Song Feature Prediction

Nafis Abeer
Erfan Khalaj
Risheet Nair
Bahar Rezaei

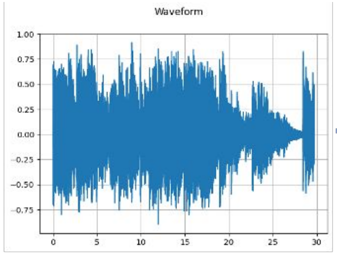
Task at Hand

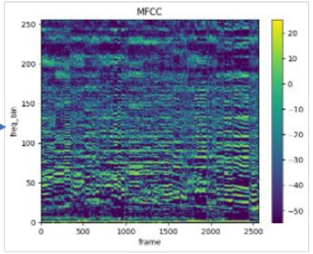
- Spotify defines various features to be used for song recommendations
    - [Get Track's Audio Features - Web API]
    - We hypothesize that these features are compiled by directly analyzing the audio
- We are specifically interested in the following features
    - Danceability
    - Acousticness
    - Instrumentalness
    - Speechiness
    - Energy

Related Work

- Many available projects predicting the Popularity metric of song
  - Use case would be to then create songs with features that correlate with high popularity
  - Using Deep Learning to Predict Hip-Hop Popularity on Spotify | by Nicholas Indorf | Towards Data Science
- Pytorch provides sufficient tools for audio processing
  - Audio Feature Extractions — Torchaudio 2.0.1 documentation
- Previous work on hit song prediction uses both high and low level features
  - High level features: release date, genre, valence, tempo
  - Low level features: MFCCs, temporal features from spectrograms
  - HIT SONG PREDICTION: LEVERAGING LOW- AND HIGH-LEVEL AUDIO FEATURES

Approach





MFCC tensor

RNN Classification

Fully Connected Layer

```python
track_df = pd.DataFrame({
    'track_id': [track_id],
    'track_name': [track_name],
    'track_artist': [track_artist],
    'danceability': [audio_features_data['danceability']],
    'instrumentalness': [audio_features_data['instrumentalness']],
    'speechiness': [audio_features_data['speechiness']],
    'acousticness': [audio_features_data['acousticness']],
    'energy': [audio_features_data['energy']],
    'valence': [audio_features_data['valence']],
    'tempo': [audio_features_data['tempo']],
    'loudness': [audio_features_data['loudness']],
    'liveness': [audio_features_data['liveness']],
    'key': [audio_features_data['key']],
    'mode': [audio_features_data['mode']],
    'time_signature': [audio_features_data['time_signature']],
    'preview_url' : [preview_url]
})
all_tracks_df = pd.concat([all_tracks_df, track_df], ignore_index=True)
```

Regression

Dataset

- Created a Spotify developers account, and acquired client ID and client secret

```python
client_credentials_manager = SpotifyClientCredentials(client_id=cid, client_secret=secret)
sp = spotipy.Spotify(client_credentials_manager=client_credentials_manager)
```

- Used spotify to search for tracks and get track features

```python
def search_tracks(query, limit, offset):
    results = sp.search(q=query, type='track', limit=limit, offset=offset)
    return results['tracks']['items']

def get_track_features(track_ids):
    features = sp.audio_features(track_ids)
    return features
```

- Grabbed desired features for 20000 songs across various genres

```python
genres = ['pop', 'rock', 'hip-hop', 'classical', 'jazz', 'country', 'electronic', 'reggae']
total_tracks_per_genre = 20000 // len(genres)
tracks_df_full = pd.DataFrame()

for genre in genres:
    query = f"genre:{genre}"
    genre_tracks_df = compile_tracks(query, total_tracks_per_genre)
    # tracks_df = tracks_df.append(genre_tracks_df, ignore_index=True)
    tracks_df_full = pd.concat([tracks_df_full, genre_tracks_df], ignore_index=True)
    print(len(tracks_df_full))

tracks_df_full.to_csv('tracks_features.csv', index=False)
```

Data Preparation

1. Grab preview URLs from spotify for 20000 songs, and save the audio
2. Split audio waveforms up into 120 frames and grab 20 MFCC coefficients for each frame

```
n_fft = 4096
win_length = None
hop_length = 8192
n_mels = 256
n_mfcc = 20
```

Notes about choices for MFCC transform:

- n_fft: number of points used for fast fourier transform
- hop_length: samples between frames
- n_mels: resolution of mel-spectrograms

```
for index, row in tqdm(all_tracks.iterrows(), total=all_tracks.shape[0]):
  track_id = row['id']
  filepath = name_of_file(track_id)
  WAVEFORM, SAMPLE_RATE = torchaudio.load(filepath)
  waveform_mono = torch.mean(WAVEFORM, dim=0, keepdim=True).cuda()
  mfcc = mfcc_transform(waveform_mono)

  # Pad the MFCC tensor to shape [1, 20, 160]
  mfcc = pad_tensor(mfcc, 160)
```

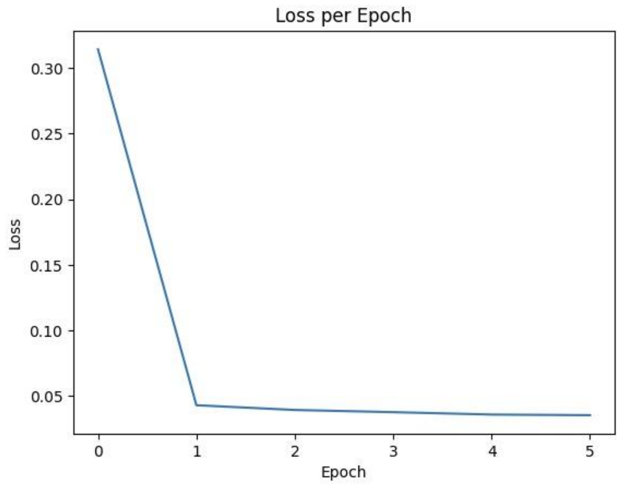
```
mfcc_transform = T.MFCC(
  sample_rate=SAMPLE_RATE,
  n_mfcc=n_mfcc,
  melkwargs={
      "n_fft": n_fft,
      "n_mels": n_mels,
      "hop_length": hop_length,
      "mel_scale": "htk",
    },
).cuda()  # Move the transform to the GPU
```

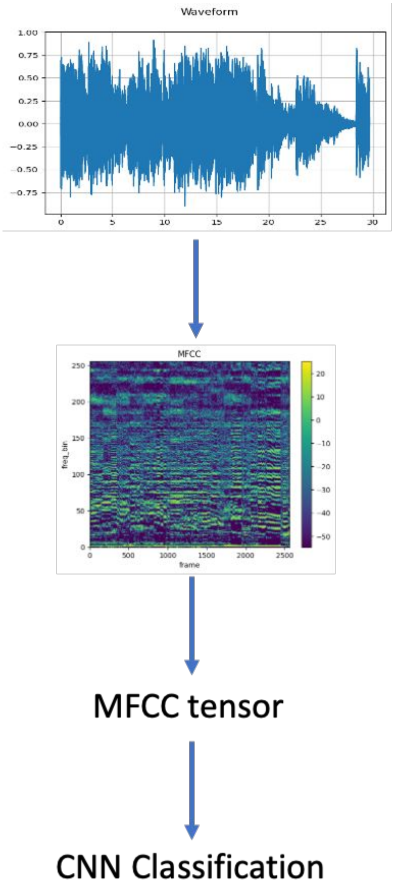
Complex CNN model

```python
class Net(nn.Module):
    def __init__(self, n_classes):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.conv3 = nn.Conv2d(64, 128, 3, 1)
        self.conv4 = nn.Conv2d(128, 256, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.pooling = nn.AdaptiveAvgPool2d((8, 8))
        self.fc1 = nn.Linear(16384, 128)
        self.fc2 = nn.Linear(128, n_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)

        x = self.conv3(x)
        x = F.relu(x)
        x = self.conv4(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = self.pooling(x)

        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        return x
```

Loss function used: nn.MSELoss()

Epochs trained: 6



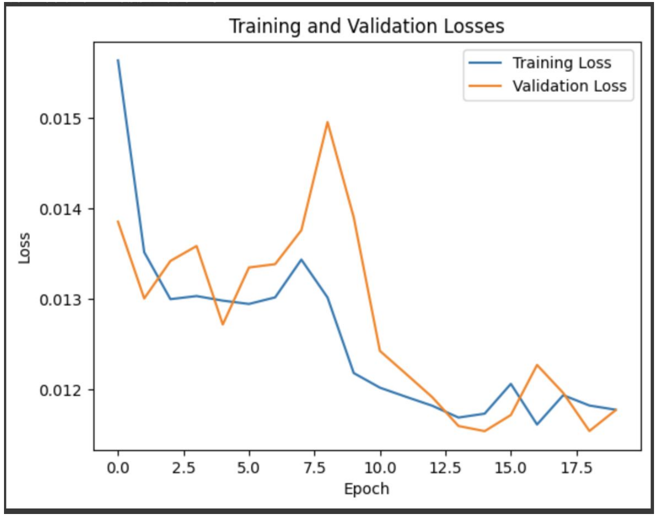


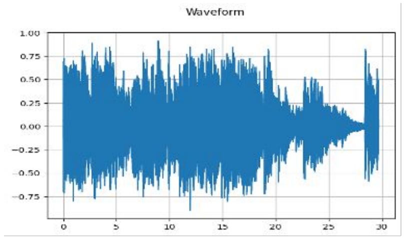
MultiTask RNN Model

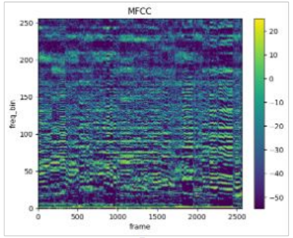
Loss function used: nn.SmoothL1Loss()

Epochs trained: 20

```python
class MultiTaskRNNModel(nn.Module):
  def __init__(self, input_size=20, hidden_size=64, num_layers=1, num_outputs=5):
    super(MultiTaskRNNModel, self).__init__()
    self.hidden_size = hidden_size
    self.num_layers = num_layers
    self.num_outputs = num_outputs
    self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
    self.fc = nn.Linear(hidden_size, num_outputs)
    self.sigmoid = nn.Sigmoid()

  def forward(self, x):
    h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
    c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
    out, _ = self.lstm(x, (h0, c0))
    out = self.fc(out[:, -1, :])
    out = self.sigmoid(out)

    # Multiply by 1000, round, and then divide by 1000 for precision purposes
    #out = torch.round(out * 1000) / 1000

    return out
```







MFCC tensor

RNN Classification

Multitask Regression Model

Architecture choice: 3 Linear layers

Loss function choice: MSE

Epochs trained for: 100

```
track_df = pd.DataFrame({
    'track_id': [track_id],
    'track_name': [track_name],
    'track_artist': [track_artist],
    'danceability': [audio_features_data['danceability']],
    'instrumentalness': [audio_features_data['instrumentalness']],
    'speechiness': [audio_features_data['speechiness']],
    'acousticness': [audio_features_data['acousticness']],
    'energy': [audio_features_data['energy']],
    'valence': [audio_features_data['valence']],
    'tempo': [audio_features_data['tempo']],
    'loudness': [audio_features_data['loudness']],
    'liveness': [audio_features_data['liveness']],
    'key': [audio_features_data['key']],
    'mode': [audio_features_data['mode']],
    'time_signature': [audio_features_data['time_signature']],
    'preview_url' : [preview_url]
})
all_tracks_df = pd.concat([all_tracks_df, track_df], ignore_index=True)
```

Regression

```
df = pd.read_csv('tracks_features.csv')
x = df[['valence', 'tempo', 'loudness', 'key', 'mode', 'time_signature']].values
y = df[['danceability', 'energy', 'speechiness', 'acousticness', 'instrumentalness']].values

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
x_train_scaled = scaler.fit_transform(x_train)
x_test_scaled = scaler.transform(x_test)

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.layer1 = nn.Linear(6, 64)
        self.layer2 = nn.Linear(64, 32)
        self.layer3 = nn.Linear(32, 5)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        x = self.layer3(x)
        return x

model = Model()

criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters())

train_dataset = TracksDataset(x_train_scaled, y_train)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_dataset = TracksDataset(x_test_scaled, y_test)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Linear Model

Architecture: Fully connected layer connecting outputs of MFCC and regression models

Loss function used: nn.MSELoss()

Epochs trained: 6

```
class CombinerModel(nn.Module):
    def __init__(self, input_size=20, hidden_size=64, num_layers=1, num_outputs=5):
        super(CombinerModel, self).__init__()
        self.fc_rnn = nn.Linear(64, 32)
        self.fc_lm = nn.Linear(32, 32)
        self.fc_combiner = nn.Linear(64, 5)

    def forward(self, x):
        x_rnn = x[:, 0:20*160]
        x_lm = x[:, 20*160:20*160+6]
        batch_size = x.shape[0]
        x_rnn = torch.reshape(x_rnn, (batch_size, 160, 20))

        out_rnn = trained_mt_rnn(x_rnn)
        out_rnn = torch.relu(self.fc_rnn(out_rnn))

        out_lm = trained_lm(x_lm)
        out_lm = torch.relu(self.fc_lm(out_lm))

        out = torch.cat((out_rnn, out_lm), 1)
        out = self.fc_combiner(out)
        out = torch.sigmoid(out)

        return out
combiner = CombinerModel().cuda()
```

Evaluation Metrics

- Mean Squared Error(MSE): can be sensitive to outliers
- Mean Absolute Error(MAE): directly quantifies the average magnitude of errors
- Root Mean Squared Error(RMSE): useful for understanding the average magnitude of the errors and penalizes larger errors more heavily
- R-squared: commonly used to evaluate the goodness-of-fit of a model
- Pearson's Correlation Coefficient:measures the linear relationship between two variables, with values ranging from -1 (perfect negative correlation) to 1 (perfect positive correlation)
  - Higher values for this means when the actual labels are of high magnitude, so are model outputs and vice versa

Results

Model	Danceability (MSE/MAE)	Energy (MSE/MAE)	Speechiness (MSE/MAE)	Acousticness (MSE/MAE)	Instrumentalness (MSE/MAE)	Average (MSE/MAE)
Multi-task Regression	0.016/0.102	0.045/0.157	0.071/0.175	0.008/0.059	0.014/0.089	0.031/0.116
Multi-task 1D RNN	0.019/0.111	0.018/0.107	0.007/0.054	0.024/0.113	0.052/0.141	0.024/0.105
Complex CNN	X	X	X	X	X	0.026/0.111
Simple 2D CNN	0.001/0.080	X	X	X	X	X
Simple 1D RNN	0.010/0.079	X	X	X	X	X
Linear Combiner	0.022/0.117	0.030/0.134	0.008/0.058	0.066/0.196	0.082/0.200	0.042/0.141

Conclusion

Things that worked:

Both numerical based model as well as the MFCC based models, along with their variations proved to be capable of learning how to predict the song features within a small window of error.

Things we tried but did not work:

We tried transfer learning (ResNet18)

- Shapes of our tensors didn’t match with what ResNet18 was trained with