Unsupervised Phoneme Discovery Using Markov Models

Programming and Synthesis 2

March 2025

This document introduces the process of unsupervised phoneme discovery using Markov Models and clustering techniques. It is designed for Master's students in Electronic Music Composition as part of the "Programming and Synthesis 2" course. The focus is on exploring creative applications for audio synthesis, machine learning, and experimental sound design.

1 Introduction

In experimental music and sound design, machine learning can provide tools for discovering patterns in sound. One such application is the **unsupervised discovery of phonemes**, the fundamental sound units of speech. This text introduces a method using **Markov Models and clustering** to segment and analyze phoneme-like structures in an unknown language, useful for both linguistic exploration and generative sound synthesis.

1.1 Creative Use Cases

- **Algorithmic Composition:** Extract phoneme-like structures and use them the synthesis of new sounds.
- **Timbre Morphing:** Apply discovered phoneme transitions to resynthesize vocal-like textures. Or if not dealing with speech audio, to resynthesize audio that mimics the structures of the input.
- **Live Performance Tools:** Build an interactive system where Markov-generated sounds influence synthesis in real time.
- **Sound Sculpting:** Use phoneme sequences as control signals for granular-like synthesis.
- **Other?:** Discuss your ideas for the creative use or misuse of this technique.

2 Conceptual Overview

The process of **unsupervised phoneme discovery** is an attempt to extract the fundamental building blocks of speech without any prior linguistic knowledge. Instead of predefined phoneme categories, we use machine learning techniques to identify patterns in the sound and represent them as probabilistic states in a Markov Model.

How to use!!: you will use this python script, create its necessary virtual environment to install required libraries and make sure you have a *train_speech.wav* and *test_speech.wav* audio files that you recorded previously, in the same folder where the script is. These audio files need to be at 16 kHz sampling rate.

The steps of our code are as follows:

- **Audio Framing:** Input is a continuous waveform. To analyze it meaningfully, we divide it into short, overlapping frames. Each frame is a snapshot of the sound over a short period (e.g., 25ms).
- **Spectral Feature Extraction**: The Fast Fourier Transform (FFT) is applied to each frame to extract spectral information, giving us a frequency-domain representation of each frame.
- **Clustering with K-means:** We use K-means clustering to group frames with similar spectral features into discrete categories, effectively forming a "codebook" of fundamental speech components. In our case we are arbitrarily choosing the number of clusters k to be 50.
- **Markov State Assignment:** Each frame is assigned to the nearest cluster center, forming a sequence of states that represent the speech.
- **Markov Transition Modeling:** By analyzing how states(clusters) transition from one to another, we compute a Markov Transition Matrix that captures the statistical relationships between different phoneme-like units.
- **Markov-Based Synthesis:** Using the learned transition probabilities, we generate new speech-like sounds by probabilistically selecting frames based on their transitions.

This approach is powerful because it does not require labeled phoneme data. Instead, it lets the structure of speech emerge naturally, making it useful for both scientific analysis and artistic exploration.

3 Implementation

3.1 Main Function: Running the Full Pipeline

The following function integrates all the steps into a complete workflow:

```
def main_pipeline(train_audio, test_audio, k_clusters=50):
    """Full pipeline: From audio to Markov matrix and synthesis
    using k-means clustering."""

    # 1. Frame the training audio and extract FFT features
    train_frames, sr = frame_audio(train_audio)
    train_features = compute_fft(train_frames)

    # 2. Create the codebook using K-means.
    kmeans_model = build_codebook(train_features, k=k_clusters)

    # 3. Process the test audio and assign states
    test_frames, _ = frame_audio(test_audio)
    test_features = compute_fft(test_frames)

    assigned_states = assign_to_codebook(test_features,
    kmeans_model)

    # 4. Compute and visualize the Markov transition matrix
    markov_matrix = compute_markov_matrix(assigned_states,
    k_clusters)
    visualize_markov_matrix(markov_matrix)

    # 5. Display sample state sequence and matrix
    print(f"\nSample State Sequence: {assigned_states[:20]}")
    print(f"\nMarkov Transition Matrix (shape {markov_matrix.shape
    }):\n", markov_matrix)

    # 6. Synthesize audio based on the Markov model
    synthesized_audio = synthesize_audio(test_frames,
    assigned_states, markov_matrix, length=2000)
    save_synthesized_audio(synthesized_audio, sr)
```

Listing 1: Main pipeline for phoneme discovery and synthesis.

Explanation: The function executes the pipeline in a structured manner:

1. It extracts **FFT features** from a reference (training) audio file.
2. It uses **K-means clustering** to create a set of representative sound units with similar spectral features (categories). This is the **codebook**
3. It processes a separate (test) audio file, **assigning states** to each of its **framed snippets** based on the trained **codebook**.
4. It computes the **Markov Transition Matrix**, which models how these states transition over time.
5. It generates a **new audio sequence** by stochastically (probabilistically) selecting frames based on the learned transitions.

3.2 Step 1: Frame the Audio

```python
import librosa

def frame_audio(audio_file, frame_size_ms=25, hop_size_ms=10, sr=16000):
    """Frame the audio file into overlapping frames."""
    y, sr = librosa.load(audio_file, sr=sr)
    frame_size = int(sr * frame_size_ms / 1000)
    hop_length = int(sr * hop_size_ms / 1000)
    frames = librosa.util.frame(y, frame_length=frame_size, hop_length=hop_length).T
    return frames, sr
```

Listing 2: Framing audio into overlapping segments.

3.3 Step 2: Compute FFT Features

```python
import numpy as np

def compute_fft(frames):
    """Compute the FFT magnitude spectrum for each frame."""
    fft_frames = np.abs(np.fft.rfft(frames, axis=1))
    return fft_frames
```

Listing 3: Computing FFT magnitude spectrum for each frame.

3.4 Step 3: Cluster FFT Features Using K-Means

```python
from sklearn.cluster import KMeans

def build_codebook(features, k=50):
    """Cluster the FFT features using K-means to create the codebook."""
    kmeans = KMeans(n_clusters=k, random_state=0, max_iter=300, n_init=10)
    kmeans.fit(features)
    return kmeans
```

Listing 4: Clustering FFT features using K-means.

3.5 Step 4: Compute Markov Transition Matrix

```python
def compute_markov_matrix(states, num_states):
    """Compute the Markov state transition matrix."""
    matrix = np.zeros((num_states, num_states))
    for (i, j) in zip(states[:-1], states[1:]):
        matrix[i, j] += 1
    matrix /= matrix.sum(axis=1, keepdims=True) + 1e-10  # Normalize
    return matrix
```

Listing 5: Computing Markov state transition probabilities.

3.6 Step 5: Generate Audio Using Markov Model

```python
def synthesize_audio(frames, assigned_states, markov_matrix, length=1000):
    """Synthesize audio by selecting frames based on Markov transitions."""
    synthesized_audio = []
    current_state = np.random.choice(np.arange(len(markov_matrix)))
    for _ in range(length):
        frame_indices = np.where(assigned_states == current_state)[0]
        if len(frame_indices) == 0:
            current_state = np.random.choice(np.arange(len(markov_matrix)))
            continue
        chosen_frame = frames[np.random.choice(frame_indices)]
        synthesized_audio.extend(chosen_frame)
        next_state_probs = markov_matrix[current_state]
        current_state = np.random.choice(np.arange(len(markov_matrix)), p=next_state_probs)
    return np.array(synthesized_audio)
```

Listing 6: Synthesizing new audio from Markov model.

3.7 Step 6: Visualize Markov Transition Matrix

```python
import matplotlib.pyplot as plt
import seaborn as sns

def visualize_markov_matrix(markov_matrix):
    plt.figure(figsize=(10, 8))
    sns.heatmap(markov_matrix, cmap="viridis", square=True)
    plt.title("Markov State Transition Matrix")
    plt.xlabel("Next State")
    plt.ylabel("Current State")
    plt.show()
```

Listing 7: Visualizing the Markov state transition matrix.

3.8 Step 8: Save Synthesized Audio

```python
import soundfile as sf

def save_synthesized_audio(audio_array, sr, output_path="synthesized_audio.wav"):
    sf.write(output_path, audio_array, sr)
    print(f"Synthesized audio saved to {output_path}")
```

Listing 8: Saving the synthesized audio to a file.

4 Future Challenges and Next Steps

This tutorial covers the fundamental steps for unsupervised phoneme discovery and Markov-based synthesis. However, there are several directions for further research and experimentation:

- **Try other sources and explore:** Instead of speech, try to use other sources of musical audio or any sound you can think of to understand what the script is doing.
- **Use different parameters:** Play with the parameters of the steps (size of snippets of audio, sampling rate, number of clusters, etc.)
- **Try Feedback:** One idea is to use the output audio as input for another run of the script. It should converge into digital noise.
- **Use other features:** Instead of computing FFTs, try with other methods, such as MEL Frequency Cepstral Coefficients (MFCC).
- **Improving Phoneme Separation:** Exploring different clustering methods such as Gaussian Mixture Models (GMMs) to achieve better phoneme separation.
- **Explore different parameters and windowing in re-synthesis:** In this example, we are concatenating snippets of audio without overlap. You could try to implement overlapping and windowing (i.e. Triangular, Hann, and other windows).
- **Explore options for [re]synthesis:** Right now we are simply concatenating snippets of audio. But audio could be re-synthesized using the spectrogram data directly. You could research how this is done.
- **Applying Deep Learning:** Using neural networks to refine phoneme representation and improve synthesis quality.
- **Enhancing Temporal Structure:** Investigating higher-order Markov models to capture longer phoneme transitions.
- **Implement HMM when phonemes = group of clusters:** As of now, we are identifying a phoneme with one of the clusters. But this means that a phoneme is assumed to be 25ms long. In relaity a phoneme is longer and made of several clusters put together. This is a different model that needs to find that relationship. It would use a training algorithm called Baum-Welch. This is of course already implemented in some libraries, is just a matter of knowing how to use it.
- **Live Performance Integration:** Using real-time Markov-based phoneme synthesis as an interactive performance tool.

5 Conclusion

This method allows us to **extract meaningful phoneme-like units from speech** and explore them creatively in our context of electronic music/sound manipulation. Potential applications include **voice-based synthesis, stochastic sound generation, and AI-assisted vocal textures, conceptual artwork, sound design, pranking your neighbours, etc**.

Further Reading

[1] Aalto University. Speech processing. `https://wiki.aalto.fi/display/ITSP/Introduction+to+Speech+Processing`, 2024. Accessed: March 5, 2025.

[2] BuiltIn. Markov chains: A gentle introduction. `https://builtin.com/machine-learning/markov-chain`, 2024. Accessed: March 5, 2025.

[3] Antoine Caillon and Philippe Esling. Rave: A variational autoencoder for fast and high-quality neural audio synthesis. *arXiv preprint arXiv:2111.05011*, 2021.

[4] Mousumi Malakar and Ravindra B Keskar. Progress of machine learning based automatic phoneme recognition and its prospect. *International Journal of Speech Technology*, 2023.

[5] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, et al. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Appendices

Complete Script

```
import librosa
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import seaborn as sns
import sys
import soundfile as sf


# ------------------------ STEP 1: FRAME AUDIO
    -------------------------
def frame_audio(audio_file, frame_size_ms=25, hop_size_ms=10, sr
    =16000):
    """Frame the audio file into overlapping frames."""
    y, sr = librosa.load(audio_file, sr=sr)
    frame_size = int(sr * frame_size_ms / 1000)
    hop_length = int(sr * hop_size_ms / 1000)
    frames = librosa.util.frame(y, frame_length=frame_size,
    hop_length=hop_length).T
    return frames, sr


# ------------------------ STEP 2: COMPUTE FFT
    ------------------------
def compute_fft(frames):
    """Compute the FFT magnitude spectrum for each frame."""
    fft_frames = np.abs(np.fft.rfft(frames, axis=1))
    return fft_frames


# ------------------------ STEP 3: K-MEANS CLUSTERING
    ------------------------
def build_codebook(features, k=50):
    """Cluster the FFT features using K-means to create the
    codebook."""
    print(f"Clustering {features.shape[0]} frames into {k} clusters
    ...")
    kmeans = KMeans(n_clusters=k, random_state=0, max_iter=300,
    n_init=10)
    kmeans.fit(features)
    print("Codebook creation complete.")
    return kmeans


# ------------------------ STEP 4: ASSIGN STATES
    ------------------------
def assign_to_codebook(features, kmeans_model):
    """Assign each frame to the nearest codebook cluster center."""
    assigned_states = kmeans_model.predict(features)
    return assigned_states


# ------------------------ STEP 5: MARKOV TRANSITION MATRIX
```

```
    ------------------------
def compute_markov_matrix(states, num_states):
    """Compute the Markov state transition matrix."""
    matrix = np.zeros((num_states, num_states))
    for (i, j) in zip(states[:-1], states[1:]):
        matrix[i, j] += 1
    matrix /= matrix.sum(axis=1, keepdims=True) + 1e-10  #
    Normalize
    return matrix


# ------------------------ STEP 6: VISUALIZE MARKOV MATRIX
    ------------------------
def visualize_markov_matrix(markov_matrix):
    """Visualize the Markov state transition matrix."""
    plt.figure(figsize=(10, 8))
    sns.heatmap(markov_matrix, cmap="viridis", square=True)
    plt.title("Markov State Transition Matrix")
    plt.xlabel("Next State")
    plt.ylabel("Current State")
    plt.show()


# ------------------------ STEP 7: SYNTHESIZE AUDIO
    ------------------------
def synthesize_audio(frames, assigned_states, markov_matrix, length
    =1000):
    """Synthesize audio by probabilistically selecting frames based
     on the Markov matrix."""
    synthesized_audio = []
    current_state = np.random.choice(np.arange(len(markov_matrix)))
    print(len(markov_matrix))
    print(np.where(assigned_states == current_state)[0])
    for _ in range(length):
        frame_indices = np.where(assigned_states == current_state)
    [0]
        if len(frame_indices) == 0:
            current_state = np.random.choice(np.arange(len(
    markov_matrix)))
            continue

        chosen_frame = frames[np.random.choice(frame_indices)]
        synthesized_audio.extend(chosen_frame)

        # Select next state based on transition probabilities
        next_state_probs = markov_matrix[current_state]
        current_state = np.random.choice(np.arange(len(
    markov_matrix)), p=next_state_probs)

    return np.array(synthesized_audio)


# ------------------------ STEP 8: SAVE SYNTHESIZED AUDIO
    ------------------------
def save_synthesized_audio(audio_array, sr, output_path="
    synthesized_audio.wav"):
    """Save the synthesized audio to a file."""
```

```
    sf.write(output_path, audio_array, sr)
    print(f"Synthesized audio saved to {output_path}")


# ------------------------ MAIN PIPELINE ------------------------
def main_pipeline(train_audio, test_audio, k_clusters=50):
    """Full pipeline: From audio to Markov matrix and synthesis
    using k-means clustering."""

    # 1. Frame the training audio and extract FFT features
    train_frames, sr = frame_audio(train_audio)
    train_features = compute_fft(train_frames)

    # 2. Create the codebook using K-means.
    kmeans_model = build_codebook(train_features, k=k_clusters)

    # ===========

    # 3. Process the test audio and assign states
    test_frames, _ = frame_audio(test_audio)
    test_features = compute_fft(test_frames)

    assigned_states = assign_to_codebook(test_features,
    kmeans_model)

    # 4. Compute and visualize the Markov transition matrix
    markov_matrix = compute_markov_matrix(assigned_states,
    k_clusters)
    visualize_markov_matrix(markov_matrix)

    # 5. Display sample state sequence and matrix
    print(f"\nSample State Sequence: {assigned_states[:20]}")
    print(f"\nMarkov Transition Matrix (shape {markov_matrix.shape
    }):\n", markov_matrix)

    # 6. Synthesize audio based on the Markov model
    synthesized_audio = synthesize_audio(test_frames,
    assigned_states, markov_matrix, length=2000)
    save_synthesized_audio(synthesized_audio, sr)


# ------------------------ RUN THE SCRIPT ------------------------
if __name__ == "__main__":
    # Provide two audio files (can be the same if testing)
    train_audio_file = "train_speech.wav"  # Reference audio for
    building the codebook
    test_audio_file = "test_speech.wav"    # Audio to analyze and
    synthesize from

    # Run the pipeline
    main_pipeline(train_audio_file, test_audio_file, k_clusters=50)
```

Listing 9: Full implementation of phoneme discovery and synthesis.