

AAI-511: PREDICTING MUSIC COMPOSERS, WITH MIDI FILES DATASET

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

| | |
|---|----------|
| Front Page..... | 0 |
| Abstract | 2 |
| Goals/Strategize..... | 3 |
| Design and methodology | 4 |
| 1. Data importing and processing (ETL)..... | 4 |
| 1.1. PICKLE FILE | 4 |
| 1.2. IMPORTING LIBRARIES | 5 |
| 1.3. LOADING THE DATASET | 6 |
| 1.3.1 The process:..... | 6 |
| 1.4. INITIAL DATA VERIFICATION AND NORMALIZATION | 7 |
| 1.5. OPTIMIZING THE DATASET | 8 |
| 1.6. DATA CLEANING..... | 8 |
| 2. BUILDING THE LSTM MODEL (Long Short-Term Memory) | 8 |
| 2.1. LSTM Model Version 2 | 11 |
| 2.2. LSTM Model Version 3 | 12 |
| 3. CNN MODEL (Convolutional Neural Network)..... | 13 |

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

The Keras libraries. (TensorFlow) is the chosen library to use in this code.

Goals/Strategize

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:

- Feature engineering of the dataset
- In turn this will determine the final dataset for extraction and loading into the dataframe
- Building a suitable model
- With some research and development, we will identify the best structure for the LSTM model, the best Optimizer, and how best to evaluate
- Build a CNN model based on the char based data

- Build an alternative dataset, based off WAV files converted from the MIDI files. From this convert to a PNG graphical file, and then create a CNN based on the dataset of PNG file (graphical based)

Final result will be three different models:

1. MODEL-1 LSTM
2. MODEL-2 CNN with Time series data
3. MODEL-3 Using Image analysis

Final results will indicate the best deep learning model suited for such a dataset.

Design and methodology

1. Data importing and processing (ETL)

The process of the Extract, Transform, Load (ETL) Step one is to create a pickle file that will be used to load into the dataframe. This approach effectively distributes the workload of model building and feature extraction. In case the model performs poorly with the current data, a team member can execute feature extraction, generate a dataframe, and save it as a pickle file. This binary file can then be shared with the rest of the team, allowing them to utilize the preprocessed data without duplicating efforts.

1.1. PICKLE FILE

A few steps in creating the pickle data file for import of the MIDI data.

Feature Extraction: First, the need to extract the relevant features from each MIDI file. The pretty_midi library to parse the MIDI files is used, and extract information like times, pitch, note density, volume, etc. A function to perform this feature extraction for each MIDI file. The import pickle library is needed to import the data from the pickle file.

There are 9 composers, each with 40-42 files.

The process continues to the data preprocessing: After extracting the features, the code requires to organize them into a structured format, such as a pandas DataFrame (df), where each row represents a sequence (e.g., a song) and each column represents a feature.

Final step is to Pickle the Data: Once the data is in a suitable format, the next step is to pickle the file using the pickle library in Python. Pickling allows the coder to serialize the data and save it to a file. Later, the pickle file is loaded to access the preprocessed data, which is faster than parsing the MIDI files and extracting features every time one needs to work with the data.


Fields of the dataset:

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

1.2. IMPORTING LIBRARIES

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:

```
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
```

1.3. LOADING THE DATASET

Load the pickle file:

- 'team8_composer_dataset.pkl'

1.3.1 The process:

A few steps in creating the pickle data file for import of the MIDI data.

Feature Extraction: First, extract the relevant features from each MIDI file. Using the `pretty_midi` library to parse the MIDI files and extract information like times, pitch, note density, volume, etc. Create a function to perform this feature extraction for each MIDI file.


There are 9 composers, each with 40-42 files.

Next, continue to the data preprocessing: After extracting the features, the data is organized into a structured format, such as a pandas DataFrame (df), where each row represents a sequence (e.g., a song) and each column represents a feature.

Final step is to Pickle the Data: Once the data is in a suitable format, one can pickle it using the pickle library in Python. Pickling allows you to serialize the data and save it to a file. Later, next load the pickle file to access the preprocessed data, which is faster than parsing the MIDI files and extracting features every time one needs to work with the data.

Data features: (8)

1. Composer,
2. Times,
3. Pitch,
4. Note,

- 
5. Density,
 6. Volume,
 7. Rhythmic_Complexit,
 8. Tempo

1.4. INITIAL DATA VERIFICATION AND NORMALIZATION

Review data field attributes and data types, ensure proper import and verify the field types, check the dataframe df as in 2.4

Code sample: 2.4

```
##title 2.4: Data Pre-processing - verifications
print(df.head())
print(df.info())
```

Verify the list of Composers and record count as in 2.6

Code sample 2.6:

```
##title 2.6: Check composer names and Count the occurrences of each 'quality' value
composer_counts = df['Composer'].value_counts()

# Sort the count by 'quality' values
sorted_composer_counts = composer_counts.sort_index()

# Print the count for each 'quality' value
print(sorted_composer_counts)
```

Confirming the results:

| | |
|-------------|----|
| bach | 42 |
| bartok | 41 |
| byrd | 42 |
| chopin | 41 |
| handel | 41 |
| hummel | 42 |
| mendelssohn | 41 |
| mozart | 41 |
| schumann | 38 |

1.5. OPTIMIZING THE DATASET

Normalization of the data by removing unnecessary columns and replacing missing values.

1.6. DATA CLEANING

Data cleaning and normalization by removing unnecessary columns and replacing missing values. Check for NaN values in the data fields, or other values that can be an obstacle in the building of a DL model.

Summary:

- The code defines a function called “calculate_features(midi_file)” that loads a MIDI file, extracts various musical features like pitch, note density, volume, rhythmic complexity, and tempo. The extracted features are returned as numpy arrays.
- The code defines another function called “process_composer_data()” that iterates over directories containing MIDI files for different composers. It uses the “calculate_features()” function to extract features from each MIDI file and appends the data into a pandas DataFrame (df)
- The DataFrame (df) is then saved to a pickle file for future use. If the pickle file already exists, the code loads the data from the pickle file instead of reprocessing the MIDI files.
- The code checks if the pickle file ('team8_composer_dataset.pkl') exists before processing the data. If the pickle file exists, it loads the data from the file, and if not, it calls the process_composer_data() function to create the dataset.

2. BUILDING THE LSTM MODEL (Long Short-Term Memory)

The project will evaluate a few versions of the DL models. This will determine the best tuning of the model to fit this project.

Starting with version 1:

Load of the related libraries, which are:

- ✓ `from tensorflow.keras.models import Sequential`
- ✓ `from tensorflow.keras.layers import LSTM, Dropout, Dense`

The code we are providing is for creating a sequential neural network model using Keras, a high-level deep learning library.

Definig the LSTM as:

`LSTM(50, activation='relu', #utilizing the ReLU activation function`

`input_shape=(num_steps, num_features))`: This line adds an LSTM (Long Short-Term Memory) layer to the model. LSTM is a type of recurrent neural network (RNN) that is particularly well-suited for sequence data. Here, 50 is the number of LSTM units or cells in the layer, `activation='relu'` specifies the activation function as Rectified Linear Unit (ReLU), and `input_shape=(num_steps, num_features)` defines the shape of the input data. `num_steps` represents the number of time steps in the sequence, and `num_features` represents the number of features at each time step.

`Dense(num_classes, activation='softmax')`: This line adds a dense (fully connected) layer to the model. The Dense layer is typically used for the final layer of a classification model. `num_classes` represents the number of classes in your classification problem, and `activation='softmax'` applies the softmax activation function to the output of this layer, which converts the output values into probabilities for each class.

Using optimizer = Adam

Item 7: LSTM Model

```
# Build the LSTM model
model = Sequential([
    LSTM(128, input_shape=(num_steps, X_train.shape[2]),
        return_sequences=True),
```

```
Dropout(0.2),
LSTM(64),
Dropout(0.2),
Dense(num_classes, activation='softmax')
```

The results:

Model: "sequential"

| Layer (type) | Output Shape | Param # |
|---------------|--------------|---------|
| ===== | | |
| lstm (LSTM) | (None, 50) | 11200 |
| dense (Dense) | (None, 9) | 459 |

Total params: 11,659

Trainable params: 11,659

Non-trainable params: 0


In the data there are under 11,700 parameters which is not a large dataset.

Model evaluation:

Breakdown Score Results for the Version 1 of the LSTM Model:

Classification Report - LSTM Model:

| | precision | recall | f1-score | support |
|--------|-----------|--------|----------|---------|
| bach | 0.62 | 0.89 | 0.73 | 9 |
| bartok | 0.20 | 0.12 | 0.15 | 8 |
| byrd | 0.71 | 0.62 | 0.67 | 8 |
| chopin | 0.47 | 0.88 | 0.61 | 8 |
| handel | 0.43 | 0.38 | 0.40 | 8 |
| hummel | 0.67 | 0.44 | 0.53 | 9 |



| | | | | |
|--------------|------|------|------|----|
| mendelssohn | 0.00 | 0.00 | 0.00 | 8 |
| mozart | 0.43 | 0.38 | 0.40 | 8 |
| schumann | 0.38 | 0.38 | 0.38 | 8 |
| accuracy | | | 0.46 | 74 |
| macro avg | 0.43 | 0.45 | 0.43 | 74 |
| weighted avg | 0.44 | 0.46 | 0.43 | 74 |

Top performer is the composer Bach with a f1 score of: 0.73, and a poor performance of composer Mendelssohn with a f1 score of: 0.

Overall f1 score of: 0.46 being at 46%.

2.1. LSTM Model Version 2

Due to an average level scoring of version 1, the next version is about introducing a few feature changes: (see Item 11)

1. Add another hidden layer (L2)
2. Increase the epochs to 200 (from 50)
3. Use optimizer = Adamax

Item 11: LSTM Model with two layers and changes Version 2

```
# Adding layers
model.add(LSTM(512, input_shape=(num_steps, X_train.shape[2]),
return_sequences=True))
model.add(Dropout(0.1))
model.add(LSTM(256))
model.add(Dense(256))
model.add(Dropout(0.1))
model.add(Dense(num_classes, activation='softmax'))

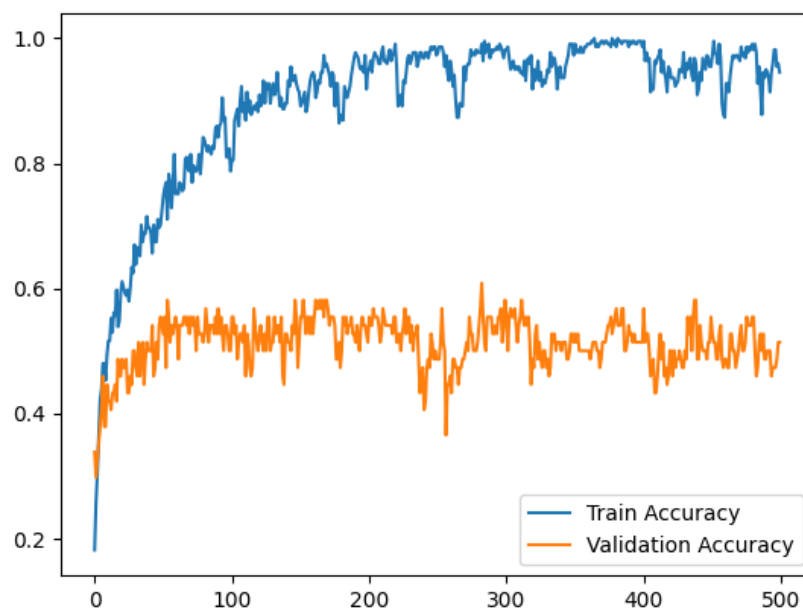
# Compiling the model for training
opt = Adamax(learning_rate=0.01)
model.compile(loss='categorical_crossentropy', optimizer=opt,
metrics=['accuracy'])
```

Results:

Modifying the model did reduce the losses, which is good, and it did increase the training accuracy. However, the difference between the validating accuracy and the training one is too much. Again, not having enough validation data is the cause of this overfitting.

This can be noticed in the accuracy convergence of the train and validation data as seen in image 13

Image 13: Train vs. Validation Accuracy (v.2)



2.2. LSTM Model Version 3

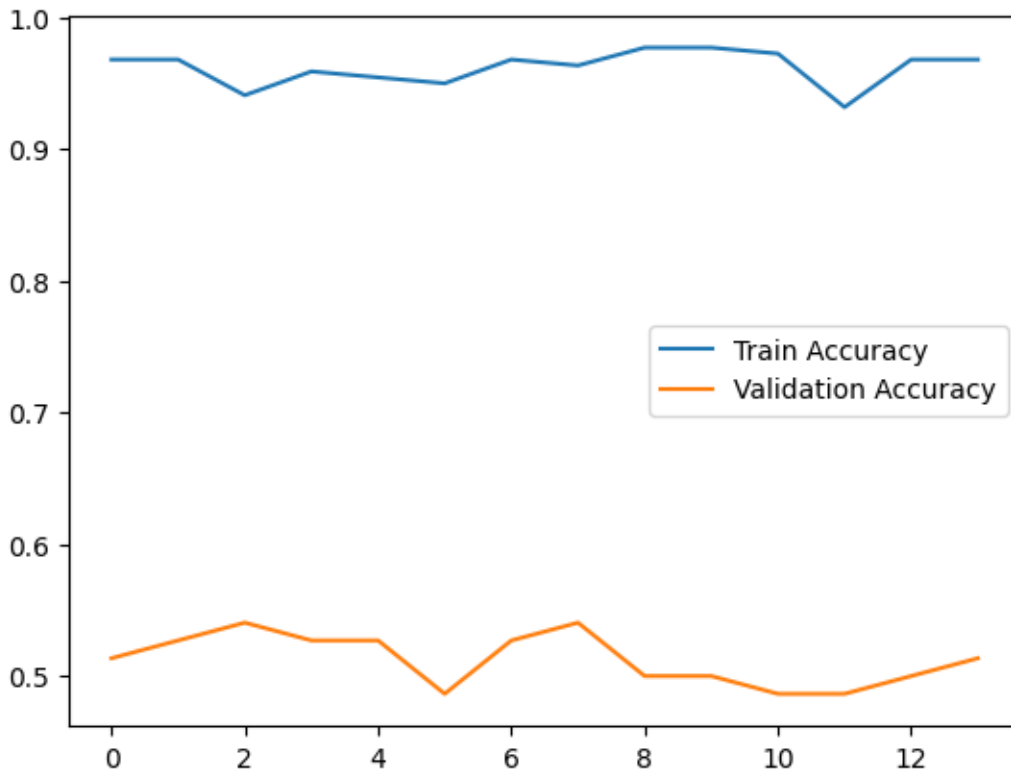
In efforts to stop a situation of overfitting of the data, a new code feature is introduced to the model this is an early stopping of the fitting of the data. This is accomplished using the library:

```
from tensorflow.keras.callbacks import EarlyStopping
and settin the number of passes to 10 in this case.
early_stopping = EarlyStopping(patience=10)
```

Results:

Introducing the early stopping criteria helped reduce overfitting and computational time. Notice how the scores are close enough to the trained data from the same model earlier with 500 epochs but without early stopping. See image 14

Image 14: Train vs. Validation Accuracy (v.3)



3. CNN MODEL (Convolutional Neural Network)

Next version is about testing the CNN model with a regular numbering dataset.

Step one is to load the related Python libraries:

```
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense, Dropout
```

The model is structured of two hidden layers, utilizing the activation function of ReLU and softmax on the classification. See Item 16.

Item 16. CNN Model Version 1

```
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')
])
```


Results:

Structure:

| Layer (type) | Output Shape | Param # |
|--------------------------------|-----------------|---------|
| conv1d (Conv1D) | (None, 25, 64) | 1024 |
| max_pooling1d (MaxPooling1D) | (None, 12, 64) | 0 |
| conv1d_1 (Conv1D) | (None, 10, 128) | 24704 |
| max_pooling1d_1 (MaxPooling1D) | (None, 5, 128) | 0 |
| flatten (Flatten) | (None, 640) | 0 |
| dense_6 (Dense) | (None, 128) | 82048 |
| dropout_7 (Dropout) | (None, 128) | 0 |
| dense_7 (Dense) | (None, 9) | 1161 |
| Total params: 108,937 | | |
| Trainable params: 108,937 | | |

Classification has improved, see Item 12.2

Classification Report - CNN Version 1 Model:



| | precision | recall | f1-score | support |
|--------------|-----------|--------|----------|---------|
| bach | 0.50 | 0.56 | 0.53 | 9 |
| bartok | 0.62 | 0.62 | 0.62 | 8 |
| byrd | 0.56 | 0.62 | 0.59 | 8 |
| chopin | 0.55 | 0.75 | 0.63 | 8 |
| handel | 0.43 | 0.38 | 0.40 | 8 |
| hummel | 0.80 | 0.44 | 0.57 | 9 |
| mendelssohn | 0.25 | 0.38 | 0.30 | 8 |
| mozart | 0.50 | 0.38 | 0.43 | 8 |
| schumann | 0.50 | 0.38 | 0.43 | 8 |
| accuracy | | 0.50 | | 74 |
| macro avg | 0.52 | 0.50 | 0.50 | 74 |
| weighted avg | 0.53 | 0.50 | 0.50 | 74 |

In this model composer Bartok has top performance with a f1 score of 0.62 (62%) and the lowest score is with composer Mendelssohn at 0.30 (30%)