DS203 Project

Autumn ‘24

Team: Billie Jean

Members:
1. Chinmay Kale, 23B1849 (Anchor)
2. Gokularamanan R S, 23B1854
3. Arya Joshi, 23B1853
4. Arash Dev Ahlawat, 23B1817

Objectives of the project

- Analysis of the 116 MFCC files, and broad classification into given groups
- Individual identification of at least 3 songs of each category

Our approach to the problem

- To achieve classifying songs from their MFCC, the model needs to handle large amounts of time-series data with spatial and temporal structures, just like what is there in MFCC files.
- We believed that a CNN-RNN model would perform good in such circumstances. https://arxiv.org/pdf/1604.04573 Served as a motivation to use it.
- More details about the processes followed are explained in the next two slides.

Processes followed

1. EDA on the dataset, including:
   a. PCA followed by scatterplot
   b. Heatmaps
   c. Elbow diagrams
   d. Attempts at classification
2. Study of MFCC coefficients
3. Reverse-engineering the coefficients into a song, attempts at manual classification
4. Creating a labelled dataset of external songs from the internet and giving them labels.

Processes followed (contd.)

5. CNN+RNN code

    a. Preparing the MFCC data and modifying it to suit our specifications
    b. Training a CNN model on these train-data songs and applying them to the official MFCC-files-v2 dataset.

6. We used Kaggle to run our model with GPUs, since the model required immense computational power, our computers simply couldn't handle it.

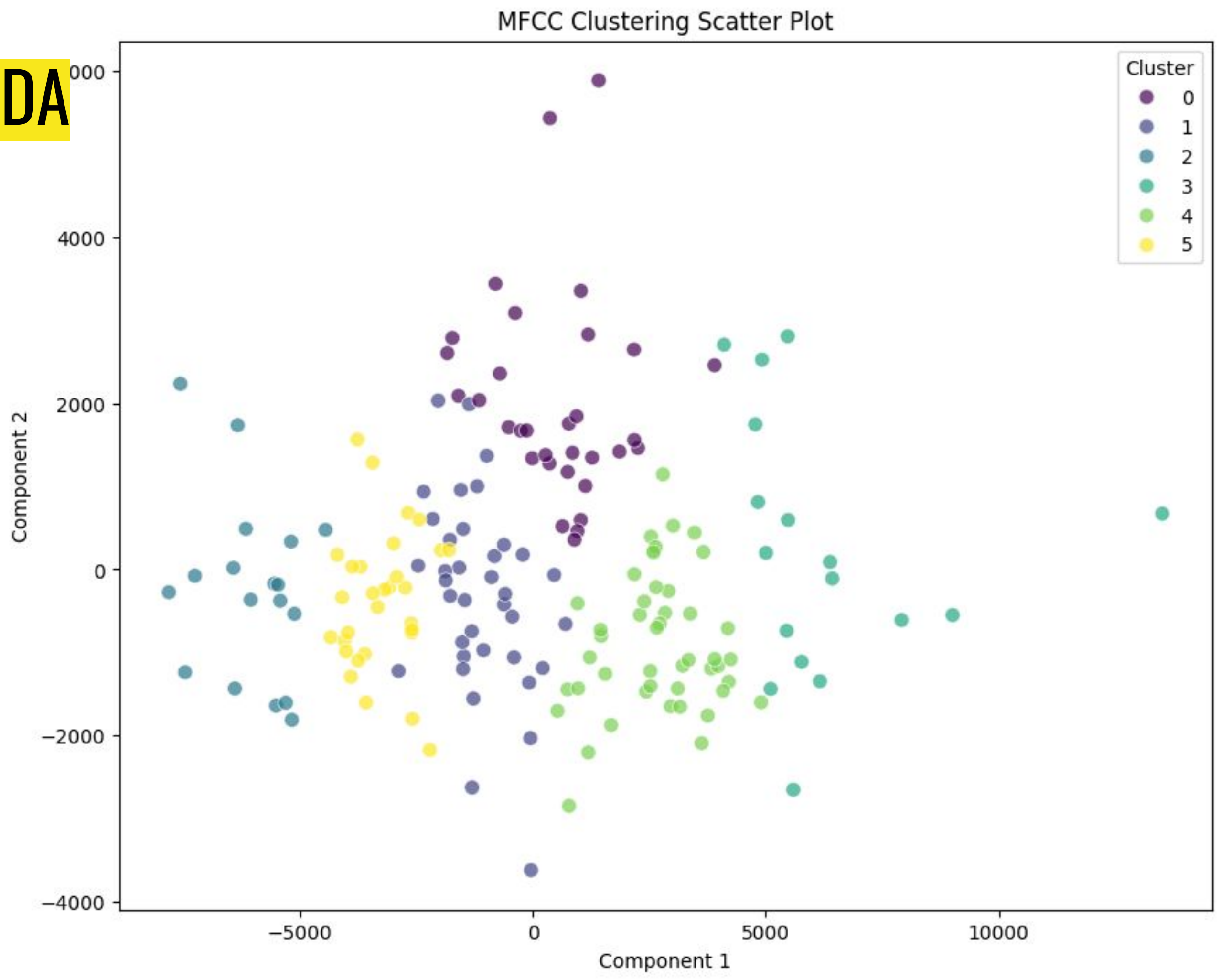
Link for our notebook and our publicly available training data
https://www.kaggle.com/code/gokularamanan/billiejean-final-ds203

EDA

In order to properly understand the data, we performed several analyses:

Analysis Step	**Description**	**Explanation**
PCA Scatter Plot	Visualizing data spread with PCA	Scatter plot showing the spread of files along the two principal components (PC1 and PC2) with the highest variance.
Elbow Plot	Explained Variance per Component	Line plot of explained variance vs. number of components to find optimal PCA components by the "elbow" point.
MFCC Heatmaps	Temporal Changes in Frequency Patterns	Heatmaps of each file's MFCC features, where x-axis represents time and y-axis represents frequency bins.

EDA

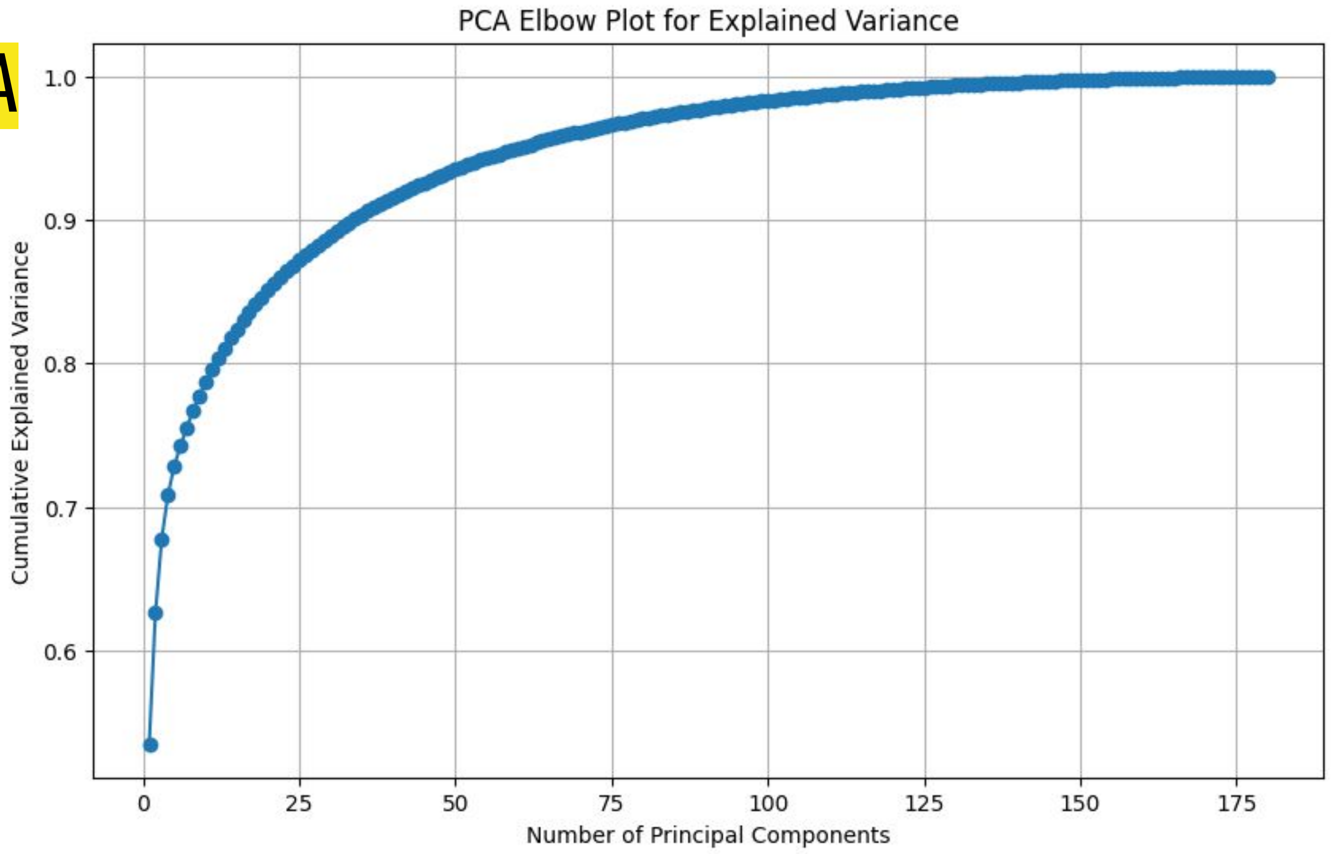


A simple clustering plot between features 1 and 2 from the **external training data**

<u>Analysis:</u>
There are definite patterns observable here.

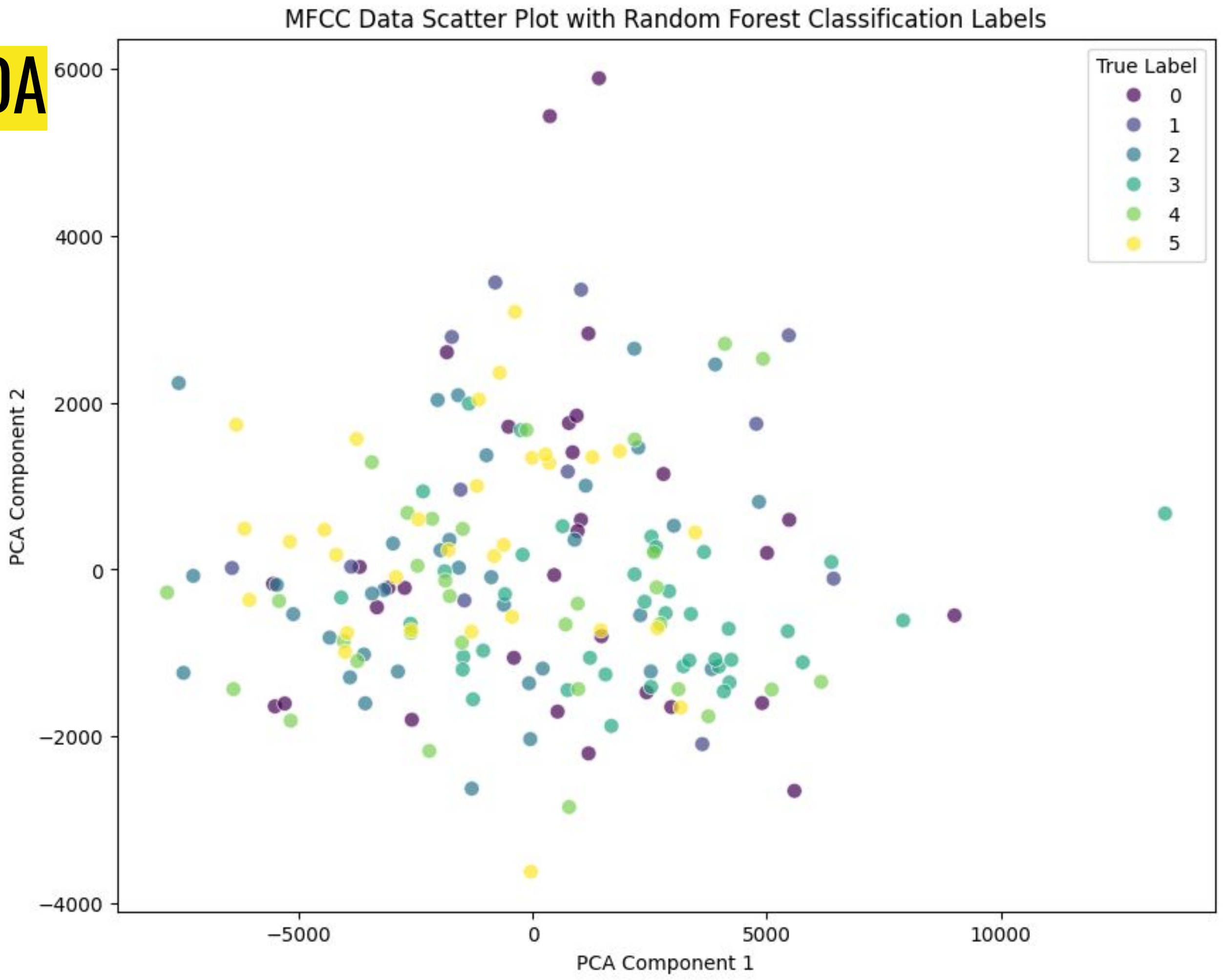
We can observe that each genre is well differentiated from the other.

EDA



An elbow plot
We can see that most of the variance can be explained in ~20 components

EDA



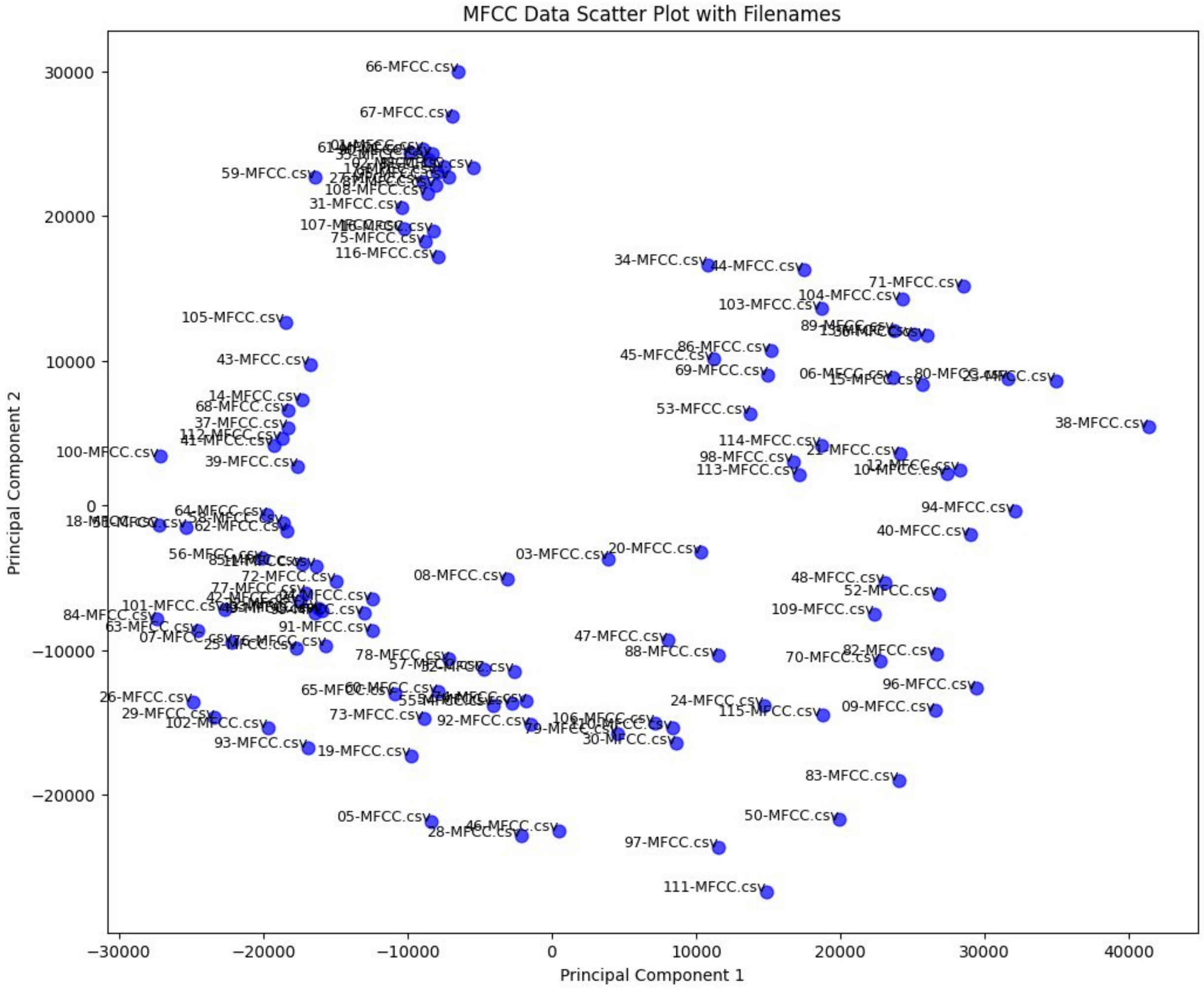
Attempted Random Forest Classification for the **test 116 songs**

<u>Analysis:</u>

This plot doesn't have properly defined clusters as in the normal scatter plot between component 1 and 2 of training data

Not comprehensible

EDA

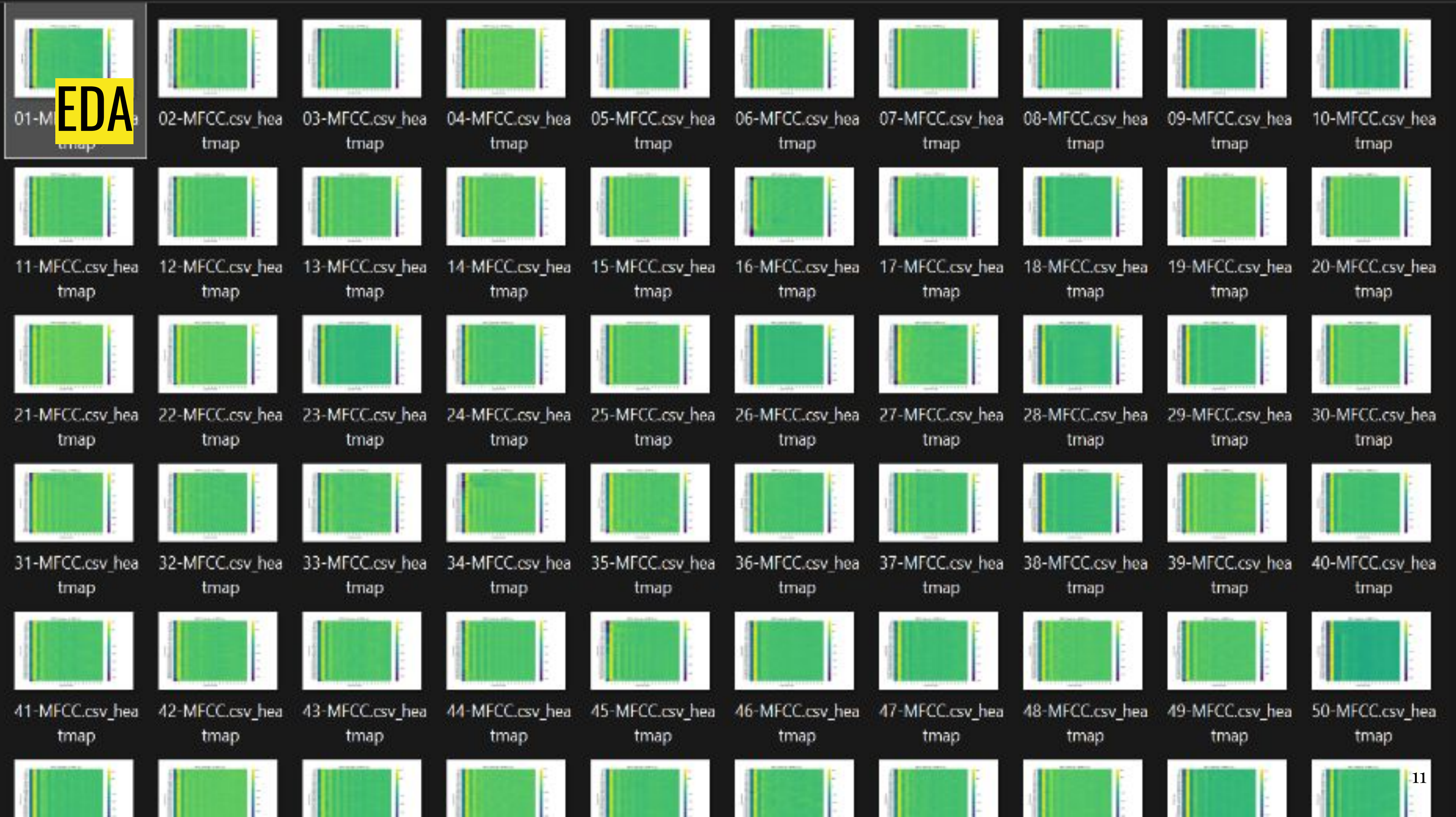


The same plot as in previous slide, just added filenames for each point.

Some basic metrics

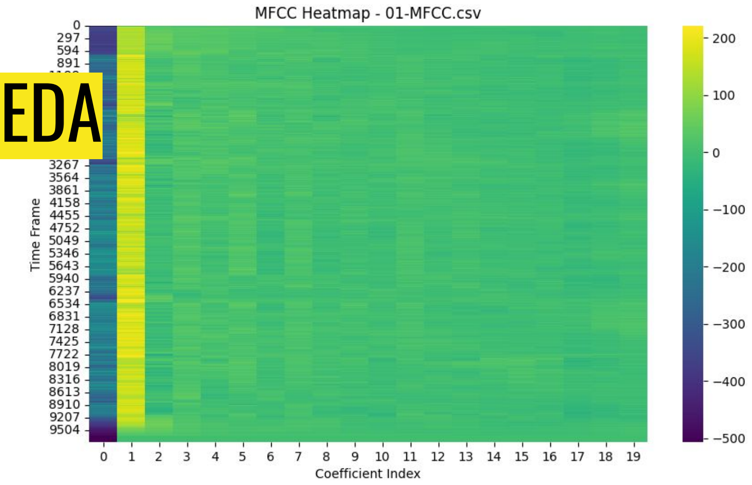
Metric	Value
Average Inter-Cluster Distance	32066.9200302
Average Intra-Cluster Difference	44531.67912857844
Average Nearest Neighbour Difference	20885.49093277749
Total Scatter	122770684650.1105

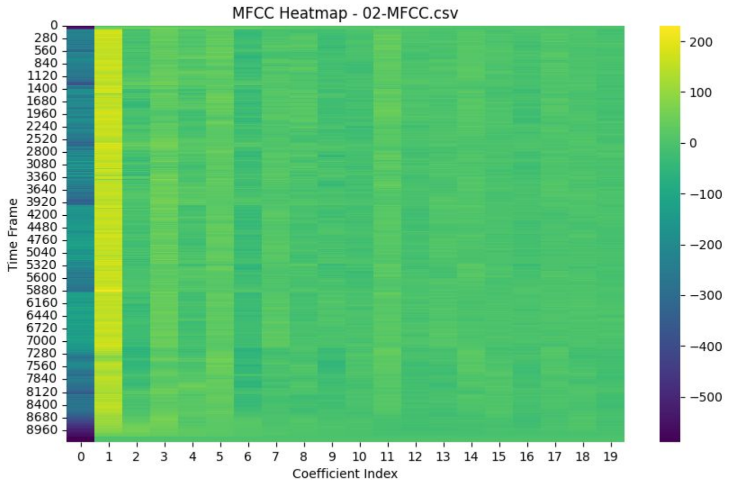
EDA

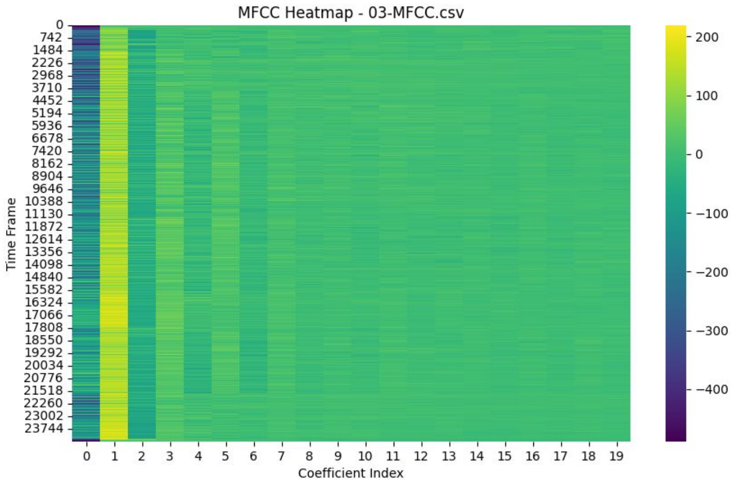


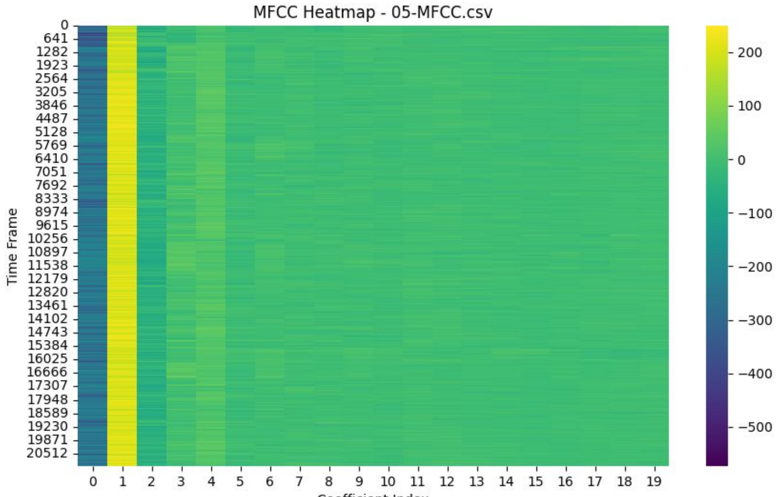
01-M[illegible]a tmap
02-MFCC.csv_heatmap
03-MFCC.csv_heatmap
04-MFCC.csv_heatmap
05-MFCC.csv_heatmap
06-MFCC.csv_heatmap
07-MFCC.csv_heatmap
08-MFCC.csv_heatmap
09-MFCC.csv_heatmap
10-MFCC.csv_heatmap
11-MFCC.csv_heatmap
12-MFCC.csv_heatmap
13-MFCC.csv_heatmap
14-MFCC.csv_heatmap
15-MFCC.csv_heatmap
16-MFCC.csv_heatmap
17-MFCC.csv_heatmap
18-MFCC.csv_heatmap
19-MFCC.csv_heatmap
20-MFCC.csv_heatmap
21-MFCC.csv_heatmap
22-MFCC.csv_heatmap
23-MFCC.csv_heatmap
24-MFCC.csv_heatmap
25-MFCC.csv_heatmap
26-MFCC.csv_heatmap
27-MFCC.csv_heatmap
28-MFCC.csv_heatmap
29-MFCC.csv_heatmap
30-MFCC.csv_heatmap
31-MFCC.csv_heatmap
32-MFCC.csv_heatmap
33-MFCC.csv_heatmap
34-MFCC.csv_heatmap
35-MFCC.csv_heatmap
36-MFCC.csv_heatmap
37-MFCC.csv_heatmap
38-MFCC.csv_heatmap
39-MFCC.csv_heatmap
40-MFCC.csv_heatmap
41-MFCC.csv_heatmap
42-MFCC.csv_heatmap
43-MFCC.csv_heatmap
44-MFCC.csv_heatmap
45-MFCC.csv_heatmap
46-MFCC.csv_heatmap
47-MFCC.csv_heatmap
48-MFCC.csv_heatmap
49-MFCC.csv_heatmap
50-MFCC.csv_heatmap

EDA


MFCC Heatmap - 01-MFCC.csv
Time Frame
Coefficient Index

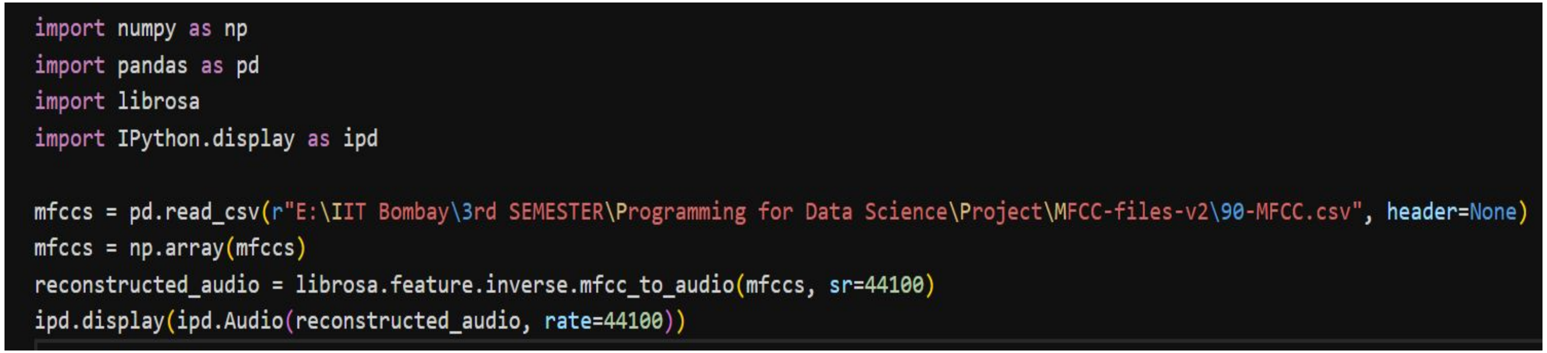

MFCC Heatmap - 02-MFCC.csv
Time Frame
Coefficient Index


MFCC Heatmap - 03-MFCC.csv
Time Frame
Coefficient Index


MFCC Heatmap - 05-MFCC.csv
Time Frame

Analysing the MFCC files and reverse engineering

- Using the following bit of code, we reverse-engineered the given MFCC files into the original audio



```
import numpy as np
import pandas as pd
import librosa
import IPython.display as ipd

mfccs = pd.read_csv(r"E:\IIT Bombay\3rd SEMESTER\Programming for Data Science\Project\MFCC-files-v2\90-MFCC.csv", header=None)
mfccs = np.array(mfccs)
reconstructed_audio = librosa.feature.inverse.mfcc_to_audio(mfccs, sr=44100)
ipd.display(ipd.Audio(reconstructed_audio, rate=44100))
```

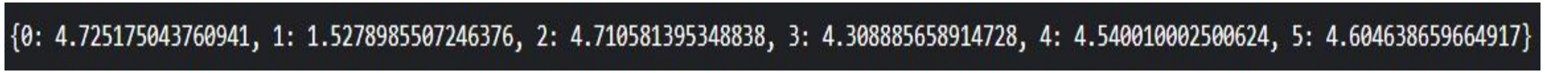
- The audio was fairly noisy, due to loss of information
- However, we were able to identify the songs, along with the singer

A brief breakdown of MFCC coefficients

- MFCCs, standing for **Mel-Frequency Cepstral Coefficients**, capture information about frequency ranges using the mel scale (a scale reflecting human pitch perception)
- Humans do not perceive frequencies on a linear scale. Therefore, the a mel-scale closely approximates the human auditory system's response, which is more sensitive to changes in lower frequencies than higher ones
- The signal is now converted to a time-domain representation called the cepstrum. The cepstrum separates the signal's periodic variation (pitch) from the slow variation (timbre), focusing on the latter which carries most of the information relevant to recognizing speech
- Often, the first MFCC (index 0) represents the **overall energy** of the audio frame while the **first few coefficients** (1–5) capture the **broad spectral shape** or general characteristics of the sound with higher coefficients capturing the finer spectral fluctuations

Reverse Engineering (contd.)

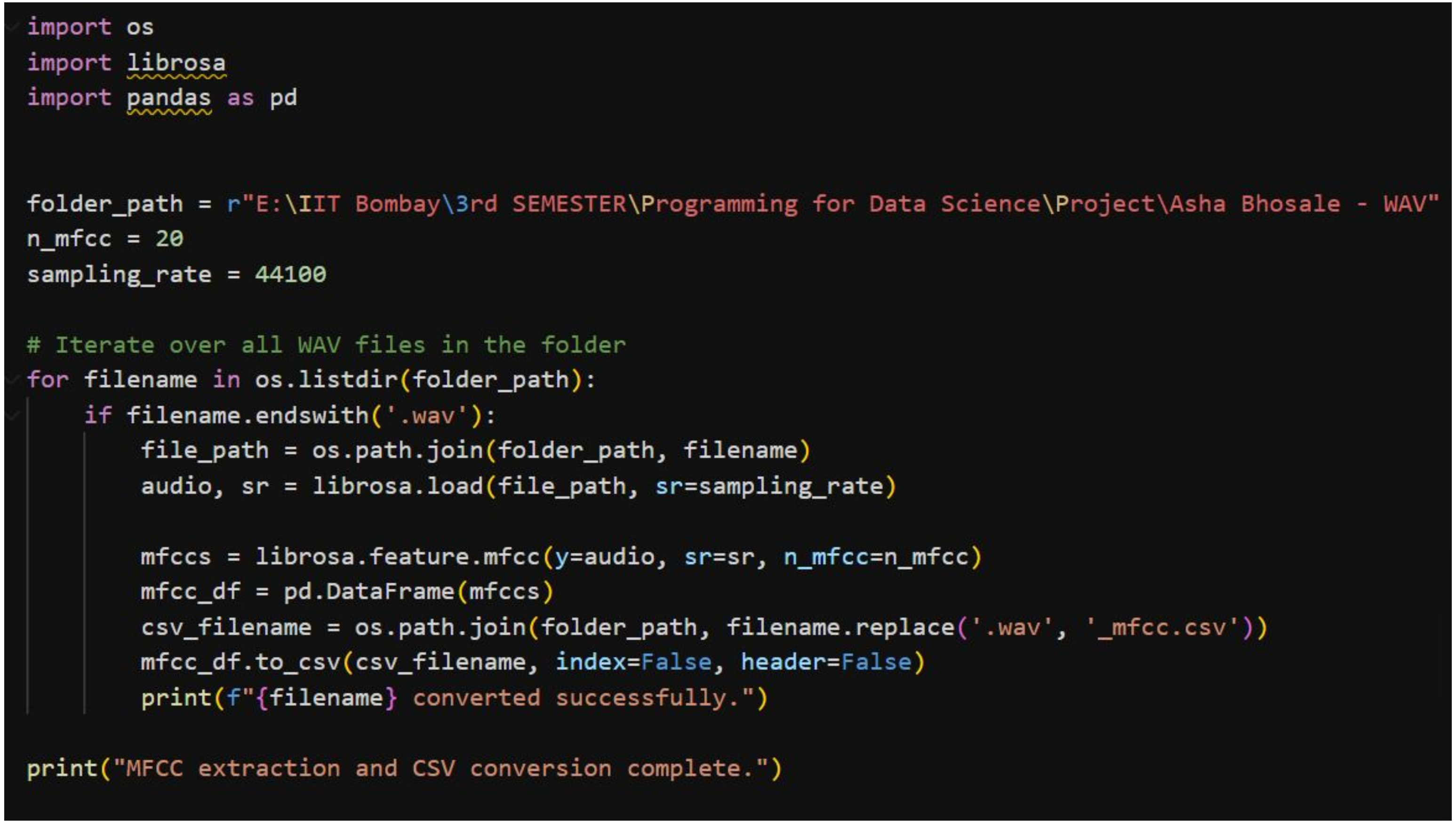
- Using the reverse engineered files; we were able to identify each (most) of the 116 files.
- Attached in this link is a sheet of each of the songs along with singer and label: https://docs.google.com/spreadsheets/d/11E85fL6itdAxA37ArHm91HsEt5B1scq_YT8_Auf4vno/edit?usp=sharing
- This allowed us to verify the outputs of our model and check accuracy.
- Below is a sample reconstructed audio file, 90-mfcc.csv, it is the national anthem. https://drive.google.com/file/d/12nwDG8slHmtNaCdnQl4RupjcD1KEpJ76/view?usp=sharing
- Further, we also analyzed basic metrics such as the average length (minutes) of each song. This helped us what kind of songs from the selected genres we had to download. Below is the average length of songs of each genre.
  0 = Asha, 1 = Jana gana, 2 = Kishore, 3 = Bhav, 4 = Lavni, 5 = MJ



```
{0: 4.725175043760941, 1: 1.5278985507246376, 2: 4.710581395348838, 3: 4.308885658914728, 4: 4.540010002500624, 5: 4.604638659664917}
```

Creating a labelled dataset of songs

- As given in the instructions, we are required to classify the songs into various labels. For this, we need to train the model on some (a lot of) external songs.
- To serve this purpose, we downloaded about 30 songs per label from the internet: *{'Asha Bhosale': 0, 'Jana Gana Mana': 1, 'Kishore Kumar':2, 'Marathi Bhavgeet: 3, 'Marathi Lavni:4,'Michael Jackson':5}*
- We then converted these .wav songs into MFCCs, using the create_MFCC_coefficients() function from the code snippet provided on moodle and saved them in a CSV file.
- The code snippet is shown in the next slide.
- The drive link containing these files: https://iitbacin-my.sharepoint.com/:f:/g/personal/gokul_ramanan_iitb_ac_in/EqWt7dMPfnRDhp5UN2y6uPwBQKo6ZzVg28vaqZBOFEtRSA?e=5Wnt9H



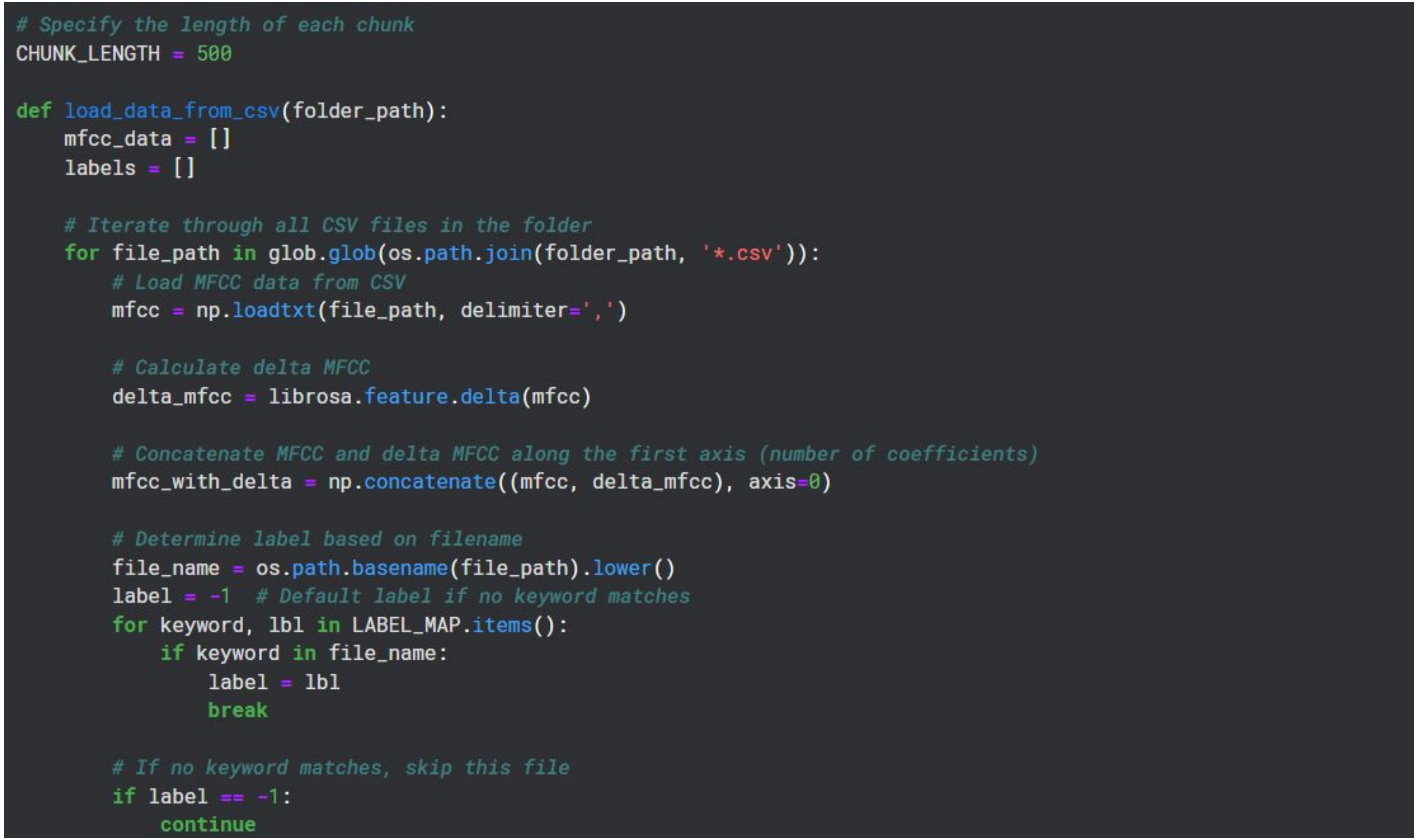
```python
import os
import librosa
import pandas as pd


folder_path = r"E:\IIT Bombay\3rd SEMESTER\Programming for Data Science\Project\Asha Bhosale - WAV"
n_mfcc = 20
sampling_rate = 44100

# Iterate over all WAV files in the folder
for filename in os.listdir(folder_path):
    if filename.endswith('.wav'):
        file_path = os.path.join(folder_path, filename)
        audio, sr = librosa.load(file_path, sr=sampling_rate)

        mfccs = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=n_mfcc)
        mfcc_df = pd.DataFrame(mfccs)
        csv_filename = os.path.join(folder_path, filename.replace('.wav', '_mfcc.csv'))
        mfcc_df.to_csv(csv_filename, index=False, header=False)
        print(f"{filename} converted successfully.")

print("MFCC extraction and CSV conversion complete.")
```

CNN Code - Explained

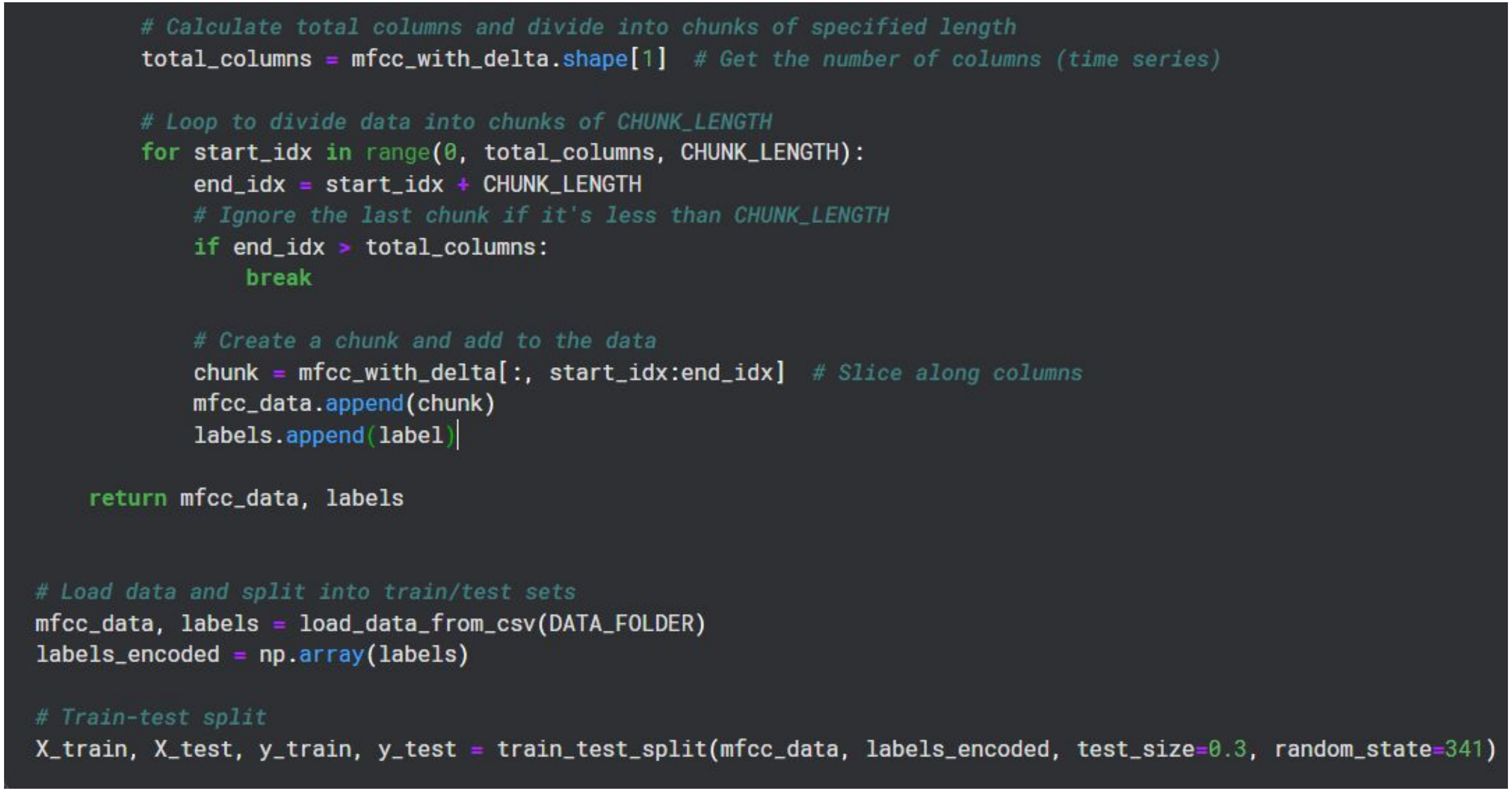
- We have broken down our code into 3 parts -
  - Preparing the MFCC files to be processed(calculating delta mfcc and chunking)
  - Defining the model along with the forward function
  - Training part of the model (train()), evaluating the model (evaluate()) on the training data
  - Finally predicting the labels by of the 116 MFCC songs by dividing them into chunks (testing phase).

Part 1. Importing the CSV files [Code Snippets in slides 22 and 23]

- We define a function ‘load_data_from_csv’, which imports the MFCC coefficients for each song from a csv file, containing 20 rows of the mfcc coefficients and variable number of columns
- The function calculates delta_mfcc, which tracks how the each coefficient changes over the columns, and adds it to the original array, now having total 40 rows
- Now the data is divided into chunks of 500 columns, to make all the inputs have a uniform shape of (40,500)
- For the training of the model, we also define a labels list, which contains the corresponding labels for each song, and then split this data into train and test datasets
- The actual mfcc files are divided into chunks without labels, and then fed to the model for predictions

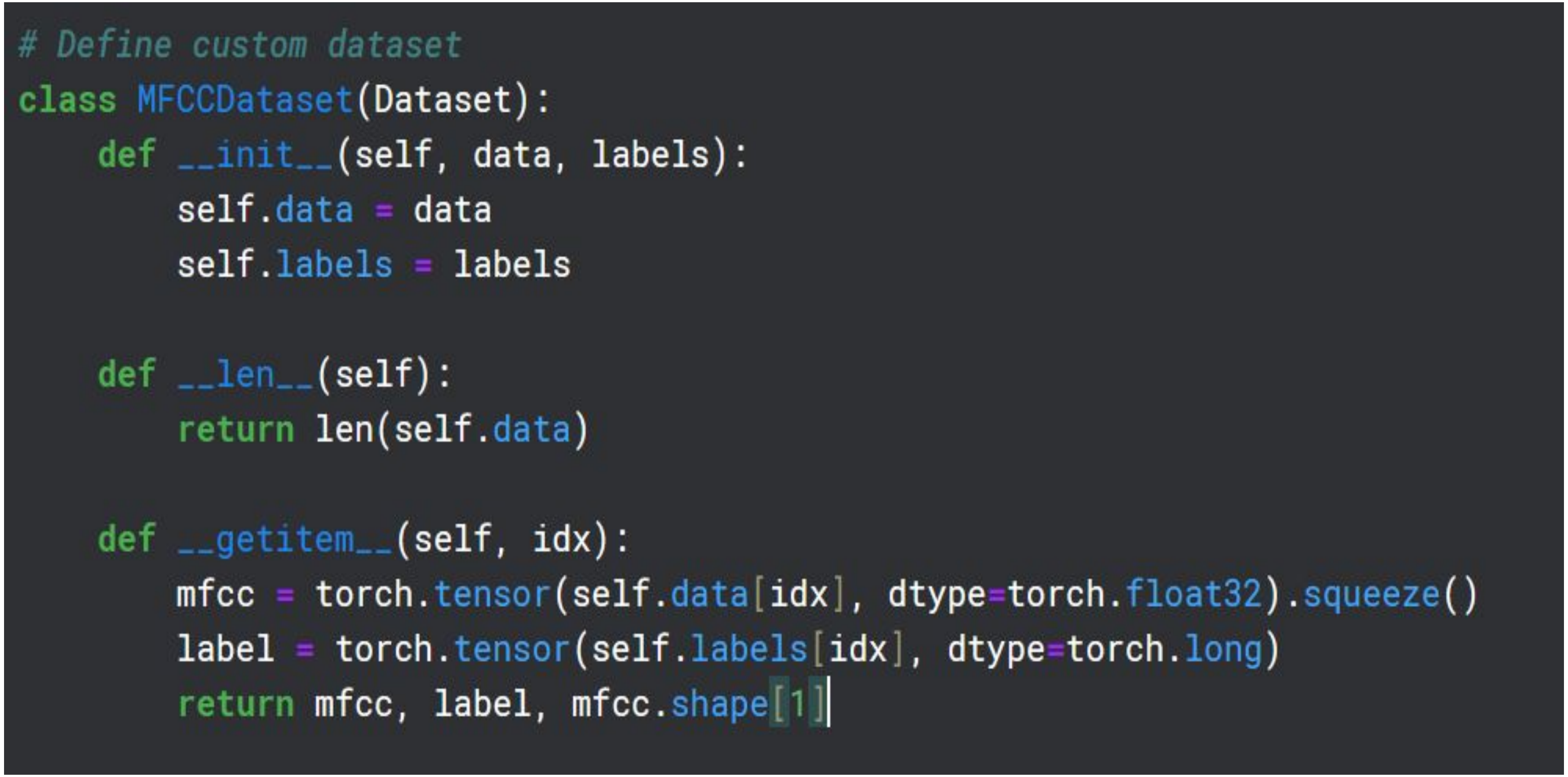


```python
# Specify the length of each chunk
CHUNK_LENGTH = 500

def load_data_from_csv(folder_path):
    mfcc_data = []
    labels = []

    # Iterate through all CSV files in the folder
    for file_path in glob.glob(os.path.join(folder_path, '*.csv')):
        # Load MFCC data from CSV
        mfcc = np.loadtxt(file_path, delimiter=',')

        # Calculate delta MFCC
        delta_mfcc = librosa.feature.delta(mfcc)

        # Concatenate MFCC and delta MFCC along the first axis (number of coefficients)
        mfcc_with_delta = np.concatenate((mfcc, delta_mfcc), axis=0)

        # Determine label based on filename
        file_name = os.path.basename(file_path).lower()
        label = -1  # Default label if no keyword matches
        for keyword, lbl in LABEL_MAP.items():
            if keyword in file_name:
                label = lbl
                break

        # If no keyword matches, skip this file
        if label == -1:
            continue
```

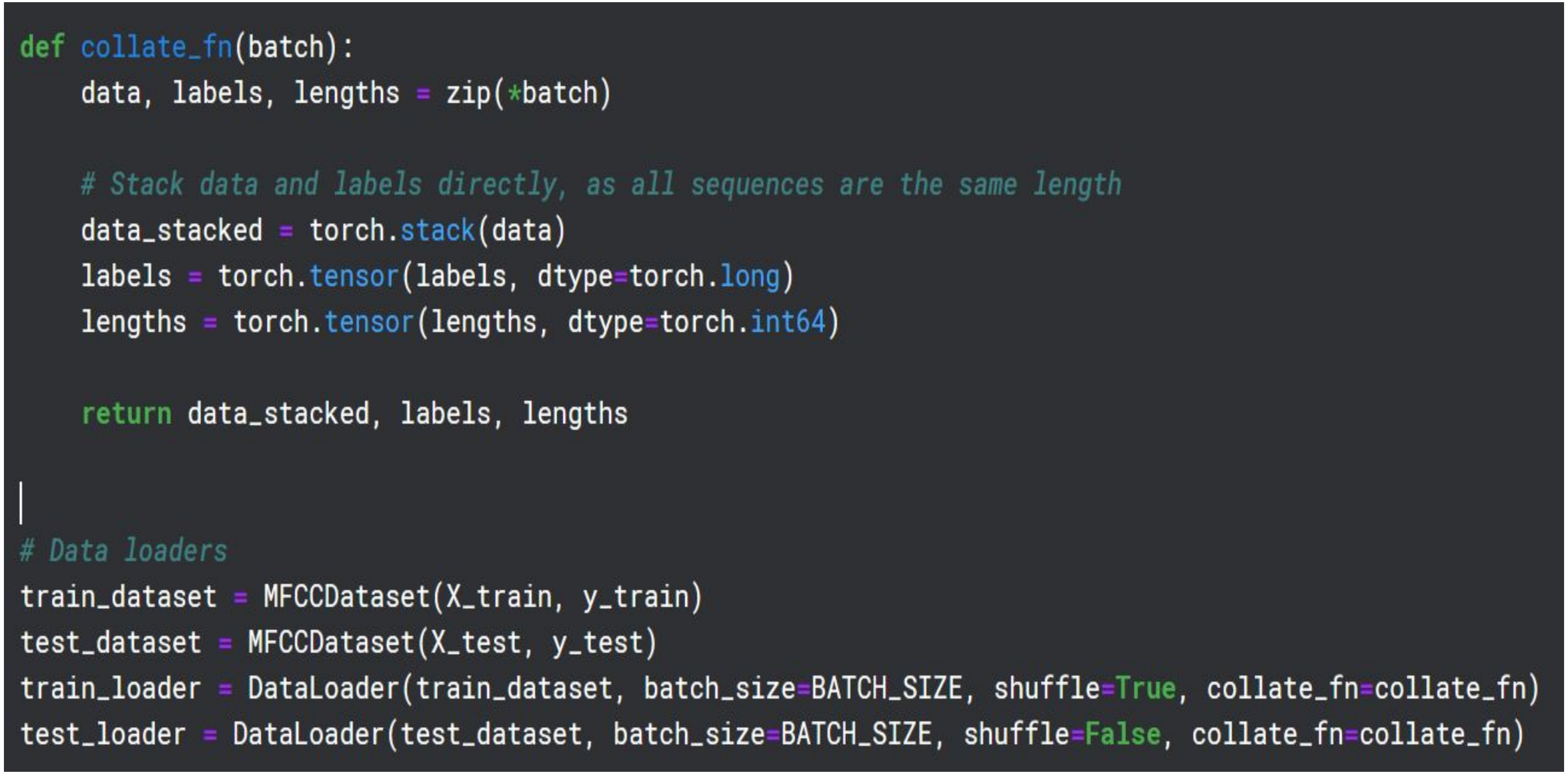

```
        # Calculate total columns and divide into chunks of specified length
        total_columns = mfcc_with_delta.shape[1]  # Get the number of columns (time series)

        # Loop to divide data into chunks of CHUNK_LENGTH
        for start_idx in range(0, total_columns, CHUNK_LENGTH):
            end_idx = start_idx + CHUNK_LENGTH
            # Ignore the last chunk if it's less than CHUNK_LENGTH
            if end_idx > total_columns:
                break

            # Create a chunk and add to the data
            chunk = mfcc_with_delta[:, start_idx:end_idx]  # Slice along columns
            mfcc_data.append(chunk)
            labels.append(label)

    return mfcc_data, labels


# Load data and split into train/test sets
mfcc_data, labels = load_data_from_csv(DATA_FOLDER)
labels_encoded = np.array(labels)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(mfcc_data, labels_encoded, test_size=0.3, random_state=341)
```

Data Handling - Custom dataset [Code Snippet in next slide, 25]

- We define a custom class "MFCCDataset" for handling the MFCC data with labels; preparing the data for testing
- This converts the data to a PyTorch tensor, having a shape (40, 500), where 40 is the number of MFCC coefficients, and 500 is the chunk size
- Since we have defined each chunk to be of equal length, the collate function directly stacks the data, and converts the labels and lengths into tensors
- Then, the DataLoader is initialised for training and test sets, using the collate function
- The batch-size is taken as the pre-defined value, and shuffle is kept True which helps the model generalize better. Code snippet is given in next slide



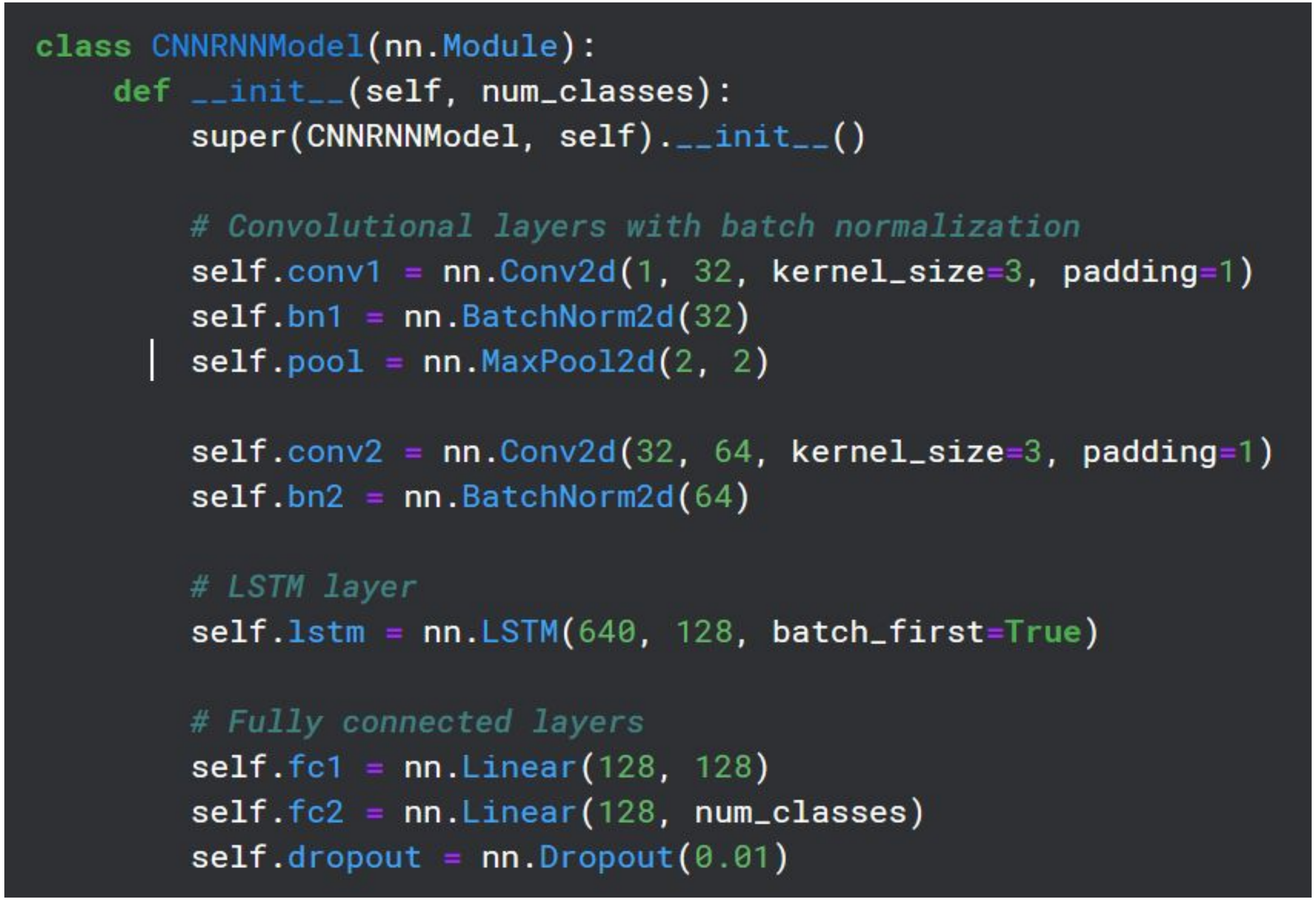
```python
# Define custom dataset
class MFCCDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        mfcc = torch.tensor(self.data[idx], dtype=torch.float32).squeeze()
        label = torch.tensor(self.labels[idx], dtype=torch.long)
        return mfcc, label, mfcc.shape[1]
```



```
def collate_fn(batch):
    data, labels, lengths = zip(*batch)

    # Stack data and labels directly, as all sequences are the same length
    data_stacked = torch.stack(data)
    labels = torch.tensor(labels, dtype=torch.long)
    lengths = torch.tensor(lengths, dtype=torch.int64)

    return data_stacked, labels, lengths


# Data loaders
train_dataset = MFCCDataset(X_train, y_train)
test_dataset = MFCCDataset(X_test, y_test)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=collate_fn)
```

Defining the model class [Code Snippet in next slide, 29]

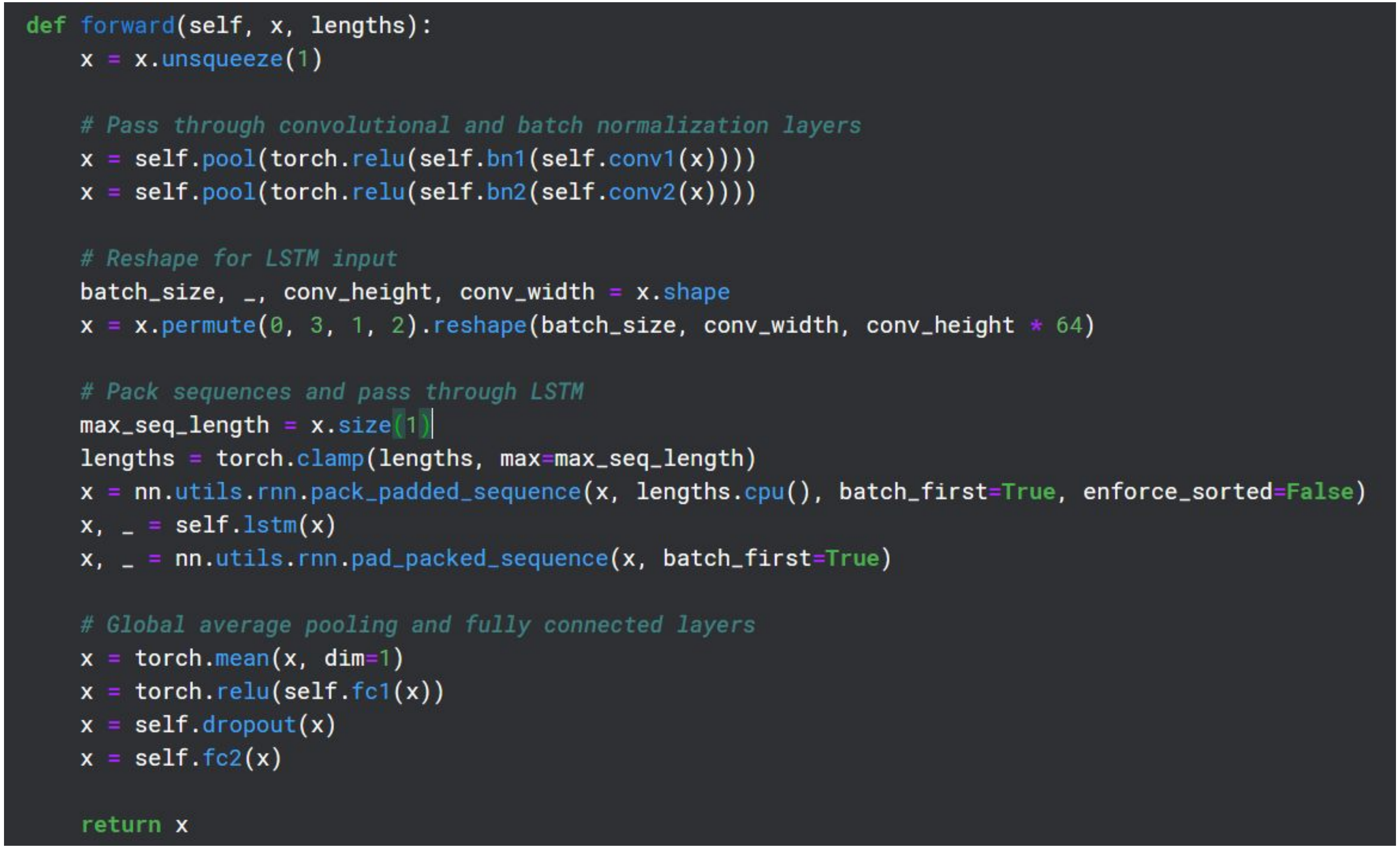
- We define a neural network model CNNRNNModel, which combines convolutional layers (CNN) for feature extraction with an LSTM (RNN) for sequential data processing
- The model starts with 2 convolutional layers with batch normalisation, by extracting features from each input
- Pooling then reduces the dimensions, downscaling the feature maps
- Then, the LSTM layer is configured, with a set input size calculated based on the convolution input
- Fully-connected layers reduces the LSTM output; mapping it to *num_ classes*
- The dropout layer helps prevent overfitting



```
class CNNRNNModel(nn.Module):
    def __init__(self, num_classes):
        super(CNNRNNModel, self).__init__()

        # Convolutional layers with batch normalization
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool = nn.MaxPool2d(2, 2)

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)

        # LSTM layer
        self.lstm = nn.LSTM(640, 128, batch_first=True)

        # Fully connected layers
        self.fc1 = nn.Linear(128, 128)
        self.fc2 = nn.Linear(128, num_classes)
        self.dropout = nn.Dropout(0.01)
```

Forward Method [Code Snippet in next slide, 31]

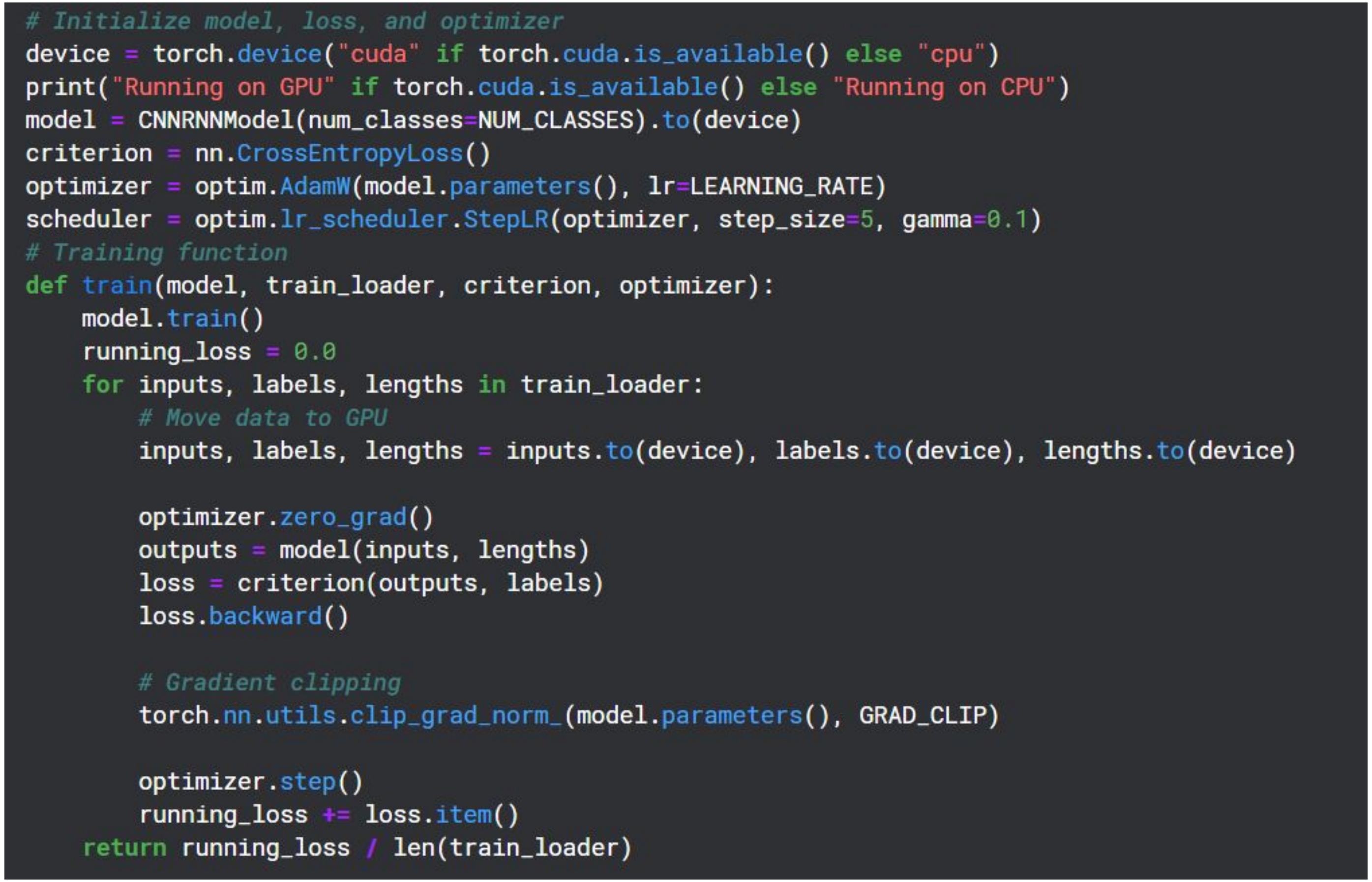
- The input is reshaped by adding a dimension, making it suitable for CNN
- The input is passed sequentially through the layers, followed by activation and pooling. This process is repeated
- The output is reshaped for the LSTM, passed through, and then unpacked
- The output is averaged across the sequence dimension, summarising the LSTM output across time
- Fully-connected layers apply transformations to the pooled features, along with functions to add non-linearity
- This model architecture is effective, as it combines feature extraction, with the LSTM layer to predict class labels



```python
def forward(self, x, lengths):
    x = x.unsqueeze(1)

    # Pass through convolutional and batch normalization layers
    x = self.pool(torch.relu(self.bn1(self.conv1(x))))
    x = self.pool(torch.relu(self.bn2(self.conv2(x))))

    # Reshape for LSTM input
    batch_size, _, conv_height, conv_width = x.shape
    x = x.permute(0, 3, 1, 2).reshape(batch_size, conv_width, conv_height * 64)

    # Pack sequences and pass through LSTM
    max_seq_length = x.size(1)
    lengths = torch.clamp(lengths, max=max_seq_length)
    x = nn.utils.rnn.pack_padded_sequence(x, lengths.cpu(), batch_first=True, enforce_sorted=False)
    x, _ = self.lstm(x)
    x, _ = nn.utils.rnn.pad_packed_sequence(x, batch_first=True)

    # Global average pooling and fully connected layers
    x = torch.mean(x, dim=1)
    x = torch.relu(self.fc1(x))
    x = self.dropout(x)
    x = self.fc2(x)

    return x
```

Training module [Code Snippet in next slide, 33]

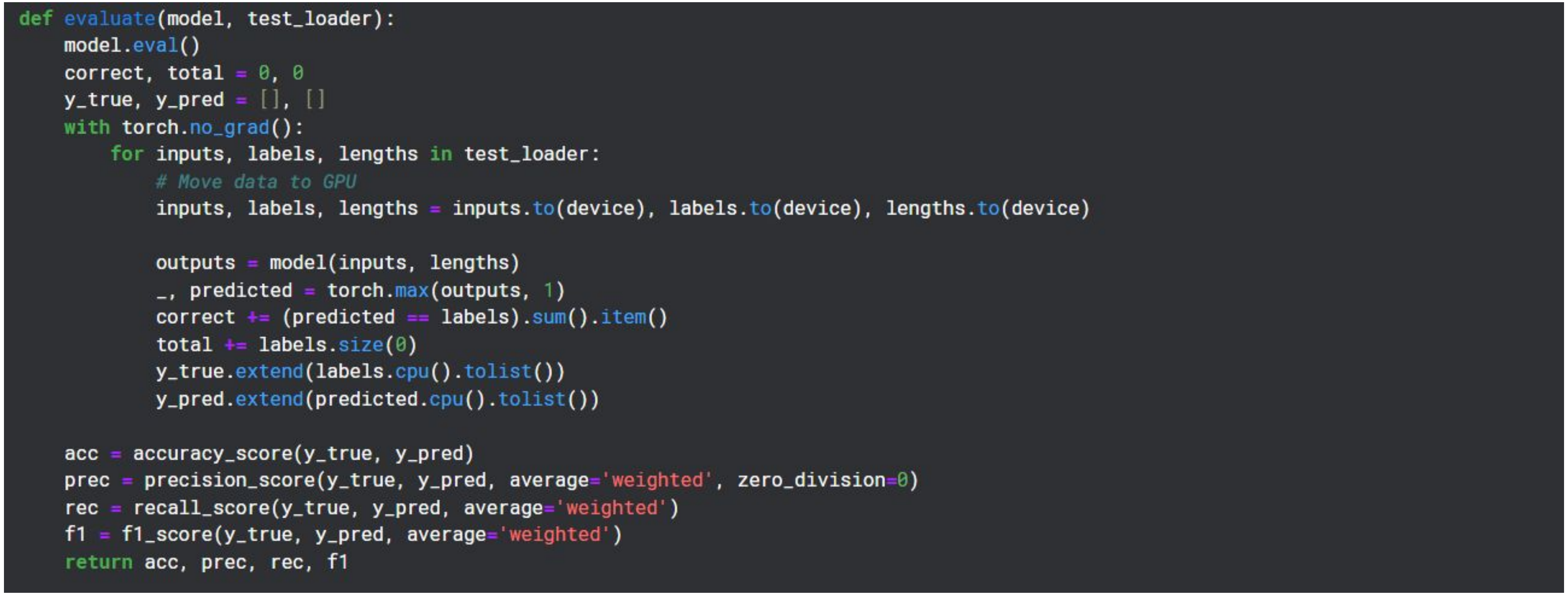
- The function, train() gets the input from trainloader(), iterates over batches of it and sends it to pytorch functional GPU (the one we're using through Kaggle, GPU P100)
- The loss for each batch is calculated through criterion(). We have used cross-entropy loss function, since it is performs for classification than compared to MAE or MSE.
- Weights are updated using backpropagation using gradient descent.
- We have used gradient clipping to prevent exploding gradients (leads to slower convergence).
- Finally, train() returns the average loss over one epoch.



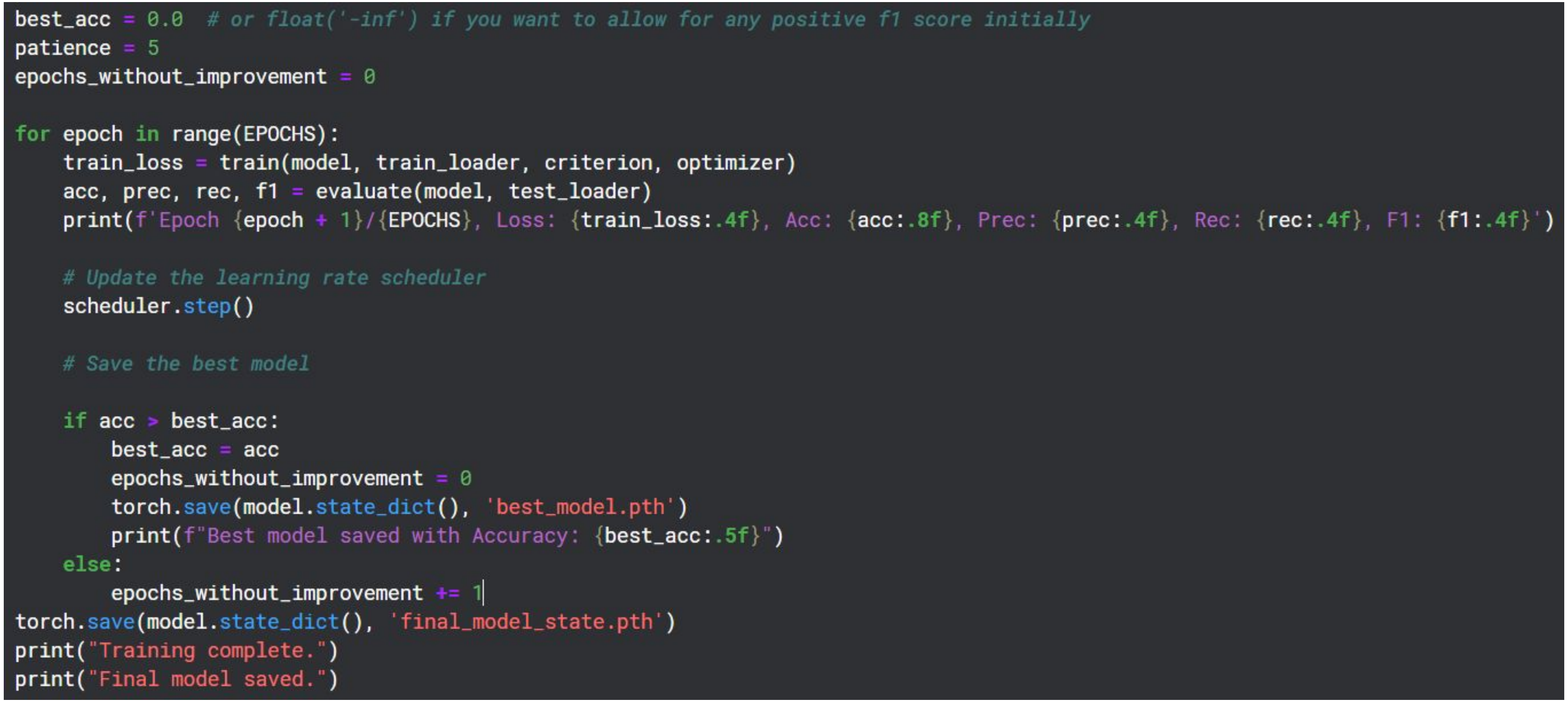
```python
# Initialize model, loss, and optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Running on GPU" if torch.cuda.is_available() else "Running on CPU")
model = CNNRNNModel(num_classes=NUM_CLASSES).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)
# Training function
def train(model, train_loader, criterion, optimizer):
    model.train()
    running_loss = 0.0
    for inputs, labels, lengths in train_loader:
        # Move data to GPU
        inputs, labels, lengths = inputs.to(device), labels.to(device), lengths.to(device)

        optimizer.zero_grad()
        outputs = model(inputs, lengths)
        loss = criterion(outputs, labels)
        loss.backward()

        # Gradient clipping
        torch.nn.utils.clip_grad_norm_(model.parameters(), GRAD_CLIP)

        optimizer.step()
        running_loss += loss.item()
    return running_loss / len(train_loader)
```

Evaluate module [Code Snippets in slides 34 & 35]

- The main difference between evaluate() and train() is that in evaluate, dropout is disabled and batch normalisation is now set to work on the learned parameters from training. Hence, gradient calculation is also not done.
- Further, metrics such as accuracy, precision, f1-score and losses are explicitly stored and returned.
- Now, we just need to run the train(), get the updated parameters, loss values and feed it into the evaluate().
- (slide 36), We set the max epochs to be 40, hence we're running the model exactly 40 times and during each epoch, the accuracy values are stored. Finally, the model with the best performance is stored as a .pth file.

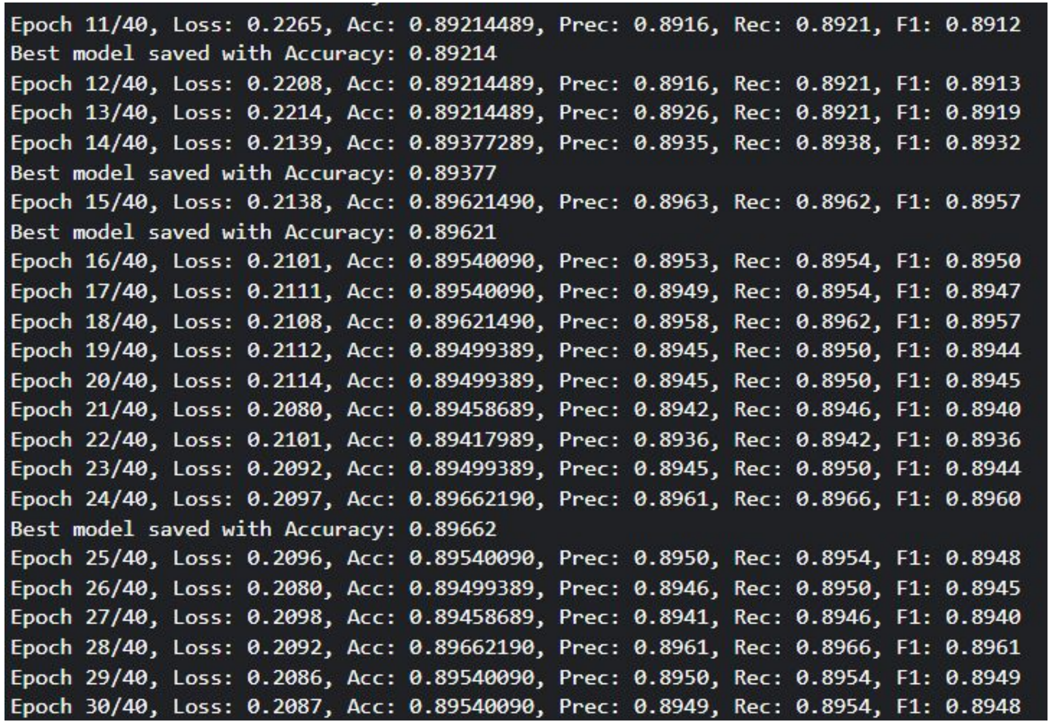


```python
def evaluate(model, test_loader):
    model.eval()
    correct, total = 0, 0
    y_true, y_pred = [], []
    with torch.no_grad():
        for inputs, labels, lengths in test_loader:
            # Move data to GPU
            inputs, labels, lengths = inputs.to(device), labels.to(device), lengths.to(device)

            outputs = model(inputs, lengths)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)
            y_true.extend(labels.cpu().tolist())
            y_pred.extend(predicted.cpu().tolist())

    acc = accuracy_score(y_true, y_pred)
    prec = precision_score(y_true, y_pred, average='weighted', zero_division=0)
    rec = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')
    return acc, prec, rec, f1
```



```python
best_acc = 0.0  # or float('-inf') if you want to allow for any positive f1 score initially
patience = 5
epochs_without_improvement = 0

for epoch in range(EPOCHS):
    train_loss = train(model, train_loader, criterion, optimizer)
    acc, prec, rec, f1 = evaluate(model, test_loader)
    print(f'Epoch {epoch + 1}/{EPOCHS}, Loss: {train_loss:.4f}, Acc: {acc:.8f}, Prec: {prec:.4f}, Rec: {rec:.4f}, F1: {f1:.4f}')

    # Update the learning rate scheduler
    scheduler.step()

    # Save the best model

    if acc > best_acc:
        best_acc = acc
        epochs_without_improvement = 0
        torch.save(model.state_dict(), 'best_model.pth')
        print(f"Best model saved with Accuracy: {best_acc:.5f}")
    else:
        epochs_without_improvement += 1
torch.save(model.state_dict(), 'final_model_state.pth')
print("Training complete.")
print("Final model saved.")
```

Metrics(train)

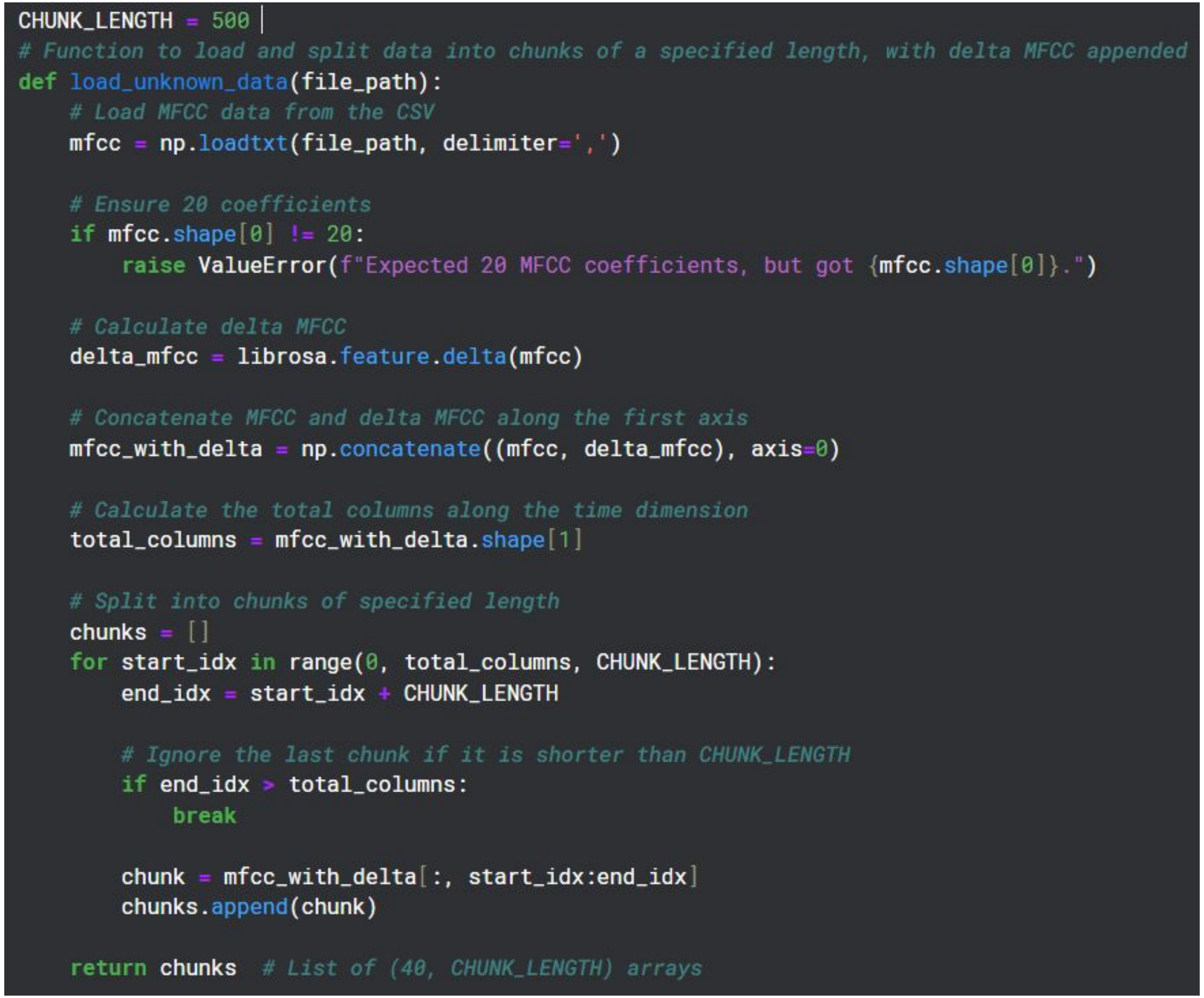


```
Epoch 11/40, Loss: 0.2265, Acc: 0.89214489, Prec: 0.8916, Rec: 0.8921, F1: 0.8912
Best model saved with Accuracy: 0.89214
Epoch 12/40, Loss: 0.2208, Acc: 0.89214489, Prec: 0.8916, Rec: 0.8921, F1: 0.8913
Epoch 13/40, Loss: 0.2214, Acc: 0.89214489, Prec: 0.8926, Rec: 0.8921, F1: 0.8919
Epoch 14/40, Loss: 0.2139, Acc: 0.89377289, Prec: 0.8935, Rec: 0.8938, F1: 0.8932
Best model saved with Accuracy: 0.89377
Epoch 15/40, Loss: 0.2138, Acc: 0.89621490, Prec: 0.8963, Rec: 0.8962, F1: 0.8957
Best model saved with Accuracy: 0.89621
Epoch 16/40, Loss: 0.2101, Acc: 0.89540090, Prec: 0.8953, Rec: 0.8954, F1: 0.8950
Epoch 17/40, Loss: 0.2111, Acc: 0.89540090, Prec: 0.8949, Rec: 0.8954, F1: 0.8947
Epoch 18/40, Loss: 0.2108, Acc: 0.89621490, Prec: 0.8958, Rec: 0.8962, F1: 0.8957
Epoch 19/40, Loss: 0.2112, Acc: 0.89499389, Prec: 0.8945, Rec: 0.8950, F1: 0.8944
Epoch 20/40, Loss: 0.2114, Acc: 0.89499389, Prec: 0.8945, Rec: 0.8950, F1: 0.8945
Epoch 21/40, Loss: 0.2080, Acc: 0.89458689, Prec: 0.8942, Rec: 0.8946, F1: 0.8940
Epoch 22/40, Loss: 0.2101, Acc: 0.89417989, Prec: 0.8936, Rec: 0.8942, F1: 0.8936
Epoch 23/40, Loss: 0.2092, Acc: 0.89499389, Prec: 0.8945, Rec: 0.8950, F1: 0.8944
Epoch 24/40, Loss: 0.2097, Acc: 0.89662190, Prec: 0.8961, Rec: 0.8966, F1: 0.8960
Best model saved with Accuracy: 0.89662
Epoch 25/40, Loss: 0.2096, Acc: 0.89540090, Prec: 0.8950, Rec: 0.8954, F1: 0.8948
Epoch 26/40, Loss: 0.2080, Acc: 0.89499389, Prec: 0.8946, Rec: 0.8950, F1: 0.8945
Epoch 27/40, Loss: 0.2098, Acc: 0.89458689, Prec: 0.8941, Rec: 0.8946, F1: 0.8940
Epoch 28/40, Loss: 0.2092, Acc: 0.89662190, Prec: 0.8961, Rec: 0.8966, F1: 0.8961
Epoch 29/40, Loss: 0.2086, Acc: 0.89540090, Prec: 0.8950, Rec: 0.8954, F1: 0.8949
Epoch 30/40, Loss: 0.2087, Acc: 0.89540090, Prec: 0.8949, Rec: 0.8954, F1: 0.8948
```

- Evaluating the training dataset, containing approximately 180 songs
- A 40 epoch run, with batch size set to 20, learning rate = 0.005 and grad clip = 5
- NOTE: The accuracy printed is not the actual accuracy of how many songs have been predicted right, since the model is trained on multiple chunks of each song

Predicting labels: Loading and dividing into chunks

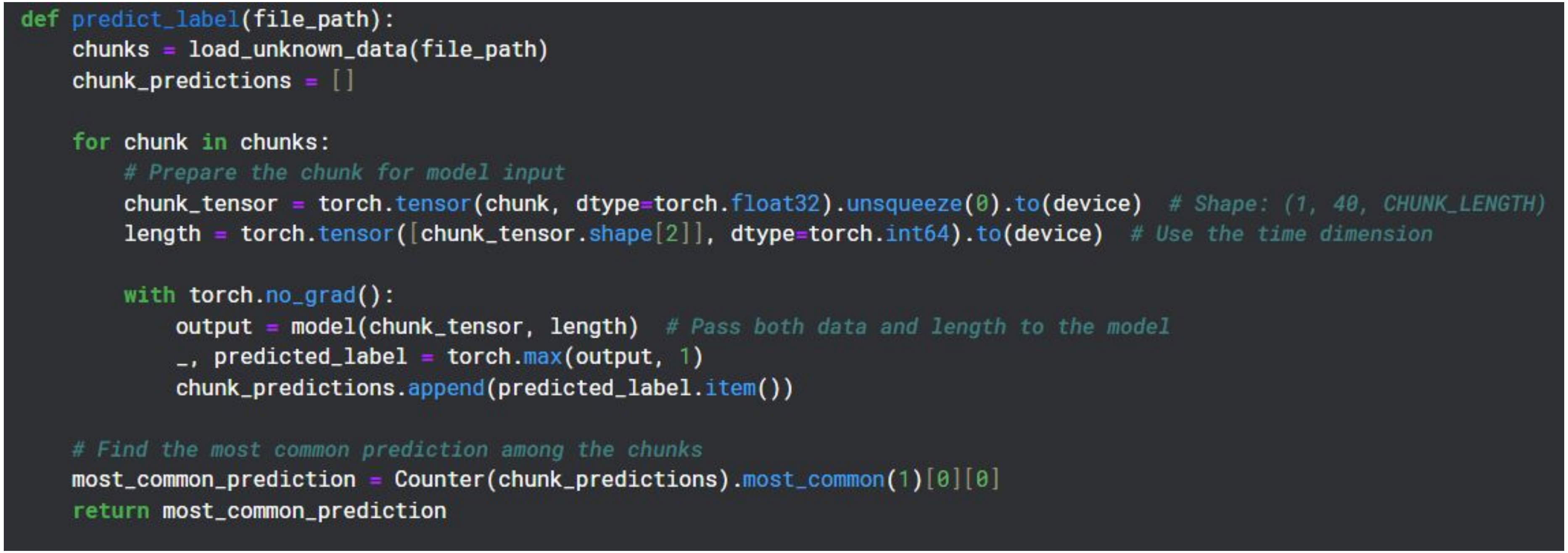
- We define a function that loads and processes MFCC data by appending delta MFCC features and splitting the data; just like we did for the train dataset
- We load the MFCC data, check the number of coefficients, verifying it is 20
- Then the first derivative is calculated, which captures the rate of change of values, telling us how the features evolve over time
- The original MFCC and delta MFCC are then concatenated
- The matrix is split into multiple chunks, each of equal width as above
- Finally, the function returns a list of arrays, each representing a chunk of data, along with the delta values



```python
CHUNK_LENGTH = 500
# Function to load and split data into chunks of a specified length, with delta MFCC appended
def load_unknown_data(file_path):
    # Load MFCC data from the CSV
    mfcc = np.loadtxt(file_path, delimiter=',')

    # Ensure 20 coefficients
    if mfcc.shape[0] != 20:
        raise ValueError(f"Expected 20 MFCC coefficients, but got {mfcc.shape[0]}.")

    # Calculate delta MFCC
    delta_mfcc = librosa.feature.delta(mfcc)

    # Concatenate MFCC and delta MFCC along the first axis
    mfcc_with_delta = np.concatenate((mfcc, delta_mfcc), axis=0)

    # Calculate the total columns along the time dimension
    total_columns = mfcc_with_delta.shape[1]

    # Split into chunks of specified length
    chunks = []
    for start_idx in range(0, total_columns, CHUNK_LENGTH):
        end_idx = start_idx + CHUNK_LENGTH

        # Ignore the last chunk if it is shorter than CHUNK_LENGTH
        if end_idx > total_columns:
            break

        chunk = mfcc_with_delta[:, start_idx:end_idx]
        chunks.append(chunk)

    return chunks  # List of (40, CHUNK_LENGTH) arrays
```

Predicting labels: Predictor function

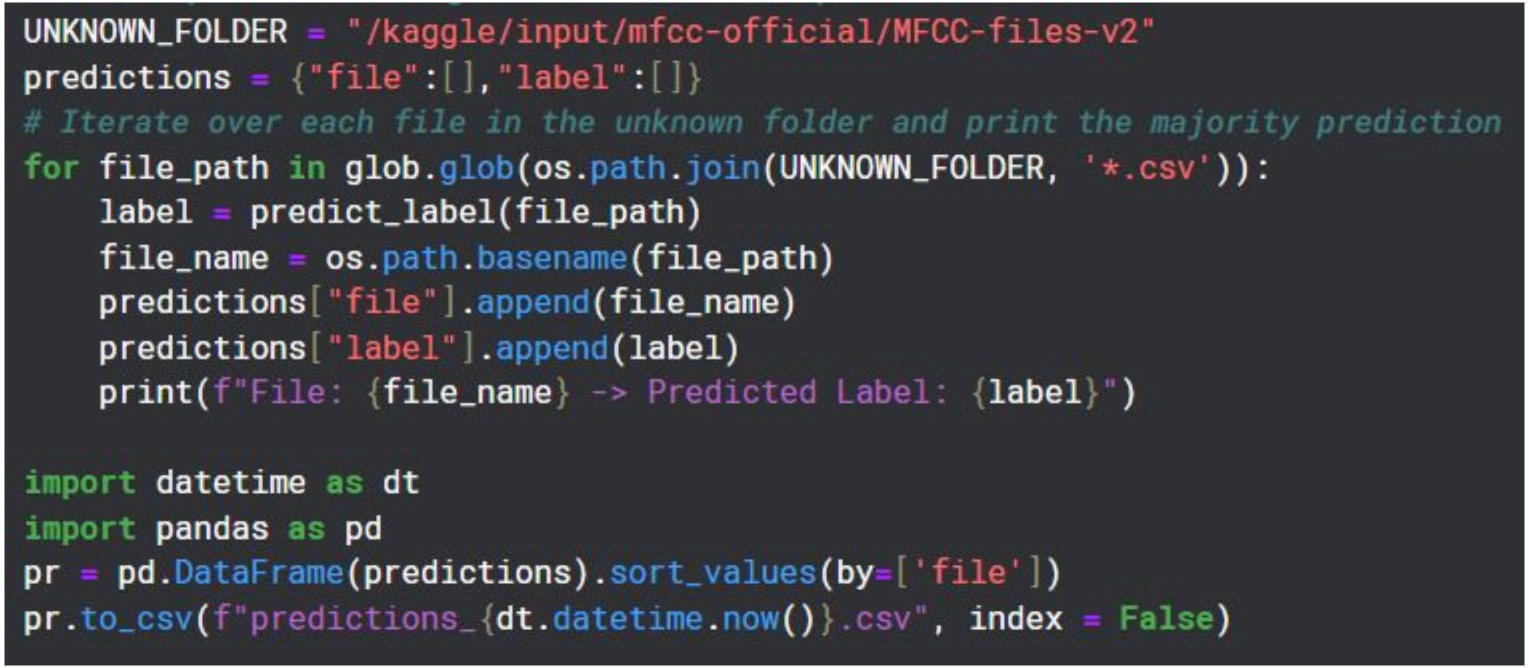
- The predict_label function makes a label prediction for the audio file
- It uses the defined load_unknown_data function, to load the data in chunks
- It loops through each chunk and converts it to a PyTorch tensor, with an added batch dimension
- This is then moved to the GPU, for faster prediction
- Then, the label for each chunk is predicted, by using torch.nograd(), which disables gradient calculation
- Then, the most common label for each file is calculated by taking the mode of each predicted label of each chunk



```python
def predict_label(file_path):
    chunks = load_unknown_data(file_path)
    chunk_predictions = []

    for chunk in chunks:
        # Prepare the chunk for model input
        chunk_tensor = torch.tensor(chunk, dtype=torch.float32).unsqueeze(0).to(device)  # Shape: (1, 40, CHUNK_LENGTH)
        length = torch.tensor([chunk_tensor.shape[2]], dtype=torch.int64).to(device)  # Use the time dimension

        with torch.no_grad():
            output = model(chunk_tensor, length)  # Pass both data and length to the model
            _, predicted_label = torch.max(output, 1)
            chunk_predictions.append(predicted_label.item())

    # Find the most common prediction among the chunks
    most_common_prediction = Counter(chunk_predictions).most_common(1)[0][0]
    return most_common_prediction
```

Predicting labels: Printing and saving to CSV

- We now call the defined functions to make our final predictions
- Using glob, we find all csv files in the uploaded folder (the 116 provided MFCC files)
- For each file, predict_label is called to predict the label, and then the filename and the label are extracted and appended
- After printing this data, we convert the dictionary to a pandas DataFrame, and save it to a CSV file with a timestamped filename



```python
UNKNOWN_FOLDER = "/kaggle/input/mfcc-official/MFCC-files-v2"
predictions = {"file":[],"label":[]}
# Iterate over each file in the unknown folder and print the majority prediction
for file_path in glob.glob(os.path.join(UNKNOWN_FOLDER, '*.csv')):
    label = predict_label(file_path)
    file_name = os.path.basename(file_path)
    predictions["file"].append(file_name)
    predictions["label"].append(label)
    print(f"File: {file_name} -> Predicted Label: {label}")

import datetime as dt
import pandas as pd
pr = pd.DataFrame(predictions).sort_values(by=['file'])
pr.to_csv(f"predictions_{dt.datetime.now()}.csv", index = False)
```

The link to the predictions.csv file:
https://iitbacin-my.sharepoint.com/:x:/g/personal/gokul_ramanan_iitb_ac_in/EdKAAV_kuydCmDFUi5V6xikBWpC_4FkfXdLm7YEPoeC7YQ?e=Z5tdbL

Results over testing data (116 songs)

The predictions.csv made by the model is uploaded here: [link]

We believe that we have correctly identified most of 116 songs to the correct genre.

Labels (0 - 5)	Asha Bhosle (0)	Jana Gana Mana (1)	Kishore Kumar (2)	Marathi Bhavgeet (3)	Marathi lavni (4)	Michael Jackson (5)
Total songs among the 116 titles	29	16	20	24	16	11
Correct model predictions	11	9	15	5	14	4
Accuracy	37.03%	56.25%	75%	20.08%	87.5%	36.36%

Results (contd.)

Labels	Asha Bhosle	Jana Gana Mana	Kishore Kumar	Marathi Bhavgeet	Marathi Lavni	Michael Jackson
Songs identified (filename)	04-MFCC 15-MFCC 23-MFCC 24-MFCC 54-MFCC 82-MFCC 102-MFCC 105-MFCC 106-MFCC 110-MFCC 115-MFCC	01-MFCC 02-MFCC 16-MFCC 17-MFCC 27-MFCC 35-MFCC 81-MFCC 90-MFCC 95-MFCC	05-MFCC 09-MFCC 18-MFCC 46-MFCC 50-MFCC 59-MFCC 63-MFCC 83-MFCC 84-MFCC 93-MFCC 96-MFCC 97-MFCC 100-MFCC 111-MFCC	37-MFCC 41-MFCC 72-MFCC 92-MFCC 104-MFCC	07-MFCC 19-MFCC 25-MFCC 30-MFCC 47-MFCC 62-MFCC 64-MFCC 70-MFCC 76-MFCC 85-MFCC 89-MFCC 94-MFCC 101-MFCC 109-MFCC	08-MFCC 20-MFCC 103-MFCC 114-MFCC

How the model performed

- Initially we expected the model to be quite good at predicting the National Anthem and Michael Jackson, due to their high variance from the other categories viz. Length, and nature of the song
- By contrast, upon observing the predictions.csv, we found out that the model was in fact extremely good at predicting the other genres, while being fairly mediocre at the other two
- It was able to accurately predict nearly all of the Marathi Lavni and Kishore Kumar songs
- Overall, the model was able to accurately predict 54/116 songs, giving an accuracy of nearly 50%

Conclusion: Results

We successfully achieved the objective of recognizing at least 3 songs from each of the genres given.

Learnings and experiences

- Learned how to effectively use CNN-RNN neural networks, and with with big datasets
- Initially, it was quite confusing as to how to proceed, how would we classify songs of variable length, singers, and size?
- Even if we were able to somehow cluster them together, how would we know which songs were which?
- We started reverse engineering each song and trying to id
- We then got the idea of training the model on externally downloaded songs
- Credits to Arya's mother for identifying a bulk of these songs with the singer and song name :-)
- Michael Jackson gained 4 new fans for life!