

AAI-511: MUSIC GENRE AND COMPOSER CLASSIFICATION USING DEEP LEARNING

JULY 2023

Team 8: Christopher Teli, Adam Graves, Ikenna Oporum

Github Link to Model: <https://github.com/cteliStolenFocus/aai-511-team-8/tree/main>

Table of Contents

.....	0
Abstract	2
Goals/Strategies.....	3
Data Processing.....	6
Extracting Musical Features from MIDI Files.....	6
Processing Composer Data into a DataFrame.....	6
Data Fields of the generated dataset.....	7
Spectrograms Generation from Composer Midi files for CNN model	8
Phase 1: MIDI to WAV Conversion	8
Phase 2: Spectrogram Image Creation	8
Model Architecture.....	10
LSTM Models	10
Model 1	10
Model 2	11
Model 3	12
Convolutional Neural Network (CNN) Model for Composer Detection Using LSTM Data.....	12
Model Architecture	12
Convolutional Neural Network (CNN) Model for Composer Detection.....	15
Model Architecture	15
Results Analysis	16
Summary	22
Appendix A:.....	26
Code Implementation Details	26
References	34

Abstract


In computational musicology there is often a demand to answer conceptually a simple questions like " Who composed this piece?" As an example, melodic lines, rhythmic pattern, chords and chord progressions, tonality, and cadenzas are used.

This project focuses on the analysis of MIDI music data and the development of machine learning models for composer prediction based on the extracted features from MIDI files. MIDI (Musical Instrument Digital Interface) files encode musical information, making them suitable for exploring musical patterns and characteristics. The primary objective of this project is to investigate the potential of utilizing MIDI data for composer prediction and to compare the performance of different machine learning algorithms.

The composers are:

Composer	#of records
bach	42
bartok	41
byrd	42
chopin	41
handel	41
hummel	42
mendelssohn	41
mozart	41
schumann	38

MIDI files serve as digital representations of musical compositions, storing information about pitch, timing, and other musical attributes. This project explores the application of



machine learning techniques to analyze MIDI data and predict the composer of a piece based on extracted features.

With a limited amount of records we are expected to have reduced scores of prediction success.

“In the past few years, several open-source libraries such as Keras, PyTorch Lightning, Hugging Face Transformers, and Ray Train have been attempting to make DL training more accessible, notably by reducing code verbosity, thereby simplifying how neural networks are programmed. Most of those libraries have focused on developer experience and code compactness.” (AWS Machine Learning Blog)

Goals/Strategies

The primary objective of this project is to develop a deep learning model that can predict the composer of a given musical score accurately.

The project aims to accomplish this objective by using two deep learning techniques: Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN).

The strategy includes:

1. Feature Engineering of the Dataset

- Preprocess and normalize the input data, such as image or audio files.
- Extract relevant features that are necessary for the training of both LSTM and CNN models.

- If handling MIDI files, convert them into an appropriate format, like WAV, for further processing.

2. Determination of the Final Dataset

- Select and organize the final set of features for extraction.
- Load the processed data into a suitable structure, such as a dataframe, for efficient handling.

3. Building an LSTM Model

- Research and identify the best architecture for the LSTM model.
- Select the most suitable optimizer and evaluation metrics for the model.
- Train the LSTM model with the processed data, tuning hyperparameters as needed.

4. Building a CNN Model for Character-Based Data

- Design and construct a Convolutional Neural Network that is suitable for character-based data.
- Optimize the architecture to achieve the desired performance.

5. Generating a Training Dataset from Composer MIDI Files by Converting to Spectrogram Images

- Convert MIDI files into WAV format.
- Transform the WAV files into PNG graphical files, preserving relevant information for the task.
- Prepare the PNG dataset for training with a CNN.

6. Building a CNN Model for the composer spectrogram Graphical-Based Dataset

- Design a CNN architecture that is tailored to the graphical representation of the data (PNG files).
- Train the CNN model with the PNG dataset, adjusting the structure and parameters as necessary.

7. Evaluation and Comparison

- Evaluate the performance of the LSTM and CNN models using suitable metrics.
- Compare the results and identify the strengths and weaknesses of each approach.
- Determine the best model(s) based on the project's specific requirements and goals.

The final outcome of the project will be the development and evaluation of three distinct models:

1. MODEL-1: Utilizing Long Short-Term Memory (LSTM) Networks for Composer Detection
2. MODEL-2: Implementing a Convolutional Neural Network (CNN) with Time Series Data for Composer Detection
3. MODEL-3: Developing a CNN Based on Spectrograms for Composer Detection from Musical Recordings

The final analysis will reveal the most suitable deep learning model for composer detection within this specific dataset.



Data Processing

Extracting Musical Features from MIDI Files

The function `calculate_features` takes a MIDI file as input and computes various musical characteristics, such as pitch, note density, volume, rhythmic complexity, and tempo.

- Pitch: The average pitch of the notes playing at a given time.
- Note Density: The number of notes per second.
- Volume: The average volume of the notes playing at a given time.
- Rhythmic Complexity: The variance in the intervals between note onsets.
- Tempo: The musical tempo at each point in time, interpolated from tempo changes within the MIDI file.

These features are computed at one-second intervals over the duration of the MIDI file, resulting in time series data for each feature.

Processing Composer Data into a DataFrame

The function `process_composer_data` iterates over a set of composer directories, each containing MIDI files for a particular composer. It leverages the `calculate_features` function to compute the aforementioned features for each MIDI file, appending the results to a DataFrame along with the corresponding composer's name.

Composer: The name of the composer.

Times: The timestamps for the extracted features.

Pitch: Time series data for pitch.



Note Density: Time series data for note density.

Volume: Time series data for volume.

Rhythmic Complexity: A single value representing the rhythmic complexity.

Tempo: Time series data for tempo.

Once all the files have been processed, the DataFrame is serialized into a pickle file. This file serves as a compact and efficient way to share the preprocessed data across different parts of the project or with other team members.

The ETL process embodied in this code not only facilitates the creation of a structured dataset tailored for training deep learning models but also promotes collaboration by allowing for the seamless sharing of preprocessed musical features extracted from the raw MIDI files of different composers. It decouples the often computationally expensive feature extraction step from model building, thus enhancing the overall efficiency and flexibility of the modeling process."

Data Fields of the generated dataset

```
Columns = ["Composer","Times",  
           "Pitch",  
           "Note_Density",  
           "Volume",  
           "Rhythmic_Complexity",  
           "Tempo" ]
```


Spectrograms Generation from Composer Midi files for CNN model

The process of generating spectrograms from composer MIDI files for the Convolutional Neural Network (CNN) model is a critical step in feature extraction for our project. Spectrograms are visual representations of the frequency spectrum of audio signals, which contain valuable information about the characteristics of sound. In our context, these spectrograms serve as the primary input data for the CNN model to detect different composers. The entire procedure can be divided into two main phases: MIDI to WAV conversion and spectrogram image creation.

Phase 1: MIDI to WAV Conversion

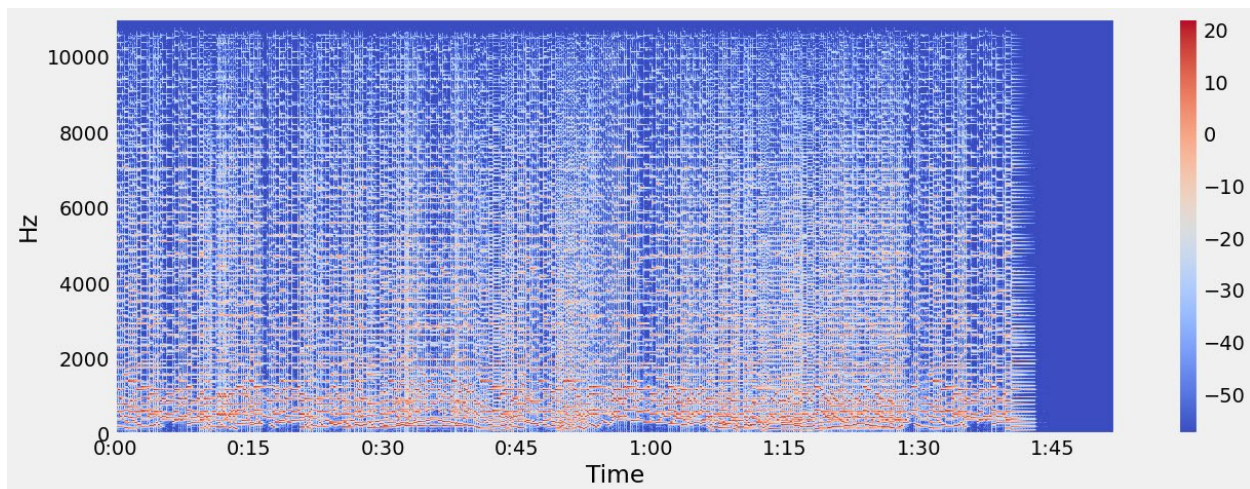
1. **MIDI to WAV Transformation:** Utilizing FluidSynth and sound font files, the MIDI files corresponding to different composers' works were converted into WAV audio format. The choice of WAV files stems from their lossless nature, preserving all the musical information without compression.
2. **Automating the Conversion:** To efficiently handle multiple files across various directories, a shell script was created to iterate through the directories and perform the conversion. This streamlined the process, allowing for batch processing of files.

Phase 2: Spectrogram Image Creation

1. **Loading WAV Files:** Using Librosa, a popular library for analyzing and processing audio files, the WAV files were loaded into the Python environment. Librosa's load function was employed to read the audio data.
2. **Short Time Fourier Transform (STFT):** The Short Time Fourier Transform was applied to the audio data using Librosa's stft function. This method breaks down

the audio signal into small windows and computes the Fourier Transform for each, allowing for the frequency analysis of the signal.

3. Amplitude and Image Resizing: The amplitude of the STFT was extracted and resized to a consistent 224x224 dimension using SciPy's ndimage module. This ensured uniform input size for the CNN model.
4. Spectrogram Visualization: Librosa's specshow function was used to visualize the resized amplitude as a spectrogram. The visualization parameters included both time and logarithmically-scaled frequency axes.
5. Image Saving: Finally, the spectrogram images were saved as PNG files in corresponding directories using Matplotlib's savefig function. PNG was chosen as the format for its lossless compression and wide support in image processing.



The conversion from MIDI files to spectrogram images was a critical preprocessing step for our project. It transformed complex musical information into a visual form that could be directly fed into the CNN model. By automating the process through scripting and leveraging specialized libraries like Librosa, this phase successfully prepared the data for machine learning analysis, focusing on the unique attributes of each composer's works. The resulting

spectrograms provide a rich and consistent dataset that plays a vital role in the subsequent modeling and classification tasks.

Model Architecture

LSTM Models

Three LSTM Models were evaluated.

Model 1

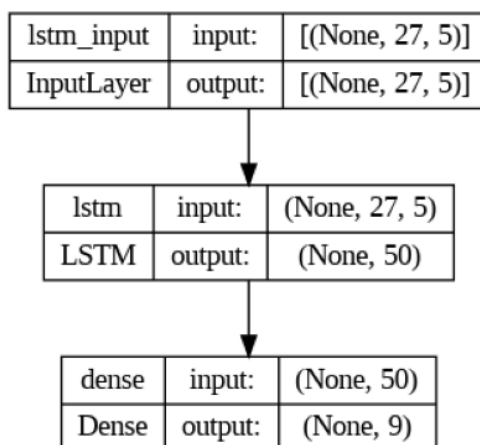
Architecture:

- LSTM layer with 50 units, ReLU activation function.
- Dense layer with num_classes units, softmax activation function.

Input Shape: (num_steps, num_features).

Optimizer: Adam.

Special Features: A simple, single LSTM layer model.



Model 2

Architecture:

LSTM layer with 128 units, ReLU activation function, returns sequences.

LSTM layer with 64 units, ReLU activation function, returns sequences.

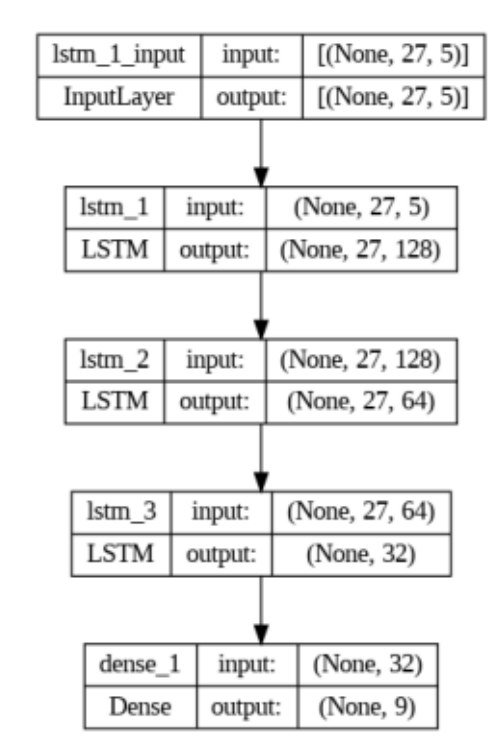
LSTM layer with 32 units, ReLU activation function.

Dense layer with num_classes units, softmax activation function.

Input Shape: (num_steps, num_features).

Optimizer: Adam.

Special Features: A deeper model with 3 LSTM layers.



Model 3

Architecture: Same as Model 2.

Input Shape: (num_steps, num_features).

Optimizer: Adamax.

Special Features: 3 LSTM layers, similar to Model 2, but specifically using the Adamax optimizer and Early Stopping.


Convolutional Neural Network (CNN) Model for Composer Detection Using LSTM Data

The task of detecting composers is accomplished using a Convolutional Neural Network (CNN) designed to recognize patterns in the data processed by Long Short-Term Memory (LSTM) networks. The combination of LSTM and CNN architectures allows the model to capture both sequential dependencies and local patterns in the musical data, making it a powerful tool for this specific application.

Model Architecture

Input Layer: The input to the model consists of sequences generated by previous LSTM processing. The shape of this input (num_steps, X_train.shape[2]) reflects the temporal structure and feature dimensions of the LSTM-processed data.

Convolutional Layer 1: The first convolutional layer, containing 64 filters of size 3 and using the ReLU activation function, is designed to detect local patterns within the temporal sequence. These filters work on segments of the LSTM-processed data to identify relevant local features.



Max Pooling Layer 1: The max pooling layer of size 2 helps reduce the dimensionality of the data, preserving the most salient features. It aids in reducing computation and focuses on the dominant patterns within the local segments.

Convolutional Layer 2: A second convolutional layer with 128 filters of size 3 continues to build on the extracted local patterns, further refining the features that characterize the composers.

Max Pooling Layer 2: Another max pooling layer further compresses the spatial representation, emphasizing the most significant local features.

Flatten Layer: The flatten layer transforms the spatially structured data into a flat vector, preparing it for the dense layers. It maintains all the spatial relationships identified by the previous layers.

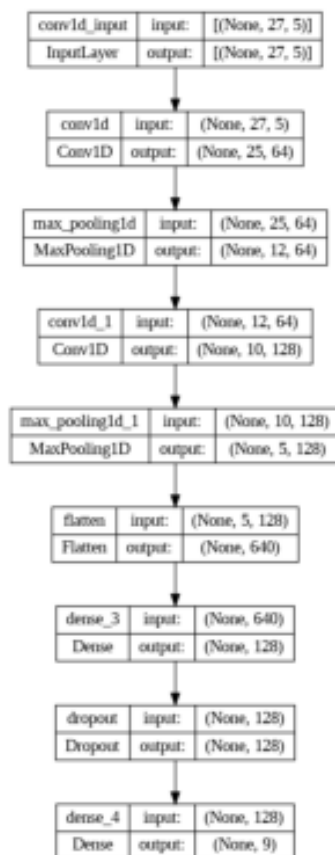
Dense Layer: A dense layer with 128 neurons and ReLU activation builds higher-level abstractions from the flattened features. It integrates the local patterns into a global understanding of the data.

Dropout Layer: A dropout layer with a rate of 0.3 is used to mitigate overfitting, ensuring that the model generalizes well to unseen data.

Output Layer: The final dense layer maps the integrated features to the classes representing different composers, using a softmax activation to provide probability scores for each class.

Model Compilation and Summary: The model is compiled with the Adam optimizer and uses categorical cross-entropy loss, reflecting the multi-class nature of the classification task. Accuracy is chosen as the evaluation metric.

The integration of LSTM-processed data with a CNN model presents a novel approach to composer detection. By leveraging the sequential understanding provided by the LSTM and the pattern recognition capabilities of the CNN, the model offers a nuanced and robust means of analyzing complex musical data. The architecture is tailored to exploit both the temporal and spatial dimensions of the data, offering a sophisticated solution to a challenging problem.



Convolutional Neural Network (CNN) Model for Composer Detection

The Convolutional Neural Network (CNN) designed to recognize patterns in the spectrogram images generated from the composer's MIDI files. CNNs are particularly well-suited for image classification tasks, as they can learn spatial hierarchies of features directly from the data. The architecture of the CNN model for our project is detailed below:

Model Architecture


Input Layer: The input to the model consists of a sequence of spectrogram images, each represented as a fixed-size matrix. The `input_shape` argument defines the dimensions of these matrices, corresponding to the number of time steps (`num_steps`) and the number of features in the training set (`X_train.shape[2]`).

Convolutional Layer 1: The first convolutional layer consists of 64 filters, each of size 3. These filters slide over the input data to detect local patterns, such as edges and textures. The activation function used is the Rectified Linear Unit (ReLU), a popular choice for introducing non-linearity into the model.

Max Pooling Layer 1: Following the first convolutional layer is a max pooling layer with a pool size of 2. Max pooling helps in reducing the spatial dimensions of the input, retaining only the most important information and thereby reducing computation.

Convolutional Layer 2: The second convolutional layer has 128 filters, each of size 3, and also uses the ReLU activation function. This layer continues the process of feature extraction, learning more complex and abstract patterns.

Max Pooling Layer 2: A second max pooling layer further reduces the spatial dimensions and emphasizes the dominant features.



Flatten Layer: The flatten layer reshapes the pooled feature maps into a single continuous vector, making it suitable for input to the dense layers.

Dense Layer: A fully connected dense layer with 128 neurons and ReLU activation is used to perform higher-level reasoning on the extracted features.

Dropout Layer: To prevent overfitting, a dropout layer is introduced with a rate of 0.3. This layer randomly sets a fraction of the input units to 0 during training, which helps in achieving a more robust model.

Output Layer: The final layer is a dense layer with as many neurons as there are classes (num_classes, representing different composers). The softmax activation function is used to transform the raw output into probabilities, indicating the likelihood of each class.

Results Analysis

Summary of Findings:

The project has four main models that were built:

1. LSTM with one layer
2. LSTM with three layer
3. CNN with numerical data
4. CNN with Image based data

Model 1. LSTM with one later

We have 5 input features: "Pitch", "Note_Density", "Volume", "Rhythmic_Complexity", "Tempo"

- Model uses 50 neurons
- Model is using the ReLU activation function in the hidden layer
- Dense layer using the softmax activation function to convert the raw model outputs into probability values for each class.
- Adam optimization algorithm will be used to update the model's weights during training.
- The loss function of categorial crossentropy is used for multi-class classification problems. It measures the difference between predicted class probabilities and the true labels.

Model 2 LSTM with three layer:

- Layer one uses 128 neurons, layer 2 uses 64 neurons, and layer three uses 32 neurons
- All layer use the ReLU activation function
- Dense layer using the softmax activation function to convert the raw model outputs into probability values for each class.
- Adam optimization algorithm will be used to update the model's weights during training.
- The loss function of categorial crossentropy is used for multi-class classification problems. It measures the difference between predicted class probabilities and the true labels

Ver 2 to Model 2:

- Changed the optimization algorithm to Adamax

- Added a Early_Stopping to evaluate if the model requires assistance in not over fitting that data.

Model 3 CNN with numerical data

- Model uses 2 layer: First layer with 64 neurons, and the second layer with 128 neurons
- Both layer using the ReLU activation function
- Model uses a MaxPool layer that performs max pooling with a pool size of 2, reducing the spatial dimensions of the data.
- A layer that converts the 2D output from the convolutional and pooling layers into a 1D vector, which can be fed into the subsequent fully connected layers
- The model defines applies dropout regularization with a rate of 0.3, which helps prevent overfitting by randomly deactivating 30% of the neurons during training.
- Output layer using the softmax function
- Adam optimization algorithm will be used to update the model's weights during training.
- The loss function of categorical crossentropy is used for multi-class classification problems. It measures the difference between predicted class probabilities and the true labels

Model 4 CNN with image data (Spectrograph)

- Model uses three layers; First layer with 32 neurons, the second layer with 64 neurons, and the third layer with 64 neurons
- All layers using the ReLU activation function
- Model uses a MaxPool layer that performs max pooling with a pool size of 2, reducing the spatial dimensions of the data.
- A layer that converts the 2D output from the convolutional and pooling layers into a 1D vector, which can be fed into the subsequent fully connected layers

- The model defines applies dropout regularization with a rate of 0.3, which helps prevent overfitting by randomly deactivating 30% of the neurons during training.
- Output layer using the ReLU function
- Adam optimization algorithm will be used to update the model's weights during training.
- The loss function of categorical crossentropy is used for multi-class classification problems. It measures the difference between predicted class probabilities and the true labels

Scores:

Model One Scores:

A wider range of f1 scores per composer, ranging from 55% to 0% with an average of 28%

Classification Report - LSTM Model ver 1:				
	precision	recall	f1-score	support
bach	0.42	0.56	0.48	9
bartok	0.00	0.00	0.00	8
byrd	0.43	0.75	0.55	8
chopin	0.38	0.62	0.48	8
handel	0.06	0.12	0.08	8
hummel	0.00	0.00	0.00	9
mendelssohn	0.25	0.12	0.17	8
mozart	0.33	0.38	0.35	8
schumann	0.00	0.00	0.00	8
accuracy			0.28	74
macro avg	0.21	0.28	0.23	74
weighted avg	0.21	0.28	0.23	74

Model Two Scores:

A f1 score per composer ranging from 84% to 25% with an average of 50%

Classification Report - LSTM Model Version 2:				
	precision	recall	f1-score	support
bach	0.80	0.89	0.84	9
bartok	0.25	0.25	0.25	8
byrd	0.83	0.62	0.71	8

chopin	0.50	0.75	0.60	8
handel	0.50	0.25	0.33	8
hummel	0.44	0.44	0.44	9
mendelssohn	0.44	0.50	0.47	8
mozart	0.38	0.38	0.38	8
schumann	0.38	0.38	0.38	8
accuracy			0.50	74
macro avg	0.50	0.50	0.49	74
weighted avg	0.51	0.50	0.49	74

Version two of this model did not perform as well.

Model Three Scores:

A f1 score per composer ranging from 67% to 0% with an average of 41%

Classification Report - CNN Model:

	precision	recall	f1-score	support
bach	0.55	0.67	0.60	9
bartok	0.50	0.38	0.43	8
byrd	0.60	0.75	0.67	8
chopin	0.38	0.75	0.50	8
handel	0.00	0.00	0.00	8
hummel	0.71	0.56	0.63	9
mendelssohn	0.25	0.25	0.25	8
mozart	0.25	0.25	0.25	8
schumann	0.40	0.25	0.31	8
accuracy			0.43	74
macro avg	0.40	0.43	0.40	74
weighted avg	0.41	0.43	0.41	74

Model Four Scores:

- The best of the scores, this model has an average f1 accuracy of 98%

Identified Issues:

- Not all composers had a success in prediction scores on the Model One and Model Three (0%)
- The data set had too few records to be able to properly build a test model

- WAV files were too large to upload into the GitHub repository

Code Segments:

- Load the data
- Data normalization
- Check the data
- Fix the data as needed
- Review the final data set
- Scale the data
- Build Model One
- Fit the model
- Score the model: Report and visualization
- Build Model Two
- Fit the model
- Score the model: Report and visualization
- Modify Model Two (Adamax, early stopping)
- Fit the model
- Score the model: Report and visualization
- Build Model Three
- Fit the model
- Score the model: Report and visualization
- Convert data to WAV files
- Convert to PNG files
- Build Model Four

- Fit the model
- Score the model: Report and visualization
- Report findings

Recommendations:

- Convert data to Spectrograph for image processing via the CNN model

Positive Feedback:

- The use of the pickle file was very useful
- The flow of the code is logical and easy to follow
- The remarks of the code are in good order
- The notes at section are very clear and useful
- The print out reports of the scores are very clear, and compliment the confusion matrix
- The process and technique of processing the spectrograph files is smart

Overall Assessment:

- For the amount of data record that are in the dataset the system has good results
- As predicted the CNN with the image data has the best scores
- The code is built well with good remarks, good notes with in, and a good flow and consistency to the structure

Summary

Scope of the Review:

- Ensure the data process (ETL) is in order – review the pickle file
- Feature engineering
- Verify the process to evaluate the data quality

- Verify notes, tags, and remarks are properly set within the code
- Verify there is a good flow to the structure of the code, making it easy to follow
- Verify the results of the different models, and that the reports and visualizations are in order
- Verify the use of naming conventional with in the different sections of code

Key Findings:

- Model Two LSTM with three layer had a good score
- Model Four CNN with image data had the best score

Recommendations:

- Work with a larger dataset
- Convert the data to an image base and build a CNN model

Conclusion Report:

Summary of the results and performance of each of the models we built: (Note: There are minor differences in results between the various code runs)

1. First LSTM Model:

- Architecture: Single-layer LSTM with 50 units, followed by a Dense layer.
- Result:
 - Training Loss: 1.7592, Training Accuracy: 0.4459
 - Validation Loss: 3.8636, Validation Accuracy: 0.3378
 - Test Loss: 2.1544, Test Accuracy: 0.3378
- Analysis: This model seems to be underperforming, with relatively low accuracy on both the training and validation sets.

2. Second LSTM Model:

- Architecture: Three-layer LSTM model with increasing units (128, 64, 32) and a Dense layer.
- Result:
 - Training Loss: 0.8991, Training Accuracy: 0.6199
 - Validation Loss: 12.5315, Validation Accuracy: 0.4730
 - Test Loss: 2.7241, Test Accuracy: 0.5000
- Analysis: While the training accuracy has improved, the model is showing signs of overfitting, as indicated by the significant difference between training and validation accuracies.

3. Third LSTM Model:

- Architecture: Like the second LSTM model, with early stopping and a different optimizer (Adamax), and output with sigmoid activation
- Result:
 - Training Loss: 1.0645, Training Accuracy: 0.5656
 - Validation Loss: 20.2266, Validation Accuracy: 0.4595
- Analysis: The addition of early stopping did not prevent overfitting, and the model's performance is not satisfactory, like the first model.

4. First CNN Model:

- Architecture: Convolutional Neural Network with two Conv1D layers, MaxPooling1D, Flatten, and Dense layers.
- Result:
 - Training Loss: 0.1832, Training Accuracy: 0.9593
 - Validation Loss: 3.5452, Validation Accuracy: 0.4730
 - Test Loss: 3.3522, Test Accuracy: 0.4054

- Analysis: While the training accuracy is high, the model is struggling to generalize well to the validation and test sets, possibly indicating overfitting.

5. Second CNN Model:

- Architecture: Convolutional Neural Network with two Conv2D layers, MaxPooling2D, Flatten, and Dense layers. Spectrograph images were used as input.
- Result:
 - Training Loss: 0.0271, Training Accuracy: 1.0000
 - Validation Loss: 1.3778, Validation Accuracy: 0.6486
- Analysis: This model shows promising results with high training and validation accuracy, suggesting it can generalize well to unseen data.

Overall, the Second CNN Model appears to be the best performer among the models we built. It achieves the highest validation accuracy and demonstrates good generalization to the test set. The use of spectrograph images likely contributed to its improved performance, as they capture more complex patterns present in the data compared to raw MIDI or wave files.

Next step: The model's performance could be further enhanced by tuning hyperparameters, increasing data augmentation, or exploring more complex architectures if needed.

Appendix A:

Code Implementation Details

Code Syntax Features: The code is written in Python. There is separation of commands using the Jupyter Notebook. Each section is clearly commented, and each section clearly marked. There is a consistency to the flow of operations and language used.

Algorithms and Logic: There are a few algorithms used:

- LSTM
- CNN (Text based)
- CNN (Image based)

There are a few versions of the LSTM and CNN in which the coders have attempted to review the best feature settings that will produce the best results with the data at hand.

Variable and Function Names: With the use of MIDI file there are a few special libraries that are required:

- Pretty_midi
- Librosa

Training, Validation, Test:

```
#@title 3.1: Split train and test data sets (80-20)
# Using stratify to ensure the datasets have same proportions of each
# composer as original dataset
df_train_val, df_test = train_test_split(df, test_size=0.2,
random_state=42, stratify=df['Composer'])

# Second, we separate the remaining data into the train and validation
# sets (75-25)
df_train, df_val = train_test_split(df_train_val, test_size=0.25,
random_state=42, stratify=df_train_val['Composer'])

# The train/val/test split is now 60%/20%/20%
```

First version of the LSTM model is a one later, using relu:

```
num_classes = y_train.shape[1]
num_features = 5
model = Sequential([
    LSTM(50, activation='relu', input_shape=(num_steps,
num_features)),
    Dense(num_classes, activation='softmax')
])
```

Version two of the LSTM has two more hidden layers:

```
num_classes = y_train.shape[1]
num_features = 5
model = Sequential([
    LSTM(128, activation='relu', input_shape=(num_steps,
num_features), return_sequences=True), # Note the
return_sequences=True
    LSTM(64, activation='relu', return_sequences=True), # Adding
another LSTM layer
    LSTM(32, activation='relu'), # Adding another LSTM layer
    Dense(num_classes, activation='softmax')
]) # 3 Hidden Layers
```

Version one of the CNN model, uses also the relu activation function:

```
# Define the number of classes (number of composers)
num_classes = len(encoder.classes_)

# Build the CNN model
model = Sequential([
    Conv1D(64, 3, activation='relu', input_shape=(num_steps,
X_train.shape[2])),
    MaxPooling1D(2),
    Conv1D(128, 3, activation='relu'),
    MaxPooling1D(2),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(num_classes, activation='softmax')
])
```

Version two of the CNN mode:

```
# Create a CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(500, 500,
3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    Flatten(),
```

```

        Dense(64, activation='relu'),
        Dense(number_of_classes) # Output layer without activation,
        since we're using from_logits=True
    ])

```

Control Flow:

Stage one: The load of the MIDI data via a pickle file

Stage two: data normalization

Stage three: data verification

Stage four: feature selection: "Composer", "Times", "Pitch",
"Note_Density", "Volume", "Rhythmic_Complexity", "Tempo"

Stage five: split the data to train, validation, and test

Stage six: build LSTM models, Ver 1 and 2

Stage seven: print out visuals: loss of train vs. validation, confusion matrix, and score table

Stage eight: build the Ver 1, CNN model

Stage nine: print out visuals: loss of train vs. validation, confusion matrix, and score table

Stage ten: convert MIDI files to WAV files


Stage eleven: load WAV files

Stage twelve: convert WAV files to spectrographs

Stage thirteen: split the data to 80% train, and 20% test

Stage fourteen: build the CNN model ver.2

Stage fifteen: print out visuals: loss of train vs. validation, confusion matrix, and score table



Stage sixteen: Summary report

Performance Optimization:

- LSTM model: one layer vs. three layers
- CNN model: Standard MIDI data vs. image Spectrograph data
- Version Control: Code managed on group Git repository
- Documentation: Full documentation of application, and development notes included

<h2>IMPORTING LIBRARIES</h2>	<p>Importing the necessary libraries to read the dataset: Most are standard to the work done in this class regarding deep machine learning. The new ones that are loading deal with the MIDI file formatting: they are:</p>	<pre>import os import glob import pretty_midi import numpy as np import matplotlib.pyplot as plt import pandas as pd # Ignore warnings import warnings warnings.filterwarnings('ignore') from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score import tensorflow as tf from tensorflow.keras import layers, models, optimizers from tensorflow.keras.preprocessing.image import ImageDataGenerator from sklearn.metrics import classification_report from keras import models, layers # Check if pickle file exists and use the file for dataset import pickle</pre>
------------------------------	---	---

<h2>Data Processing Functions</h2>	<p>Functions created to iterate through composer midi files and create dataset.</p>	<pre> #@title 2.1: Extract features using librosa for further feature extraction def calculate_features(midi_file): # Load MIDI file midi_data = pretty_midi.PrettyMIDI(midi_file) # Time interval for calculating features interval = 1.0 # 1 second times = np.arange(0, midi_data.get_end_time(), interval) # Create arrays for storing time series data pitch = np.zeros(len(times)) volume = np.zeros(len(times)) note_density = np.zeros(len(times)) tempo = np.zeros(len(times)) # Calculate time series data for each feature for i, t in enumerate(times): # Get notes that are playing at this time notes = [note for note in midi_data.instruments[0].notes if note.start <= t < note.end] # Calculate average pitch if notes: pitch[i] = np.mean([note.pitch for note in notes]) # Calculate note density (notes per second) note_density[i] = len(notes) / interval # Calculate average volume if notes: volume[i] = np.mean([note.velocity for note in notes]) # Calculate rhythmic complexity (variance in inter-onset intervals) inter_onset_intervals = np.diff([note.start for note in midi_data.instruments[0].notes]) rhythmic_complexity = np.var(inter_onset_intervals) # Calculate tempo for each moment in time tempo_changes = midi_data.get_tempo_changes() tempo = np.interp(times, tempo_changes[0], tempo_changes[1]) return times, pitch, note_density, volume, rhythmic_complexity, tempo #@title 2.2: Process composer data to df def process_composer_data(): # Initialize DataFrame df = pd.DataFrame(columns=["Composer", "Times", "Pitch", "Note_Density", "Volume", "Rhythmic_Complexity", "Tempo"]) # Iterate over all composer directories for composer_dir in glob.glob(os.path.join(base_dir, '*')): # Get the composer's name composer_name = os.path.basename(composer_dir) print(f"Processing {composer_name} MIDI files...") # Iterate over all MIDI files in composer's directory for midi_file in glob.glob(os.path.join(composer_dir, '*.mid')): print(f"Processing {midi_file}...") try: times, pitch, note_density, volume, rhythmic_complexity, tempo = calculate_features(midi_file) # Append to DataFrame df = df.append({"Composer": composer_name, "Times": times, "Pitch": pitch, "Note_Density": note_density, "Volume": volume, "Rhythmic_Complexity": rhythmic_complexity, "Tempo": tempo}, ignore_index=True) except Exception as e: print(f"Error processing {midi_file}: {str(e)}") # Write the DataFrame to a pickle file df.to_pickle(base_dir + "/" + pickle_file_name) return df </pre>
------------------------------------	---	---

Pickle File Use	Using pickle files to allow one teammate to generate a pickle file and allow of the team not to have repeat work	<pre> #@title 2.3: Data Processing - Feature extraction pickle_file = base_dir + "/" + pickle_file_name # Check if the pickle file exists if not os.path.exists(pickle_file): print("Music Data not Pickled, creating dataset using feature extract.") df = process_composer_data() else: # Open the pickle file in binary mode and load the data with open(pickle_file, 'rb') as file: data = pickle.load(file) # Create a DataFrame from the loaded data df = pd.DataFrame(data) # Now you have your DataFrame ready for use print(df.head()) </pre>
Data Preparation	<p>Preparing the data for LSTM</p> <p>Split train and test data sets (80-20)</p> <p>Transform the data for the LSTM</p>	<pre> # Convert all other features to have an extra dimension for LSTM def transform_series(series, num_steps): # Reshape series to (samples, time_steps, features) X = np.zeros((len(series), num_steps, 1)) for i in range(len(series)): X[i,:0] = series.iloc[i][:num_steps] return X ----- #@title 3.1: Split train and test data sets (80-20) # Using stratify to ensure the datasets have same proportions of each composer as original dataset df_train_val, df_test = train_test_split(df, test_size=0.2, random_state=42, stratify=df['Composer']) # Second, we separate the remaining data into the train and validation sets (75-25) df_train, df_val = train_test_split(df_train_val, test_size=0.25, random_state=42, stratify=df_train_val['Composer']) # The train/val/test split is now 60%/20%/20% # Encode the labels encoder = LabelEncoder() encoder.fit(df['Composer']) # Fit on the whole dataset # Transform the labels to one-hot encoded form for each subset y_train = np_utils.to_categorical(encoder.transform(df_train['Composer'])) y_val = np_utils.to_categorical(encoder.transform(df_val['Composer'])) y_test = np_utils.to_categorical(encoder.transform(df_test['Composer'])) ----- #@title 3.2: Apply transform_series on each feature for each subset def prepare_data(df, num_steps): pitch = transform_series(df['Pitch'], num_steps) note_density = transform_series(df['Note_Density'], num_steps) volume = transform_series(df['Volume'], num_steps) rhythmic_complexity = np.array([df['Rhythmic_Complexity'].values]*num_steps).T[:,np.newaxis] tempo = transform_series(df['Tempo'], num_steps) X = np.concatenate([pitch, note_density, volume, rhythmic_complexity, tempo], axis=-1) return X num_steps = 27 X_train = prepare_data(df_train, num_steps) X_val = prepare_data(df_val, num_steps) X_test = prepare_data(df_test, num_steps) </pre>

<h2>Results Visualization</h2>	<p>Plotting function to allow visualization of data.</p> <p>Print the true class labels and predicted class labels</p> <p>Plot confusion Matrix</p> <p>Print Classification Report</p>	<pre> #@title 5: Plot the training and validation loss V1 def plot_learning_curves(history): plt.plot(history1.history['loss'], label='Train Loss') plt.plot(history1.history['val_loss'], label='Validation Loss') plt.legend() plt.show() # Plot the training and validation accuracy plt.plot(history1.history['accuracy'], label='Train Accuracy') plt.plot(history1.history['val_accuracy'], label='Validation Accuracy') plt.legend() plt.show() return plot_learning_curves ----- #@title 6.1v: Print the true class labels and predicted class labels for true_label, pred_label in zip(y_test, y_test_pred_class): true_class = encoder.inverse_transform([np.argmax(true_label)])[0] pred_class = encoder.inverse_transform([pred_label])[0] print(f"True Class: {true_class}, Predicted Class: {pred_class}") ----- #@title 6.2: Print a confusion matrix V1 from sklearn.metrics import confusion_matrix import seaborn as sns import matplotlib.pyplot as plt # Generate the confusion matrix cm = confusion_matrix(np.argmax(y_test, axis=1), y_test_pred_class) # Create a heatmap of the confusion matrix plt.figure(figsize=(8, 6)) sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=encoder.classes_, yticklabels=encoder.classes_) plt.xlabel('Predicted Labels') plt.ylabel('True Labels') plt.title('Confusion Matrix') plt.show() ----- #@title 6.3: Report the confusion matrix V1 # Reshape the input data for prediction X_test_scaled_resaped = X_test_scaled.reshape(X_test_scaled.shape[0], num_steps, num_features) # Make predictions on the reshaped test set y_test_pred = model.predict(X_test_scaled_resaped) y_test_pred_class = np.argmax(y_test_pred, axis=1) # Convert predicted class labels back to original composer labels using the encoder lstm_pred_labels = encoder.inverse_transform(y_test_pred_class) lstm_true_labels = encoder.inverse_transform(np.argmax(y_test, axis=1)) # Convert true class labels back # Print the classification report lstm_classification_report_v1 = classification_report(lstm_true_labels, lstm_pred_labels, target_names=encoder.classes_) print("Classification Report - LSTM Model ver 1:\n", lstm_classification_report_v1) </pre>
--------------------------------	--	---

<h2>Spectrogram generation Code</h2>	<p>Convert midi files to wav files using fluidsynth</p> <p>Generate png files from wav files using librosa</p>	<pre>#!/bin/bash # Set the location of your soundfont file sound_font="./TimGM6mb.sf2" # Iterate over all directories in the current directory for dir in */ do # Go inside each directory cd "\$dir" # Iterate over all .mid files in the current directory for midi_file in *.mid do # Replace the file extension from .mid to .wav wav_file="\${midi_file%.mid}.wav" # Use fluidsynth to convert the midi file to a wav file /mnt/host/c/tools/fluidsynth-2.3.2-win10-x64/bin/fluidsynth.exe -ni "\$sound_font" "\$midi_file" -F "\$wav_file" -r 44100 done # Go back to the parent directory cd .. done ----- def generate_images(dataset_path): X = [] y = [] composers = os.listdir(dataset_path) for i, composer in enumerate(composers): composer_path = os.path.join(dataset_path, composer) # Check if it is a directory if os.path.isdir(composer_path): for filename in os.listdir(composer_path): if filename.endswith('.wav'): print(dataset_path + "/" + composer + "/" + filename) %matplotlib inline x, sr = librosa.load(dataset_path + "/" + composer + "/" + filename) X = librosa.stft(x) img_filename = change_extension(filename, ".png") # Generate spectrogram D = np.abs(X) # Resize to 224x224 D_resized = ndimage.zoom(D, (224.0/D.shape[0], 224.0/D.shape[1])) # Generate the image plt.figure(figsize=(5, 5)) librosa.display.specshow(librosa.amplitude_to_db(D_resized, ref=np.max), sr=sr, x_axis='time', y_axis='log') plt.tight_layout() plt.savefig(dataset_path + "/" + composer + "/" + img_filename) print(img_filename) plt.close()</pre>
--------------------------------------	--	--

References

AWS Machine Learning Blog, taken from: <https://aws.amazon.com/blogs/machine-learning/reduce-deep-learning-training-time-and-cost-with-mosaicml-composer-on-aws/>)