AudioProcessor Code Explanation

Overview

This code implements a complete audio processing pipeline that:

1. Processes audio files (denoising, voice activity detection)
2. Performs speaker diarization (separating different speakers)
3. Extracts speaker embeddings and acoustic features
4. Stores everything in a vector database (Milvus) for similarity search

Import Libraries

```python
import os, glob, librosa, noisereduce as nr, matplotlib.pyplot as plt
import soundfile as sf, numpy as np, torch, torch.nn.functional as F
import nemo.collections.asr as nemo_asr  # NVIDIA's speech AI toolkit
from pyannote.audio import Pipeline, Model, Inference  # Speaker diarization
from pydub import AudioSegment  # Audio manipulation
from pymilvus import *  # Vector database
```

Key libraries:

- **librosa**: Audio analysis and processing
- **noisereduce**: Removes background noise
- **nemo_asr**: NVIDIA's AI models for speech processing
- **pyannote.audio**: Advanced speaker diarization and embedding extraction
- **pymilvus**: Vector database for storing and searching audio features

---

Class Initialization

`__init__` Method

```python
def __init__(self, input_folder, output_folder, auth_token, milvus_host, milvus_port):
```

Purpose: Sets up the entire processing pipeline

What it does:

1. **Stores configuration**: Input/output folders, database connection details
2. **Creates output directory**: Where processed files will be saved
3. **Initializes AI models**: Loads pre-trained models for speech processing
4. **Connects to Milvus**: Sets up the vector database connection
5. **Prepares data storage**: Creates lists to store extracted features

Key components initialized:

- Voice Activity Detection (VAD) model
- Speaker diarization pipeline
- Speaker embedding model
- Vector database collections

Model Setup Functions

`setup_models` Method

```python
def setup_models(self):
```

Purpose: Loads and configures three main AI models

Models loaded:

1. **VAD Model (Voice Activity Detection)**
   - Model: `nvidia/frame_vad_multilingual_marblenet_v2.0`
   - Purpose: Detects which parts of audio contain speech vs silence
   - Output: Probability scores for each audio frame
2. **Diarization Pipeline**
   - Model: `pyannote/speaker-diarization@2.1`
   - Purpose: Separates different speakers in the audio
   - Output: Time segments labeled by speaker ID
3. **Speaker Embedding Model**
   - Model: `pyannote/embedding`

- Purpose: Converts speech segments into numerical vectors (embeddings)
- Output: 512-dimensional vectors representing speaker characteristics

Why these models?

- **VAD**: Removes silence and non-speech, improving processing efficiency
- **Diarization**: Essential for multi-speaker scenarios
- **Embeddings**: Enable speaker similarity comparison and identification

---

Database Setup Functions

`setup_milvus` Method

```python
def setup_milvus(self):
```

Purpose: Establishes connection to Milvus vector database

What Milvus does:

- Stores high-dimensional vectors (embeddings)
- Enables fast similarity search
- Scales to millions of audio samples

`setup_collections` Method

```python
def setup_collections(self):
```

Purpose: Creates two database collections with specific schemas

Collection 1: Speaker Embeddings

```python
```

```
# Fields stored for each speaker embedding:
- id: Unique identifier
- audio_name: Original file name
- speaker_id: Which speaker (SPEAKER_00, SPEAKER_01, etc.)
- audio_path: File location
- embedding_vector: 512D numerical vector
- timestamp: When processed
```

Collection 2: Log-Mel Features

```python
# Fields stored for each log-mel feature:
- Similar fields but with 192D logmel_vector instead
```

Why two collections?

- **Speaker embeddings**: Better for speaker identification/verification
- **Log-mel features**: Traditional acoustic features, kept for comparison

---

Audio Processing Pipeline

`find_audio_files` Method

```python
def find_audio_files(self):
```

Purpose: Discovers all audio files in the input directory

Process:

1. Searches for multiple audio formats: .wav, .mp3, .flac, .m4a, .aac
2. Uses recursive search (includes subdirectories)
3. Validates each file by attempting to load a small sample
4. Filters out corrupted or empty files

Output: List of valid audio file paths

`preprocess_audio` Method

```
python

def preprocess_audio(self, audio_path, output_folder):
```

Purpose: Step 1 of processing - cleans up the audio

Process:

1. **Load audio**: Converts to 16kHz sample rate (standard for speech)
2. **Create waveform visualization**: Saves plot of original audio
3. **Apply noise reduction**: Uses AI to remove background noise
4. **Save results**: Stores both original and denoised audio + visualizations

Why preprocessing?

- **Standardization**: Ensures consistent format for AI models
- **Noise removal**: Improves accuracy of downstream processing
- **Visualization**: Helps users understand what was processed

---

`apply_vad` Method

```
python

def apply_vad(self, audio_path, output_folder, threshold=0.99):
```

Purpose: Step 2 - removes non-speech portions

Process:

1. **Load audio** into tensor format for neural network
2. **Run VAD model**: Gets probability of speech for each 20ms frame
3. **Apply threshold**: Frames above 0.99 probability are considered speech
4. **Merge segments**: Combines nearby speech segments
5. **Extract speech**: Removes silence, keeps only speech portions
6. **Add padding**: Small buffers around speech segments

Parameters explained:

- `threshold=0.99`: Very high confidence required (reduces false positives)
- `min_speech_duration=0.15`: Ignores very short segments (reduces noise)

Output: Audio file containing only speech segments

`perform_diarization` Method

```python
def perform_diarization(self, audio_path, output_folder):
```

Purpose: Step 3 - separates different speakers

Process:

1. **Run diarization**: AI identifies when different speakers are talking
2. **Generate RTTM file**: Standard format showing speaker timing
   ```
   SPEAKER filename 1 start_time duration <NA> <NA> speaker_id <NA>
   ```
3. **Separate speakers**: Creates individual audio files for each speaker
4. **Combine segments**: Merges all segments from same speaker into one file

Example output:

- `speaker_SPEAKER_00.wav`: All speech from first speaker
- `speaker_SPEAKER_01.wav`: All speech from second speaker
- `diarization.rttm`: Timeline of who spoke when

Feature Extraction Functions

`extract_speaker_embedding` Method

```python
def extract_speaker_embedding(self, audio_path, audio_name, speaker_id):
```

Purpose: Step 4A - converts speech to numerical representation

Process:

1. **Load audio**: Ensures mono, 16kHz format
2. **Create audio dictionary**: Format required by pyannote model
3. **Extract embedding**: Neural network converts speech → 512D vector
4. **Prepare data**: Creates structured record for database storage

What are embeddings?

- Numerical "fingerprints" of speaker characteristics
- Similar voices produce similar embedding vectors
- Enable speaker recognition and similarity search

Output: Dictionary containing embedding vector and metadata

`extract_logmel_features` Method

```python
def extract_logmel_features(self, audio_path, audio_name, speaker_id):
```

Purpose: Step 4B - extracts traditional acoustic features

Process:

1. **Compute Mel Spectrogram**: Frequency analysis mimicking human hearing
2. **Apply logarithm**: Compresses dynamic range
3. **Calculate deltas**: First and second derivatives (capture dynamics)
4. **Normalize**: Standardizes feature values
5. **Aggregate**: Averages over time to get fixed-size vector

Technical details:

- **64 mel bands**: Frequency resolution
- **Delta features**: Capture how sound changes over time
- **Final dimension**: 64 + 64 + 64 = 192D vector

Why both embedding and log-mel?

- **Embeddings**: State-of-the-art, learned representations
- **Log-mel**: Traditional features, interpretable, good baseline

Database Operations

`insert_to_milvus` Method

```python
def insert_to_milvus(self, embedding_data, logmel_data):
```

Purpose: Stores extracted features in vector database

Process:

1. **Insert embeddings**: Adds to speaker_embeddings collection
2. **Insert log-mel**: Adds to logmel_features collection
3. **Update local cache**: Keeps copy in memory
4. **Error handling**: Catches and reports database issues

`search_similar_speakers` Method

```python
def search_similar_speakers(self, query_embedding, top_k=5):
```

Purpose: Finds speakers similar to a query

Process:

1. **Load collection**: Ensures database is ready
2. **Configure search**: Uses cosine similarity metric
3. **Execute search**: Finds k most similar embeddings
4. **Return results**: Includes similarity scores and metadata

Search algorithm:

- **Metric**: Cosine similarity (measures vector angle)
- **Index type**: IVF_FLAT (inverted file index for speed)
- **Output**: Ranked list of similar speakers

Main Processing Function

`process_single_audio` Method

```python
def process_single_audio(self, audio_path):
```

Purpose: Complete pipeline for one audio file

Full pipeline:

1. **Validation**: Checks file exists and is loadable
2. **Preprocessing**: Denoising + visualization
3. **VAD**: Speech detection
4. **Diarization**: Speaker separation
5. **Feature extraction**: Both embedding and log-mel for each speaker
6. **Database storage**: Insert all features into Milvus
7. **JSON export**: Save features to local file

Error handling:

- Validates file at each step
- Provides detailed error messages
- Continues processing other files if one fails

`process_all_audios` Method

```python
def process_all_audios(self):
```

Purpose: Batch processes all audio files

Process:

1. **Discovery**: Finds all valid audio files
2. **Progress tracking**: Uses tqdm for progress bars
3. **Individual processing**: Calls `process_single_audio` for each file

4. **Database flush**: Ensures all data is written to Milvus
5. **Export combined data**: Creates master JSON file with all features
6. **Statistics**: Reports success/failure counts

Utility Functions

`demo_similarity_search` Method

```python
def demo_similarity_search(self, query_audio_path, top_k=5):
```

Purpose: Demonstrates speaker similarity search

Use case:

- Input: New audio file
- Output: List of most similar speakers from database
- Application: Speaker identification, verification

`get_collection_stats` Method

```python
def get_collection_stats(self):
```

Purpose: Reports database statistics

Output:

- Number of speaker embeddings stored
- Number of log-mel features stored
- Database health status

Complete Workflow Summary

1. **Initialize**: Load AI models, connect to database
2. **Discover**: Find all audio files to process
3. **For each audio file:**

- Clean up audio (denoise)
- Remove silence (VAD)
- Separate speakers (diarization)
- Extract features for each speaker
- Store in database

4. **Export**: Save all data to JSON files
5. **Search**: Enable similarity queries

Key Innovations

1. **End-to-end pipeline**: From raw audio to searchable database
2. **Multiple AI models**: VAD + diarization + embeddings
3. **Dual feature types**: Modern (embeddings) + traditional (log-mel)
4. **Scalable storage**: Vector database for millions of samples
5. **Visual feedback**: Waveform plots at each processing step

This system enables applications like:

- Speaker identification across large audio collections
- Voice biometrics and authentication
- Audio content organization and search
- Forensic audio analysis