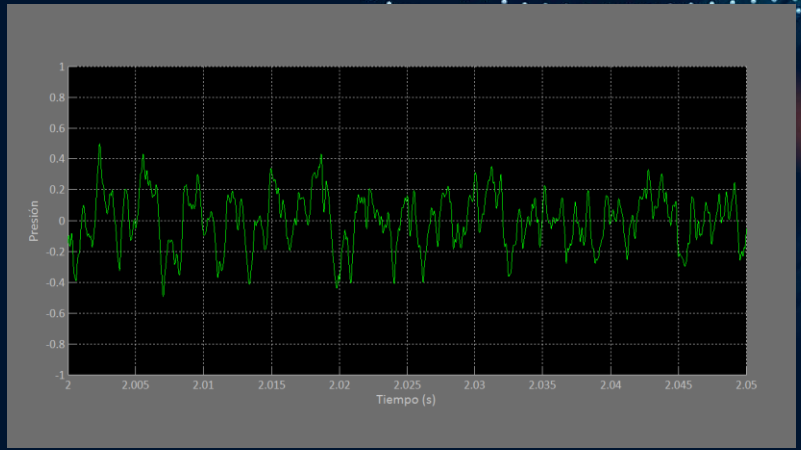
ANALISIS DE AUDIO

Jarol Vidal Angulo

¿QUÉ ES UNA SEÑAL DE AUDIO?

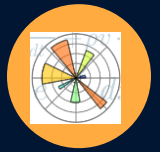
Se llama audio a la señal correspondiente a los sonidos. Puede decirse que un audio es una señal analógica que, a nivel eléctrico, equivale a una señal sonora. Su frecuencia se ubica entre 20 y 20.000 Hz



LIBRERIAS USADAS

NUMPY

Crear vectores y matrices grandes multidimensionales



MATPLOTLIB

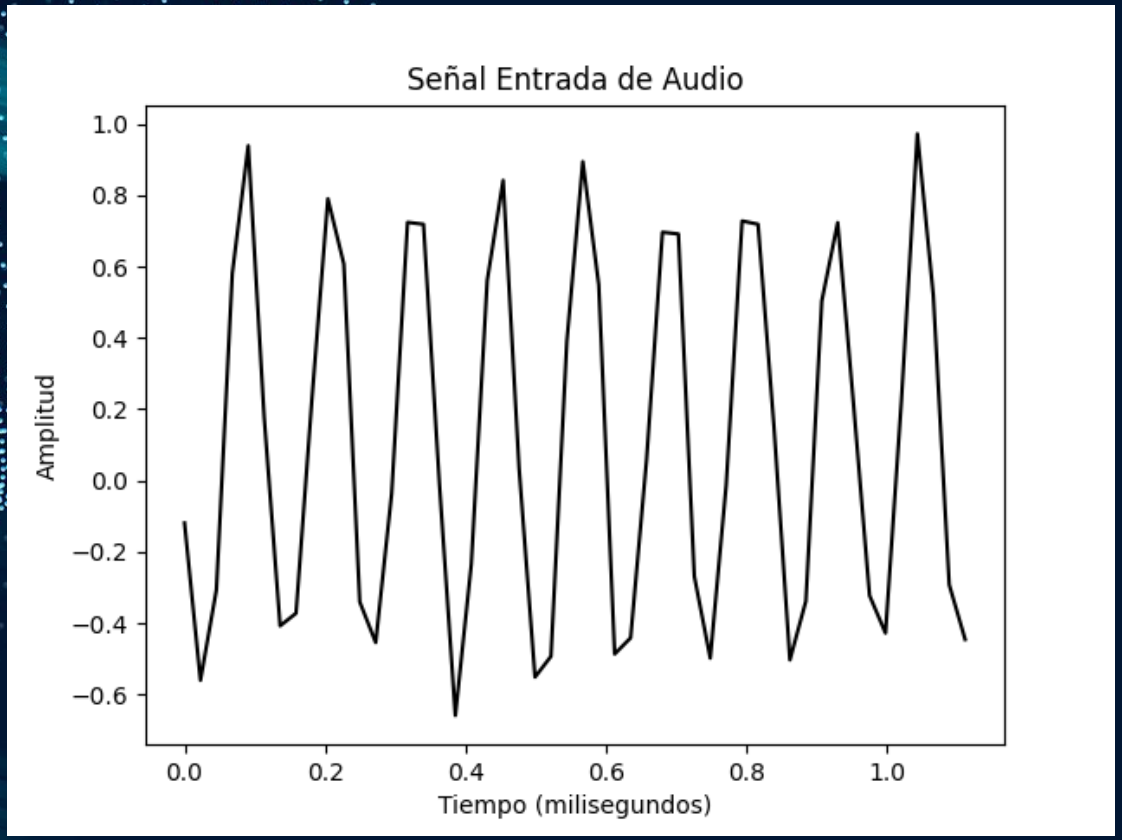
librería especializada en la creación de gráficos en dos dimensiones.

SCIPY

Libreria que se compone de herramientas y algoritmos matemáticos

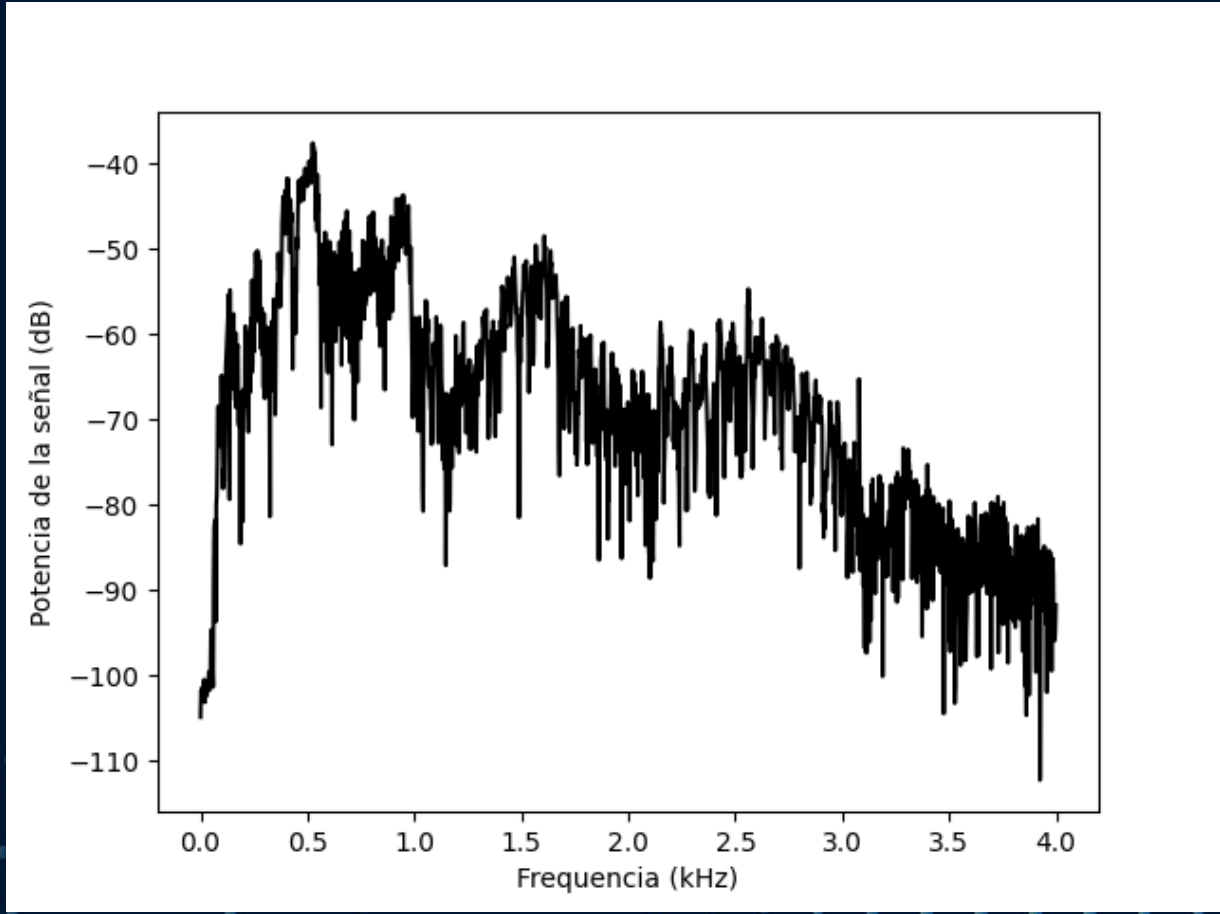
GRAFICAR AUDIO

```python
# GRAFICAR ARCHIVO DE AUDIO

# Importar las librerías
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile

# Lee el archivo de audio
frecuencia_muestreo, senial = wavfile.read('sonido_aleatorio.wav')

# Display the params
print('\nTamaño señal:', senial.shape)
print('Tipo de dato:', senial.dtype)
print('Duracción de la señal:', round(senial.shape[0] / float(frecuencia_muestreo), 2), 'seconds')
print('Frecuencia de muestreo: ', frecuencia_muestreo)

# Normalizar la señal
senial = senial / np.power(2, 15)

# Extraer los primeros 50 valores
senial = senial[:50]

# Construir el eje de tiempo en milisegundos
eje_del_tiempo = 1000 * np.arange(0, len(senial), 1) / float(frecuencia_muestreo)

# Dibujar la señal de audio
plt.plot(eje_del_tiempo, senial, color='black')
plt.xlabel('Tiempo (milisegundos)')
plt.ylabel('Amplitud')
plt.title('Señal Entrada de Audio')
plt.show()
```


Señal Entrada de Audio
Amplitud
Tiempo (milisegundos)
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0.0
0.2
0.4
0.6
0.8
1.0

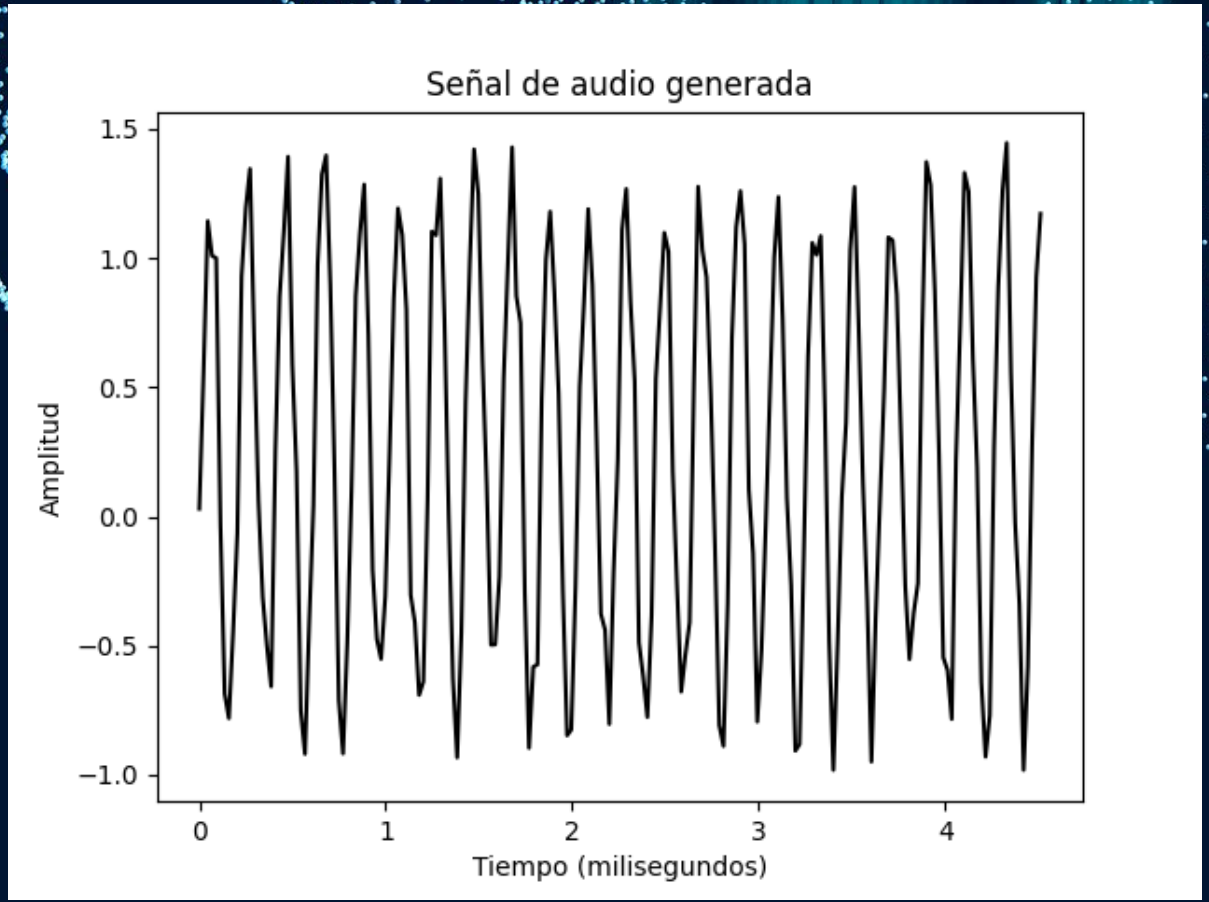
TRANSFORMAR A FRECUENCIA

```python
# TRANSFORMACIÓN AL DOMINIO DE LA FRECUENCIA
# Importar librerías
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
# Leer el archivo de audio
frecuencia_de_muestreo, senial = wavfile.read('palabra_hablada.wav')
# Normalizar los valores
senial = senial / np.power(2, 15)
# Extraer la longitud de la señal de audio
longitud_senial = len(senial)
# Extraer la mitad de la longitud
mitad_longitud = np.ceil((longitud_senial + 1) / 2.0).astype(np.int)
# Aplicar la Transformada de Fourier
frecuencia_senial = np.fft.fft(senial)
# Normalización
frecuencia_senial = abs(frecuencia_senial[0:mitad_longitud]) / longitud_senial
# Cuadrado
frecuencia_senial **= 2
# Extrae la longitud de la señal de frecuencia transformada
len_fts = len(frecuencia_senial)
# Ajustar la señal para casos pares e impares
if longitud_senial % 2:
    frecuencia_senial[1:len_fts] *= 2
else:
    frecuencia_senial[1:len_fts-1] *= 2
# Extraer el valor de potencia en dB
potencia_senial = 10 * np.log10(frecuencia_senial)
# Construir el eje X
eje_x = np.arange(0, mitad_longitud, 1) * (frecuencia_de_muestreo / longitud_senial) / 1000.0
# Graficar la figura
plt.figure()
plt.plot(eje_x, potencia_senial, color='black')
plt.xlabel('Frequencia (kHz)')
plt.ylabel('Potencia de la señal (dB)')
plt.show()
```


Potencia de la señal (dB)
−40
−50
−60
−70
−80
−90
−100
−110
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Frequencia (kHz)

GENERAR AUDIO

```python
# Importar librerías
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import write

# Archivo de salida en el cual se grabará el audio
archivo_salida = 'audio_generado.wav'
# Especificar los parámetros del audio
duracion = 4  # in seconds
frecuencia_muestreo = 44100  # in Hz
frecuencia_tono = 784
valor_minimo = -4 * np.pi
valor_maximo = 4 * np.pi
# Generar la señal de audio
t = np.linspace(valor_minimo, valor_maximo, duracion * frecuencia_muestreo)
senial = np.sin(2 * np.pi * frecuencia_tono * t)
# Agregar algún ruido a la señal
ruido = 0.5 * np.random.rand(duracion * frecuencia_muestreo)
senial += ruido
# Escalar a valores enteros de 16 bits
factor_escalamiento = np.power(2, 15) - 1
senial_normalizada = senial / np.max(np.abs(senial))
senial_escalada = np.int16(senial_normalizada * factor_escalamiento)
# Almacenar la señal de audio en el archivo de salida
write(archivo_salida, frecuencia_muestreo, senial_escalada)
# Extraer los primeros 200 valores de la señal de audio
senial = senial[:200]
# Construir el eje del tiempo en milisegundos
eje_tiempo = 1000 * np.arange(0, len(senial), 1) / float(frecuencia_muestreo)
# Graficar la señal de audio
plt.plot(eje_tiempo, senial, color='black')
plt.xlabel('Tiempo (milisegundos)')
plt.ylabel('Amplitud')
plt.title('Señal de audio generada')
plt.show()
```


Señal de audio generada
1.5
1.0
0.5
0.0
−0.5
−1.0
Amplitud
0
1
2
3
4
Tiempo (milisegundos)

SINTETIZAR TONO

```python
# SINTETIZAR TONOS
# Generar librerías
import json
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import write
# Sintetizar el tono basado en los parámetros de entrada
def sintetizador_tono(frecuencia, duracion, amplitud=1.0, frecuencia_muestreo=44100):
    # Construir el eje de tiempo
    eje_tiempo = np.linspace(0, duracion, duracion * frecuencia_muestreo)
    # Construir la señal de audio
    senial = amplitud * np.sin(2 * np.pi * frecuencia * eje_tiempo)
    return senial.astype(np.int16)
if __name__=='__main__':
    # Nombres de los archivos de salida
    archivo_tono_generado = 'tono_generado.wav'
    archivo_secuencia_tono_generada = 'secuencia_de_tono_generada.wav'
    # Source: http://www.phy.mtu.edu/~suits/notefrecuencias.html
    archivo_mapeo = 'tone_mapping.json'
    # Cargue el mapa de tono a frecuencia desde el archivo de mapeo
    with open(archivo_mapeo, 'r') as f:
        mapa_tonos = json.loads(f.read())
    # Configure los parámetros de entrada para generar el tono 'F'
    nombre_tono = 'F'
    duracion = 3     # segundos
    amplitud = 12000
    frecuencia_muestreo = 44100    # Hz
    # Extrae la frecuencia del tono
    frecuencia_tono = mapa_tonos[nombre_tono]
    # Genere el tono usando los parámetros anteriores
    tono_sintetizado = sintetizador_tono(frecuencia_tono, duracion, amplitud, frecuencia_muestreo)
    # Escribe la señal de audio en el archivo de salida.
    write(archivo_tono_generado, frecuencia_muestreo, tono_sintetizado)
    # Defina la secuencia de tonos junto con las duraciones correspondientes en segundos
    tono_secuencia = [('G', 0.4), ('D', 0.5), ('F', 0.3), ('C', 0.6), ('A', 0.4)]
    # Construya la señal de audio basándose en la secuencia anterior
    senial = np.array([])
```

```
    for item in tono_secuencia:
        # Obtiene el nombre del tono
        nombre_tono = item[0]
        # Extrae la frecuencia correspondiente del tono.
        frecuencia = int(mapa_tonos[nombre_tono])
        # Extrae la duración
        duracion = item[1]
        duracion = int(duracion)
        # Sintetizar el tono
        tono_sintetizado = sintetizador_tono(frecuencia, duracion, amplitud, frecuencia_muestreo)
        # Añadir la señal de salida
        senial = np.append(senial, tono_sintetizado, axis=0)
    # Guarda el audio en el archivo de salida
    write(archivo_secuencia_tono_generada, frecuencia_muestreo, senial)
```

RECONOCER PALABRAS

```python
import os
import argparse
import warnings
import numpy as np
from scipy.io import wavfile
from hmmlearn import hmm
from python_speech_features import mfcc
# Define una función para analizar los argumentos de entrada
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Entrena la voz basada en HMM: \
            sistema de reconocimiento')
    parser.add_argument("--input-folder", dest="input_folder", required=True,
            help="Carpeta de entrada que contiene los archivos de audio para entrenamiento.")
    return parser
# Define una clase para entrenar el HMM
class ModelHMM(object):
    def __init__(self, num_components=4, num_iter=1000):
        self.n_components = num_components
        self.n_iter = num_iter
        self.cov_type = 'diag'
        self.model_name = 'GaussianHMM'
        self.models = []
        self.model = hmm.GaussianHMM(n_components=self.n_components,
                covariance_type=self.cov_type, n_iter=self.n_iter)
    # 'training_data' es un array numpy 2D donde cada fila es 13-dimensional
    def train(self, training_data):
        np.seterr(all='ignore')
        cur_model = self.model.fit(training_data)
        self.models.append(cur_model)
    # corre el modelo HMM para realizar inferencia sobre la entrada de datos
    def compute_score(self, input_data):
        return self.model.score(input_data)

# Define una función para construir un modelo para cada palabra
def build_models(input_folder):
    # Inicializar la variable para almacenar todos los modelos
    speech_models = []
    # Analiza el directorio de entrada
```

```
    for dirname in os.listdir(input_folder):
        # Obtiene el nombre del subfolder
        subfolder = os.path.join(input_folder, dirname)
        if not os.path.isdir(subfolder):
            continue
        # Extrae la etiqueta
        label = subfolder[subfolder.rfind('/') + 1:]
        # Inicializa las variables
        X = np.array([])
        # Crea una lista de archivos a ser utilizados para el entrenamiento
        # Se deja un archivo por folder para validación
        training_files = [x for x in os.listdir(subfolder) if x.endswith('.wav')][:-1]
        # Se itera a través de los archivos de entrenamiento y se construyen los modelos
        for filename in training_files:
            # Se extrae el path actual
            filepath = os.path.join(subfolder, filename)
            # Se lee la señal lde audio desde el archivo de entrada
            sampling_freq, signal = wavfile.read(filepath)
            # Se extraen las características MFCC
            with warnings.catch_warnings():
                warnings.simplefilter('ignore')
                features_mfcc = mfcc(signal, sampling_freq)
            # Se agrega a la variable X
            if len(X) == 0:
                X = features_mfcc
            else:
                X = np.append(X, features_mfcc, axis=0)
        # Se crea el modelo HMM
        model = ModelHMM()
        # Se entrena el HMM
        model.train(X)
        # Se almacena el modelo para la palabra actual
        speech_models.append((model, label))
        # Se reinicia la variable
        model = None
    return speech_models
# Define una función para ejecutar pruebas sobre los archivos de entrada
```

```
def run_tests(test_files):
    # Clasifica los datos de entrada
    for test_file in test_files:
        # Lee el archivo de entrada
        sampling_freq, signal = wavfile.read(test_file)
        # Extrae las características MFCC
        with warnings.catch_warnings():
            warnings.simplefilter('ignore')
            features_mfcc = mfcc(signal, sampling_freq)
        # Define variables
        max_score = -float('inf')
        output_label = None
        # Ejecuta el vector de características actual a través
        # de todos los modelos HMM y elige el que tenga el
        # puntaje más alto
        for item in speech_models:
            model, label = item
            score = model.compute_score(features_mfcc)
            if score > max_score:
                max_score = score
                predicted_label = label
        # Muestra la salida prevista
        start_index = test_file.find('/') + 1
        end_index = test_file.rfind('/')
        original_label = test_file[start_index:end_index]
        print('\nOriginal: ', original_label)
        print('Predicción:', predicted_label)

if __name__=='__main__':
    args = build_arg_parser().parse_args()
    input_folder = args.input_folder
    # Construye el modelo HMM para cada palabra
    speech_models = build_models(input_folder)
    #  Verifica los archivos -- el archivo 15 en cada subfolder
    test_files = []
    for root, dirs, files in os.walk(input_folder):
        for filename in (x for x in files if '15' in x):
            filepath = os.path.join(root, filename)
            test_files.append(filepath)
    run_tests(test_files)
```

GRACIAS!