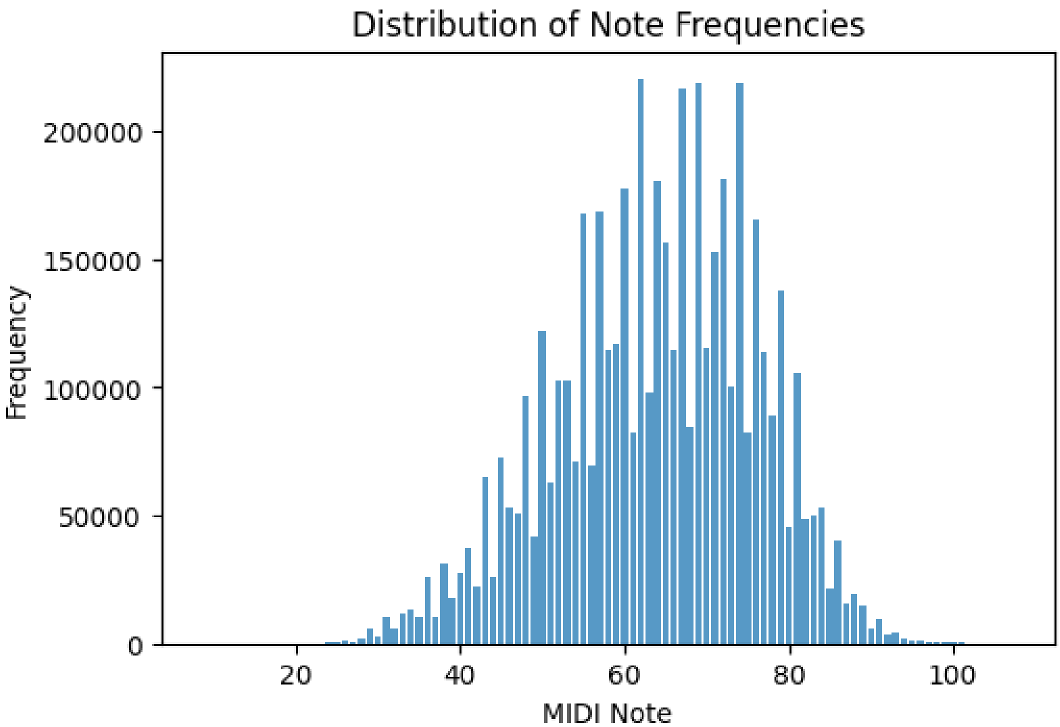


[ ]:
```python
# Number of notes per MIDI file by composer
plt.figure(figsize=(6, 4))
features_list_df.boxplot(by='artist', column=['num_notes'], grid=False)
plt.title('Number of Notes per MIDI File by Composer')
plt.suptitle('')
plt.xlabel('Composer')
plt.ylabel('Number of Notes')
plt.xticks(rotation=45)
plt.show()
```

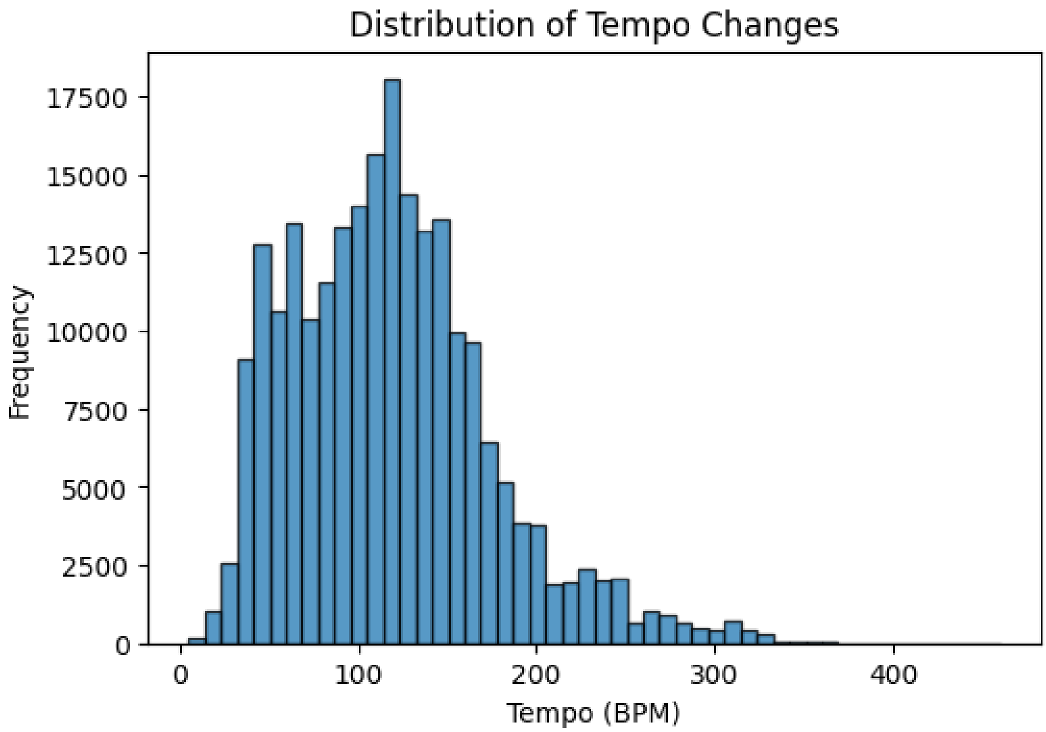
```
<Figure size 600x400 with 0 Axes>
```



[ ]:
```
# Distribution of note frequencies
note_freqs = Counter()
for note_counter in features_list_df['note_freq']:
    note_freqs.update(note_counter)

plt.figure(figsize=(6, 4))
plt.bar(note_freqs.keys(), note_freqs.values(), alpha=0.75)
plt.title('Distribution of Note Frequencies')
plt.xlabel('MIDI Note')
plt.ylabel('Frequency')
plt.show()
```

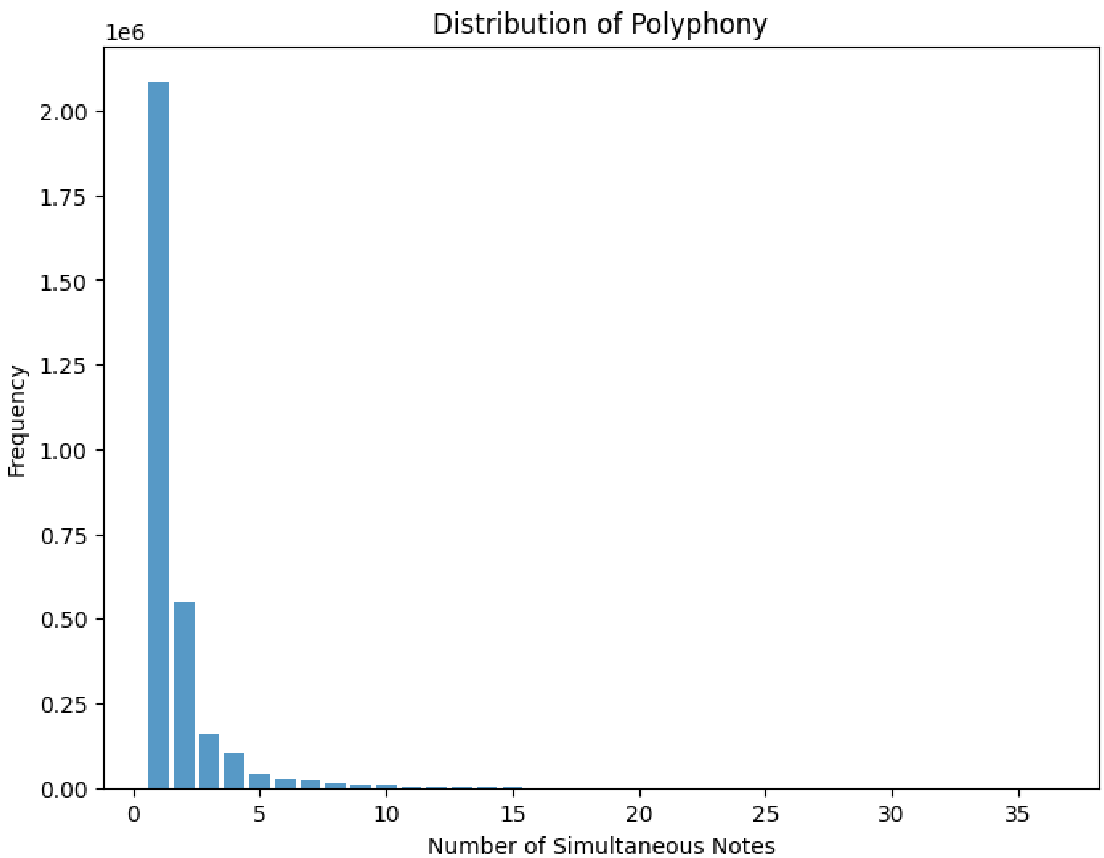


[ ]:
```
# Distribution of tempo changes
tempo_changes = [
    tempo for sublist in features_list_df["tempo_changes"] for tempo in sublist
]

plt.figure(figsize=(6, 4))
plt.hist(tempo_changes, bins=50, alpha=0.75, edgecolor="black")
plt.title("Distribution of Tempo Changes")
plt.xlabel("Tempo (BPM)")
plt.ylabel("Frequency")
plt.show()
```



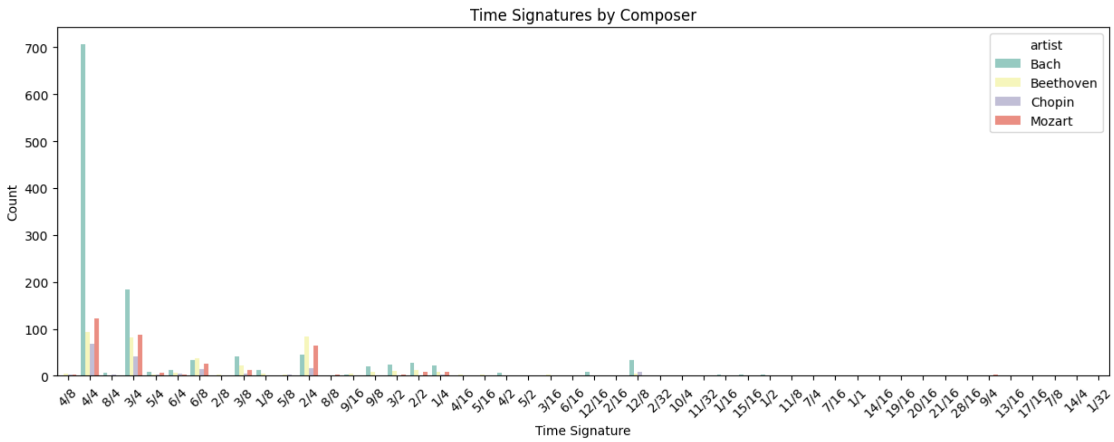
[ ]:

```
# Distribution of polyphony
polyphony_count = Counter()
for polyphony_counter in features_list_df['polyphony']:
    polyphony_count.update(polyphony_counter)

plt.figure(figsize=(8, 6))
plt.bar(polyphony_count.keys(), polyphony_count.values(), alpha=0.75)
plt.title('Distribution of Polyphony')
plt.xlabel('Number of Simultaneous Notes')
plt.ylabel('Frequency')
plt.show()
```

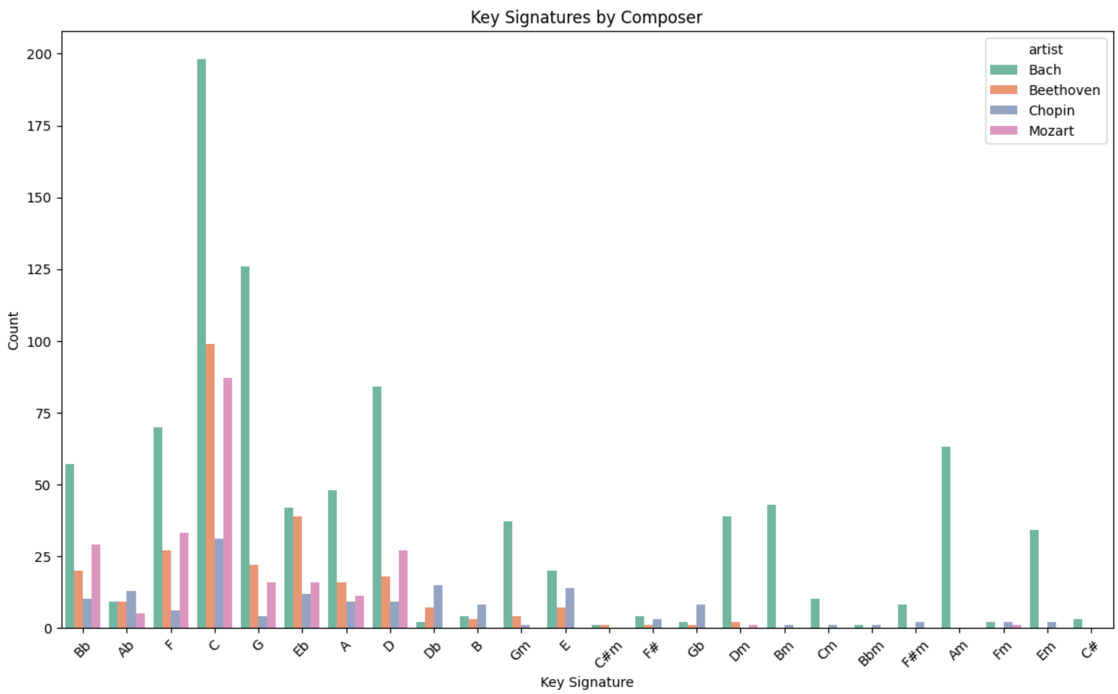


[ ]:
```
# Time signatures by composer
time_sigs_flat = []
for idx, row in features_list_df.iterrows():
    for time_sig, count in row["time_sigs"].items():
        time_sigs_flat.append(
            {
                "artist": row["artist"],
                "time_signature": f"{time_sig[0]}/{time_sig[1]}",
                "count": count,
            }
        )

time_sigs_df = pd.DataFrame(time_sigs_flat)

plt.figure(figsize=(15, 5))
sns.countplot(data=time_sigs_df, x="time_signature", hue="artist",␣
 ↪palette="Set3")
plt.title("Time Signatures by Composer")
plt.xlabel("Time Signature")
```

```python
plt.ylabel("Count")
plt.xticks(rotation=45)
plt.show()
```



[ ]:
```python
# Key signatures by composer
key_sigs_flat = []
for idx, row in features_list_df.iterrows():
    for key_sig, count in row["key_sigs"].items():
        key_sigs_flat.append(
            {"artist": row["artist"], "key_signature": key_sig, "count": count}
        )

key_sigs_df = pd.DataFrame(key_sigs_flat)

plt.figure(figsize=(14, 8))
sns.countplot(data=key_sigs_df, x="key_signature", hue="artist", palette="Set2")
plt.title("Key Signatures by Composer")
plt.xlabel("Key Signature")
plt.ylabel("Count")
plt.xticks(rotation=45)
plt.show()
```



[ ]:
```
features_list_df.to_pickle('extracted_features.pkl')
```

Ten Second Random Audio Samples

[ ]:
```
# Function to list MIDI files for a composer
def list_midi_files(directory, composer):
    composer_dir = os.path.join(directory, composer)
    return [
        os.path.join(composer_dir, file)
        for file in os.listdir(composer_dir)
        if file.endswith(".mid")
    ]


# Function to play a MIDI file for a specified duration
def play_midi(file_path, duration=10):
    pygame.mixer.init()
    pygame.mixer.music.load(file_path)
    pygame.mixer.music.play()
    time.sleep(duration)
    pygame.mixer.music.stop()


# Dictionary to hold a randomly selected MIDI file for each composer
selected_files = {}
```

```python
# Select one random MIDI file for each composer
for composer in specific_artists:
    midi_files = list_midi_files(raw_data_extracted, composer)
    if midi_files:
        selected_files[composer] = random.choice(midi_files)
    else:
        print(f"No MIDI files found for {composer}")

# Play the selected MIDI files
for composer, file_path in selected_files.items():
    print(f"Playing {composer}'s selected MIDI file: {file_path}")
    play_midi(file_path)
```

```
Playing Bach's selected MIDI file: raw_data_unzipped/Bach/Bwv0544 Prelude and
Fugue.mid
Playing Beethoven's selected MIDI file: raw_data_unzipped/Beethoven/Bagatella
op33 n5.mid
Playing Chopin's selected MIDI file: raw_data_unzipped/Chopin/Prelude n03 op28
''Thou Art So Like A Flower''.mid
Playing Mozart's selected MIDI file: raw_data_unzipped/Mozart/K393 Solfeggi
n1.mid
```

0.4.2 Loading Dataset

[ ]:
```python
## In case if we need to directly load in
# features_list_df = pd.read_pickle('extracted_features.pkl')
# features_list_df.head()
```

0.4.3 Preparing Data

[ ]:
```python
# Handle missing values if any
features_list_df.fillna(0, inplace=True)

# Encode the artist labels
label_encoder = LabelEncoder()
features_list_df["artist_encoded"] = label_encoder.fit_transform(
    features_list_df["artist"]
)

# Standardize the features
scaler = StandardScaler()
numeric_features = ["length", "num_notes"]
scaled_features = scaler.fit_transform(features_list_df[numeric_features])

# Prepare sequences
X = []
```

```
y = []
sequence_length = 10  # Adjust as necessary

for i in range(len(scaled_features) - sequence_length):
    X.append(scaled_features[i : i + sequence_length])
    y.append(features_list_df["artist_encoded"].iloc[i + sequence_length])

X = np.array(X)
y = np.array(y)
y = to_categorical(y, num_classes=len(label_encoder.classes_))

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(1298, 10, 2) (325, 10, 2) (1298, 4) (325, 4)
```

0.4.4 Defining the LSTM Model

[ ]:
```
# Define the LSTM model
model = Sequential()
model.add(
    LSTM(128, input_shape=(X_train.shape[1], X_train.shape[2]),␣
 ↪return_sequences=True)
)
model.add(Dropout(0.2))
model.add(LSTM(64))
model.add(Dropout(0.2))
model.add(Dense(len(label_encoder.classes_), activation="softmax"))

model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss="categorical_crossentropy",
    metrics=["accuracy"],
)
model.summary()
```

```
Model: "sequential_59"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 10, 128)           67072

 dropout_177 (Dropout)       (None, 10, 128)           0
```

```
 lstm_1 (LSTM)               (None, 64)                49408

 dropout_178 (Dropout)       (None, 64)                0

 dense_118 (Dense)           (None, 4)                 260

=================================================================
Total params: 116,740
Trainable params: 116,740
Non-trainable params: 0
_________________________________________________________________
```

0.4.5 Training the Model

[ ]:
```
backend.clear_session()
tf.compat.v1.reset_default_graph()
```

[ ]:
```
history = model.fit(
    X_train,
    y_train,
    epochs=50,
    batch_size=32,
    validation_data=(X_test, y_test),
    verbose=0,
)
```

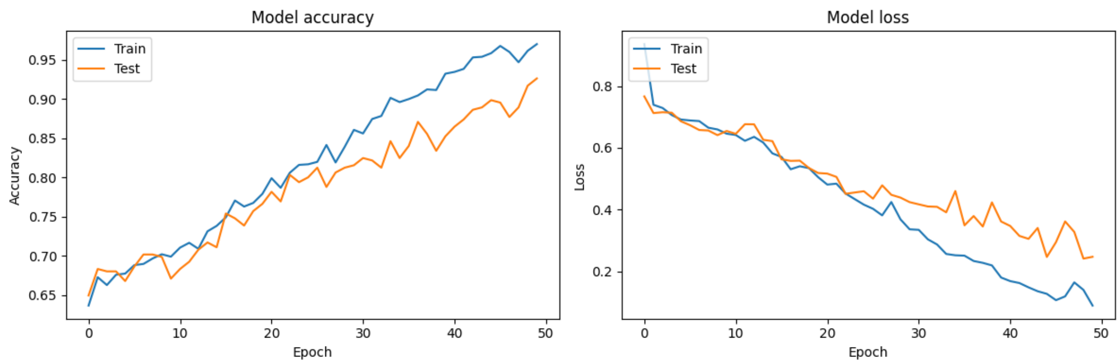
0.4.6 Evaluating the Model

Visualizing Training History

[ ]:
```
def plot_training_history(history, figsize=(12, 4)):
    metrics = ["accuracy", "loss"]
    plt.figure(figsize=figsize)
    for i, metric in enumerate(metrics):
        plt.subplot(1, 2, i + 1)

        if metric in history.history:
            plt.plot(history.history[metric])
            plt.plot(history.history[f"val_{metric}"])
            plt.title(f"Model {metric}")
            plt.ylabel(metric.capitalize())
            plt.xlabel("Epoch")
            plt.legend(["Train", "Test"], loc="upper left")
        else:
            plt.text(
                0.5,
                0.5,
                f"No {metric} data available",
                horizontalalignment="center",
```

```
                verticalalignment="center",
                transform=plt.gca().transAxes,
            )
            plt.title(f"Model {metric}")
            plt.ylabel(metric.capitalize())
            plt.xlabel("Epoch")
    plt.tight_layout()
    plt.show()
```

[ ]:
```
plot_training_history(history)
```



[ ]:
```
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss}, Test Accuracy: {accuracy}")

# Make predictions
predictions = model.predict(X_test)
predicted_classes = np.argmax(predictions, axis=1)
true_classes = np.argmax(y_test, axis=1)

# Convert encoded labels back to original
predicted_labels = label_encoder.inverse_transform(predicted_classes)
true_labels = label_encoder.inverse_transform(true_classes)

# Display some predictions
for i in range(10):
    print(f"True: {true_labels[i]}, Predicted: {predicted_labels[i]}")
```

```
11/11 [==============================] - 0s 2ms/step - loss: 0.2468 - accuracy:
0.9262
Test Loss: 0.24684257805347443, Test Accuracy: 0.926153838634491
True: Beethoven, Predicted: Beethoven
True: Bach, Predicted: Bach
True: Bach, Predicted: Bach
```

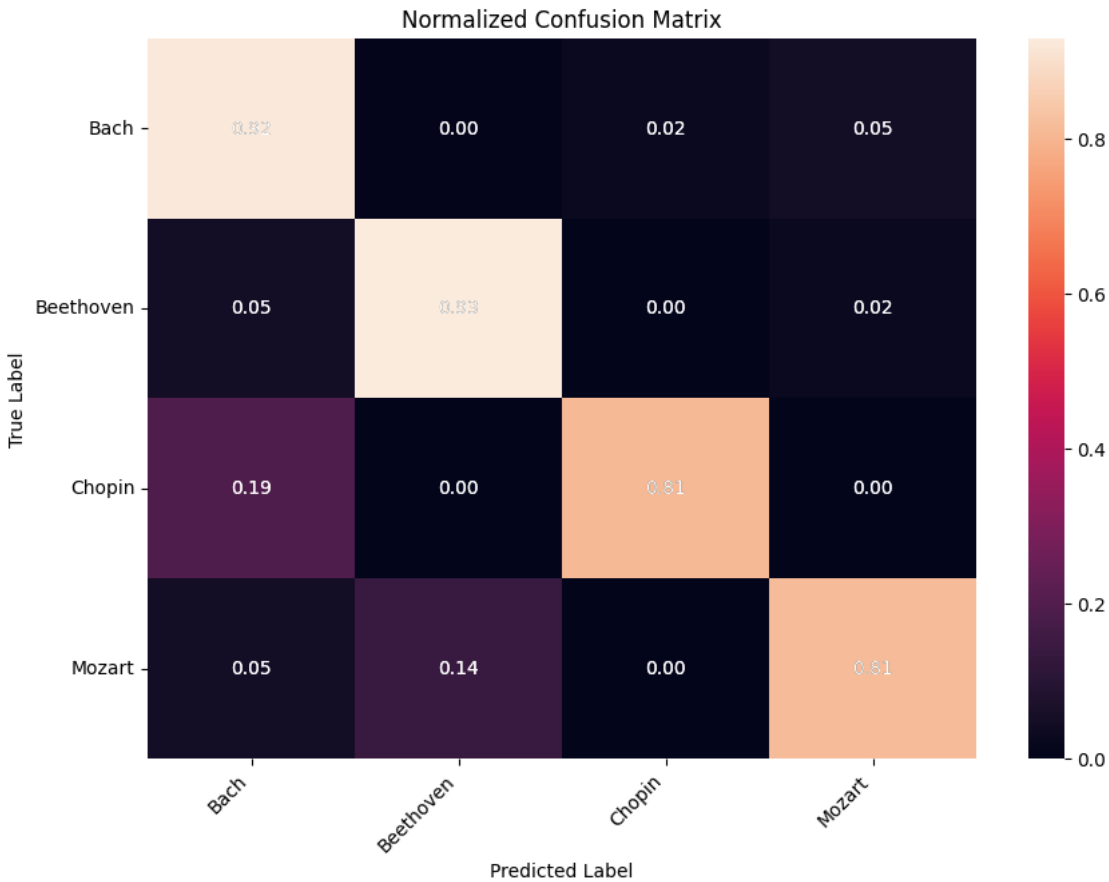
```
True: Bach, Predicted: Bach
True: Bach, Predicted: Bach
True: Bach, Predicted: Bach
True: Mozart, Predicted: Mozart
True: Bach, Predicted: Bach
True: Bach, Predicted: Bach
True: Bach, Predicted: Bach
```

Evaluation Metrics

[ ]:
```python
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss}, Test Accuracy: {accuracy}")

# Make predictions
predictions = model.predict(X_test)
predicted_classes = np.argmax(predictions, axis=1)
true_classes = np.argmax(y_test, axis=1)

# Convert encoded labels back to original
predicted_labels = label_encoder.inverse_transform(predicted_classes)
true_labels = label_encoder.inverse_transform(true_classes)

# Classification report
print("Classification Report:")
print(
    classification_report(
        true_classes, predicted_classes, target_names=label_encoder.classes_
    )
)

# Confusion matrix
conf_matrix = confusion_matrix(true_classes, predicted_classes)

# Normalize the confusion matrix
conf_matrix_normalized = (
    conf_matrix.astype("float") / conf_matrix.sum(axis=1)[:, np.newaxis]
)

# Plot normalized confusion matrix
plt.figure(figsize=(10, 7))
heatmap = sns.heatmap(
    conf_matrix_normalized,
    annot=True,
    fmt=".2f",
    cmap="rocket",
    xticklabels=label_encoder.classes_,
    yticklabels=label_encoder.classes_,
```

```python
)
heatmap.set_yticklabels(heatmap.get_yticklabels(), rotation=0, ha="right")
heatmap.set_xticklabels(heatmap.get_xticklabels(), rotation=45, ha="right")

# Annotate each cell with the numeric value
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        plt.text(
            j + 0.5,
            i + 0.5,
            f"{conf_matrix_normalized[i, j]:.2f}",
            horizontalalignment="center",
            verticalalignment="center",
            color="white",
        )

plt.title("Normalized Confusion Matrix")
plt.ylabel("True Label")
plt.xlabel("Predicted Label")
plt.show()
```

```
11/11 [==============================] - 0s 3ms/step - loss: 0.2891 - accuracy:
0.9015
Test Loss: 0.28913936018943787, Test Accuracy: 0.9015384912490845
Classification Report:
              precision    recall  f1-score   support

        Bach       0.96      0.92      0.94       213
   Beethoven       0.85      0.93      0.89        43
      Chopin       0.81      0.81      0.81        26
      Mozart       0.76      0.81      0.79        43

    accuracy                           0.90       325
   macro avg       0.84      0.87      0.86       325
weighted avg       0.90      0.90      0.90       325
```



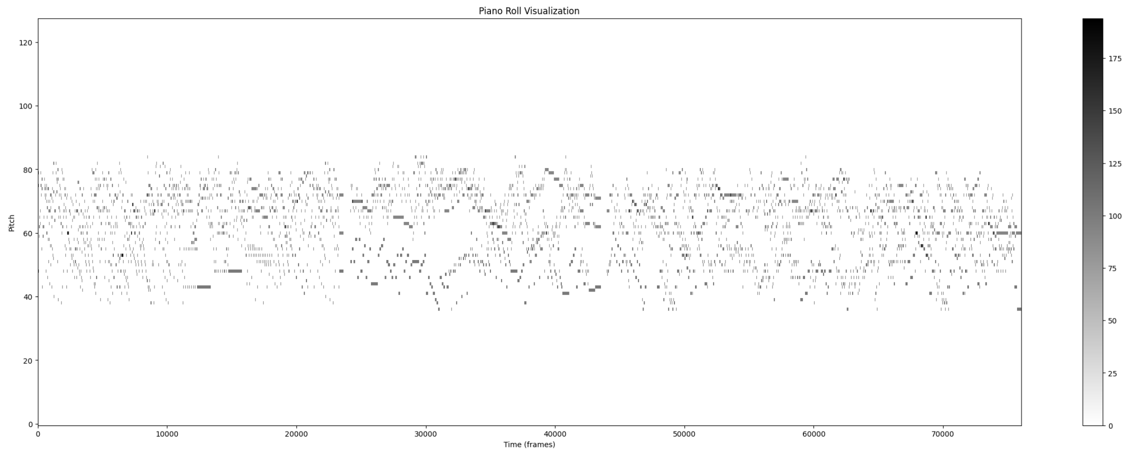
0.5 Convolutional Neural Network (CNN)

0.5.1 Data Exploration

Understanding Structure Understanding how data can be used for CNN using a `test_file`.

[ ]:
```
# checking how the data will look like
test_file = raw_data_extracted + "/Bach/Bwv0526 Sonate en trio n2.mid"
# Load MIDI file
midi_data = pretty_midi.PrettyMIDI(test_file)

# Generate piano roll
piano_roll = midi_data.get_piano_roll(fs=100)

# Plot piano roll
plt.figure(figsize=(30, 10))
plt.imshow(
    piano_roll, aspect="auto", origin="lower", cmap="gray_r",␣
 ↪interpolation="nearest"
)
```

```
plt.xlabel("Time (frames)")
plt.ylabel("Pitch")
plt.title("Piano Roll Visualization")
plt.colorbar()
plt.show()
```



0.5.2 Feature Extraction

In this feature extraction process, we convert MIDI files into a multichannel piano roll to capture various aspects of the musical performance:

1. **Binary Roll**: Captures note presence.
2. **Velocity Roll**: Reflects note intensity.
3. **Instrumentation Roll**: Shows which instruments play each note.
4. **Expressive Timing Roll**: Details the timing of notes.

[ ]:
```
# Processes a MIDI file into a multichannel piano roll (binary, velocity,␣
↪instrumentation, timing).
def process_multichannel_midi(file_path, fs=10, max_length=100):
    midi_data = pretty_midi.PrettyMIDI(file_path)

    # Binary and velocity piano rolls
    piano_roll = midi_data.get_piano_roll(fs=fs)
    binary_piano_roll = (piano_roll > 0).astype(int)
    velocity_roll = piano_roll / 127  # Normalize velocity

    # Combining instrument rolls, adjusting for length
    instrument_rolls = []
    for instrument in midi_data.instruments:
        inst_roll = instrument.get_piano_roll(fs=fs)
        instrument_rolls.append(inst_roll)
```

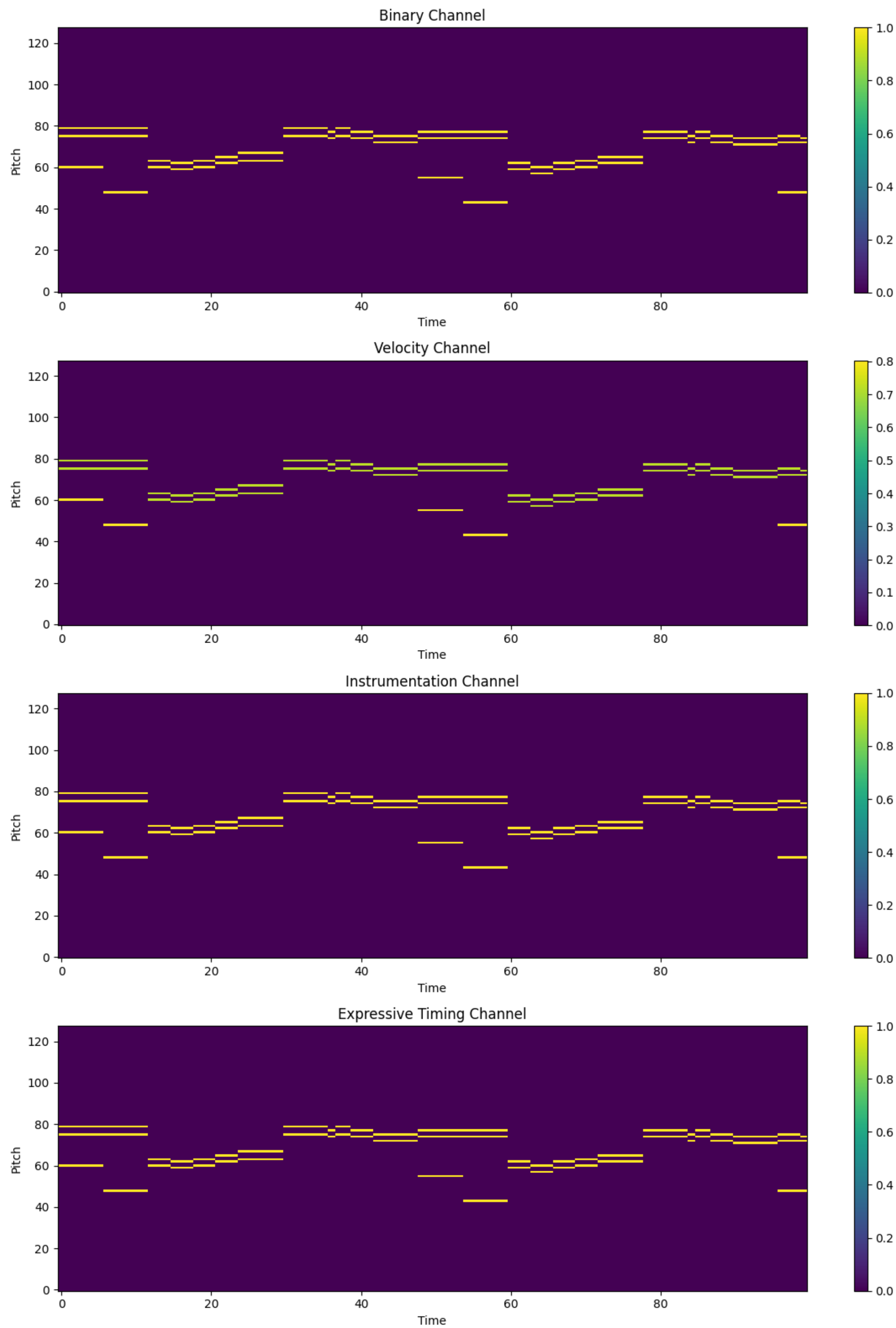
```
    max_instrument_length = max(inst.shape[1] for inst in instrument_rolls)
    combined_instrument_roll = np.zeros((128, max_instrument_length))
    for inst_roll in instrument_rolls:
        if inst_roll.shape[1] < max_instrument_length:
            # Pad to the right if shorter
            padding = np.zeros((128, max_instrument_length - inst_roll.
↪shape[1]))
            inst_roll = np.hstack((inst_roll, padding))
        combined_instrument_roll += (inst_roll > 0).astype(int)

    # Creating expressive timing roll
    expressive_timing_roll = np.zeros((128, max_instrument_length))
    for instrument in midi_data.instruments:
        for note in instrument.notes:
            start = int(note.start * fs)
            end = int(note.end * fs)
            expressive_timing_roll[note.pitch, start:end] = 1

    # Adjusting rolls to match the maximum length
    if max_instrument_length > max_length:
        combined_instrument_roll = combined_instrument_roll[:, :max_length]
        expressive_timing_roll = expressive_timing_roll[:, :max_length]
    elif max_instrument_length < max_length:
        padding = np.zeros((128, max_length - max_instrument_length))
        combined_instrument_roll = np.hstack((combined_instrument_roll,␣
↪padding))
        expressive_timing_roll = np.hstack((expressive_timing_roll, padding))

    binary_piano_roll = binary_piano_roll[:, :max_length]
    velocity_roll = velocity_roll[:, :max_length]

    # Stacking all channels into a multichannel roll
    multichannel_roll = np.stack(
        [
            binary_piano_roll,
            velocity_roll,
            combined_instrument_roll,
            expressive_timing_roll,
        ],
        axis=-1,
    )

    return multichannel_roll
```

[ ]:
```
# Plotting each channel of the processed multichannel piano roll data
def plot_multichannel_piano_roll(processed_data):
    # Unpacking the channels
```

```
    binary_channel = processed_data[:, :, 0]
    velocity_channel = processed_data[:, :, 1]
    instrument_channel = processed_data[:, :, 2]
    expressive_timing_channel = processed_data[:, :, 3]

    # Setting up the plot
    fig, axes = plt.subplots(nrows=4, ncols=1, figsize=(12, 16))
    titles = [
        "Binary Channel",
        "Velocity Channel",
        "Instrumentation Channel",
        "Expressive Timing Channel",
    ]

    # Plotting each channel
    for ax, channel, title in zip(
        axes,
        [
            binary_channel,
            velocity_channel,
            instrument_channel,
            expressive_timing_channel,
        ],
        titles,
    ):
        cax = ax.imshow(channel, aspect="auto", origin="lower",␣
↪interpolation="nearest")
        ax.set_title(title)
        ax.set_xlabel("Time")
        ax.set_ylabel("Pitch")
        fig.colorbar(cax, ax=ax, orientation="vertical")

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

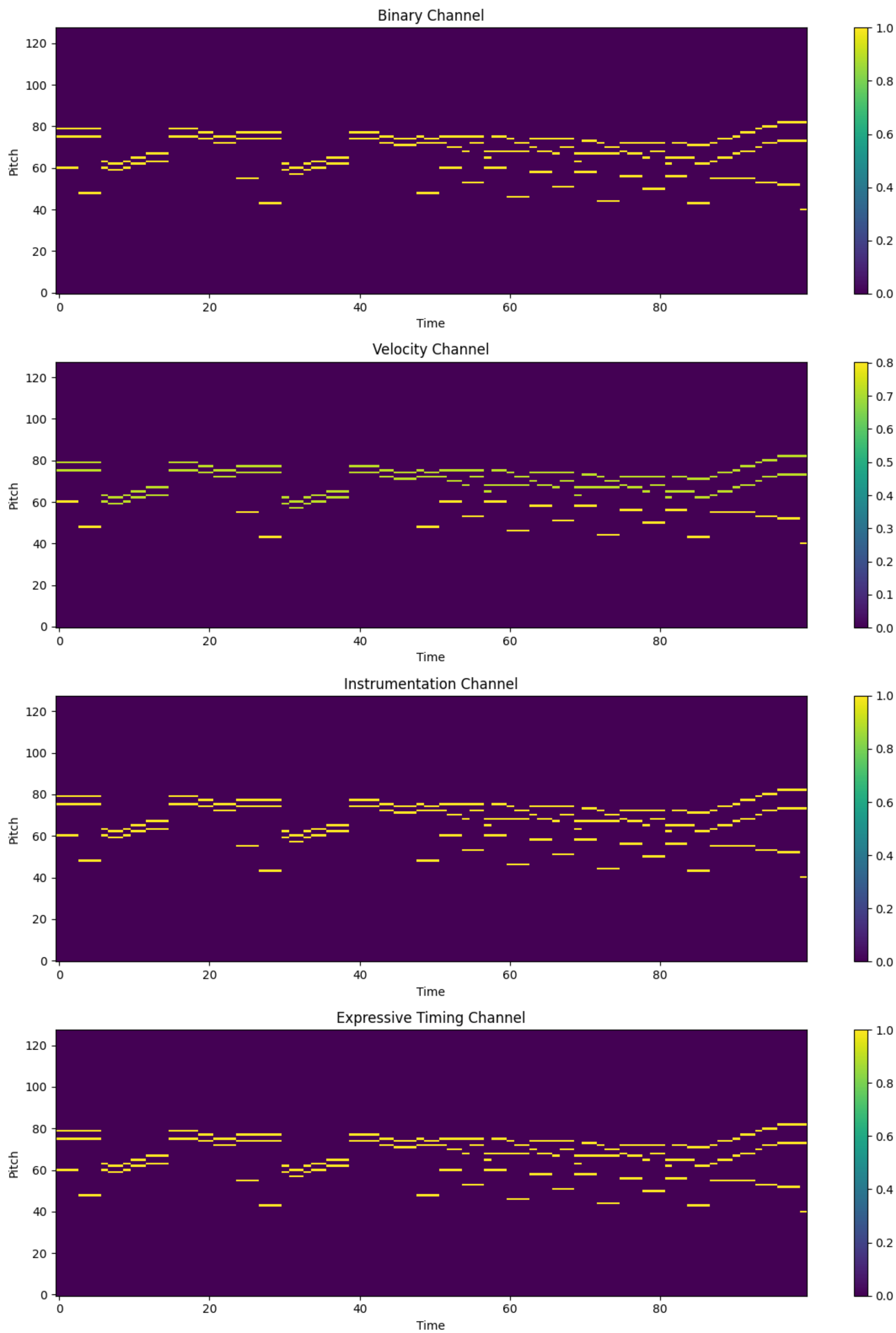
Testing Frames Per Second (FPS) Determining the optimal placement for frames per second (FPS) using visual aids. Trying to see how much of the visual detail is being compressed.

[ ]:
```
processed_data = process_multichannel_midi(test_file, fs=16)
plot_multichannel_piano_roll(processed_data)
```

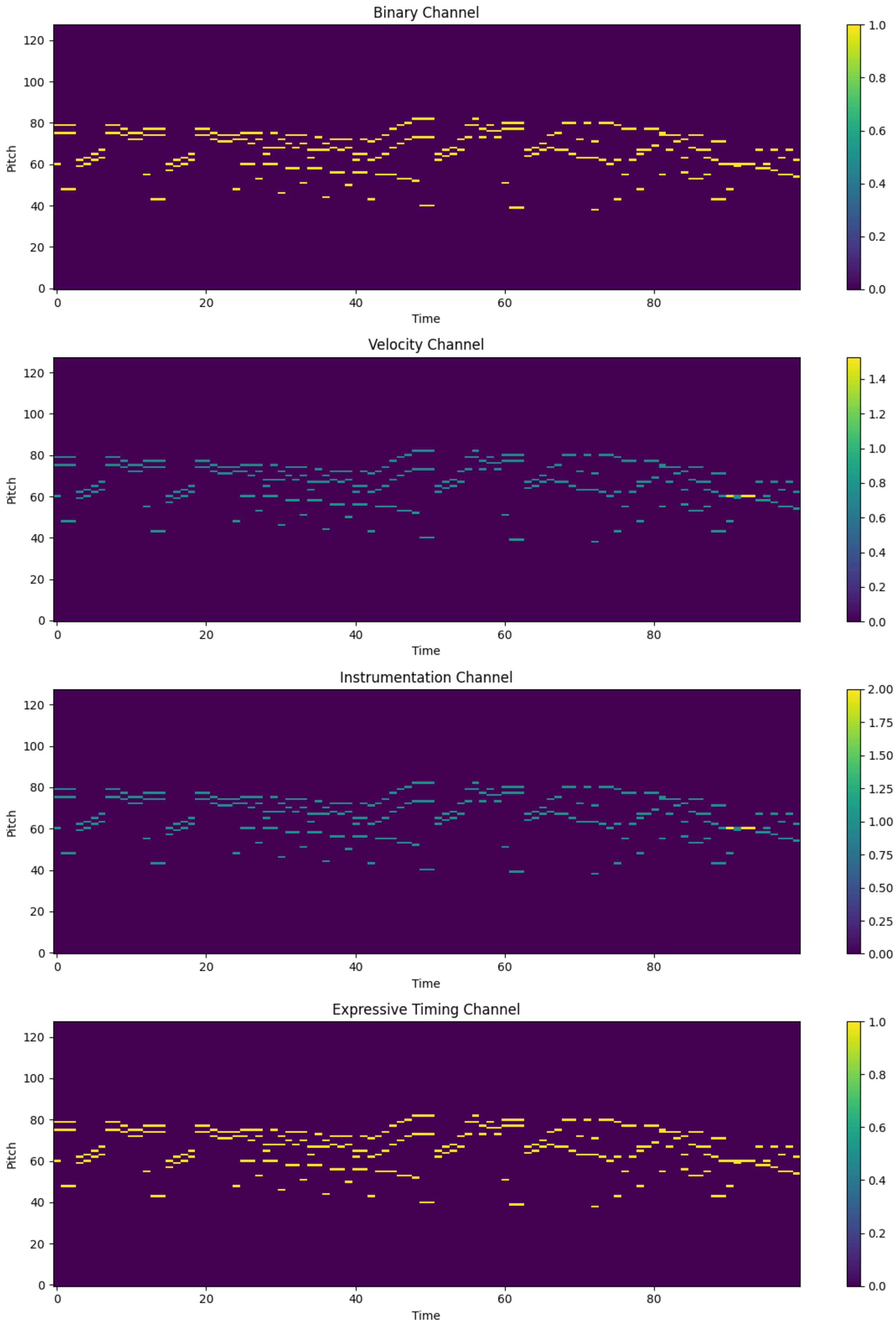


Binary Channel
Velocity Channel
Instrumentation Channel
Expressive Timing Channel
Pitch
Time

[ ]:
```
processed_data = process_multichannel_midi(test_file, fs=8)
plot_multichannel_piano_roll(processed_data)
```


Binary Channel
Pitch
Time
Velocity Channel
Pitch
Time
Instrumentation Channel
Pitch
Time
Expressive Timing Channel
Pitch
Time

[ ]:
```
processed_data = process_multichannel_midi(test_file, fs=4)
plot_multichannel_piano_roll(processed_data)
```


Binary Channel
Pitch
Time
Velocity Channel
Pitch
Time
Instrumentation Channel
Pitch
Time
Expressive Timing Channel
Pitch
Time

A number around 8 sounds good, will be using 10, just to make it a little compressed.

0.5.3 Preparing Data

Chunks We divided MIDI files into chunks to classify segments by the artist. This approach reduces bias from different file lengths and focuses on smaller sections instead of the full piece. Also, there is a function to visualize the chunk if needed.

Also, there is an overlap of information between Binary and Expressive Timing Roll and also between Velocity and Instrumentation Roll. So, we ended up only using the Binary and Velocity.

[ ]:
```
def midi_to_chunks(file_path, chunk_size=150, fs=10):
    midi_data = pretty_midi.PrettyMIDI(file_path)
    piano_roll = midi_data.get_piano_roll(fs=fs)
    num_chunks = piano_roll.shape[1] // chunk_size

    chunks = []

    # Creating fixed-size chunks
    for i in range(num_chunks):
        start = i * chunk_size
        end = start + chunk_size
        chunk = piano_roll[:, start:end]

        # Converting to binary and velocity channels
        binary = (chunk > 0).astype(int)
        velocity = chunk / 127

        # Stacking channels
        multichannel_chunk = np.stack([binary, velocity], axis=-1)
        chunks.append(multichannel_chunk)

    # Handling the last chunk if it doesn't fit perfectly
    if piano_roll.shape[1] % chunk_size != 0:
        last_chunk = piano_roll[:, num_chunks * chunk_size :]
        if last_chunk.shape[1] < chunk_size:
            padding = np.zeros((128, chunk_size - last_chunk.shape[1], 2))
            last_chunk_padded = np.stack(
                [(last_chunk > 0).astype(int), last_chunk / 127], axis=-1
            )
            last_chunk_padded = np.concatenate([last_chunk_padded, padding],␣
↪axis=1)
            chunks.append(last_chunk_padded)

    return chunks
```

```python
file_path = test_file
chunks = midi_to_chunks(file_path)
```

[ ]:
```python
def visualize_chunks(chunks):
    num_chunks = len(chunks)
    fig, axes = plt.subplots(
        num_chunks, 2, figsize=(15, 3 * num_chunks)
    )  # 2 columns for binary and velocity

    if num_chunks == 1:
        axes = [axes]

    for i, chunk in enumerate(chunks):
        # Binary Channel
        ax1 = axes[i][0] if num_chunks > 1 else axes[0]
        binary_channel = chunk[:, :, 0]  # Assuming binary channel is the first␣
↪channel
        cax1 = ax1.imshow(
            binary_channel,
            aspect="auto",
            origin="lower",
            cmap="gray",
            interpolation="none",
        )
        ax1.set_title(f"Chunk {i+1} - Binary Channel")
        ax1.set_xlabel("Time (frames)")
        ax1.set_ylabel("Pitch")
        fig.colorbar(cax1, ax=ax1, orientation="vertical")

        # Velocity Channel
        ax2 = axes[i][1] if num_chunks > 1 else axes[1]
        velocity_channel = chunk[
            :, :, 1
        ]  # Assuming velocity channel is the second channel
        cax2 = ax2.imshow(
            velocity_channel,
            aspect="auto",
            origin="lower",
            cmap="viridis",
            interpolation="none",
        )
        ax2.set_title(f"Chunk {i+1} - Velocity Channel")
        ax2.set_xlabel("Time (frames)")
        ax2.set_ylabel("Pitch")
        fig.colorbar(cax2, ax=ax2, orientation="vertical")

    plt.tight_layout()
```

```
    plt.show()
```

Creating DataFrame of MIDI Files

```
[ ]: # In case if we want to read from a previous run file.
     # paths_artist_length_data  = pd.read_pickle('paths_artist_length_data.pkl')
```

```
[ ]: # Constructing the full file path if necessary
     def construct_file_path(base_url, relative_path):
         if not relative_path.startswith(base_url):
             return f"{base_url}/{relative_path}"
         return relative_path


     # Iterating over each file to create chunks
     def process_all_files(df, base_url, fs=10, chunk_size=150):
         all_chunks = []

         for idx, row in df.iterrows():
             file_path = construct_file_path(base_url, row["path"])
             artist = row["artist"]
             chunks = midi_to_chunks(file_path, chunk_size=chunk_size, fs=fs)

             # Collecting chunks with additional metadata
             for i, chunk in enumerate(chunks):
                 all_chunks.append(
                     [chunk, artist, row["path"], i + 1, chunk.shape[1] < chunk_size]
                 )

         # Creating a DataFrame
         columns = ["Chunk", "Artist", "Original Path", "Chunk Number", "Padding␣
      ↪Added"]
         chunk_df = pd.DataFrame(all_chunks, columns=columns)

         return chunk_df
```

```
[ ]: processed_chunk_df = process_all_files(
         paths_artist_length_data, raw_data_extracted, fs=8, chunk_size=150
     )
     processed_chunk_df.to_pickle("processed_chunk_df.pkl")
```

Synthetic Data Check

```
[ ]: print('How many chunks has padding')
     print(processed_chunk_df["Padding Added"].value_counts())
     print(processed_chunk_df["Padding Added"].value_counts(normalize=True) * 100)
```

```
How many chunks has padding
No     39720
```

```
Yes     1633
Name: Padding Added, dtype: int64
No     96.051072
Yes     3.948928
Name: Padding Added, dtype: float64
```

Pretty good percentage.

[ ]:
```
processed_chunk_df['Chunk'].iloc[0].shape
```

[ ]:
```
(128, 150)
```

Input Feature (X)

[ ]:
```
# Preprocessing chunks from a DataFrame into a format suitable (numpy array)␣
↪for CNN input
def preprocess_chunks(dataframe, chunk_size=150):

    processed_chunks = []

    for chunk in dataframe["Chunk"]:
        if isinstance(chunk, np.ndarray):
            if chunk.shape[1] != chunk_size:
                if chunk.shape[1] < chunk_size:
                    padding = np.zeros((128, chunk_size - chunk.shape[1]))
                    chunk = np.hstack((chunk, padding))
                else:
                    chunk = chunk[:, :chunk_size]
            processed_chunks.append(chunk)
        else:
            print("Chunk is not a numpy array. Check data preparation steps.")

    # Normalizing data as well
    X = np.stack(processed_chunks) / 127.0
    X = X.reshape(-1, 128, chunk_size, 1)
    return X
```

[ ]:
```
X = preprocess_chunks(processed_chunk_df)
X.shape
```

[ ]:
```
(41353, 128, 150, 1)
```

Target Variable (y)

[ ]:
```
# Encoding labels into one-hot format and returning the encoder
def encode_labels(labels):
    label_encoder = LabelEncoder()
    integer_encoded = label_encoder.fit_transform(labels)
```

```
    onehot_encoder = OneHotEncoder(sparse=False)
    integer_encoded = integer_encoded.reshape(-1, 1)
    onehot_encoded = onehot_encoder.fit_transform(integer_encoded)

    return onehot_encoded, label_encoder
```

[ ]:
```
y, label_encoder = encode_labels(processed_chunk_df["Artist"])
y.shape
```

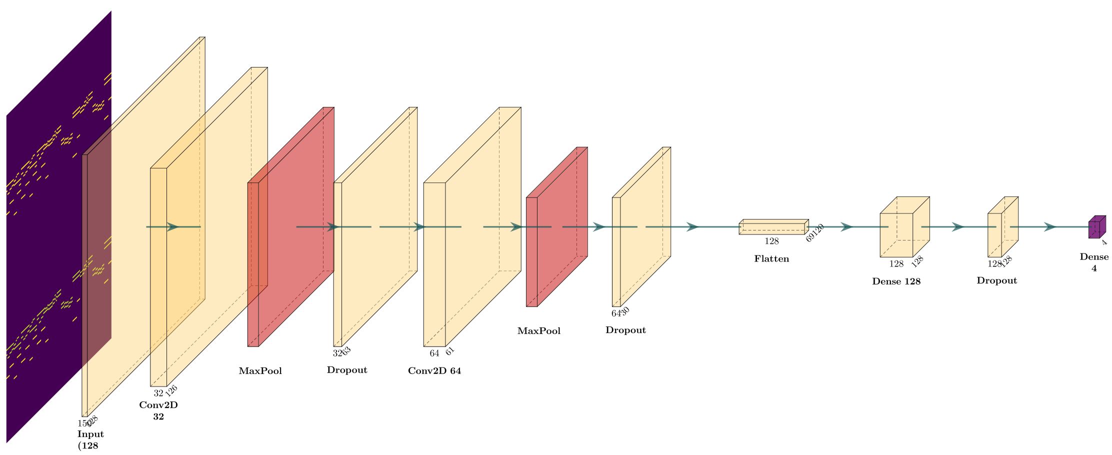
[ ]: (41353, 4)

[ ]:
```
X, y = shuffle(X, y, random_state=42)
```

[ ]:
```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

0.5.4 Defining the CNN Model

Here is what the architecture looks like:



We start with the piano rolls of Binary and Velocity channels and it goes through several layers, ending up within one of the Artists in the end.

[ ]:
```
model = Sequential(
    [
        Conv2D(32, (3, 3), activation="relu", input_shape=X.shape[1:]),
        MaxPooling2D((2, 2)),
        Dropout(0.25),
        Conv2D(64, (3, 3), activation="relu"),
        MaxPooling2D((2, 2)),
        Dropout(0.25),
        Flatten(),
        Dense(128, activation="relu"),
```

```
        Dropout(0.5),
        Dense(len(np.unique(processed_chunk_df["Artist"])),␣
 ↪activation="softmax"),
    ]
)

model.compile(optimizer="adam", loss="categorical_crossentropy",␣
 ↪metrics=["accuracy"])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 126, 148, 32)      320

 max_pooling2d (MaxPooling2D  (None, 63, 74, 32)       0
 )

 dropout (Dropout)           (None, 63, 74, 32)        0

 conv2d_1 (Conv2D)           (None, 61, 72, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 30, 36, 64)       0
 2D)

 dropout_1 (Dropout)         (None, 30, 36, 64)        0

 flatten (Flatten)           (None, 69120)             0

 dense (Dense)               (None, 128)               8847488

 dropout_2 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 4)                 516

=================================================================
Total params: 8,866,820
Trainable params: 8,866,820
Non-trainable params: 0
_________________________________________________________________
```

0.5.5 Training the Model

[ ]:
```
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_val, y_val))
```

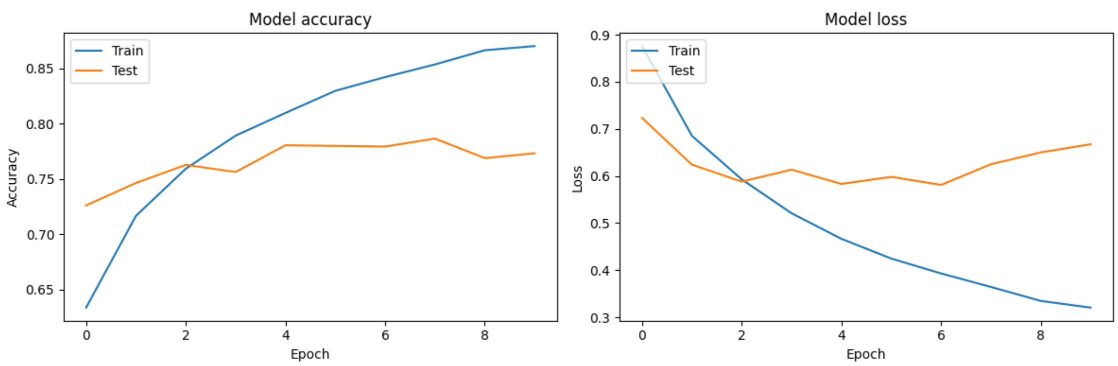
```
Epoch 1/10
1034/1034 [==============================] - 11s 9ms/step - loss: 0.8562 -
```

```
accuracy: 0.6424 - val_loss: 0.6854 - val_accuracy: 0.7225
Epoch 2/10
1034/1034 [==============================] - 8s 8ms/step - loss: 0.6538 -
accuracy: 0.7341 - val_loss: 0.6077 - val_accuracy: 0.7631
Epoch 3/10
1034/1034 [==============================] - 8s 8ms/step - loss: 0.5437 -
accuracy: 0.7789 - val_loss: 0.5947 - val_accuracy: 0.7704
Epoch 4/10
1034/1034 [==============================] - 8s 7ms/step - loss: 0.4617 -
accuracy: 0.8185 - val_loss: 0.5790 - val_accuracy: 0.7820
Epoch 5/10
1034/1034 [==============================] - 8s 8ms/step - loss: 0.3891 -
accuracy: 0.8448 - val_loss: 0.5915 - val_accuracy: 0.7900
Epoch 6/10
1034/1034 [==============================] - 8s 8ms/step - loss: 0.3249 -
accuracy: 0.8718 - val_loss: 0.6694 - val_accuracy: 0.7861
Epoch 7/10
1034/1034 [==============================] - 8s 8ms/step - loss: 0.2917 -
accuracy: 0.8881 - val_loss: 0.6093 - val_accuracy: 0.7913
Epoch 8/10
1034/1034 [==============================] - 8s 8ms/step - loss: 0.2574 -
accuracy: 0.9024 - val_loss: 0.6763 - val_accuracy: 0.7999
Epoch 9/10
1034/1034 [==============================] - 8s 7ms/step - loss: 0.2248 -
accuracy: 0.9124 - val_loss: 0.6757 - val_accuracy: 0.7994
Epoch 10/10
1034/1034 [==============================] - 8s 8ms/step - loss: 0.2069 -
accuracy: 0.9183 - val_loss: 0.7263 - val_accuracy: 0.8021
```

0.5.6 Evaluating the Model

Visualizing Training History

[ ]:
```
plot_training_history(history)
```



Evaluation Metrics

```
[ ]: val_loss, val_accuracy = model.evaluate(X_val, y_val)
print(f'Validation accuracy: {val_accuracy:.2f}, Validation loss: {val_loss:.
↪2f}')
```

```
259/259 [==============================] - 1s 4ms/step - loss: 0.7263 -
accuracy: 0.8021
Validation accuracy: 0.80, Validation loss: 0.73
```

```
[ ]: predictions = model.predict(X_val)
predicted_labels = label_encoder.inverse_transform(np.argmax(predictions,␣
↪axis=1))
# Print the top 5 predictions
for i in range(min(5, len(predictions))):
    print(f"{predicted_labels[i]}")
```

Beethoven

Bach

Bach

Bach

Bach

0.5.7 Optimization

We built a CNN model with adjustable parameters and used `RandomizedSearchCV` to find the optimal combination of hyperparameters like optimizer, initializer, dropout rate, epochs, and batch size to achieve the best model performance.

```
[ ]: # Model to test different parameters against
def create_model(optimizer="adam", init="glorot_uniform", dropout_rate=0.5):
    model = Sequential(
        [
            Conv2D(
                32,
                (3, 3),
                activation="relu",
                kernel_initializer=init,
                input_shape=X.shape[1:],
            ),
            MaxPooling2D((2, 2)),
            Dropout(dropout_rate),
            Conv2D(64, (3, 3), activation="relu", kernel_initializer=init),
            MaxPooling2D((2, 2)),
            Dropout(dropout_rate),
            Flatten(),
            Dense(128, activation="relu", kernel_initializer=init),
            Dropout(dropout_rate),
```

```
            Dense(
                len(np.unique(processed_chunk_df["Artist"])),
                activation="softmax",
                kernel_initializer=init,
            ),
        ]
    )

    model.compile(
        optimizer=optimizer, loss="categorical_crossentropy",␣
↪metrics=["accuracy"]
    )
    return model


model = KerasClassifier(build_fn=create_model, verbose=1)
```

[ ]:
```
# Parameter grid for RandomizedSearchCV
param_grid = {
    "optimizer": ["adam", "sgd"],
    "init": ["glorot_uniform", "he_normal"],
    "dropout_rate": [0.4, 0.5],
    "epochs": [10, 20],
    "batch_size": [20, 30],
}

# Initialize and run RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=model, param_distributions=param_grid, n_iter=10, cv=3, verbose=1
)
random_search_result = random_search.fit(X, y)

print(
    "Best: %f using %s"
    % (random_search_result.best_score_, random_search_result.best_params_)
)

# Please do not mind the scrolling. We initially decided to remove the output
# but decided to keep the results at the last minute as they provide useful
# information to track back.
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
Epoch 1/20
1379/1379 [==============================] - 7s 5ms/step - loss: 1.1217 -
accuracy: 0.5268
Epoch 2/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9988 -
accuracy: 0.5866
```

```
Epoch 3/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9364 -
accuracy: 0.6076
Epoch 4/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8814 -
accuracy: 0.6290
Epoch 5/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8344 -
accuracy: 0.6504
Epoch 6/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7896 -
accuracy: 0.6678
Epoch 7/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7596 -
accuracy: 0.6793
Epoch 8/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7280 -
accuracy: 0.6960
Epoch 9/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6983 -
accuracy: 0.7081
Epoch 10/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6793 -
accuracy: 0.7176
Epoch 11/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6567 -
accuracy: 0.7276
Epoch 12/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6372 -
accuracy: 0.7370
Epoch 13/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6145 -
accuracy: 0.7459
Epoch 14/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5989 -
accuracy: 0.7525
Epoch 15/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5775 -
accuracy: 0.7602
Epoch 16/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5587 -
accuracy: 0.7713
Epoch 17/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5435 -
accuracy: 0.7768
Epoch 18/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5286 -
accuracy: 0.7836
```

```
Epoch 19/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5158 -
accuracy: 0.7900
Epoch 20/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.4948 -
accuracy: 0.7962
690/690 [==============================] - 2s 3ms/step - loss: 0.6479 -
accuracy: 0.7370
Epoch 1/20
1379/1379 [==============================] - 7s 5ms/step - loss: 1.1276 -
accuracy: 0.5323
Epoch 2/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9934 -
accuracy: 0.5881
Epoch 3/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9255 -
accuracy: 0.6115
Epoch 4/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8779 -
accuracy: 0.6281
Epoch 5/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8351 -
accuracy: 0.6512
Epoch 6/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8063 -
accuracy: 0.6625
Epoch 7/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7681 -
accuracy: 0.6757
Epoch 8/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7419 -
accuracy: 0.6876
Epoch 9/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7195 -
accuracy: 0.6982
Epoch 10/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7006 -
accuracy: 0.7097
Epoch 11/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6805 -
accuracy: 0.7168
Epoch 12/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6642 -
accuracy: 0.7233
Epoch 13/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6477 -
accuracy: 0.7290
Epoch 14/20
```

```
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6309 -
accuracy: 0.7419
Epoch 15/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6108 -
accuracy: 0.7437
Epoch 16/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6014 -
accuracy: 0.7514
Epoch 17/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5824 -
accuracy: 0.7599
Epoch 18/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5623 -
accuracy: 0.7681
Epoch 19/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5529 -
accuracy: 0.7727
Epoch 20/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5425 -
accuracy: 0.7767
690/690 [==============================] - 3s 4ms/step - loss: 0.6180 -
accuracy: 0.7562
Epoch 1/20
1379/1379 [==============================] - 7s 5ms/step - loss: 1.1616 -
accuracy: 0.4998
Epoch 2/20
1379/1379 [==============================] - 7s 5ms/step - loss: 1.0234 -
accuracy: 0.5652
Epoch 3/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9496 -
accuracy: 0.5930
Epoch 4/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9016 -
accuracy: 0.6182
Epoch 5/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8552 -
accuracy: 0.6381
Epoch 6/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8201 -
accuracy: 0.6518
Epoch 7/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7883 -
accuracy: 0.6682
Epoch 8/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7583 -
accuracy: 0.6842
Epoch 9/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7352 -
```

```
accuracy: 0.6923
Epoch 10/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7107 -
accuracy: 0.7013
Epoch 11/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6900 -
accuracy: 0.7134
Epoch 12/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6706 -
accuracy: 0.7234
Epoch 13/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6492 -
accuracy: 0.7317
Epoch 14/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6278 -
accuracy: 0.7406
Epoch 15/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6068 -
accuracy: 0.7501
Epoch 16/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5924 -
accuracy: 0.7552
Epoch 17/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5744 -
accuracy: 0.7652
Epoch 18/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5634 -
accuracy: 0.7658
Epoch 19/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5480 -
accuracy: 0.7739
Epoch 20/20
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5342 -
accuracy: 0.7848
690/690 [==============================] - 3s 4ms/step - loss: 0.6285 -
accuracy: 0.7485
Epoch 1/10
919/919 [==============================] - 6s 6ms/step - loss: 0.8985 -
accuracy: 0.6279
Epoch 2/10
919/919 [==============================] - 6s 7ms/step - loss: 0.7052 -
accuracy: 0.7150
Epoch 3/10
919/919 [==============================] - 6s 7ms/step - loss: 0.6149 -
accuracy: 0.7521
Epoch 4/10
919/919 [==============================] - 6s 6ms/step - loss: 0.5424 -
accuracy: 0.7779
```

```
Epoch 5/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4894 -
accuracy: 0.8025
Epoch 6/10
919/919 [==============================] - 6s 6ms/step - loss: 0.4531 -
accuracy: 0.8194
Epoch 7/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4070 -
accuracy: 0.8365
Epoch 8/10
919/919 [==============================] - 6s 7ms/step - loss: 0.3712 -
accuracy: 0.8511
Epoch 9/10
919/919 [==============================] - 6s 7ms/step - loss: 0.3420 -
accuracy: 0.8624
Epoch 10/10
919/919 [==============================] - 6s 7ms/step - loss: 0.3169 -
accuracy: 0.8735
460/460 [==============================] - 2s 4ms/step - loss: 0.6870 -
accuracy: 0.7673
Epoch 1/10
919/919 [==============================] - 6s 7ms/step - loss: 0.8718 -
accuracy: 0.6402
Epoch 2/10
919/919 [==============================] - 6s 7ms/step - loss: 0.6672 -
accuracy: 0.7285
Epoch 3/10
919/919 [==============================] - 6s 6ms/step - loss: 0.5709 -
accuracy: 0.7694
Epoch 4/10
919/919 [==============================] - 6s 7ms/step - loss: 0.5000 -
accuracy: 0.8004
Epoch 5/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4317 -
accuracy: 0.8269
Epoch 6/10
919/919 [==============================] - 6s 7ms/step - loss: 0.3793 -
accuracy: 0.8477
Epoch 7/10
919/919 [==============================] - 6s 6ms/step - loss: 0.3425 -
accuracy: 0.8594
Epoch 8/10
919/919 [==============================] - 6s 7ms/step - loss: 0.3038 -
accuracy: 0.8777
Epoch 9/10
919/919 [==============================] - 6s 6ms/step - loss: 0.2796 -
accuracy: 0.8865
Epoch 10/10
```

```
919/919 [==============================] - 6s 6ms/step - loss: 0.2570 -
accuracy: 0.8979
460/460 [==============================] - 2s 4ms/step - loss: 0.7348 -
accuracy: 0.7741
Epoch 1/10
919/919 [==============================] - 7s 7ms/step - loss: 0.8757 -
accuracy: 0.6348
Epoch 2/10
919/919 [==============================] - 6s 6ms/step - loss: 0.6888 -
accuracy: 0.7154
Epoch 3/10
919/919 [==============================] - 6s 7ms/step - loss: 0.5777 -
accuracy: 0.7652
Epoch 4/10
919/919 [==============================] - 6s 6ms/step - loss: 0.4988 -
accuracy: 0.8013
Epoch 5/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4359 -
accuracy: 0.8256
Epoch 6/10
919/919 [==============================] - 6s 7ms/step - loss: 0.3757 -
accuracy: 0.8511
Epoch 7/10
919/919 [==============================] - 6s 6ms/step - loss: 0.3366 -
accuracy: 0.8641
Epoch 8/10
919/919 [==============================] - 6s 6ms/step - loss: 0.3011 -
accuracy: 0.8817
Epoch 9/10
919/919 [==============================] - 6s 7ms/step - loss: 0.2782 -
accuracy: 0.8922
Epoch 10/10
919/919 [==============================] - 6s 7ms/step - loss: 0.2549 -
accuracy: 0.9004
460/460 [==============================] - 2s 4ms/step - loss: 0.7375 -
accuracy: 0.7693
Epoch 1/20
919/919 [==============================] - 6s 6ms/step - loss: 1.0702 -
accuracy: 0.5498
Epoch 2/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9372 -
accuracy: 0.6186
Epoch 3/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8596 -
accuracy: 0.6508
Epoch 4/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8108 -
accuracy: 0.6694
```

```
Epoch 5/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7664 -
accuracy: 0.6860
Epoch 6/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7331 -
accuracy: 0.7036
Epoch 7/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6958 -
accuracy: 0.7192
Epoch 8/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6686 -
accuracy: 0.7302
Epoch 9/20
919/919 [==============================] - 6s 7ms/step - loss: 0.6348 -
accuracy: 0.7459
Epoch 10/20
919/919 [==============================] - 6s 7ms/step - loss: 0.6061 -
accuracy: 0.7540
Epoch 11/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5761 -
accuracy: 0.7681
Epoch 12/20
919/919 [==============================] - 6s 7ms/step - loss: 0.5486 -
accuracy: 0.7835
Epoch 13/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5197 -
accuracy: 0.7933
Epoch 14/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4936 -
accuracy: 0.8058
Epoch 15/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4716 -
accuracy: 0.8097
Epoch 16/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4432 -
accuracy: 0.8257
Epoch 17/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4180 -
accuracy: 0.8364
Epoch 18/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4011 -
accuracy: 0.8424
Epoch 19/20
919/919 [==============================] - 6s 6ms/step - loss: 0.3739 -
accuracy: 0.8533
Epoch 20/20
919/919 [==============================] - 6s 6ms/step - loss: 0.3562 -
accuracy: 0.8625
```

```
460/460 [==============================] - 2s 4ms/step - loss: 0.6140 -
accuracy: 0.7685
Epoch 1/20
919/919 [==============================] - 6s 6ms/step - loss: 1.0635 -
accuracy: 0.5513
Epoch 2/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9324 -
accuracy: 0.6209
Epoch 3/20
919/919 [==============================] - 6s 7ms/step - loss: 0.8559 -
accuracy: 0.6510
Epoch 4/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8024 -
accuracy: 0.6740
Epoch 5/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7582 -
accuracy: 0.6922
Epoch 6/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7243 -
accuracy: 0.7054
Epoch 7/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6917 -
accuracy: 0.7209
Epoch 8/20
919/919 [==============================] - 6s 7ms/step - loss: 0.6522 -
accuracy: 0.7368
Epoch 9/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6232 -
accuracy: 0.7491
Epoch 10/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5999 -
accuracy: 0.7598
Epoch 11/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5729 -
accuracy: 0.7716
Epoch 12/20
919/919 [==============================] - 6s 7ms/step - loss: 0.5463 -
accuracy: 0.7826
Epoch 13/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5197 -
accuracy: 0.7918
Epoch 14/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4900 -
accuracy: 0.8061
Epoch 15/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4667 -
accuracy: 0.8153
Epoch 16/20
```

```
919/919 [==============================] - 6s 7ms/step - loss: 0.4466 -
accuracy: 0.8237
Epoch 17/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4211 -
accuracy: 0.8342
Epoch 18/20
919/919 [==============================] - 6s 6ms/step - loss: 0.3942 -
accuracy: 0.8480
Epoch 19/20
919/919 [==============================] - 6s 6ms/step - loss: 0.3771 -
accuracy: 0.8562
Epoch 20/20
919/919 [==============================] - 6s 6ms/step - loss: 0.3577 -
accuracy: 0.8594
460/460 [==============================] - 2s 4ms/step - loss: 0.6114 -
accuracy: 0.7717
Epoch 1/20
919/919 [==============================] - 6s 6ms/step - loss: 1.0634 -
accuracy: 0.5486
Epoch 2/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9291 -
accuracy: 0.6203
Epoch 3/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8573 -
accuracy: 0.6471
Epoch 4/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8014 -
accuracy: 0.6721
Epoch 5/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7644 -
accuracy: 0.6862
Epoch 6/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7305 -
accuracy: 0.7039
Epoch 7/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6982 -
accuracy: 0.7169
Epoch 8/20
919/919 [==============================] - 6s 7ms/step - loss: 0.6682 -
accuracy: 0.7284
Epoch 9/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6364 -
accuracy: 0.7416
Epoch 10/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6097 -
accuracy: 0.7533
Epoch 11/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5808 -
```

```
accuracy: 0.7693
Epoch 12/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5541 -
accuracy: 0.7785
Epoch 13/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5298 -
accuracy: 0.7865
Epoch 14/20
919/919 [==============================] - 6s 7ms/step - loss: 0.5041 -
accuracy: 0.7997
Epoch 15/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4808 -
accuracy: 0.8087
Epoch 16/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4540 -
accuracy: 0.8213
Epoch 17/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4293 -
accuracy: 0.8318
Epoch 18/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4059 -
accuracy: 0.8404
Epoch 19/20
919/919 [==============================] - 6s 6ms/step - loss: 0.3875 -
accuracy: 0.8496
Epoch 20/20
919/919 [==============================] - 6s 6ms/step - loss: 0.3703 -
accuracy: 0.8563
460/460 [==============================] - 3s 5ms/step - loss: 0.6157 -
accuracy: 0.7652
Epoch 1/10
1379/1379 [==============================] - 7s 5ms/step - loss: 1.0651 -
accuracy: 0.5470
Epoch 2/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9427 -
accuracy: 0.6152
Epoch 3/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8750 -
accuracy: 0.6415
Epoch 4/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8313 -
accuracy: 0.6624
Epoch 5/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7873 -
accuracy: 0.6792
Epoch 6/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7477 -
accuracy: 0.6944
```

```
Epoch 7/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7142 -
accuracy: 0.7122
Epoch 8/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6851 -
accuracy: 0.7253
Epoch 9/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6569 -
accuracy: 0.7349
Epoch 10/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6326 -
accuracy: 0.7438
690/690 [==============================] - 3s 3ms/step - loss: 0.6523 -
accuracy: 0.7363
Epoch 1/10
1379/1379 [==============================] - 8s 5ms/step - loss: 1.0606 -
accuracy: 0.5560
Epoch 2/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9340 -
accuracy: 0.6183
Epoch 3/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8763 -
accuracy: 0.6417
Epoch 4/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8322 -
accuracy: 0.6598
Epoch 5/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7957 -
accuracy: 0.6738
Epoch 6/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7610 -
accuracy: 0.6906
Epoch 7/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7290 -
accuracy: 0.7018
Epoch 8/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7000 -
accuracy: 0.7149
Epoch 9/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6743 -
accuracy: 0.7273
Epoch 10/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6522 -
accuracy: 0.7355
690/690 [==============================] - 2s 3ms/step - loss: 0.7158 -
accuracy: 0.7252
Epoch 1/10
1379/1379 [==============================] - 8s 5ms/step - loss: 1.0643 -
```

```
accuracy: 0.5482
Epoch 2/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9331 -
accuracy: 0.6164
Epoch 3/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8823 -
accuracy: 0.6432
Epoch 4/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8411 -
accuracy: 0.6573
Epoch 5/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7925 -
accuracy: 0.6747
Epoch 6/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7626 -
accuracy: 0.6894
Epoch 7/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7331 -
accuracy: 0.7040
Epoch 8/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7012 -
accuracy: 0.7139
Epoch 9/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6787 -
accuracy: 0.7234
Epoch 10/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6560 -
accuracy: 0.7321
690/690 [==============================] - 7s 3ms/step - loss: 0.7162 -
accuracy: 0.7081
Epoch 1/10
919/919 [==============================] - 7s 7ms/step - loss: 1.1346 -
accuracy: 0.5143
Epoch 2/10
919/919 [==============================] - 6s 6ms/step - loss: 1.0101 -
accuracy: 0.5776
Epoch 3/10
919/919 [==============================] - 6s 6ms/step - loss: 0.9558 -
accuracy: 0.6043
Epoch 4/10
919/919 [==============================] - 6s 7ms/step - loss: 0.9104 -
accuracy: 0.6268
Epoch 5/10
919/919 [==============================] - 6s 6ms/step - loss: 0.8697 -
accuracy: 0.6420
Epoch 6/10
919/919 [==============================] - 6s 6ms/step - loss: 0.8325 -
accuracy: 0.6448
```

```
Epoch 7/10
919/919 [==============================] - 6s 6ms/step - loss: 0.8088 -
accuracy: 0.6625
Epoch 8/10
919/919 [==============================] - 6s 6ms/step - loss: 0.7738 -
accuracy: 0.6722
Epoch 9/10
919/919 [==============================] - 6s 6ms/step - loss: 0.7566 -
accuracy: 0.6831
Epoch 10/10
919/919 [==============================] - 6s 7ms/step - loss: 0.7336 -
accuracy: 0.6951
460/460 [==============================] - 2s 4ms/step - loss: 0.7324 -
accuracy: 0.6947
Epoch 1/10
919/919 [==============================] - 6s 7ms/step - loss: 1.1332 -
accuracy: 0.5223
Epoch 2/10
919/919 [==============================] - 6s 6ms/step - loss: 0.9879 -
accuracy: 0.5849
Epoch 3/10
919/919 [==============================] - 6s 6ms/step - loss: 0.9371 -
accuracy: 0.6108
Epoch 4/10
919/919 [==============================] - 6s 7ms/step - loss: 0.8886 -
accuracy: 0.6288
Epoch 5/10
919/919 [==============================] - 6s 6ms/step - loss: 0.8483 -
accuracy: 0.6441
Epoch 6/10
919/919 [==============================] - 6s 6ms/step - loss: 0.8184 -
accuracy: 0.6574
Epoch 7/10
919/919 [==============================] - 6s 6ms/step - loss: 0.7885 -
accuracy: 0.6686
Epoch 8/10
919/919 [==============================] - 6s 6ms/step - loss: 0.7582 -
accuracy: 0.6808
Epoch 9/10
919/919 [==============================] - 6s 6ms/step - loss: 0.7429 -
accuracy: 0.6885
Epoch 10/10
919/919 [==============================] - 6s 6ms/step - loss: 0.7205 -
accuracy: 0.6975
460/460 [==============================] - 2s 4ms/step - loss: 0.6986 -
accuracy: 0.7149
Epoch 1/10
919/919 [==============================] - 6s 6ms/step - loss: 1.1145 -
```

```
accuracy: 0.5245
Epoch 2/10
919/919 [==============================] - 6s 6ms/step - loss: 0.9991 -
accuracy: 0.5846
Epoch 3/10
919/919 [==============================] - 6s 6ms/step - loss: 0.9554 -
accuracy: 0.6064
Epoch 4/10
919/919 [==============================] - 6s 6ms/step - loss: 0.9105 -
accuracy: 0.6213
Epoch 5/10
919/919 [==============================] - 6s 6ms/step - loss: 0.8730 -
accuracy: 0.6359
Epoch 6/10
919/919 [==============================] - 6s 6ms/step - loss: 0.8400 -
accuracy: 0.6523
Epoch 7/10
919/919 [==============================] - 6s 6ms/step - loss: 0.8068 -
accuracy: 0.6593
Epoch 8/10
919/919 [==============================] - 6s 6ms/step - loss: 0.7819 -
accuracy: 0.6726
Epoch 9/10
919/919 [==============================] - 6s 6ms/step - loss: 0.7602 -
accuracy: 0.6823
Epoch 10/10
919/919 [==============================] - 6s 6ms/step - loss: 0.7435 -
accuracy: 0.6855
460/460 [==============================] - 2s 4ms/step - loss: 0.7502 -
accuracy: 0.6846
Epoch 1/10
1379/1379 [==============================] - 8s 5ms/step - loss: 0.9636 -
accuracy: 0.6075
Epoch 2/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7691 -
accuracy: 0.6795
Epoch 3/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6863 -
accuracy: 0.7163
Epoch 4/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6125 -
accuracy: 0.7461
Epoch 5/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5613 -
accuracy: 0.7684
Epoch 6/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5099 -
accuracy: 0.7909
```

```
Epoch 7/10
1379/1379 [==============================] - 8s 6ms/step - loss: 0.4609 -
accuracy: 0.8129
Epoch 8/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.4280 -
accuracy: 0.8251
Epoch 9/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.3922 -
accuracy: 0.8413
Epoch 10/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.3675 -
accuracy: 0.8514
690/690 [==============================] - 3s 3ms/step - loss: 0.6960 -
accuracy: 0.7648
Epoch 1/10
1379/1379 [==============================] - 8s 5ms/step - loss: 0.9863 -
accuracy: 0.6102
Epoch 2/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7781 -
accuracy: 0.6755
Epoch 3/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6900 -
accuracy: 0.7141
Epoch 4/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6227 -
accuracy: 0.7419
Epoch 5/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5666 -
accuracy: 0.7669
Epoch 6/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5169 -
accuracy: 0.7901
Epoch 7/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.4753 -
accuracy: 0.8020
Epoch 8/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.4402 -
accuracy: 0.8222
Epoch 9/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.4120 -
accuracy: 0.8252
Epoch 10/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.3874 -
accuracy: 0.8407
690/690 [==============================] - 3s 4ms/step - loss: 0.6294 -
accuracy: 0.7694
Epoch 1/10
1379/1379 [==============================] - 8s 5ms/step - loss: 0.9734 -
```

```
accuracy: 0.6076
Epoch 2/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7601 -
accuracy: 0.6853
Epoch 3/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6625 -
accuracy: 0.7263
Epoch 4/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5903 -
accuracy: 0.7603
Epoch 5/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5299 -
accuracy: 0.7852
Epoch 6/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.4820 -
accuracy: 0.8054
Epoch 7/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.4364 -
accuracy: 0.8261
Epoch 8/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.4052 -
accuracy: 0.8349
Epoch 9/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.3735 -
accuracy: 0.8500
Epoch 10/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.3446 -
accuracy: 0.8626
690/690 [==============================] - 2s 3ms/step - loss: 0.6590 -
accuracy: 0.7609
Epoch 1/20
919/919 [==============================] - 6s 7ms/step - loss: 1.0751 -
accuracy: 0.5469
Epoch 2/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9593 -
accuracy: 0.6106
Epoch 3/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9004 -
accuracy: 0.6336
Epoch 4/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8586 -
accuracy: 0.6518
Epoch 5/20
919/919 [==============================] - 6s 7ms/step - loss: 0.8165 -
accuracy: 0.6675
Epoch 6/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7912 -
accuracy: 0.6782
```

```
Epoch 7/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7621 -
accuracy: 0.6910
Epoch 8/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7366 -
accuracy: 0.7010
Epoch 9/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7120 -
accuracy: 0.7108
Epoch 10/20
919/919 [==============================] - 6s 7ms/step - loss: 0.6936 -
accuracy: 0.7206
Epoch 11/20
919/919 [==============================] - 6s 7ms/step - loss: 0.6751 -
accuracy: 0.7258
Epoch 12/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6488 -
accuracy: 0.7348
Epoch 13/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6298 -
accuracy: 0.7443
Epoch 14/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6099 -
accuracy: 0.7543
Epoch 15/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5910 -
accuracy: 0.7631
Epoch 16/20
919/919 [==============================] - 6s 7ms/step - loss: 0.5727 -
accuracy: 0.7737
Epoch 17/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5597 -
accuracy: 0.7758
Epoch 18/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5387 -
accuracy: 0.7854
Epoch 19/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5228 -
accuracy: 0.7902
Epoch 20/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5069 -
accuracy: 0.7986
460/460 [==============================] - 2s 4ms/step - loss: 0.6129 -
accuracy: 0.7590
Epoch 1/20
919/919 [==============================] - 7s 7ms/step - loss: 1.0696 -
accuracy: 0.5449
Epoch 2/20
```

```
919/919 [==============================] - 6s 6ms/step - loss: 0.9521 -
accuracy: 0.6114
Epoch 3/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8994 -
accuracy: 0.6342
Epoch 4/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8612 -
accuracy: 0.6503
Epoch 5/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8255 -
accuracy: 0.6653
Epoch 6/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7962 -
accuracy: 0.6778
Epoch 7/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7680 -
accuracy: 0.6858
Epoch 8/20
919/919 [==============================] - 6s 7ms/step - loss: 0.7444 -
accuracy: 0.7013
Epoch 9/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7151 -
accuracy: 0.7094
Epoch 10/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6940 -
accuracy: 0.7194
Epoch 11/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6732 -
accuracy: 0.7262
Epoch 12/20
919/919 [==============================] - 6s 7ms/step - loss: 0.6514 -
accuracy: 0.7385
Epoch 13/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6348 -
accuracy: 0.7432
Epoch 14/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6146 -
accuracy: 0.7510
Epoch 15/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5983 -
accuracy: 0.7605
Epoch 16/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5798 -
accuracy: 0.7654
Epoch 17/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5644 -
accuracy: 0.7740
Epoch 18/20
```

```
919/919 [==============================] - 6s 6ms/step - loss: 0.5484 -
accuracy: 0.7765
Epoch 19/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5305 -
accuracy: 0.7903
Epoch 20/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5139 -
accuracy: 0.7942
460/460 [==============================] - 2s 4ms/step - loss: 0.6036 -
accuracy: 0.7611
Epoch 1/20
919/919 [==============================] - 6s 6ms/step - loss: 1.0832 -
accuracy: 0.5371
Epoch 2/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9619 -
accuracy: 0.6077
Epoch 3/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8957 -
accuracy: 0.6337
Epoch 4/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8465 -
accuracy: 0.6496
Epoch 5/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8093 -
accuracy: 0.6675
Epoch 6/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7848 -
accuracy: 0.6783
Epoch 7/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7489 -
accuracy: 0.6918
Epoch 8/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7215 -
accuracy: 0.7039
Epoch 9/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6947 -
accuracy: 0.7169
Epoch 10/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6695 -
accuracy: 0.7268
Epoch 11/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6479 -
accuracy: 0.7392
Epoch 12/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6341 -
accuracy: 0.7442
Epoch 13/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6067 -
```

```
accuracy: 0.7542
Epoch 14/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5853 -
accuracy: 0.7636
Epoch 15/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5693 -
accuracy: 0.7699
Epoch 16/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5536 -
accuracy: 0.7765
Epoch 17/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5398 -
accuracy: 0.7810
Epoch 18/20
919/919 [==============================] - 5s 6ms/step - loss: 0.5192 -
accuracy: 0.7899
Epoch 19/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4993 -
accuracy: 0.8017
Epoch 20/20
919/919 [==============================] - 6s 6ms/step - loss: 0.4798 -
accuracy: 0.8066
460/460 [==============================] - 3s 4ms/step - loss: 0.5976 -
accuracy: 0.7657
Epoch 1/10
1379/1379 [==============================] - 7s 5ms/step - loss: 1.0446 -
accuracy: 0.5586
Epoch 2/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9103 -
accuracy: 0.6336
Epoch 3/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8497 -
accuracy: 0.6549
Epoch 4/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7971 -
accuracy: 0.6721
Epoch 5/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7493 -
accuracy: 0.6981
Epoch 6/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7081 -
accuracy: 0.7140
Epoch 7/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6681 -
accuracy: 0.7304
Epoch 8/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6298 -
accuracy: 0.7453
```

```
Epoch 9/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5972 -
accuracy: 0.7587
Epoch 10/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5579 -
accuracy: 0.7771
690/690 [==============================] - 2s 3ms/step - loss: 0.6216 -
accuracy: 0.7509
Epoch 1/10
1379/1379 [==============================] - 8s 5ms/step - loss: 1.0432 -
accuracy: 0.5625
Epoch 2/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9067 -
accuracy: 0.6302
Epoch 3/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8332 -
accuracy: 0.6588
Epoch 4/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7782 -
accuracy: 0.6812
Epoch 5/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7311 -
accuracy: 0.7007
Epoch 6/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6899 -
accuracy: 0.7198
Epoch 7/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6541 -
accuracy: 0.7334
Epoch 8/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6139 -
accuracy: 0.7526
Epoch 9/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5817 -
accuracy: 0.7655
Epoch 10/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5457 -
accuracy: 0.7819
690/690 [==============================] - 3s 4ms/step - loss: 0.6214 -
accuracy: 0.7520
Epoch 1/10
1379/1379 [==============================] - 7s 5ms/step - loss: 1.0419 -
accuracy: 0.5634
Epoch 2/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.9027 -
accuracy: 0.6326
Epoch 3/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.8290 -
```

```
accuracy: 0.6583
Epoch 4/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7775 -
accuracy: 0.6808
Epoch 5/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.7278 -
accuracy: 0.7025
Epoch 6/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6820 -
accuracy: 0.7219
Epoch 7/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6425 -
accuracy: 0.7395
Epoch 8/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.6087 -
accuracy: 0.7545
Epoch 9/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5754 -
accuracy: 0.7645
Epoch 10/10
1379/1379 [==============================] - 7s 5ms/step - loss: 0.5393 -
accuracy: 0.7817
690/690 [==============================] - 3s 3ms/step - loss: 0.6311 -
accuracy: 0.7507
Epoch 1/20
919/919 [==============================] - 6s 6ms/step - loss: 1.1031 -
accuracy: 0.5251
Epoch 2/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9864 -
accuracy: 0.5946
Epoch 3/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9298 -
accuracy: 0.6167
Epoch 4/20
919/919 [==============================] - 6s 7ms/step - loss: 0.8887 -
accuracy: 0.6313
Epoch 5/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8471 -
accuracy: 0.6449
Epoch 6/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8181 -
accuracy: 0.6578
Epoch 7/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7873 -
accuracy: 0.6698
Epoch 8/20
919/919 [==============================] - 6s 7ms/step - loss: 0.7616 -
accuracy: 0.6811
```

```
Epoch 9/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7406 -
accuracy: 0.6928
Epoch 10/20
919/919 [==============================] - 6s 7ms/step - loss: 0.7105 -
accuracy: 0.7069
Epoch 11/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6975 -
accuracy: 0.7119
Epoch 12/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6859 -
accuracy: 0.7182
Epoch 13/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6649 -
accuracy: 0.7267
Epoch 14/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6383 -
accuracy: 0.7370
Epoch 15/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6227 -
accuracy: 0.7444
Epoch 16/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6080 -
accuracy: 0.7511
Epoch 17/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5931 -
accuracy: 0.7606
Epoch 18/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5761 -
accuracy: 0.7676
Epoch 19/20
919/919 [==============================] - 6s 7ms/step - loss: 0.5624 -
accuracy: 0.7723
Epoch 20/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5493 -
accuracy: 0.7754
460/460 [==============================] - 2s 4ms/step - loss: 0.6186 -
accuracy: 0.7573
Epoch 1/20
919/919 [==============================] - 6s 6ms/step - loss: 1.1206 -
accuracy: 0.5241
Epoch 2/20
919/919 [==============================] - 6s 6ms/step - loss: 1.0101 -
accuracy: 0.5828
Epoch 3/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9633 -
accuracy: 0.6023
Epoch 4/20
```

```
919/919 [==============================] - 6s 6ms/step - loss: 0.9155 -
accuracy: 0.6220
Epoch 5/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8802 -
accuracy: 0.6327
Epoch 6/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8523 -
accuracy: 0.6403
Epoch 7/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8222 -
accuracy: 0.6587
Epoch 8/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7993 -
accuracy: 0.6679
Epoch 9/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7755 -
accuracy: 0.6742
Epoch 10/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7504 -
accuracy: 0.6849
Epoch 11/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7335 -
accuracy: 0.6966
Epoch 12/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7191 -
accuracy: 0.7003
Epoch 13/20
919/919 [==============================] - 5s 6ms/step - loss: 0.7046 -
accuracy: 0.7067
Epoch 14/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6752 -
accuracy: 0.7191
Epoch 15/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6622 -
accuracy: 0.7241
Epoch 16/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6520 -
accuracy: 0.7322
Epoch 17/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6261 -
accuracy: 0.7426
Epoch 18/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6174 -
accuracy: 0.7488
Epoch 19/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6033 -
accuracy: 0.7517
Epoch 20/20
```

```
919/919 [==============================] - 6s 6ms/step - loss: 0.5923 -
accuracy: 0.7577
460/460 [==============================] - 2s 4ms/step - loss: 0.6262 -
accuracy: 0.7468
Epoch 1/20
919/919 [==============================] - 6s 6ms/step - loss: 1.1306 -
accuracy: 0.5124
Epoch 2/20
919/919 [==============================] - 6s 6ms/step - loss: 1.0134 -
accuracy: 0.5784
Epoch 3/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9570 -
accuracy: 0.6030
Epoch 4/20
919/919 [==============================] - 6s 6ms/step - loss: 0.9097 -
accuracy: 0.6216
Epoch 5/20
919/919 [==============================] - 6s 7ms/step - loss: 0.8725 -
accuracy: 0.6350
Epoch 6/20
919/919 [==============================] - 6s 7ms/step - loss: 0.8402 -
accuracy: 0.6467
Epoch 7/20
919/919 [==============================] - 6s 6ms/step - loss: 0.8129 -
accuracy: 0.6593
Epoch 8/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7844 -
accuracy: 0.6708
Epoch 9/20
919/919 [==============================] - 6s 7ms/step - loss: 0.7604 -
accuracy: 0.6841
Epoch 10/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7334 -
accuracy: 0.6939
Epoch 11/20
919/919 [==============================] - 6s 6ms/step - loss: 0.7140 -
accuracy: 0.7036
Epoch 12/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6981 -
accuracy: 0.7104
Epoch 13/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6787 -
accuracy: 0.7198
Epoch 14/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6681 -
accuracy: 0.7259
Epoch 15/20
919/919 [==============================] - 6s 7ms/step - loss: 0.6445 -
```

```
accuracy: 0.7356
Epoch 16/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6253 -
accuracy: 0.7437
Epoch 17/20
919/919 [==============================] - 6s 6ms/step - loss: 0.6127 -
accuracy: 0.7473
Epoch 18/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5946 -
accuracy: 0.7540
Epoch 19/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5887 -
accuracy: 0.7586
Epoch 20/20
919/919 [==============================] - 6s 6ms/step - loss: 0.5713 -
accuracy: 0.7671
460/460 [==============================] - 2s 4ms/step - loss: 0.6269 -
accuracy: 0.7506
Epoch 1/10
919/919 [==============================] - 7s 6ms/step - loss: 1.0237 -
accuracy: 0.5877
Epoch 2/10
919/919 [==============================] - 6s 7ms/step - loss: 0.8080 -
accuracy: 0.6647
Epoch 3/10
919/919 [==============================] - 6s 7ms/step - loss: 0.7186 -
accuracy: 0.7000
Epoch 4/10
919/919 [==============================] - 6s 7ms/step - loss: 0.6488 -
accuracy: 0.7291
Epoch 5/10
919/919 [==============================] - 6s 7ms/step - loss: 0.5924 -
accuracy: 0.7549
Epoch 6/10
919/919 [==============================] - 6s 6ms/step - loss: 0.5430 -
accuracy: 0.7763
Epoch 7/10
919/919 [==============================] - 6s 7ms/step - loss: 0.5035 -
accuracy: 0.7920
Epoch 8/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4523 -
accuracy: 0.8097
Epoch 9/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4307 -
accuracy: 0.8240
Epoch 10/10
919/919 [==============================] - 6s 6ms/step - loss: 0.4052 -
accuracy: 0.8325
```

```
460/460 [==============================] - 2s 4ms/step - loss: 0.6449 -
accuracy: 0.7697
Epoch 1/10
919/919 [==============================] - 7s 7ms/step - loss: 0.9981 -
accuracy: 0.6025
Epoch 2/10
919/919 [==============================] - 6s 7ms/step - loss: 0.7905 -
accuracy: 0.6731
Epoch 3/10
919/919 [==============================] - 6s 6ms/step - loss: 0.7077 -
accuracy: 0.7029
Epoch 4/10
919/919 [==============================] - 6s 7ms/step - loss: 0.6452 -
accuracy: 0.7322
Epoch 5/10
919/919 [==============================] - 6s 6ms/step - loss: 0.5961 -
accuracy: 0.7539
Epoch 6/10
919/919 [==============================] - 6s 7ms/step - loss: 0.5474 -
accuracy: 0.7700
Epoch 7/10
919/919 [==============================] - 6s 6ms/step - loss: 0.5148 -
accuracy: 0.7861
Epoch 8/10
919/919 [==============================] - 6s 6ms/step - loss: 0.4812 -
accuracy: 0.7975
Epoch 9/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4525 -
accuracy: 0.8105
Epoch 10/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4288 -
accuracy: 0.8197
460/460 [==============================] - 2s 4ms/step - loss: 0.6230 -
accuracy: 0.7600
Epoch 1/10
919/919 [==============================] - 7s 7ms/step - loss: 1.1029 -
accuracy: 0.5848
Epoch 2/10
919/919 [==============================] - 6s 7ms/step - loss: 0.8009 -
accuracy: 0.6740
Epoch 3/10
919/919 [==============================] - 6s 7ms/step - loss: 0.7060 -
accuracy: 0.7085
Epoch 4/10
919/919 [==============================] - 6s 6ms/step - loss: 0.6385 -
accuracy: 0.7352
Epoch 5/10
919/919 [==============================] - 6s 7ms/step - loss: 0.5865 -
```

```
accuracy: 0.7531
Epoch 6/10
919/919 [==============================] - 6s 7ms/step - loss: 0.5372 -
accuracy: 0.7768
Epoch 7/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4902 -
accuracy: 0.7949
Epoch 8/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4615 -
accuracy: 0.8104
Epoch 9/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4282 -
accuracy: 0.8217
Epoch 10/10
919/919 [==============================] - 6s 7ms/step - loss: 0.4030 -
accuracy: 0.8322
460/460 [==============================] - 2s 4ms/step - loss: 0.6389 -
accuracy: 0.7710
Epoch 1/10
1379/1379 [==============================] - 10s 7ms/step - loss: 0.8287 -
accuracy: 0.6567
Epoch 2/10
1379/1379 [==============================] - 10s 7ms/step - loss: 0.6491 -
accuracy: 0.7359
Epoch 3/10
1379/1379 [==============================] - 9s 6ms/step - loss: 0.5523 -
accuracy: 0.7795
Epoch 4/10
1379/1379 [==============================] - 9s 7ms/step - loss: 0.4809 -
accuracy: 0.8075
Epoch 5/10
1379/1379 [==============================] - 9s 7ms/step - loss: 0.4229 -
accuracy: 0.8306
Epoch 6/10
1379/1379 [==============================] - 9s 7ms/step - loss: 0.3817 -
accuracy: 0.8477
Epoch 7/10
1379/1379 [==============================] - 9s 6ms/step - loss: 0.3379 -
accuracy: 0.8670
Epoch 8/10
1379/1379 [==============================] - 9s 7ms/step - loss: 0.3130 -
accuracy: 0.8769
Epoch 9/10
1379/1379 [==============================] - 9s 7ms/step - loss: 0.2829 -
accuracy: 0.8886
Epoch 10/10
1379/1379 [==============================] - 9s 7ms/step - loss: 0.2595 -
accuracy: 0.8999
```

```
Best: 0.770222 using {'optimizer': 'adam', 'init': 'glorot_uniform', 'epochs':
10, 'dropout_rate': 0.4, 'batch_size': 30}
```

Best Model Parameters Following are the specific value of the best model:

Parameter	Value
Optimizer	adam
Initialization	glorot_uniform
Epochs	10
Dropout Rate	0.4
Batch Size	30

```
[ ]: # Saving the best model
best_model = random_search_result.best_estimator_
best_model.model.save('best_trained_model.h5')
print("Model saved to best_trained_model.h5")
```

```
Model saved to best_trained_model.h5
```

Best Model Evaluation

```
[ ]: # Evaluate on the validation set
val_loss, val_accuracy = best_model.model.evaluate(X_val, y_val)
print(f'Validation accuracy: {val_accuracy:.2f}, Validation loss: {val_loss:.
 ↪2f}')
```

```
259/259 [==============================] - 1s 4ms/step - loss: 0.0836 -
accuracy: 0.9797
Validation accuracy: 0.98, Validation loss: 0.08
```

The best cross-validation accuracy achieved during optimization was 0.7702.

In terms of the Validation and Training, here is the comparison:

Metric	Before Optimization	After Optimization	Improvement Amount
Validation Accuracy	0.80	0.98	+0.18
Validation Loss	0.73	0.08	-0.65
Training Accuracy	0.8021	0.9797	+0.1776
Training Loss	0.7263	0.0836	-0.6427

Overall a huge increase in both accuracies and decrease in losses.

0.6 All Artists Inclusive Analysis

Rather than just ending the project with small part of the data, we decided to take the best CNN model and test the full data set against it.

0.6.1 Loading Dataset

Loading all the artists.

[ ]:
```
all_artists = get_all_artists(raw_data_extracted)

paths_artist_length_data_all = get_midi_lengths_for_artists(
    raw_data_extracted, all_artists, graph=False, debug=False
)
paths_artist_length_data_all.to_pickle("paths_artist_length_data_all.pkl")
```

Not a lot of files within some of artists, but since we have chunks, we should have more than one data points of each artist.

0.6.2 Preparing Data

[ ]:
```
processed_chunk_all_df = process_all_files(
    paths_artist_length_data_all, raw_data_extracted, fs=8, chunk_size=150
)
processed_chunk_all_df.to_pickle("processed_chunk_all_artist.pkl")
```

[ ]:
```
processed_chunk_all_df = pd.read_pickle('processed_chunk_all_artist.pkl')
```

[ ]:
```
X_all = preprocess_chunks(processed_chunk_all_df)
y_all, label_encoder_all = encode_labels(processed_chunk_all_df['Artist'])
```

[ ]:
```
X_all, y_all = shuffle(X_all, y_all, random_state=42)
```

[ ]:
```
X_train_all, X_val_all, y_train_all, y_val_all = train_test_split(
    X_all, y_all, test_size=0.2, random_state=42
)
```

[ ]:
```
X_all.shape
```

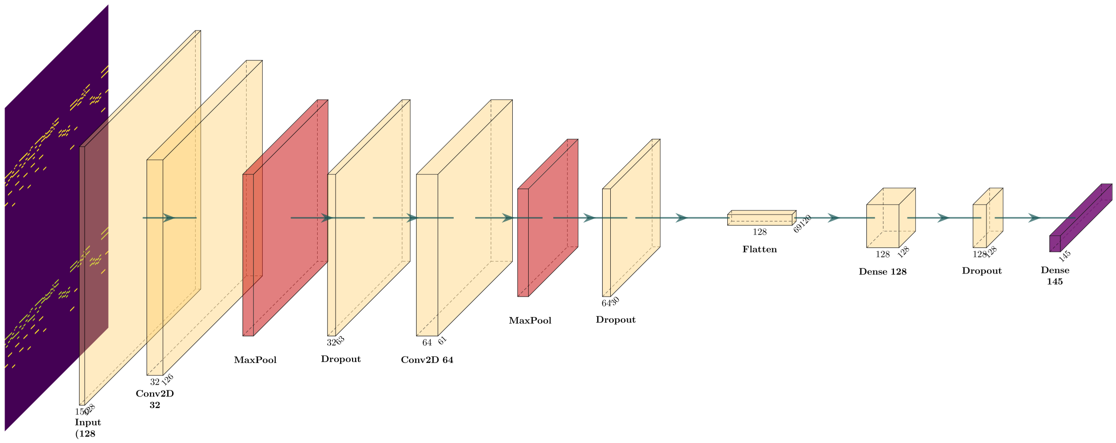
[ ]:
```
(164247, 128, 150, 1)
```

[ ]:
```
y_all.shape
```

[ ]:
```
(164247, 145)
```

0.6.3 Defining Model

Here is what the architecture looks like:




```
[ ]: def create_best_model(input_shape, output_shape, optimizer="adam",␣
     ↪init="glorot_uniform", dropout_rate=0.4):
         model = Sequential(
             [
                 Conv2D(
                     32,
                     (3, 3),
                     activation="relu",
                     kernel_initializer=init,
                     input_shape=input_shape,
                 ),
                 MaxPooling2D((2, 2)),
                 Dropout(dropout_rate),
                 Conv2D(64, (3, 3), activation="relu", kernel_initializer=init),
                 MaxPooling2D((2, 2)),
                 Dropout(dropout_rate),
                 Flatten(),
                 Dense(128, activation="relu", kernel_initializer=init),
                 Dropout(dropout_rate),
                 Dense(
                     output_shape,
                     activation="softmax",
                     kernel_initializer=init,
                 ),
             ]
         )

         model.compile(
             optimizer=optimizer, loss="categorical_crossentropy",␣
     ↪metrics=["accuracy"]
         )
         return model
```

[ ]:
```
import warnings
warnings.filterwarnings('ignore')
```

[ ]:
```
model = create_best_model(X_all.shape[1:], len(processed_chunk_all_df['Artist'].
    ↪unique()))
```

[ ]:
```
model.summary()
```

```
Model: "sequential_1"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 126, 148, 32)      320

 max_pooling2d_2 (MaxPooling  (None, 63, 74, 32)       0
 2D)

 dropout_3 (Dropout)         (None, 63, 74, 32)        0

 conv2d_3 (Conv2D)           (None, 61, 72, 64)        18496

 max_pooling2d_3 (MaxPooling  (None, 30, 36, 64)       0
 2D)

 dropout_4 (Dropout)         (None, 30, 36, 64)        0

 flatten (Flatten)           (None, 69120)             0

 dense_2 (Dense)             (None, 128)               8847488

 dropout_5 (Dropout)         (None, 128)               0

 dense_3 (Dense)             (None, 145)               18705

=================================================================
Total params: 8,885,009
Trainable params: 8,885,009
Non-trainable params: 0
_________________________________________________________________
```

0.6.4 Training Model

[ ]:
```
history = model.fit(
    X_train_all,
    y_train_all,
    validation_data=(X_val_all, y_val_all),
    epochs=10,
    batch_size=30,
```

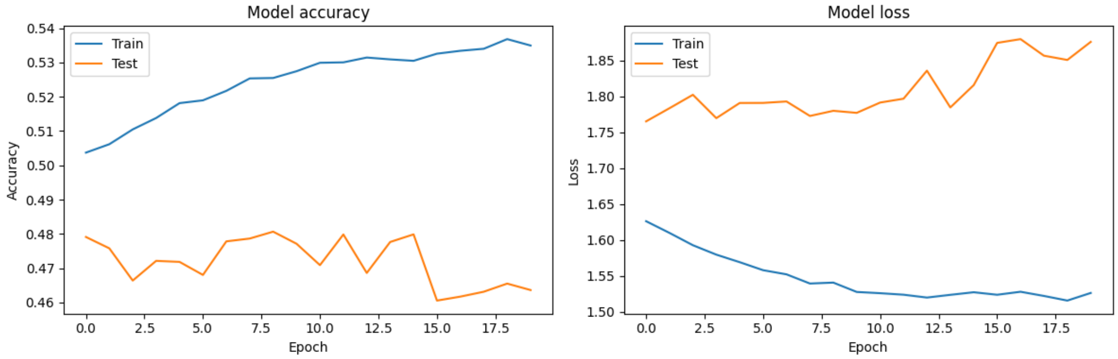
```
    verbose=1
)
```

```
Epoch 1/10
4380/4380 [==============================] - 34s 8ms/step - loss: 2.3138 -
accuracy: 0.4042 - val_loss: 2.0607 - val_accuracy: 0.4152
Epoch 2/10
4380/4380 [==============================] - 28s 6ms/step - loss: 2.0278 -
accuracy: 0.4342 - val_loss: 1.9014 - val_accuracy: 0.4452
Epoch 3/10
4380/4380 [==============================] - 28s 6ms/step - loss: 1.9087 -
accuracy: 0.4493 - val_loss: 1.8607 - val_accuracy: 0.4484
Epoch 4/10
4380/4380 [==============================] - 27s 6ms/step - loss: 1.8198 -
accuracy: 0.4642 - val_loss: 1.8181 - val_accuracy: 0.4628
Epoch 5/10
4380/4380 [==============================] - 28s 6ms/step - loss: 1.7569 -
accuracy: 0.4752 - val_loss: 1.7613 - val_accuracy: 0.4710
Epoch 6/10
4380/4380 [==============================] - 28s 6ms/step - loss: 1.7002 -
accuracy: 0.4861 - val_loss: 1.7718 - val_accuracy: 0.4753
Epoch 7/10
4380/4380 [==============================] - 28s 6ms/step - loss: 1.6513 -
accuracy: 0.4959 - val_loss: 1.7395 - val_accuracy: 0.4825
Epoch 8/10
4380/4380 [==============================] - 29s 7ms/step - loss: 1.6141 -
accuracy: 0.5033 - val_loss: 1.7529 - val_accuracy: 0.4795
Epoch 9/10
4380/4380 [==============================] - 29s 7ms/step - loss: 1.5797 -
accuracy: 0.5121 - val_loss: 1.7257 - val_accuracy: 0.4859
Epoch 10/10
4380/4380 [==============================] - 29s 7ms/step - loss: 1.5569 -
accuracy: 0.5169 - val_loss: 1.7492 - val_accuracy: 0.4859
```

[ ]:
```
model.save("best_model_all_artists_test.h5")
```

0.6.5 Evaluating the Model

[ ]:
```
plot_training_history(history)
```



[ ]:
```
print(f"Training Loss: {history.history['loss'][-1]}")
print(f"Training Accuracy: {history.history['accuracy'][-1]}")
print(f"Validation Loss: {history.history['val_loss'][-1]}")
print(f"Validation Accuracy: {history.history['val_accuracy'][-1]}")
```

```
Training Loss: 2.0304393768310547
Training Accuracy: 0.42764294147491455
Validation Loss: 1.9776721000671387
Validation Accuracy: 0.4288584589958191
```

These are quite lower than the first four artists, especially with the best-case model.

Metric	Four Artists	All Artists (Threshold 0)	Difference (All - Four)
Training Loss	0.0836	2.0304	+1.9468
Training Accuracy	97.97%	42.76%	-55.21%
Validation Loss	0.08	1.9777	+1.8977
Validation Accuracy	98%	42.89%	-55.11%

This is a huge change, there might be an issue with presence of low chunks quantity.

Checking Chunk Issue There might be an issue with artists with less chunks causing a huge increase

[ ]:
```
def filter_using_threshold(processed_chunk_all_df, threshold):
    # Group by 'Artist' and count the number of rows
    artist_counts = (
        processed_chunk_all_df.groupby("Artist").size().
↪reset_index(name="Count")
    )
    total_rows = len(processed_chunk_all_df)
    artist_counts["Percentage"] = (artist_counts["Count"] / total_rows) * 100
    print("Artist Counts and Percentages:")
    print(artist_counts)
```

```python
    filtered_artists = artist_counts[artist_counts["Percentage"] > threshold]

    # Print counts of artists below and above the threshold
    below_threshold_count = len(artist_counts[artist_counts["Percentage"] <=␣
↪threshold])
    above_threshold_count = len(filtered_artists)
    print(
        f"\nNumber of artists below the {threshold}% threshold:␣
↪{below_threshold_count}"
    )
    print(
        f"Number of artists above the {threshold}% threshold:␣
↪{above_threshold_count}"
    )

    # Filter the original DataFrame
    filtered_df = processed_chunk_all_df[
        processed_chunk_all_df["Artist"].isin(filtered_artists["Artist"])
    ]

    # Print size of the filtered DataFrame
    print(f"\nFiltered DataFrame size: {len(filtered_df)} rows")

    # Save the filtered DataFrame as a pickle file
    filtered_df.to_pickle("filtered_artists_df.pkl")

    return filtered_df


filtered_df = filter_using_threshold(processed_chunk_all_df, 0.1)
```

```
Artist Counts and Percentages:
             Artist  Count  Percentage
0           Albeniz    660    0.401834
1             Alkan    710    0.432276
2          Ambroise     86    0.052360
3           Arensky    152    0.092544
4             Arndt     28    0.017047
..              …      …          …
140        Vivaldi   2790    1.698661
141         Wagner    119    0.072452
142           Wolf     22    0.013394
143  augmented_pitch  59638   36.309948
144      meditation     28    0.017047

[145 rows x 3 columns]
```

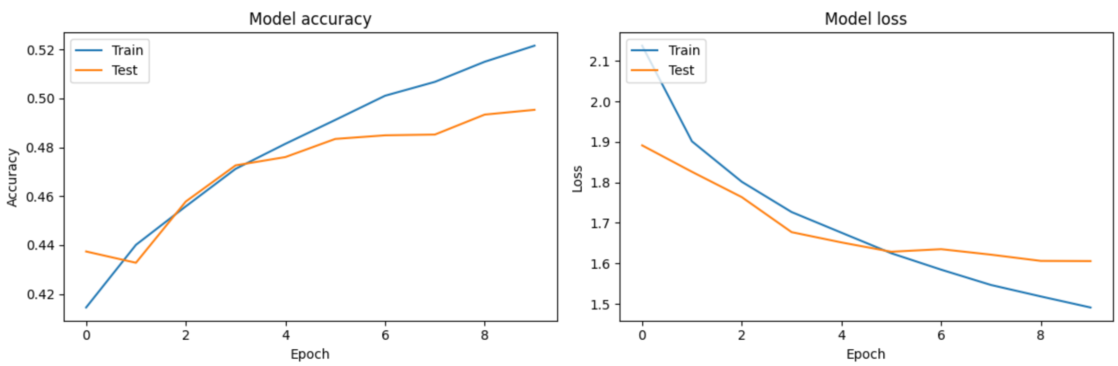
```
Number of artists below the 0.1% threshold: 96
Number of artists above the 0.1% threshold: 49

Filtered DataFrame size: 159257 rows
```

[ ]:
```
def build_and_analyse_model(filtered_df, info = ""):

    X_all = preprocess_chunks(filtered_df)
    y_all, label_encoder_all = encode_labels(filtered_df['Artist'])
    X_all, y_all = shuffle(X_all, y_all, random_state=42)
    X_train_all, X_val_all, y_train_all, y_val_all = train_test_split(
        X_all, y_all, test_size=0.2, random_state=42
    )
    model = create_best_model(X_all.shape[1:], len(filtered_df['Artist'].
  ↪unique()))
    model.summary()
    history = model.fit(
        X_train_all,
        y_train_all,
        validation_data=(X_val_all, y_val_all),
        epochs=10,
        batch_size=30,
        verbose=1
    )
    model.save(f"best_model_all_artists_{info}.h5")
    plot_training_history(history)
    print(f"Training Loss: {history.history['loss'][-1]}")
    print(f"Training Accuracy: {history.history['accuracy'][-1]}")
    print(f"Validation Loss: {history.history['val_loss'][-1]}")
    print(f"Validation Accuracy: {history.history['val_accuracy'][-1]}")

    return model

model_01 = build_and_analyse_model(filtered_df)
```

```
2024-08-10 06:59:47.813583: I tensorflow/core/platform/cpu_feature_guard.cc:151]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2024-08-10 06:59:48.978788: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1525] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 38416 MB memory:  -> device:
0, name: NVIDIA A100-SXM4-40GB, pci bus id: 0000:10:1c.0, compute capability:
8.0
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 126, 148, 32)      320

 max_pooling2d (MaxPooling2D  (None, 63, 74, 32)       0
 )

 dropout (Dropout)           (None, 63, 74, 32)        0

 conv2d_1 (Conv2D)           (None, 61, 72, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 30, 36, 64)       0
 2D)

 dropout_1 (Dropout)         (None, 30, 36, 64)        0

 flatten (Flatten)           (None, 69120)             0

 dense (Dense)               (None, 128)               8847488

 dropout_2 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 49)                6321

=================================================================
Total params: 8,872,625
Trainable params: 8,872,625
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10

2024-08-10 07:00:11.831576: I tensorflow/stream_executor/cuda/cuda_dnn.cc:366]
Loaded cuDNN version 8903
2024-08-10 07:00:13.061915: I tensorflow/stream_executor/cuda/cuda_blas.cc:1774]
TensorFloat-32 will be used for the matrix multiplication. This will only be
logged once.

4247/4247 [==============================] - 39s 8ms/step - loss: 2.1376 -
accuracy: 0.4145 - val_loss: 1.8914 - val_accuracy: 0.4374
Epoch 2/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.9014 -
accuracy: 0.4401 - val_loss: 1.8262 - val_accuracy: 0.4328
Epoch 3/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.8015 -
accuracy: 0.4558 - val_loss: 1.7636 - val_accuracy: 0.4578
Epoch 4/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.7268 -
```

```
accuracy: 0.4712 - val_loss: 1.6772 - val_accuracy: 0.4726
Epoch 5/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.6758 -
accuracy: 0.4814 - val_loss: 1.6518 - val_accuracy: 0.4760
Epoch 6/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.6251 -
accuracy: 0.4912 - val_loss: 1.6288 - val_accuracy: 0.4834
Epoch 7/10
4247/4247 [==============================] - 28s 7ms/step - loss: 1.5846 -
accuracy: 0.5010 - val_loss: 1.6351 - val_accuracy: 0.4849
Epoch 8/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.5468 -
accuracy: 0.5067 - val_loss: 1.6216 - val_accuracy: 0.4852
Epoch 9/10
4247/4247 [==============================] - 28s 7ms/step - loss: 1.5184 -
accuracy: 0.5149 - val_loss: 1.6062 - val_accuracy: 0.4933
Epoch 10/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.4911 -
accuracy: 0.5215 - val_loss: 1.6057 - val_accuracy: 0.4953
```



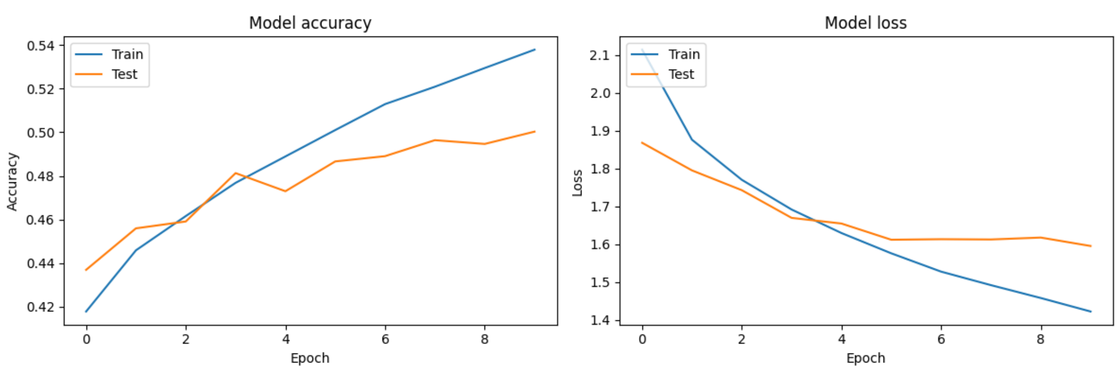

```
Training Loss: 1.491101622581482
Training Accuracy: 0.5215023159980774
Validation Loss: 1.6057184934616089
Validation Accuracy: 0.49529072642326355
```

[ ]:
```
filtered_df_1 = filter_using_threshold(processed_chunk_all_df, 1)
model_1 = build_and_analyse_model(filtered_df)
```

```
Artist Counts and Percentages:
           Artist  Count  Percentage
0         Albeniz    660    0.401834
1           Alkan    710    0.432276
2        Ambroise     86    0.052360
3         Arensky    152    0.092544
4           Arndt     28    0.017047
```

```
..                …     …          …
140          Vivaldi   2790    1.698661
141           Wagner    119    0.072452
142             Wolf     22    0.013394
143  augmented_pitch  59638   36.309948
144       meditation     28    0.017047

[145 rows x 3 columns]

Number of artists below the 1% threshold: 131
Number of artists above the 1% threshold: 14

Filtered DataFrame size: 144721 rows
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 126, 148, 32)      320

 max_pooling2d_2 (MaxPooling  (None, 63, 74, 32)       0
 2D)

 dropout_3 (Dropout)         (None, 63, 74, 32)        0

 conv2d_3 (Conv2D)           (None, 61, 72, 64)        18496

 max_pooling2d_3 (MaxPooling  (None, 30, 36, 64)       0
 2D)

 dropout_4 (Dropout)         (None, 30, 36, 64)        0

 flatten_1 (Flatten)         (None, 69120)             0

 dense_2 (Dense)             (None, 128)               8847488

 dropout_5 (Dropout)         (None, 128)               0

 dense_3 (Dense)             (None, 49)                6321

=================================================================
Total params: 8,872,625
Trainable params: 8,872,625
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
4247/4247 [==============================] - 34s 8ms/step - loss: 2.1142 -
accuracy: 0.4177 - val_loss: 1.8682 - val_accuracy: 0.4368
Epoch 2/10
```

```
4247/4247 [==============================] - 30s 7ms/step - loss: 1.8764 -
accuracy: 0.4458 - val_loss: 1.7953 - val_accuracy: 0.4559
Epoch 3/10
4247/4247 [==============================] - 28s 7ms/step - loss: 1.7705 -
accuracy: 0.4615 - val_loss: 1.7431 - val_accuracy: 0.4590
Epoch 4/10
4247/4247 [==============================] - 28s 7ms/step - loss: 1.6920 -
accuracy: 0.4768 - val_loss: 1.6699 - val_accuracy: 0.4813
Epoch 5/10
4247/4247 [==============================] - 28s 7ms/step - loss: 1.6299 -
accuracy: 0.4889 - val_loss: 1.6550 - val_accuracy: 0.4729
Epoch 6/10
4247/4247 [==============================] - 28s 7ms/step - loss: 1.5763 -
accuracy: 0.5010 - val_loss: 1.6122 - val_accuracy: 0.4866
Epoch 7/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.5280 -
accuracy: 0.5129 - val_loss: 1.6136 - val_accuracy: 0.4890
Epoch 8/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.4923 -
accuracy: 0.5209 - val_loss: 1.6128 - val_accuracy: 0.4964
Epoch 9/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.4583 -
accuracy: 0.5295 - val_loss: 1.6179 - val_accuracy: 0.4947
Epoch 10/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.4226 -
accuracy: 0.5380 - val_loss: 1.5957 - val_accuracy: 0.5003
```

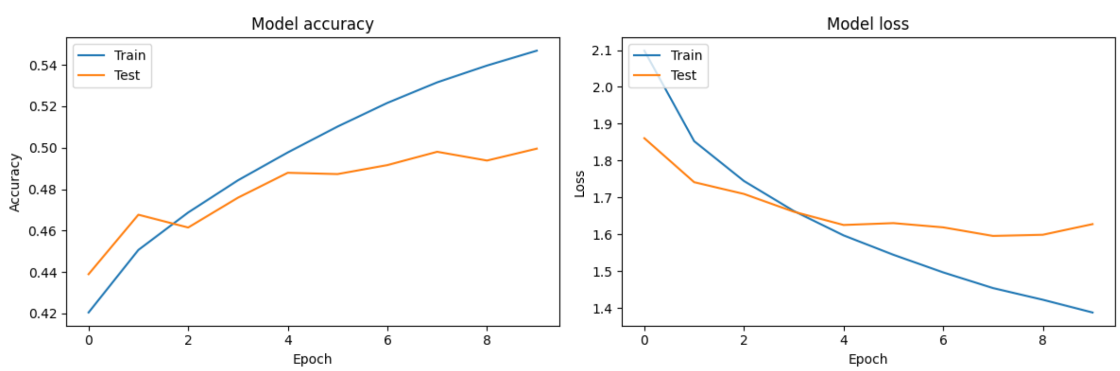



```
Training Loss: 1.4226192235946655
Training Accuracy: 0.537953794002533
Validation Loss: 1.5956977605819702
Validation Accuracy: 0.500313937664032
```

[ ]:
```
filtered_df_10 = filter_using_threshold(processed_chunk_all_df, 10)
model_10 = build_and_analyse_model(filtered_df)
```

```
Artist Counts and Percentages:
             Artist  Count  Percentage
0           Albeniz    660    0.401834
1             Alkan    710    0.432276
2          Ambroise     86    0.052360
3           Arensky    152    0.092544
4             Arndt     28    0.017047
..              …     …          …
140         Vivaldi   2790    1.698661
141          Wagner    119    0.072452
142            Wolf     22    0.013394
143  augmented_pitch  59638   36.309948
144      meditation     28    0.017047

[145 rows x 3 columns]

Number of artists below the 10% threshold: 143
Number of artists above the 10% threshold: 2

Filtered DataFrame size: 76328 rows
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 126, 148, 32)      320

 max_pooling2d_4 (MaxPooling  (None, 63, 74, 32)       0
 2D)

 dropout_6 (Dropout)         (None, 63, 74, 32)        0

 conv2d_5 (Conv2D)           (None, 61, 72, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 30, 36, 64)       0
 2D)

 dropout_7 (Dropout)         (None, 30, 36, 64)        0

 flatten_2 (Flatten)         (None, 69120)             0

 dense_4 (Dense)             (None, 128)               8847488

 dropout_8 (Dropout)         (None, 128)               0

 dense_5 (Dense)             (None, 49)                6321

=================================================================
Total params: 8,872,625
```

```
Trainable params: 8,872,625
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
4247/4247 [==============================] - 35s 8ms/step - loss: 2.0983 -
accuracy: 0.4204 - val_loss: 1.8607 - val_accuracy: 0.4389
Epoch 2/10
4247/4247 [==============================] - 28s 7ms/step - loss: 1.8527 -
accuracy: 0.4506 - val_loss: 1.7413 - val_accuracy: 0.4677
Epoch 3/10
4247/4247 [==============================] - 28s 7ms/step - loss: 1.7446 -
accuracy: 0.4687 - val_loss: 1.7094 - val_accuracy: 0.4615
Epoch 4/10
4247/4247 [==============================] - 28s 7ms/step - loss: 1.6633 -
accuracy: 0.4843 - val_loss: 1.6612 - val_accuracy: 0.4759
Epoch 5/10
4247/4247 [==============================] - 27s 6ms/step - loss: 1.5970 -
accuracy: 0.4977 - val_loss: 1.6252 - val_accuracy: 0.4879
Epoch 6/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.5444 -
accuracy: 0.5102 - val_loss: 1.6303 - val_accuracy: 0.4873
Epoch 7/10
4247/4247 [==============================] - 27s 6ms/step - loss: 1.4965 -
accuracy: 0.5217 - val_loss: 1.6188 - val_accuracy: 0.4916
Epoch 8/10
4247/4247 [==============================] - 27s 6ms/step - loss: 1.4540 -
accuracy: 0.5316 - val_loss: 1.5955 - val_accuracy: 0.4981
Epoch 9/10
4247/4247 [==============================] - 29s 7ms/step - loss: 1.4223 -
accuracy: 0.5397 - val_loss: 1.5987 - val_accuracy: 0.4938
Epoch 10/10
4247/4247 [==============================] - 30s 7ms/step - loss: 1.3877 -
accuracy: 0.5469 - val_loss: 1.6276 - val_accuracy: 0.4996
```



```
Training Loss: 1.387671947479248
Training Accuracy: 0.546870231628418
Validation Loss: 1.6275787353515625
Validation Accuracy: 0.49956047534942627
```

Here is the table summarizing of accuracy at different thresholds:

Threshold	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Artists Below Threshold	Artists Above Threshold
0	2.0304	0.4276	1.9777	0.4289	0	145
0.1	1.4911	0.5215	1.6057	0.4953	96	49
1	1.4226	0.5380	1.5957	0.5003	131	14
10	1.3877	0.5469	1.6276	0.4996	143	2

Two major thing to note:

- Both training and validation accuracy increase with higher thresholds, while training loss decreases. Validation loss, however, shows mixed results.
- The number of artists below the threshold increases as the threshold rises, while those above it decrease sharply.

It would be great to explore this further.

Overall, this project was quite fun for all of us. Not only did we learn quite a lot, but we also achieved great accuracy and optimization. We also got to try on full data, which was initially the main wish, as our data preparation was designed to include all the MIDI files and structure all the files quite nicely.

0.7 Future Plan

If we had more GPU power, it would be great to optimization the best model for all the artists. Just the small optimization of CNN took us several hours of training within our machines, and even on NVIDIA A100 (40 GB), it took quite a while to get everything running and optimized. We had to pull some parameters out due to minimum resources.

Additionally, it would be great to create a demo where we can give it a random MIDI chunk and get a prediction, similar to Shazam. We had written some code for this, but nothing was complete for an MVP. It would be great to go back and get the MVP done.