# Noise Detector

## noise_detector.py

```
# Threshold / Sliding window
# https://raw.githubusercontent.com/jeysonmc/python-google-speech-scripts/master/stt_google.py

# WebSocket streaming:
# https://gist.github.com/fopina/3cefaed1b2d2d79984ad7894aef39a68

import pyaudio
import wave
import audioop
import subprocess
import os
import time
import math
import struct
import threading
import io
import numpy as np
from collections import deque

from lock_manager import Lock_Manager
from util import Util

class Noise_Detector(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)

        self.name           = self.__class__.__name__

        self.FORMAT         = pyaudio.paFloat32
        self.RATE           = 48000 # Hz, so samples (bytes) per second
        self.CHUNK_SIZE             = 8192 # How many bytes to read from mic each time (stream.read())
        self.CHUNKS_PER_SEC         = math.floor(self.RATE / self.CHUNK_SIZE) # How many chunks make a
second? (16.000 bytes/s, each chunk is 1.024 bytes, so 1s is 15 chunks)
        self.CHANNELS       = 1
        self.HISTORY_LENGTH    = 2 # Seconds of audio cache for prepending to records to prevent chopped
phrases (history length + observer length = min record length)
        self.OBSERVER_LENGTH   = 5    # Time in seconds to be observed for noise
        self.NOTIFICATION_LIMIT = 1 # Seconds before a notification is sent
        self.LIMIT_RECODING    = 100 # 최대 Recoding chunk 수
        self.CURRENT_RECODING_TIME = 0 # 현재 Recoding chunk 수
        self.REMAIN_RECORDING_FILES = 3 # 10이상 부터 삭제 후 저장
```

```
        self.RECODING_OVER_THRESHOLD = 5 # Recoding 임계값을 연속 넘는 회수로 저장 여부 판단

        self.archive          = os.path.join(os.path.dirname(os.path.realpath(__file__)), 'archive')
        self.current_file     = None
        self.chunk            = None
        self.record           = [] # Stores audio chunks
        self.notified         = False # If we already sent a notification

        self.audio            = pyaudio.PyAudio()
        self.stream           = self.get_stream()

        self.threshold        = self.determine_threshold()

        self.lock_manager     = Lock_Manager("noise")
        self.detected_at      = None

    def __del__(self):
        # Stop recording
        if self.stream:
            self.stream.close()

        if self.audio:
            self.audio.terminate()

        # Remove lock if exists
        self.lock_manager.remove()

    def get_stream(self):
        """
        Open audio stream

        @return PyAudio
        """
        return self.audio.open(
            format=self.FORMAT,
            channels=self.CHANNELS,
            rate=self.RATE,
            input=True,
            frames_per_buffer=self.CHUNK_SIZE
        )

    def determine_threshold(self):
        """
        Determine threshold noise intensity using RMS
        Anything below the threshold is considered silence

        @return float
        """
        Util.log(self.name, "Determining threshold...")
```

```
        res = []
        for x in range(50):
            block = self.stream.read(self.CHUNK_SIZE)
            rms = self.get_rms(block)
            res.append(rms)

        # Set threshold to 20% above avergae
        threshold = (sum(res) / len(res)) * 1.7 #1.2
        Util.log(self.name, "Setting threshold to: " + str(threshold))

        return threshold

    def get_rms(self, block):
        """
        Calculate Root Mean Square (noise level) for audio chunk

        @param bytes block
        @return float
        """
        d = np.frombuffer(block, np.float32).astype(np.float)
        return np.sqrt((d * d).sum() / len(d))

    def start_recording(self):
        """
        Setup the recorder
        """
        self.current_file = self.archive + "/" + self.detected_at + ".wav"

        Util.log(self.name, "Noise detected! Recording...")

    def stop_recording(self):
        """
        Reset variables to default
        """
        self.current_file = None
        self.detected_at = None
        self.notified = False
        self.record = []
        self.CURRENT_RECODING_TIME = 0

    def run(self):
        """
        Detect noise from microphone and record
        Noise is defined as sound surrounded by silence (according to threshold)
        """

        # Stores audio intensity of previous sound-chunks
        # If one of these chunks is above threshold, recording gets triggered
        # Keep the last {OBSERVER_LENGTH} seconds in observer
        observer = deque(maxlen=self.OBSERVER_LENGTH * self.CHUNKS_PER_SEC)
```

```
        # Prepend audio from before noise was detected
        # Keep the last {HISTORY_LENGTH} seconds in history
        history = deque(maxlen=self.HISTORY_LENGTH * self.CHUNKS_PER_SEC)

        Util.log(self.name, "Listening...")

        try:
            while True:
                # Current chunk of audio data
                self.chunk = self.stream.read(self.CHUNK_SIZE, exception_on_overflow = False)
                history.append(self.chunk)

                # Add noise level of this chunk to the sliding-window
                rms = self.get_rms(self.chunk)
                #Util.log(self.name, "Noise threshold=" + str(rms))
                observer.append(rms)

                if self.detected(sum([x > self.threshold for x in observer]) > self.RECODING_OVER_THRESHOLD)
and self.LIMIT_RECODING > self.CURRENT_RECODING_TIME:
                    self.CURRENT_RECODING_TIME = self.CURRENT_RECODING_TIME + 1
                    # There's at least one chunk in the sliding-window above threshold
                    if not self.recording():
                        self.start_recording()

                    #Util.log(self.name, "Record.append noise level="+ str(sum([x > self.threshold for x in
observer])) + ", time=" + str(self.CURRENT_RECODING_TIME) )
                    self.record.append(self.chunk)

                    if not self.notified and len(self.record) > self.NOTIFICATION_LIMIT * self.CHUNKS_PER_SEC:
                        self.notify()
                elif self.recording():
                    # Silence limit was reached, finish recording and save
                    self.delete()
                    self.save(list(history) + self.record)
                    self.stop_recording()

                    Util.log(self.name, "Listening...")
        except KeyboardInterrupt:
            Util.log(self.name, "Interrupted.")

    def get_chunk(self):
        """
        Return the current chunk of audio data

        @return bytes
        """
        return self.chunk

    def delete(self):
```

```
        """
        delete mic data to a WAV file.
        @param list data
        """
        count = 0
        Util.log(self.name, "Delete audio...")
        for filename in sorted(os.listdir(self.archive), reverse=True):
            if not filename.startswith('.'):
                type = self.get_type(filename)
                if type == "audio":
                    count = count + 1
                    if self.REMAIN_RECORDING_FILES < count:
                        Util.log(self.name, "Delete audio filename=" + filename + ", type=" + type + ", count=" +
str(count))
                        os.remove(self.archive + "/" + filename)

    def get_type(self, filename):
        name, extension = os.path.splitext(filename)
        return 'video' if extension == '.mp4' else 'video' if extension == '.avi' else 'audio' if extension == '.wav' else
'audio' if extension == '.mp3' else 'photo'

    def save(self, data):
        """
        Save mic data to a WAV file.

        @param list data
        """
        Util.log(self.name, "Saving audio...")

        # Flatten the list
        data = b''.join(data)

        # Write converted data to file
        with open(self.current_file, "wb+") as file:
            file.write(self.generate_wav(data))

        # Convert 음질 개떡
        #self.convert_to_mp3(self.current_file)

    def bytes_to_array(self, bytes, type):
        """
        Convert raw audio data to TypedArray

        @param bytes bytes
        @return numpy-Array
        """
        return np.frombuffer(bytes, dtype=type)

    def generate_wav(self, raw):
        """
```

```
Create WAVE-file from raw audio chunks

@param bytes raw
@return bytes
"""
# Check if input format is supported
if self.FORMAT not in (pyaudio.paFloat32, pyaudio.paInt16):
    print("Unsupported format")
    return

# Convert raw audio bytes to typed array
samples = self.bytes_to_array(raw, np.float32)

# Get sample size
sample_size = pyaudio.get_sample_size(self.FORMAT)

# Get data-length
byte_count = (len(samples)) * sample_size

# Get bits/sample
bits_per_sample = sample_size * 8

# Calculate frame-size
frame_size = int(self.CHANNELS * ((bits_per_sample + 7) / 8))

# Container for WAVE-content
wav = bytearray()

# Start RIFF-Header
wav.extend(struct.pack('<cccc', b'R', b'I', b'F', b'F'))
# Add chunk size (data-size minus 8)
wav.extend(struct.pack('<I', byte_count + 0x2c - 8))
# Add RIFF-type ("WAVE")
wav.extend(struct.pack('<cccc', b'W', b'A', b'V', b'E'))

# Start "Format"-part
wav.extend(struct.pack('<cccc', b'f', b'm', b't', b' '))
# Add header length (16 bytes)
wav.extend(struct.pack('<I', 0x10))
# Add format-tag (e.g. 1 = PCM, 3 = FLOAT)
wav.extend(struct.pack('<H', 3))
# Add channel count
wav.extend(struct.pack('<H', self.CHANNELS))
# Add sample rate
wav.extend(struct.pack('<I', self.RATE))
# Add bytes/second
wav.extend(struct.pack('<I', self.RATE * frame_size))
# Add frame size
wav.extend(struct.pack('<H', frame_size))
# Add bits/sample
```

```
        wav.extend(struct.pack('<H', bits_per_sample))

        # Start data-part
        wav.extend(struct.pack('<cccc', b'd', b'a', b't', b'a'))
        # Add data-length
        wav.extend(struct.pack('<I', byte_count))

        # Add data
        for sample in samples:
            wav.extend(struct.pack("<f", sample))

        return bytes(wav)

    def convert_to_mp3(self, path):
        """
        Convert wav-file to mp3

        @param string path
        """
        Util.log(self.name, "Converting audio...")

        try:
            cmd = 'lame --preset insane "{}" 2> /dev/null && rm "{}"'.format(path, path)
            p = subprocess.Popen(cmd, shell=True)
            (output, err) = p.communicate()

        except subprocess.CalledProcessError:
            Util.log(self.name, "Error converting audio")

    def detected(self, has_noise):
        """
        Check if this or another detector detected something

        @param boolean has_noise
        @return boolean
        """
        if has_noise:
            self.lock_manager.set()
        else:
            self.lock_manager.remove()

        self.detected_at = self.lock_manager.get_lock_time()

        return self.detected_at is not None

    def recording(self):
        """
        Check if currently recording

        @return boolean
```

```
            """
            return len(self.record) > 0

        def notify(self):
            """
            Notify
            """
            Util.log(self.name, "Notifying")
            self.notified = True

    if __name__ == "__main__":
        nd = Noise_Detector()
        nd.start()
```

---

Revision #1
★Created 1 June 2023 14:38:23 by Hyeon Su Ryu
Updated 1 June 2023 14:39:19 by Hyeon Su Ryu