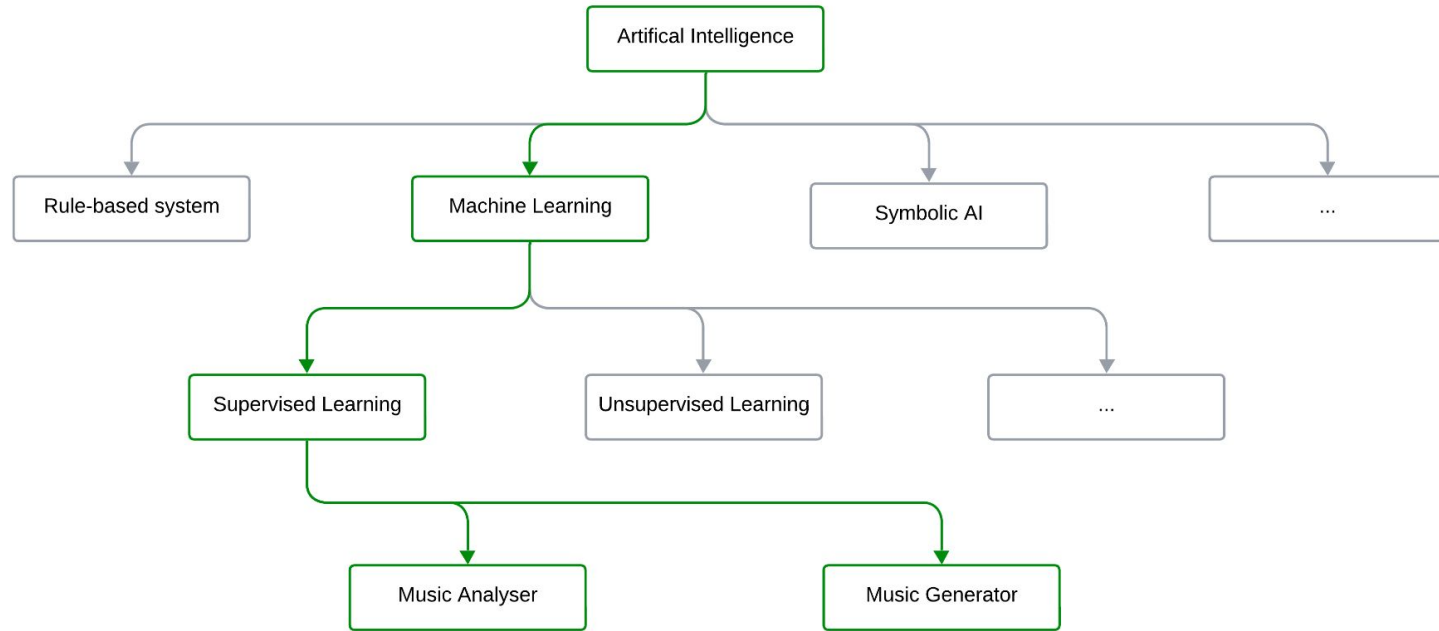


Emotions in AI Music Composition

Angelo Antona

Introduction: Choice of AI Approach



System Configuration

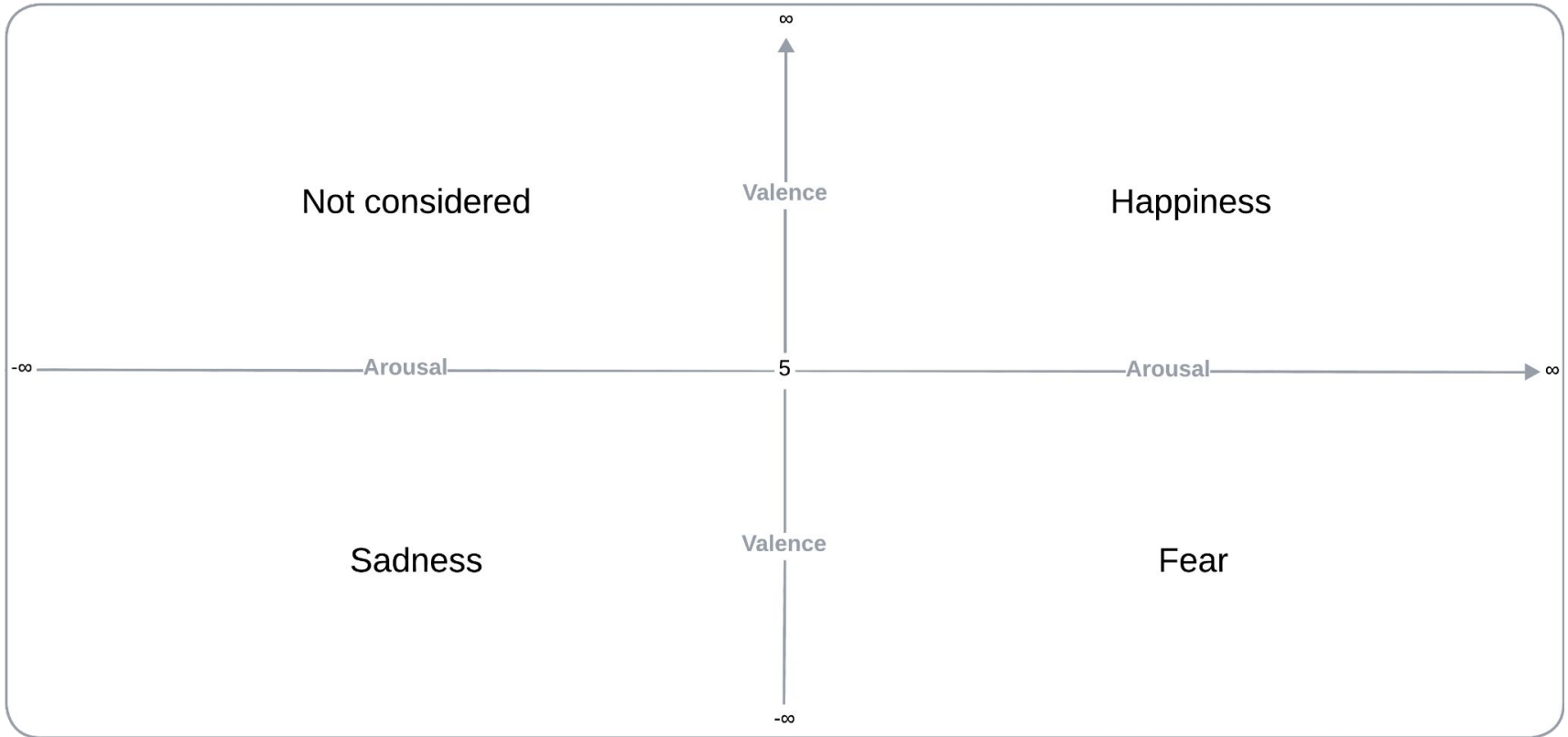
The most important libraries I used are:

- NumPy
- Pandas
- Scikit-Learn
- TensorFlow
- Librosa
- Music21

The dataset I used in the scripts are:

- DEAM
- EMOPIA

Dataset Feature Interpretation

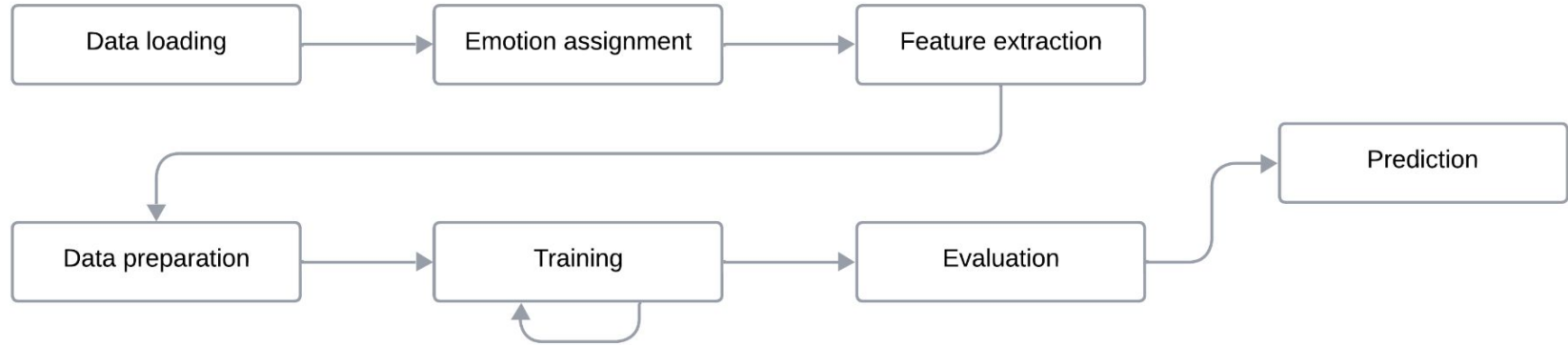


CODE1: Song Emotion Analysis

The goal is to create and train a model that recognizes the primary type of emotion a given song conveys to the listener.



Script Execution Algorithm



Emotion Assignment

We apply this function:

```
def assign_emotion(row):  
    valence = row[' valence_mean']  
    arousal = row[' arousal_mean']  
    if valence > 5 and arousal > 5:  
        return 'happiness'  
    elif valence <= 5 and arousal <= 5:  
        return 'sadness'  
    elif valence <= 5 and arousal > 5:  
        return 'fear'  
    else:  
        return None # Esclude altri stati emotivi
```



to each song in dataset, getting the respective emotion. At the end, we obtain something like:

	song_id	valence_mean	valence_std	arousal_mean	arousal_std	emotion
0	2	3.1	0.94	3.0	0.63	sadness
3	5	4.4	2.01	5.3	1.85	fear
4	7	5.8	1.47	6.4	1.69	happiness



Features Extraction

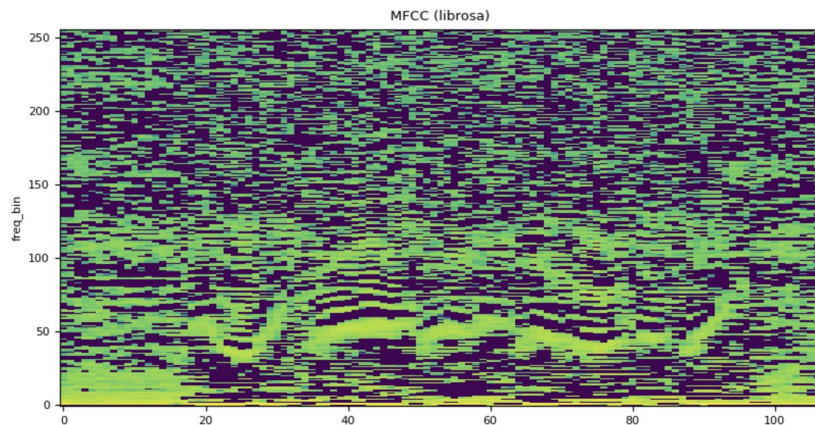
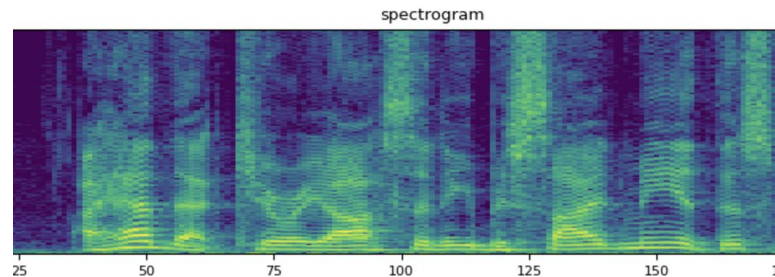
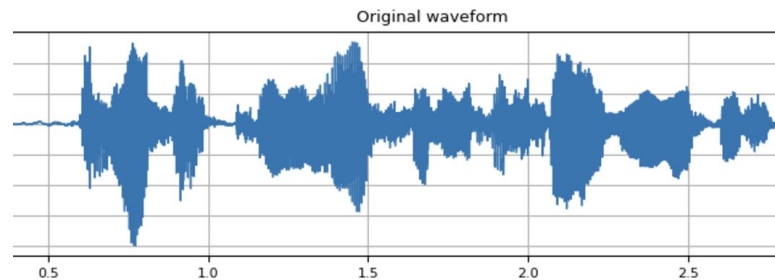
```
def extract_features(file_name):  
    try:  
        audio_data, sample_rate = librosa.load(file_name, res_type='kaiser_fast')  
        # Estrazione delle MFCC  
        mfccs = librosa.feature.mfcc(y=audio_data, sr=sample_rate, n_mfcc=40)  
        mfccs_scaled = np.mean(mfccs.T, axis=0)  
        return mfccs_scaled  
    except Exception as e:  
        print(f"Errore nell'elaborazione del file {file_name}: {e}")  
        return None
```



Features Extraction

Mel-Frequency Cepstrum Coefficients are a representation of the short-term power spectrum.

1. Take FFT.
2. Map the power of the spectrum onto the mel scale using triangle overlapping windows.
3. Take the logs of the powers at each of the mel frequencies, simulating human ear.
4. Take the discrete cosine transform of the list of mel log powers, as if it were a signal, reducing data dimensionality.
5. The MFCCs are the amplitudes of the resulting spectrum.



Data Preparation

We convert the audio features and labels into DataFrame format to be suitable for the NN:

```
features_df = pd.DataFrame(features, columns=['feature', 'label'])
```



We derive **X** and **yy**:

```
# Otteniamo X
X = np.array(features_df['feature'].tolist())
# Otteniamo yy
y = np.array(features_df['label'].tolist())
le = LabelEncoder()
yy = to_categorical(le.fit_transform(y))
```



We partition the elements to obtain a **train set** and a **test set**:

```
x_train, x_test, y_train, y_test = train_test_split(X, yy, test_size=0.2, random_state=42)
```



Neural Network

I used a feed-forward sequential Neural Network, whose architecture is:


Layer (type)	Output Shape	Param #
dense (Dense)	(None, 256)	10,496
activation (Activation)	(None, 256)	0
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 128)	32,896
activation_1 (Activation)	(None, 128)	0
dropout_1 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 3)	387
activation_2 (Activation)	(None, 3)	0



The input layer of this network is the 1D Vector of the 40 MFCC features extracted from each audio files.

Training and Test of the Model

The model is trained using the .fit method:

```
checkpointer = ModelCheckpoint(filepath='saved_models/audio_classification.keras', verbose=1,   
history = model.fit(x_train, y_train, batch_size=32, epochs=20, validation_data=(x_test, y_test)
```

At the end of the training, we use the test-set:

```
model.load_weights('saved_models/audio_classification.keras')   
score = model.evaluate(x_test, y_test, verbose=0)
```

The obtained result is:

```
Accuracy sul test set: 70.49% 
```

Prediction

From the initial song dataset, I removed the samples 3.mp3 and 4.mp3 and placed these samples in a separate test folder, apart from the other dataset elements.

I used these two songs to test if the model's prediction matches the emotion I perceived while listening to these two songs.



4.mp3



3.mp3

The terminal output is:

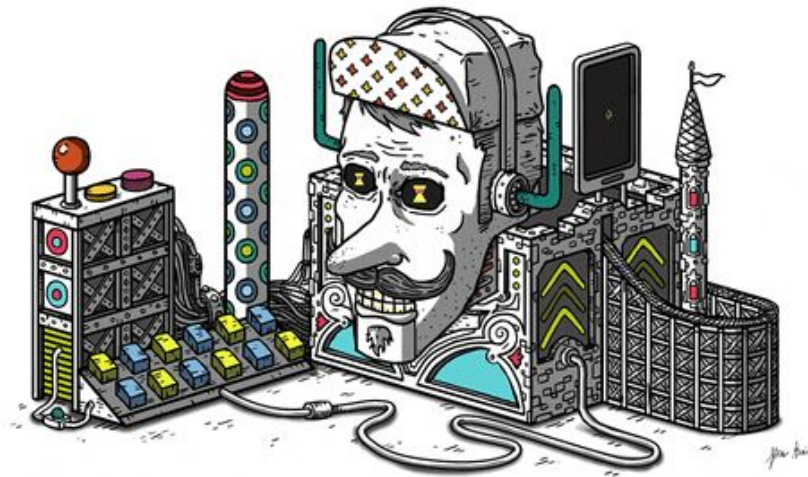
```
L'emozione predetta per 'test/4.mp3' è: happiness
```

```
L'emozione predetta per 'test/3.mp3' è: sadness
```

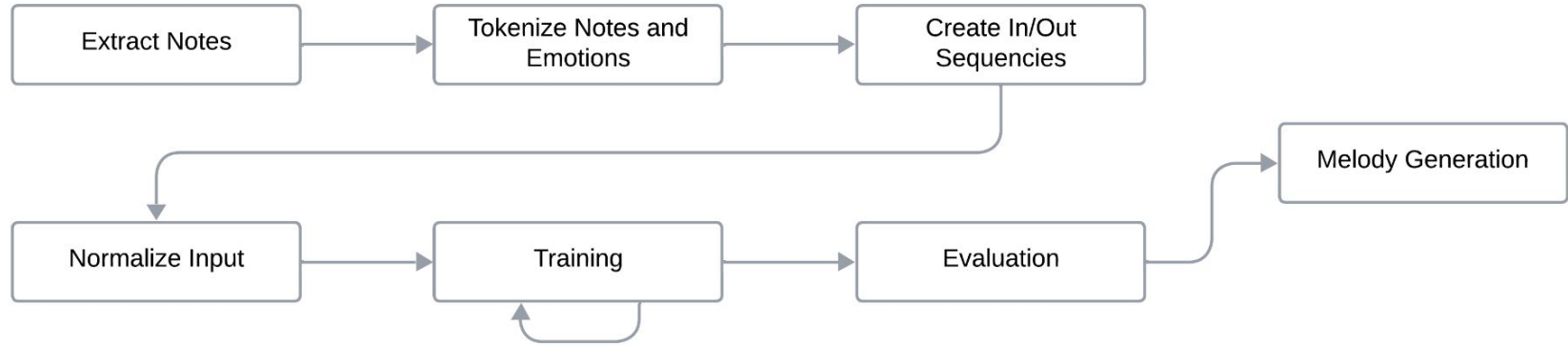


CODE2: Song Emotions Generation

The goal is to create a system that can produce emotionally expressive music by analyzing patterns in the dataset.



Script Execution Algorithm



Notes Extraction

I defined a function to extract the notes from the MIDI files located in data_midi/<emotion>/.

```
def get_notes(emotion):
    notes = []
    files = glob.glob(f"data_midi/{emotion}/*.mid")
    for file in tqdm(files, desc=f"Processing {emotion} files"):
        midi = converter.parse(file)

        try:
            s2 = instrument.partitionByInstrument(midi)
            notes_to_parse = s2.parts[0].recurse()
        except:
            notes_to_parse = midi.flat.notes

        for element in notes_to_parse:
            if isinstance(element, note.Note):
                notes.append(str(element.pitch))
            elif isinstance(element, chord.Chord):
                if len(element.pitches) == 1:
                    notes.append(str(element.pitches[0]))
                else:
                    notes.append('.'.join(str(n) for n in element.normalOrder))

    return notes
```



Notes Extraction

At this point, we collect all the notes and emotions.

```
def get_all_notes_and_emotions():  
    all_notes = []  
    all_emotions = []  
    for emotion in EMOTIONS:  
        notes = get_notes(emotion)  
        all_notes.extend(notes)  
        all_emotions.extend([emotion]*len(notes))  
    return all_notes, all_emotions  
  
all_notes, all_emotions = get_all_notes_and_emotions()
```



Data Preparation

Tokenization of Notes and Emotions

```
# Tokenization of notes
unique_notes = sorted(set(all_notes))
note_to_int = {note: number for number, note in enumerate(unique_notes)}
notes_as_int = [note_to_int[note] for note in all_notes]

# Encoding of emotions
label_encoder = LabelEncoder()
encoded_emotions = label_encoder.fit_transform(all_emotions)
```



Creating Input and Output Sequences

```
network_input = []
network_output = []
network_emotion = []

for i in range(len(notes_as_int) - SEQUENCE_LENGTH):
    seq_in = notes_as_int[i:i + SEQUENCE_LENGTH]
    seq_out = notes_as_int[i + SEQUENCE_LENGTH]
    emotion = encoded_emotions[i + SEQUENCE_LENGTH]
    network_input.append(seq_in)
    network_output.append(seq_out)
    network_emotion.append(emotion)
```



Data Preparation

```
# Convert input to numpy arrays
network_input = np.array(network_input)
network_output = np.array(network_output)
network_emotion = np.array(network_emotion)

# Normalization of notes
n_vocab = len(unique_notes)
network_input = network_input / float(n_vocab)
network_input = network_input.reshape((network_input.shape[0], SEQUENCE_LENGTH, 1))

# Normalization and repetition of emotion
emotion_normalized = network_emotion / float(max(network_emotion))
emotion_input = emotion_normalized.reshape(-1, 1, 1)
emotion_input = np.repeat(emotion_input, SEQUENCE_LENGTH, axis=1)

# Concatenation of notes and emotions as features
network_input = np.concatenate((network_input, emotion_input), axis=2)

# Conversion of output to categorical
network_output = to_categorical(network_output, num_classes=n_vocab)
```

The shape of the resulting structures are:

```
Input dimensions: (50000, 50, 2)
Output dimensions: (50000, 685)
```

Model Definition

Model: "functional"



Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 50, 2)	0
gru (GRU)	(None, 50, 256)	199,680
gru_1 (GRU)	(None, 256)	394,752
dropout (Dropout)	(None, 256)	0
dense (Dense)	(None, 128)	32,896
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 685)	88,365

Total params: 715,693 (2.73 MB)

Trainable params: 715,693 (2.73 MB)

Non-trainable params: 0 (0.00 B)

Training the NN

The model is trained using the `.fit` method:

```
# Training the model
history = model.fit(network_input, network_output, epochs=50, batch_size=32, callbacks=callbacks_list)
```

The obtained result is:

```
Epoch 1/50
1563/1563 [=====] - 233s 148ms/step - loss: 5.0591
Epoch 1: loss improved from inf to 5.0591, saving model to saved_models/audio_generation.keras
...
Epoch 50/50
1563/1563 [=====] - 243s 156ms/step - loss: 3.4806
Epoch 50: loss did not improve from 3.49500
```



Preparation for Melody Generation

```
int_to_note = {number: note for note, number in note_to_int.items()}
```



```
def sample_with_temperature(preds, temperature=1.0):  
    preds = np.asarray(preds).astype('float64')  
    if temperature == 0:  
        temperature = 1e-10  
    preds = np.log(preds + 1e-10) / temperature  
    exp_preds = np.exp(preds)  
    preds = exp_preds / np.sum(exp_preds)  
    probas = preds  
    return np.random.choice(len(probas), p=probas)
```

Preparation for Melody Generation

```
def generate_notes_by_emotion(model, network_input, int_to_note, n_vocab, desired_emotion, num_notes=100, temperature=0.7):  
    # Encode and normalize the desired emotion  
    emotion_encoded = label_encoder.transform([desired_emotion])[0]  
    emotion_normalized = emotion_encoded / float(max(label_encoder.transform(label_encoder.classes_)))  
  
    # Choose a random starting point  
    start = np.random.randint(0, len(network_input)-1)  
    pattern = network_input[start]  
    pattern = pattern.reshape(1, SEQUENCE_LENGTH, 2)  
  
    # Set the desired emotion in the pattern  
    pattern[0, :, 1] = emotion_normalized  
  
    prediction_output = []  
  
    for note_index in range(num_notes):  
        prediction = model.predict(pattern, verbose=0)  
        # Use temperature sampling  
        index = sample_with_temperature(prediction[0], temperature)  
        result = int_to_note[index]  
        prediction_output.append(result)  
  
        # Create new input  
        new_note = index / float(n_vocab)  
        new_input = np.array([[new_note, emotion_normalized]])  
        pattern = np.concatenate((pattern[:, 1:, :], new_input.reshape(1, 1, 2)), axis=1)  
    return prediction_output
```

Preparation for Melody Generation

```
def create_midi(prediction_output, output_filename='output.mid'):
    offset = 0
    output_notes = []

    for pattern in prediction_output:
        # If the pattern is a chord
        if ('.' in pattern) or pattern.isdigit():
            notes_in_chord = pattern.split('.')
            notes = []
            for current_note in notes_in_chord:
                try:
                    new_note = note.Note(int(current_note))
                except:
                    new_note = note.Note(current_note)
                new_note.storedInstrument = instrument.Piano()
                notes.append(new_note)
            new_chord = chord.Chord(notes)
            new_chord.offset = offset
            output_notes.append(new_chord)
        else:
            # If the pattern is a single note
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            output_notes.append(new_note)
        # Increase offset to prevent notes from overlapping
        offset += 0.5

    midi_stream = stream.Stream(output_notes)
    midi_stream.write('midi', fp=output_filename)
    print(f"Melody generated and saved to {output_filename}")
```



Melody Generation

```
# Select the desired emotion
desired_emotion = 'sadness' # Can be 'happiness', 'sadness', or 'fear'

# Generate the sequence
prediction_output = generate_notes_by_emotion(
    model,
    network_input,
    int_to_note,
    n_vocab,
    desired_emotion,
    num_notes=200, # Number of notes to generate
    temperature=0.8 # Temperature value for sampling
)

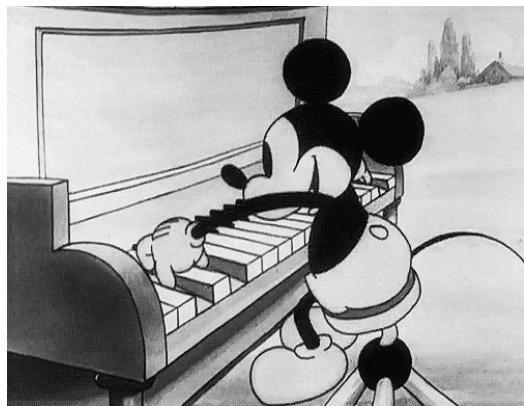
# Create the MIDI file
output_filename = os.path.join(RESULTS_DIR, f'melody_{desired_emotion}.mid')
create_midi(prediction_output, output_filename)
```



Resulting melodies



melody_happiness.mid



melody_sadness.mid



melody_fear.mid

