PROCESAMIENTO DE AUDIO

JULIAN GIRALDO CARDONA

JUAN DAVID ALVAREZ

GRAFICAR AUDIO

- Importamos librerías

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
```

- Creamos una variable de muestreo que también se define como una señal.

```
frecuencia_muestreo, senial = wavfile.read('Prueba.wav')
```

IMPRIMIR EN PANTALLA

```
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)
```

- Se muestra en pantalla los datos importantes de la señal recibida, esto se hace gracias a las funciones de numpy, se calcula la duración,

- Como la variable Senial tiene valores de forma matricial, entonces se permite el manejo con numpy

```
Senial: [[  4   4]
 [ -6  -6]
 [-12 -12]
 ...
 [-24 -24]
 [-17 -17]
 [-12 -12]]
```

NORMALIZACIÓN DE LA SEÑAL

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

- Hacemos una división del contenido de la variable senial con la instrucción np.power de numpy

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

- Se extraen los primeros 50 valores

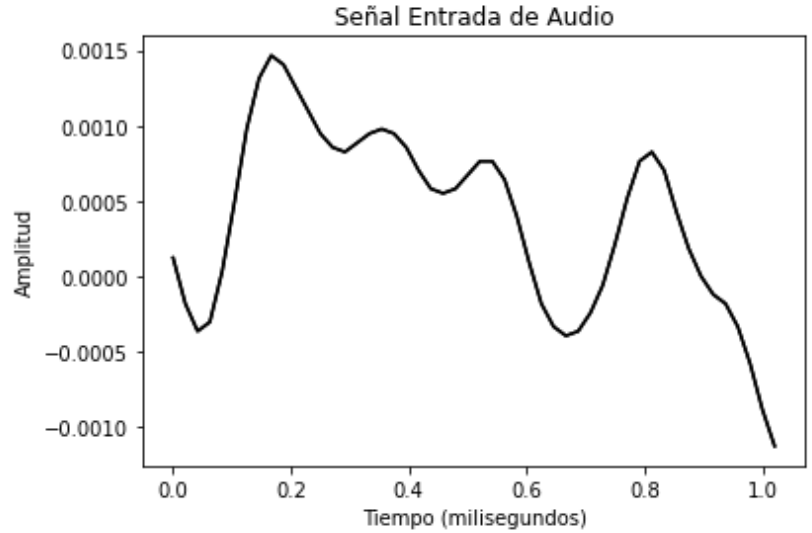
CONSTRUCCIÓN DE LA GRAFICA

```
# 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()
```

- Primero construimos el eje del tiempo (eje x) , este lo llevaremos a milisegundos con una simple operación.
- Después de construir el eje x, podemos dibujar la grafica con la librería matplotlib

RESULTADO FINAL

```
Senial: [[  4   4]
 [ -6  -6]
 [-12 -12]
 ...
 [-24 -24]
 [-17 -17]
 [-12 -12]]

Tamaño señal: (773760, 2)
Tipo de dato: int16
Duracción de la señal: 16.12 seconds
Frecuencia de muestreo:  48000
```



TRANSFORMAR A FRECUENCIA

- Leemos el archivo de audio y normalizamos los valores

```
# Leer el archivo de audio
frecuencia_de_muestreo, senial = wavfile.read('palabra_hablada.wav')
```

```
# Normalizar los valores
senial = senial / np.power(2, 15)
```

❖ Después de normalizar los valores, extraemos la longitud de la señal de audio

```
# Extraer la longitud de la señal de audio
longitud_senial = len(senial)
```

TRANSFORMADA DE FOURIER

- Extracción de la mitad de la longitud

```
# Extraer la mitad de la longitud
mitad_longitud = np.ceil((longitud_senial + 1) / 2.0).astype(np.int)
```

❖ Hallamos la transformada de Fourier mediante numpy y normalizamos.

```
# Aplicar la Transformada de Fourier
frecuencia_senial = np.fft.fft(senial)

# Normalización
frecuencia_senial = abs(frecuencia_senial[0:mitad_longitud]) / longitud_senial
```

CUADRADO DE LA TRANSFORMADA

```
# Cuadrado
frecuencia_senial **= 2

# Extrae la longitud de la señal de frecuencia transformada
len_fts = len(frecuencia_senial)
```

- Extraemos la longitud de la señal de frecuencia transformada

```
# 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
```

- Se ajusta la señal para casos pares e impares mediante la longitud de la señal de frecuencia transformada

EXTRACCIÓN DE VALOR Y CREACIÓN EJE X

```
# Extraer el valor de potencia en dB
potencia_senial = 10 * np.log10(frecuencia_senial)
```

❖ Definimos una variable para guardar el valor de la potencia mediante numpy

```
# Construir el eje X
eje_x = np.arange(0, mitad_longitud, 1) * (frecuencia_de_muestreo / longitud_senial) / 1000.0
```

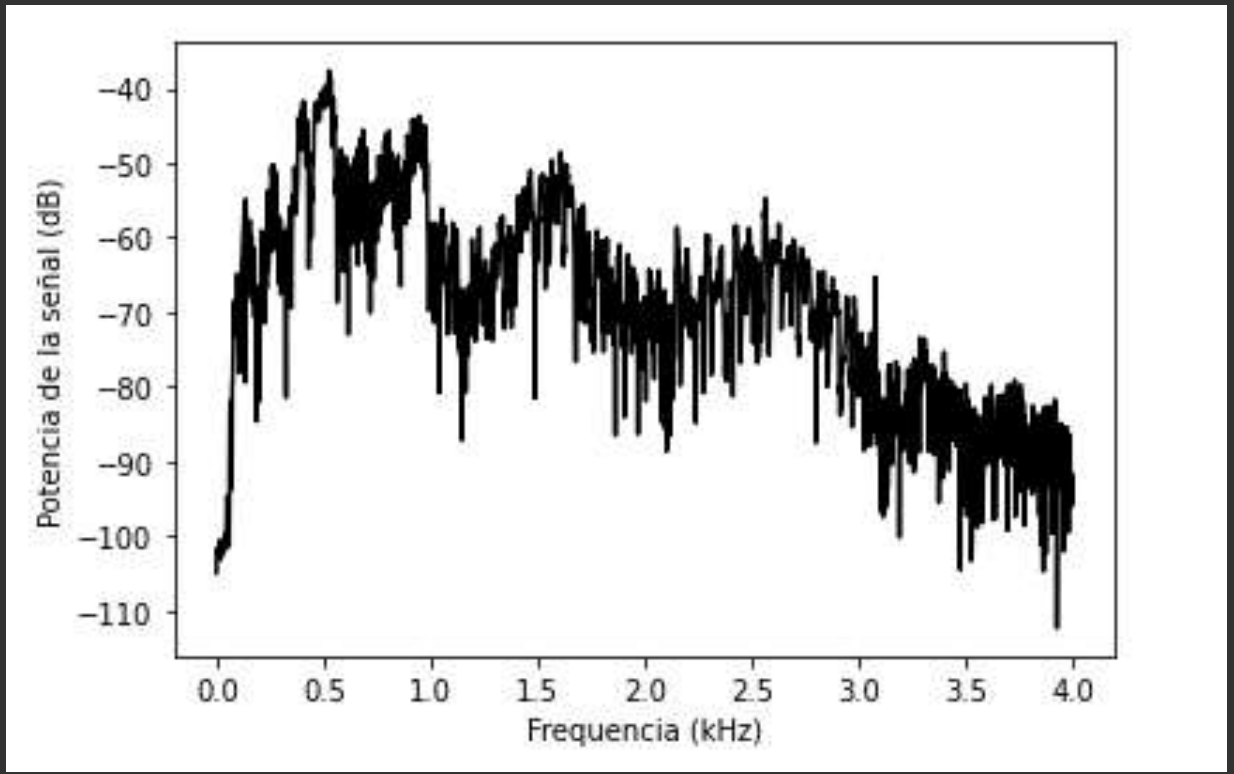
❖ Creación del eje x, dividendo entre 1000 para acomodar la frecuencia en kHz

GRAFICACIÓN DE LA FIGURA

```
# 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()
```

- Graficamos estableciendo el color, los títulos de cada eje y por último ejecutando plt.show()

RESULTADO FINAL



GENERAR AUDIO

```
# 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
```

- Especificamos los parámetros de salida que tendrá el audio que vamos a generar.

```
# Generar la señal de audio
t = np.linspace(valor_minimo, valor_maximo, duracion * frecuencia_muestreo)
senial = np.sin(2 * np.pi * frecuencia_tono * t)
```

- Generamos la señal de audio mediante la función de numpy con sus valores de entrada

AGREGAR RUIDO A LA SEÑAL

```
# Agregar algún ruido a la señal
ruido = 0.5 * np.random.rand(duracion * frecuencia_muestreo)
senial += ruido
```

- Creamos la variable ruido con la función rand de numpy, incluyendo la duración y la frecuencia de muestreo. Después se le suma a la variable senial el 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)
```

- Escalamos los valores enteros de 16 bits

ALMACENAR LA SEÑAL DE AUDIO

```
# Almacenar la señal de audio en el archivo de salida
write(archivo_salida, frecuencia_muestreo, senial_escalada)
```

- Almacenamos la señal de audio en el archivo de salida que definimos anteriormente

```
# Extraer los primeros 200 valores de la señal de audio
senial = senial[:200]
```

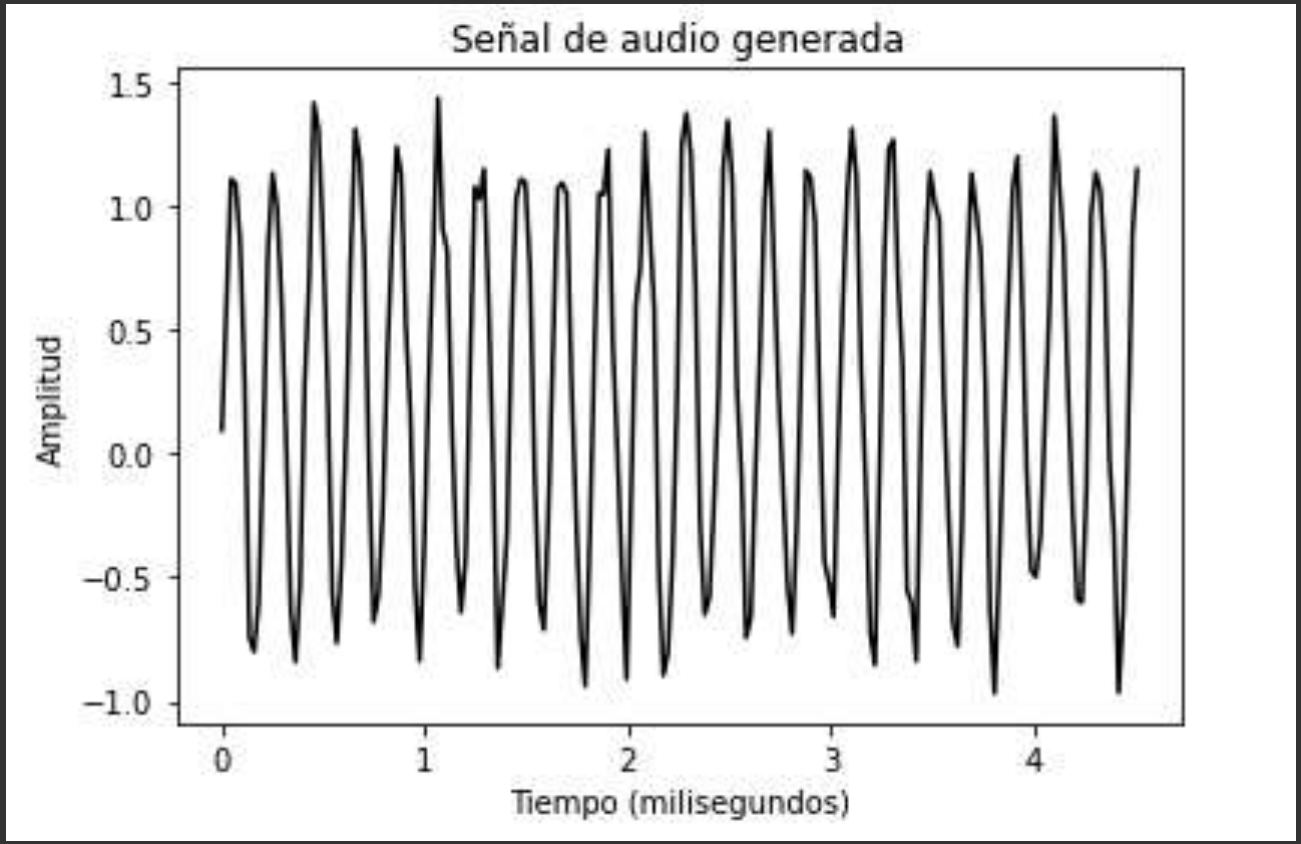
- Se extraen los primeros 200 valores de la señal de audio

CONSTRUCCIÓN DEL EJE X Y GRAFICACIÓN

```
# 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()
```

- Construimos el eje x para así transformar el tiempo a milisegundos, después graficamos la señal de audio utilizando la librería matplotlib, indicando los ejes y el titulo de grafica.

RESULTADO FINAL



SINTETIZAR TONOS

```python
# 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)
```

- Definimos la clase sintetizador_tono con unos parámetros de entrada, esenciales para la construcción del audio.

FUNCION MAIN

```
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'
```

- Creamos los dos archivos de salida, uno para el tono generado y otro para la secuencia de tono generada.
- Llamamos el archivo tone_mapping.json, este nos ayudará para el mapeo del tono

IMPORT DE MAPA DE TONO O FRECUENCIA

```
# 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())
```

- Cargamos el archivo_mapeo en modo lectura y creamos la variable mapa_tonos la cual contiene las lecturas de json

```
# Configure los parámetros de entrada para generar el tono 'F'
nombre_tono = 'F'
duracion = 3     # segundos

amplitud = 12000
frecuencia_muestreo = 44100    # Hz
```

- Configuramos los parámetros de entrada para generar el tono 'F'

EXTRAE LA FRECUENCIA DEL TONO

```
# Extrae la frecuencia del tono
frecuencia_tono = mapa_tonos[nombre_tono]
```

- Extrae la frecuencia del tono asignando el valor a la variable llamada frecuencia_tono

```
# Genere el tono usando los parámetros anteriores
tono_sintetizado = sintetizador_tono(frecuencia_tono, duracion, amplitud, frecuencia_muestreo)
```

- Se genera el tono utilizando los parámetros y llamando a la clase sintetizador_tono

ESCRITURA DE ARCHIVO DE AUDIO

```
# Escribe la señal de audio en el archivo de salida.
write(archivo_tono_generado, frecuencia_muestreo, tono_sintetizado)
```

- Escribe la señal de audio en el archivo salida para guardarlo con los parámetros.

```
# 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)]
```

- Define la secuencia de tonos junto con las duraciones correspondientes en segundos, esto para establecer unos parámetros para construir una secuencia.

CONSTRUCCIÓN DE LA SEÑAL DE AUDIO BASÁNDOSE EN LA SECUENCIA ANTERIOR

```
# 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)
```

GUARDADO DE AUDIO

```
# Guarda el audio en el archivo de salida
write(archivo_secuencia_tono_generada, frecuencia_muestreo, senial)
```

- Guardamos el audio en el archivo que definimos anteriormente, enviando los parámetros necesarios

RECONOCER PALABRAS

```
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
```

- Importamos las librerías necesarias

- Definimos una función para analizar los argumentos de entrada

```
# Define una función para analizar los argumentos de entrada
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Trains the HMM-based speech \
            recognition system')
    parser.add_argument("--input-folder", dest="input_folder", required=True,
            help="Input folder containing the audio files for training")
    return parser
```

DEFINICIÓN DE CLASE

```python
# 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)
```

- Definimos la clase para el entrenamiento de hmm con unos valores de entrada esenciales.

```python
# '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)
```

- Definimos el método train para trabajar con numpy 2D

```python
# corre el modelo HMM para realizar inferencia sobre la entrada de datos
def compute_score(self, input_data):
    return self.model.score(input_data)
```

- Definimos el método compute_score para realizar la inferencia sobre la entrada de datos (input_data)

```python
# 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 = []
```

- Definimos la función para construir un modelo para cada palabra, recibe como entrada input_folder

ANALIZAR 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]
```

- Creamos un ciclo for donde recorre el directorio local para obtener el nombre del subfolder.
- Si aun no encuentra el nombre entonces debe continuar buscando, pero si lo encuentra extrae la etiqueta e inicializa las variables

- Se crea una lista de archivos a ser utilizados para el entrenamiento y se deja un archivo folder para validación

ITERACIÓN PARA CONSEGUIR LOS MODELOS

```
# 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 de 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)
```

- Iteramos a través de la variable anterior llamada training_files
- Extraemos el path actual
- Se lee la señal de audio desde el archivo de entrada
- Se extraen las características MFCC
- Se agrega a la variable X

CREACIÓN DE LOS MODELOS

```
    # 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
```

- Se crea el modelo HMM
- Se entrena el modelo HMM
- Se almacena el modelo para la palabra actual
- Se reinicia la variable
- Y por último se retorna speech_models para cerrar la función build_models

PRUEBAS DE ARCHIVOS DE ENTRADA

```
# Definir una función para ejecutar pruebas en archivos de entrada
def run_tests(test_files):
    # Clasificar datos de entrada
    for test_file in test_files:
        # Leer archivo de entrada
        sampling_freq, signal = wavfile.read(test_file)

        # Extraer características de MFCC
        with warnings.catch_warnings():
            warnings.simplefilter('ignore')
            features_mfcc = mfcc(signal, sampling_freq)

        # Definir variables
        max_score = -float('inf')
        output_label = None
```

- Definimos una función para ejecutar pruebas en archivos de entrada
- Clasificamos los datos de entrada
- Leemos el archivo de entrada
- Extraemos características de MFCC
- Definimos variables

ELECCIÓN DE PUNTUACIÓN MÁS ALTA

```
# Ejecute el vector de características actual a través de todos los HMM
# modelos y elija el que tenga la puntuación más alta
for item in speech_models:
    model, label = item
    score = model.compute_score(features_mfcc)
    if score > max_score:
        max_score = score
        predicted_label = label

# Imprima 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('Predicted:', predicted_label)
```

- Ejecutamos el vector speech_models y vamos registrando el score de cada uno
- Procedemos a comparar el score con el máximo y guardamos la predicción
- Después de salir del ciclo imprimimos la salida prevista

CONSTRUCCIÓN MAIN

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    input_folder = args.input_folder

    # Construya un modelo HMM para cada palabra
    speech_models = build_models(input_folder)

    # Archivos de prueba: el archivo número 15 en cada subcarpeta
    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)
```

- Definimos el método main para la ejecución del código, creamos la variable args y el input_folder
- Construimos un modelo HMM para cada palabra
- Utilizamos los archivos de prueba y los registramos en test_files
- Ejecutamos cada archivo de prueba con la función run_tests(test_files)