U03 PRÁCTICA 4

AUDIO A TEXTO CON MODELOS OCULTOS DE MARKOV

SISTEMAS DE APRENDIZAJE AUTOMÁTICO

IES SERRA PERENXISA
TORRENT (VALENCIA)

MODELOS OCULTOS DE MARKOV

ENTREGA 1: Debes grabarte pronunciando los números del 0 al 9 al menos dos veces. Cada número estará en su propio fichero .wav con los nombres:

`"<digito>-<primeras3LetrasTuNombre><primeras3LetrasTusApellidos>-<contador>.wav"`

Deja los archivos en la carpeta compartida de aules (usa este enlace). Por ejemplo, si yo participara, dejaría los ficheros `"0-JOSROSROD-0.wav"` y `"0-JOSROSROD-1.wav"` para el cero, `"1-JOSROSROD-0.wav"` y `"1-JOSROSROD-1.wav"` para el 1, y así sucesivamente hasta llegar al 9 para el que dejaría los ficheros `"9-JOSROSROD-0.wav"` y `"9-JOSROSROD-1.wav"`. Yo he aportado 10 ficheros (aunque cada alumno debe crear 20 y llamarlos con ese formato para simplificar la creación del dataset).

ACTIVIDAD 1: COMPRENDER EL AUDIO.

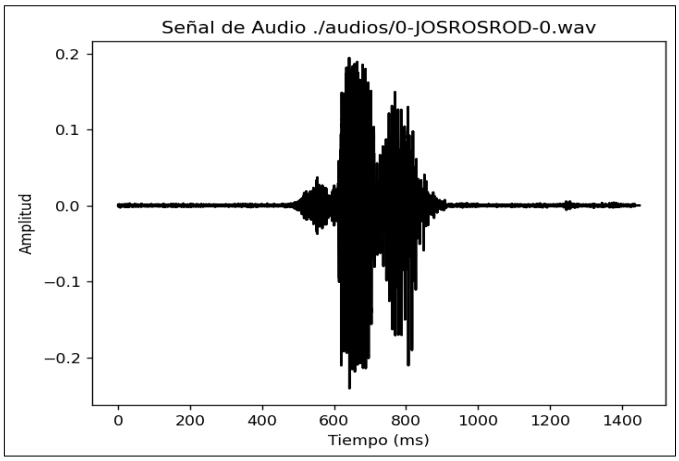
Un fichero de audio contiene mediciones en binario (muestras) de la fuerza de la vibración del aire medida cada cierto tiempo. A la cantidad de veces que se mide el audio cada segundo se le llama frecuencia de muestreo (***sampling rate***) y se mide en ***Hertzios*** (***Hz***). Un Hz es 1 vez por segundo y tiene múltiplos: *1KHz*. Son 1000 Hz y *1MHz* es un millón de Hz., etc. Si la frecuencia de muestreo es *44100 Hz*, un audio de 60 segundos contiene 2.646.000 muestras o mediciones de la fuerza del sonido.

Crea el fichero **u03_p04_actividad1.py** en la carpeta donde tengas descargada los ficheros de audios (supongamos que se llama *"audios"*) y escribe en él el siguiente código:

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile

fichero_audio = "./audios/0-JOSROSROD-0.wav"
frecuencia, audio = wavfile.read(fichero_audio)
duracion = round(audio.shape[0] / float(frecuencia), 3)
print( '\nMuestras:', audio.shape)
print ('Tipo de dato de cada muestra:', audio.dtype)
print ('Duración:', duracion, 'segundos')
```

Si abrimos un fichero de audio podemos conocer su frecuencia de muestreo y su duración. Cada muestra es un valor de 16 bits que al representarse en complemento a 2 usa 15 bits para el valor y un bit para el signo. Si normalizamos las muestras (reducimos los valores a [-1,1]) y mostramos las primeras 240 muestras, aproximadamente mostramos los primeros 5 milisegundos del audio.



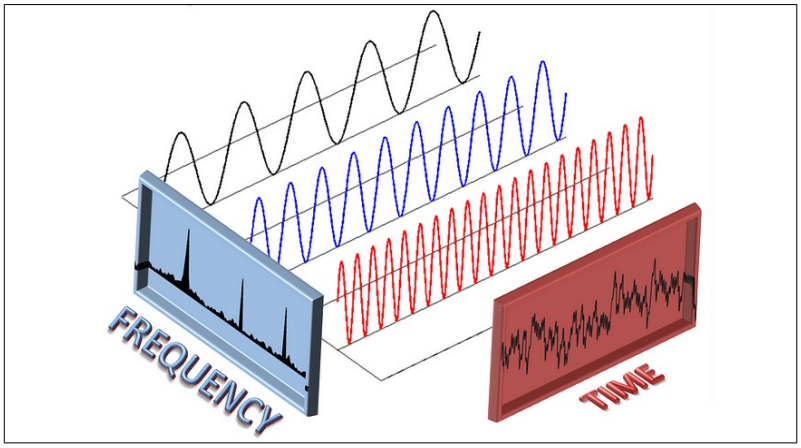

```
audio= audio / 2.**15 # Normalizar
audio1= audio[:240]   # Dibujar 240 muestras
x_val = np.arange(0, len(audio1), 1) / float(frecuencia)
x_val *= 1000
plt.plot(x_val, audio1, color='black')
plt.xlabel('Tiempo (ms)')
plt.ylabel('Amplitud')
plt.title('Señal de Audio ' + fichero_audio)
plt.show()
```

ENTREGA 2: Copia el código y lo modificas para que coja tu fichero cuando pronuncias el cero.

a) ¿La onda de tu gráfico es exactamente igual que la de mi pronunciación o tiene diferencias?

b) Entrega el gráfico de tu onda de audio junto a la mía.

El gráfico del audio que ha generado el código se dice que está en el dominio del tiempo. En estas representaciones vemos como en un punto concreto del espacio va cambiando la energía (amplitud) de la señal a medida que cambia el tiempo. Por eso el tiempo está en el eje X (horizontal) y la energía en el eje vertical (Y). Pero las señales a veces son más fáciles de estudiar en lo que se denomina el ***dominio de las frecuencias***, para lo cual debemos romper nuestra onda original en muchas ondas elementales a distintas frecuencias y amplitudes de manera que al sumarlas nos generan de nuevo nuestra onda como se intuye en la figura de abajo. A las ondas elementales se les denomina ***armónicos***. El dominio de las frecuencias nos dice cuanta energía lleva cada armónico, que tendrá frecuencias diferentes al resto de armónicos. Ahora en el eje X se representa el valor de la frecuencia, no el tiempo. En el eje Y sigue estando la energía. Y a estos gráficos se les llama ***espectros de la señal***.



¿Cómo cambiamos al dominio de las frecuencias? ¿Cómo calcular esos armónicos? Podemos usar una operación denominada ***transformada***. Hay muchas transformadas, cada una con sus características, ventajas e inconvenientes como por ejemplo la de *Fourier*, la del *coseno*, la de *Laplace*, etc.

La ***Transformada de Fourier*** es una operación que descompone cualquier señal en ondas elementales de tipo seno y coseno. Esta operación también tiene una inversa que es la ***Transformada Inversa de Fourier*** que hace el proceso contrario, unir todos los armónicos y volver a transformarlos al dominio del tiempo. Para señales continuas en el tiempo (como el audio) tiene estas fórmulas:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\exp(-j\omega t)dt$$

(1)

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\exp(j\omega t)d\omega$$

En `Python` podemos calcularla usando `Numpy` que implementa el algoritmo **`Fast Fourier Transform (FFT)`**, una versión rápida:

```
audio_transformado = np.fft.fft(audio)
```

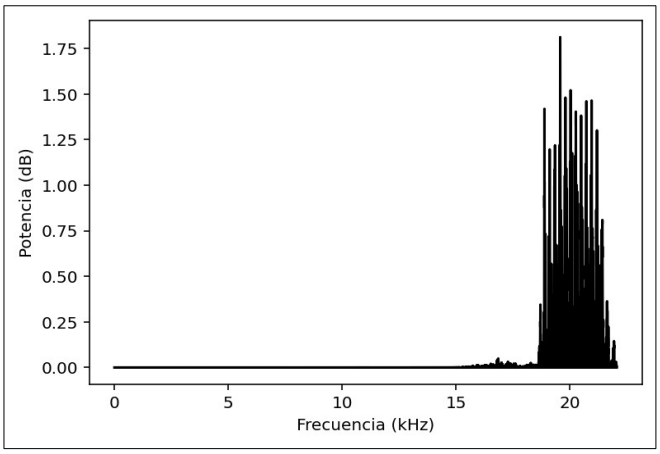
La ***`FFT`*** recibe una forma de onda de números complejos (por ejemplo: `[.5+.1j, .4+.7j, .4+.6j, …]`) y devuelve otra secuencia de números complejos. Pero si la entrada es real (no tiene valores con parte imaginaria) la ***`FFT`*** se aprovecha de que hay una simetría en el eje 0 y solamente hay que calcular las frecuencias mayores de cero.

Los valores devueltos por la ***`FFT`*** son complejos (tienen parte real y parte imaginaria, pueden expresarse como magnitud y fase). Para señales de audio lo más interesante es la magnitud que es principalmente lo que oímos, aunque la fase puede ser también muy importante en algunos tipos de problemas.

```
audio_transformado = np.fft.fft(audio)          # FFT
mitad = int(np.ceil( (len(audio) + 1) / 2.0))
audio_transformado = abs(audio_transformado[0:mitad])
audio_transformado **= 2
```

Ahora tenemos la potencia de la señal y la vamos a expresar en decibelios y representamos toda la onda de sonido pero en el dominio de las frecuencias. Cada onda tiene un patrón de ondas más fácil de diferenciar de otra que en el dominio del tiempo.

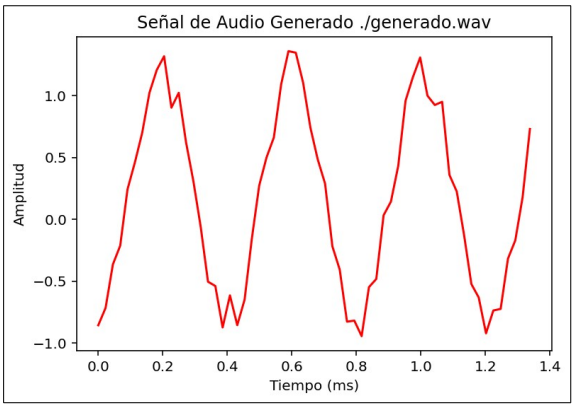
```
potencia = 20 * np.log10(audio_transformado + 1)
x_val = np.arange(0, mitad, 1) * (frecuencia / len(audio) / 1000.0)
plt.figure()
plt.plot(x_val, potencia, color='black')
plt.xlabel('Frecuencia (kHz)')
plt.ylabel('Potencia (dB)')
plt.show()
```



GENERAR NUESTRO PROPIO AUDIO

Para generar una onda de sonido podemos usar la función seno e ir generando valores a lo largo del eje tiempo a una frecuencia constante ***y=sin(2π f t) + ruido*** a la que añadimos algún ruido para que parezca más natural y no demasiado perfecta y sintética.

```
duracion = 3
frecuencia_tono = 587
min_val = -2 * np.pi
max_val = 2 * np.pi
t = np.linspace(min_val, max_val, duracion * frecuencia)
audio = np.sin(2 * np.pi * frecuencia_tono * t)
ruido = 0.4 * np.random.rand(duracion * frecuencia)
audio += ruido
escala = pow(2, 15) - 1
audio_normalizado = audio / np.max( np.abs(audio) )
audio_escalado = np.int16(audio_normalizado * escala)
x_val = np.arange(0, len(audio), 1) / float(frecuencia)
x_val *= 1000
plt.plot(x_val[:60], audio[:60], color='red')
plt.xlabel('Tiempo (ms)')
plt.ylabel('Amplitud')
# Guardar el audio a un fichero
fichero_salida = './generado.wav'
wavfile.write(fichero_salida, frecuencia, audio_escalado)
plt.title('Señal de Audio Generado ' + fichero_salida)
plt.show()
```



EXTRACCIÓN DE CARACTERÍSTICAS DESDE UN FICHERO DE AUDIO

Para construir un sistema de reconocimiento de voz, lo primero que tenemos que hacer es extraer las características importantes y eliminar el ruido del audio. Generalmente, el sonido producido por los humanos se genera en las cuerdas vocales y la boca, el tracto vocal humano emite una ***envoltura del ESPECTRO DE POTENCIA*** de poca duración como el gráfico frecuencia-potencia que hemos generado antes. Nosotros usaremos valores ***Mel*** (***Coeficientes cepstral de frecuencia: MFCC***) para representar con precisión esa envolvente del espectro.

¿Cómo extraer los ***coeficientee cepstrales de frecuencias Mel (MFCC) de un audio***?

1. Trocear la señal de audio en este caso en fragmentos (frames o ventanas) cortos.
2. Para cada frame, se calcula la estimación del periodograma de la potencia del espectro.
3. Aplicar el banco de filtros ***mel*** a los espectros de potencia y sumar la energía de cada filtro.
4. Aplicar el logaritmo de todas las energías del banco de filtros.
5. Aplicar la ***DCT (Transformada Directa del Coseno***) de las energías del banco de filtros de registro.
6. Quedarse con los coeficientes DCT2 al DCT13 y descartar el resto.

Si todo esto parece chino mira el vídeo de este enlace. Si aún no estás del todo confortable tras explorar estas sugerencias, no te preocupes. La idea es que una señal de audio cambia constantemente en el tiempo. Por eso se divide la señal en frames de entre 20 a 40 ms. El siguiente paso es calcular el espectro de potencia de cada frame. El motivo de esto es la cóclea humana (un órgano del oído) que vibra en diferentes puntos dependiendo de la frecuencia de los sonidos entrantes. Este efecto se vuelve más pronunciado a medida que aumentan las frecuencias. Por esta razón, tomamos grupos de frames de voz y los resumimos para tener una idea de cuánta energía existe en varias frecuencias muy importantes para lo que oímos. Esto lo realiza el banco de filtros ***Mel***. Sólo nos interesa aproximar cuánta energía ocurre en cada uno de esos puntos. No hay o no conozco ningún criterio que te pueda indicar qué tamaño de frames o cuántos factores de Mel debes usar pero la escala ***Mel*** nos ayuda usando los bancos de filtros y cómo de anchos debemos hacerlos.

¿Qué es la escala *Mel*?

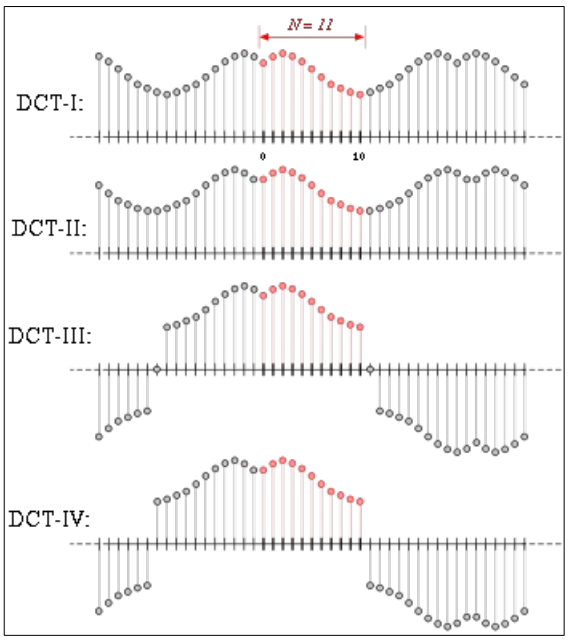
Relaciona la frecuencia o tono percibido de un tono puro con la frecuencia real medida en el audio. Los humanos somos buenos distinguiendo pequeños cambios de tono en las frecuencias bajas y no tan buenos en las frecuencias altas. La incorporación de esta escala hace que las características que vamos a extraer del audio se parezcan más a lo que escuchan nuestros oídos. La fórmula para convertir de frecuencia a escala *Mel* es:

$$M(f) = 1125 * \ln(1 + f/700)$$

El paso final es calcular la ***DCT*** (*Transformada Discreta del Coseno*) del logaritmo del banco de energías filtradas. Las razón más importante para hacer esto es porque las energías de nuestros filtros están todas solapadas y muy correlacionadas unas con otras. La *DCT* elimina estas correlaciones de energías de manera que la matriz de covarianzas puede utilizarse como modelo de las características en por ejemplo un clasificador ***HMM*.**

¿Qué es la *DCT*?

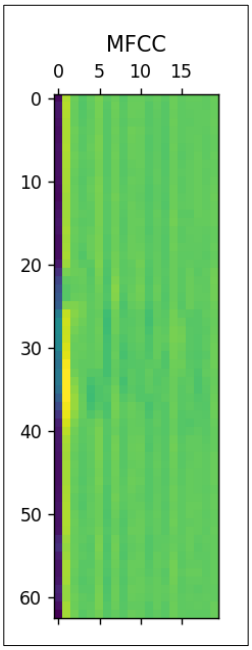
Está relacionada con la transformada discreta de *Fourier* y fue propuesta en 1974 por el ingeniero *Nasir Ahmed*. Como cualquier transformada descompone una función o señal como la suma de otras funciones o señales de tipo sinusoidal con diferentes frecuencias y amplitudes. Como la *DFT*, la *DCT* trabaja como una función aplicada a un número finito de puntos de datos discretos. La diferencia es que la *DCT* solamente usa funciones elementales de tipo coseno y por tanto números reales mientras que *DFT* usa tanto senos como cosenos (en forma de exponenciales de números complejos). Sin embargo, la diferencia más importante es que cada transformada se comporta de una manera diferente, *DCT* por ejemplo resume más la energía (comprime mejor) y no falla en los bordes de las señales.



Y lo mejor de todo es que hay muchas librerías que nos calculan las características usando tanto *DCT* como MFCC, una de ellas se llama ***Librosa***. Si quieres saber más de la *Transformada Discreta del Coseno* y en sus aplicaciones para trabajar con audio, imágenes y vídeo puedes consultar *wikipedia*: https://en.wikipedia.org/wiki/Discrete_cosine_transform

```
# Extraer las características de un audio
from librosa.feature import mfcc
import librosa
audio, frecuencia = librosa.load(fichero_audio)
caracteristicas_mfcc = mfcc(sr=frecuencia, y=audio)
print("Número de frames =", caracteristicas_mfcc.shape[0])
print('Longitud de cada característica =', caracteristicas_mfcc.shape[1])
# Dibujarlas
caracteristicas_mfcc = caracteristicas_mfcc.T
plt.matshow(caracteristicas_mfcc)
plt.title('MFCC')
plt.show()
```

```
Muestras: (63920, 2)
Tipo de dato de cada muestra: int16
Duración: 1.449 segundos
Número de frames = 20
Longitud de cada característica = 63
```



ENTREGA 3: Haz lo mismo con tu fichero de audio del número cero y entregas captura del resultado de la ejecución.

EL PAPEL DEL MODELO HMM

El modelo *HMM* es un modelo probabilístico generativo, en el que una secuencia de variables observables ***X*** se genera mediante una secuencia de estados ocultos ***Z***. Los estados ocultos no son

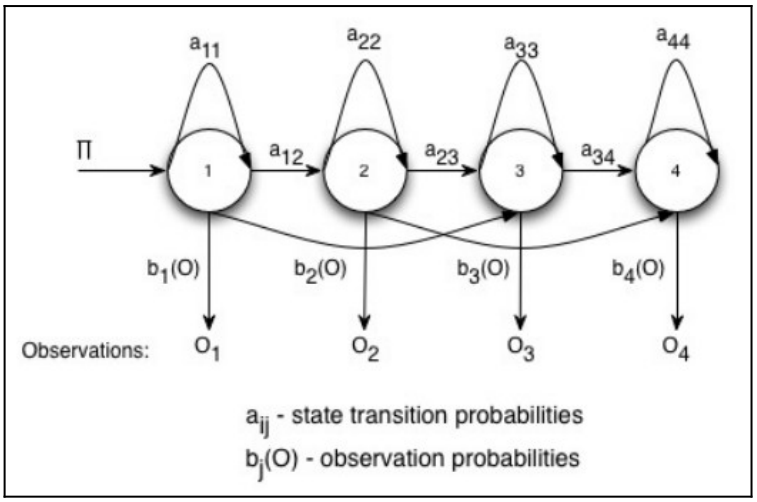
directamente observados. Las transiciones entre estados ocultos se asume que tienen la forma de una cadena de Markov (primer orden). Pueden especificarse mediante un vector de probabilidades de inicio $\boldsymbol{\pi}$ y una matriz de probabilidades de transición $\boldsymbol{A}$. La probabilidad de emisión de observable puede ser cualquier distribución de probabilidades con parámetros $\boldsymbol{\theta}$ condicionado al actual estado oculto. El *HMM* es completamente determinado por $\boldsymbol{\pi}$, $\boldsymbol{A}$ y $\boldsymbol{\theta}$.

Las cadenas de Markov (y los *HMM*) modelan secuencias con estados discretos. Dada una secuencia, podríamos querer saber por ejemplo cuál es el símbolo más probable que vendrá a continuación, o cuál es la probabilidad de que aparezca una nueva secuencia determinada. Las cadenas de Markov nos dan una forma de responder a estas preguntas.

$$P(X_1 X_2 X_3 \ldots X_N) = P(X_1) \prod_{i=1}^{N} P(X_{i+1}|X_i)$$

¿Porqué usar un *HMM* para reconocer audio?
Las cadenas de Markov solamente funcionan bien si los estados son discretos. La voz satisface esta propiedad porque hay un número finito de características *mfcc* en una secuencia. Si usas una serie temporal continua entonces las cadenas de Markov no se pueden utilizar para reconocimiento de voz.

Los estados son fenómenos por ejemplo un pequeño número de sonidos que pueden ser generados. Las observaciones o símbolos son los frames de audio que se representan usando *MFCC*. Dada una secuencia de *MFCC* por ejemplo de audio, queremos saber a qué secuencia de fonemas pertenece. Una vez que tenemos los fonemas podemos transformarlos a palabras escritas usando un diccionario donde la clave es el fonema y el valor es la palabra. Calcular la probabilidad de las observaciones de un *MFCC* dado un estado es lo que realizamos usando modelos ***Gaussian Mixture Model*** (***GMM***) o bien ***Gaussian*** simples. La diferencia es que los ***Mixture*** asumen que los símbolos emitidos tienen asociado una mezcla de distribuciones aleatorias y los simples solo una.

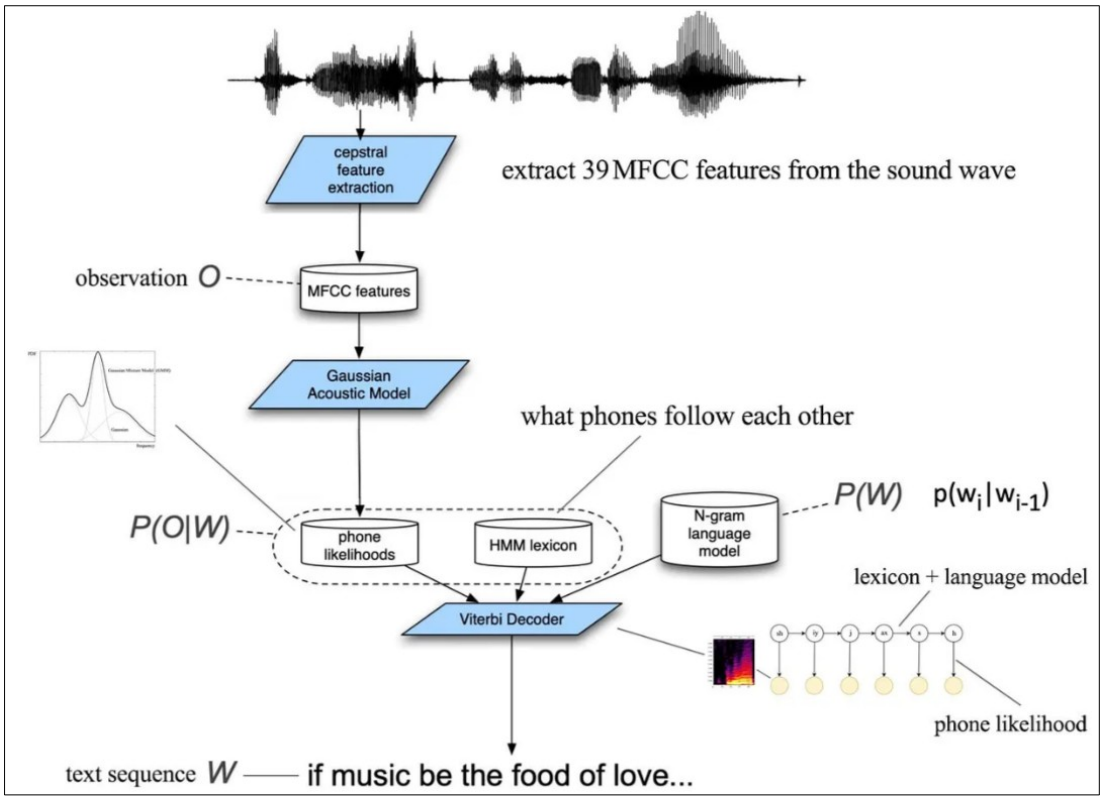


ACTIVIDAD 2: ENTRENAR NUESTRO MODELO.

Los ficheros de audio que tendremos cuando todos hayamos dejado nuestras aportaciones consistirán en varios dicheros `.wav` de cada número del "*0*" al "*9*". En primer lugar instalamos las librerías ***hmmlearn*** y **python_speech_features**.

```
python -m pip list                  # Comprobar si ya las tienes (en entorno virtual)
python -m pip install hmmlearn      # En caso de que no
python -m pip install python_speech_features
```

El esquema general:



Creamos el fichero ***u03_p04_actividad2.py*** donde comenzamos a añadir estas importaciones:

```
# -*- coding: utf-8 -*-
import itertools
import os
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from hmmlearn import hmm
from scipy.io import wavfile
from python_speech_features import mfcc, logfbank, delta
# import librosa                    # Otra posibilidad de leer audio: audio, fm = librosa.read(fichero_audio)
# from librosa.feature import mfcc    # Otra posibilidad de extraer escalas mel del audio
import matplotlib.pyplot as plt
import pickle
import warnings
warnings.filterwarnings("ignore")
```

Definimos variables que podemos manipular para adaptar el código a cada situación, por ejemplo como llegar a la carpeta que contiene los ficheros de audio para entrenar y testar el modelo, los parámetros que podemos usar para crear el dataset, los hiperparámetros del modelo, el propio modelo a utilizar, etc. Copia este código:

```
#============ PARÁMETROS PARA DEFINIR EL DATASET ==========
carpeta = "./audios/"
n_caracteristicas = 10              # Num. de características mel de cada trozo de audio
tama_caracteristicas = 10           # Cantidad de datos de cada característica
valor_imputado= 0.1                 # Mecanismo de seguridad para no alimentar HMM con valores NaN
mucha_informacion = True            # Mostrar progreso de cada acción paso a paso
ratio_de_train = 0.1                # Porcentaje de ejemplos para train ente 0.0 y 1.0
#========== HIPERPARÁMETROS DE LOS MODELOS: ==========
# Hay dos posibles modelos:
#     - hmm.GMMHMM()         Gausiano Mixture con HMM. La emisión de un símbolo se modela como una
#                            distriibución aleatoria que es una mezcla de otras distribuciones.
#     - hmm.GaussianHMM()    Gausiano con HMM. La emisión obedece a una única distribución.
#  Aunque tienen más, solamemnte usaremos estos:
n_estados_ocultos = 1           # Estados ocultos usados en el HMM
n_iteraciones = 10              # Iteraciones en el algoritmo (viterbi por defecto)
#=====================================================
```

En primer lugar preparamos los datos para entrenar y testar en una función:

```python
def definir_dataset(ruta_audios=carpeta):
    ficheros = sorted(os.listdir(ruta_audios))
    x_train = []
    y_train = []
    x_test = []
    y_test = []
    datos = dict()
    n = len(ficheros)
    digito_actual = " "
    for i in range(n):
        if not ficheros[i].lower().endswith('.wav'):
            continue
        digito = ficheros[i][0]    # digito-nombre.wav
        if mucha_informacion and digito_actual != digito:
            digito_actual = digito
            print("Procesando ficheros del dígito " + digito)
        carac = extraer_caracteristicas(ruta_audio=ruta_audios + ficheros[i])
        if digito not in datos.keys():
            datos[digito] = [carac]
            x_train.append(carac)
            y_train.append(digito)
        else:
            if np.random.rand() < ratio_de_train: # Añadirlo a train o a test
                x_test.append(carac)
                y_test.append(digito)
            else:
                datos[digito].append(carac)
                x_train.append(carac)
                y_train.append(digito)
    return x_train, y_train, x_test, y_test, datos
```

El método anterior utiliza a la función ***extraer_caracteristicas()*** cuyo código es el siguiente. Observa que puedes controlar la cantidad de características a extraer. La cantidad de información con la que alimentas al *HMM* es importante porque influye en la cantidad de estados que puedes usar (muchos o pocos y obtendrás errores de probabilidades o valores *NaN* en estados iniciales, o estados a los que no se puede saltar o fallos de convergencia del algoritmo).

```python
def extraer_caracteristicas(ruta_audio):
    # audio, frec_muestreo = librosa.load(ruta_audio)
    # mfcc_carac = mfcc(y=audio, sr=frec_muestreo, n_mfcc=n_caracteristicas)
    frec_muestreo, audio = wavfile.read(ruta_audio)
    mfcc_carac= mfcc(audio, frec_muestreo, nfft=2048, numcep=n_caracteristicas, nfilt=tama_caracteristicas)
    if mucha_informacion:
        print(f"     {ruta_audio} Dimensiones de mfcc {mfcc_carac.shape}")
    return mfcc_carac
```

El entrenamiento lo realizaremos en otra función de Python:

```python
def entrenar_modelo(datos):
    hmm_aprendido = dict()
    for label in datos.keys():
        # modelo= hmm.GMMHMM(n_components=n_estados_ocultos, n_iter=n_iteraciones)
        modelo = hmm.GaussianHMM(n_components=n_estados_ocultos, n_iter=n_iteraciones, verbose=mucha_informacion)
        caracteristica = None
        for cada_caracteristica in datos[label]:
            cada_caracteristica = np.nan_to_num(cada_caracteristica, nan=valor_imputado)
            if caracteristica is None:
                caracteristica =  cada_caracteristica
            else:
                caracteristica = np.vstack((caracteristica, cada_caracteristica))
        obj = modelo.fit(caracteristica)
        if mucha_informacion:
            print("**** Modelo de", label)
            print("Prob. de inicio:", obj.startprob_)
            print("Matriz de transición:\n", obj.transmat_)
        hmm_aprendido[label] = obj
    return hmm_aprendido
```

Cuando necesitemos hacer predicciones utilizaremos la siguiente función:

```
93    def hacer_prediccion(datos_test, entrenado):
94        label_predicha = []                          # predecir una lista de test
95        nombres = []
96        if type(datos_test) == type([]):
97            for test in datos_test:
98                scores = []
99                for nodo in entrenado.keys():
100                   scores.append(entrenado[nodo].score(test))
101                   nombres.append(nodo)
102               label_predicha.append(scores.index(max(scores)))
103       else:
104           scores = []
105           for nodo in entrenado.keys():
106               scores.append(entrenado[nodo].score(datos_test))
107               nombres.append(nodo)
108           label_predicha.append(scores.index(max(scores)))
109       return nombres[label_predicha[0]]
```

En los test dibujaremos la matriz de confusión con esta función:

```
111   def plot_matriz_confusion(cm, clases, normaliza=False, titulo='Matriz de Confusión', cmap=plt.cm.Blues):
112       plt.imshow(cm, interpolation='nearest', cmap=cmap)
113       plt.title(titulo)
114       plt.colorbar()
115       tick_marks = np.arange( len(clases) )
116       plt.xticks(tick_marks, clases, rotation=45)
117       plt.yticks(tick_marks, clases)
118       fmt = '.2f' if normaliza else 'd'
119       limite = cm.max() / 2.
120       for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
121           plt.text(j, i, format(cm[i, j], fmt),
122                    horizontalalignment="center",
123                    color="white" if cm[i, j] > limite else "black")
124       plt.tight_layout()
125       plt.ylabel('Label real')
126       plt.xlabel('Label predicha')
127       plt.show()
```

Y una función para imprimir la matriz de confusión, un informe individual de cada dígito y opcionalmente su gráfico:

```
129   def informe(y_test, y_pred, mostrar_grafico=True):
130       print("Matriz de confusión:\n\n", confusion_matrix(y_test, y_pred))
131       print("-------------------------------------------------------")
132       print("Informe de clasificación:\n\n", classification_report(y_test, y_pred))
133       print("-------------------------------------------------------\n")
134       print("Accuracy:", accuracy_score(y_test, y_pred))
135       print("-------------------------------------------------------\n")
136       if mostrar_grafico:
137           plot_matriz_confusion(confusion_matrix(y_test, y_pred), [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

El programa que llama a todas estas funciones está dividido en tres partes. Una primera parte en la cual preparamos el dataset:

```
139   # PASO 1: Definir el dataset
140   print("====== PASO 1: Definir el Dataset ======")
141   x_train, y_train, x_test, y_test, datos = definir_dataset()
142   print("  Datos de entrenamiento: ", len(x_train))
143   print("  Datos de test: ", len(x_test))
144   print("  Diccionario de datos contiene datos de:", datos.keys())
```

Una segunda parte donde entrenamos el modelo y lo guardamos en disco:

```
146   # PASO 2: Entrenar el modelo
147   print("====== PASO 2: Entrenar el modelo ======")
148   hmm = entrenar_modelo(datos)
149   # Guardar modelo
150   with open("modelo_aprendido.pkl", "wb") as fichero:
151       pickle.dump(hmm, fichero)
152   print("   Entrenamiento realizado...")
```

Una tercera parte donde realizamos el test cargando primero el modelo almacenado en disco, simulando lo que se haría en caso de usarlo en producción:

```
154   # PASO 3: Usar el modelo para predecir
155   print("====== PASO 3: Usar el modelo ======")
156   # Leer modelo de disco si queremos usarlo tras crearlo
157   with open("u03_p04_modelo_hmm.pkl", "rb") as fichero:
158       hmm = pickle.load(fichero)
159   ficheros = sorted(os.listdir(carpeta))
160   tot_test = 0
161   tot_train = 0
162   n = len(x_test)
163   m = len(x_train)
164   pred_test = []
165   pred_train = []
166   for i in range(m):
167       y_pred = hacer_prediccion(x_train[i], hmm)
168       if y_pred == y_train[i]:
169           tot_train += 1
170       pred_train.append(y_pred)
171   for i in range(n):
172       y_pred = hacer_prediccion(x_test[i], hmm)
173       if y_pred == y_test[i]:
174           tot_test += 1
175       pred_test.append(y_pred)
176   informe(y_train,pred_train)
177   informe(y_test,pred_test)
178   print('######################### TRAIN ACCURACY #########################')
179   print(tot_train/m)
180   print('#########################  TEST ACCURACY #########################')
181   print(tot_test/n)
```

ENTREGA 4: Entrega:

a) El código del programa Python.

b) Captura de pantalla del resultado de una ejecución.

ACTIVIDAD 3: ELEGIR MODELO, DATASET E HIPERPARÁMETROS.

Ahora debes intentar mejorar todo lo que puedas la configuración para alcanzar una accuracy de alrededor del 80% en los datos de test. Para ello deberás ir probando diferentes configuraciones de:

- **Generación del dataset** (aunque los datos son diferentes a los que yo utilicé, su cantidad es similar a la que vosotros tenéis y la calidad de los míos posiblemente peor, siempre que las grabaciones las hagáis de manera razonable y sin haceros trampas (grabo uno, copio y tengo dos audios equivale para el modelo a solo tienes un audio por ejemplo) y también podéis modificar la cantidad de características, el valor imputado a los *NaN* iniciales.
- **Configuraciones de hiperparámetros**: Podéis jugar con el número de estados y el número de iteraciones. Pero si tenéis pocos datos tendréis limitaciones para cambiar estos parámetros.
- **El modelo a elegir**: podéis elegir entre *GMMHMM (Gaussian Mixture Model Hide Markov Model)* y *GaussianHMM (Gaussian HMM)*. Según el modelo Gausiano que se escoja la mejor cantidad de estados del *HMM* también cambiará.

ENTREGA 5: Entrega:

a) Captura de pantalla del resultado de la ejecución con los mejores parámetros.

b) Valor establecido para:

- `n_características`: _____
- `valor_imputado`: _____
- `ratio_de_train`: _____
- `modelo usado`: ( ) `GMMHMM.` ( ) `GaussianHMM.`
- `n_estados_ocultos`: _____
- `n_iteraciones`: _____

c) Valor conseguido de (ver figuras para valores de referencia):

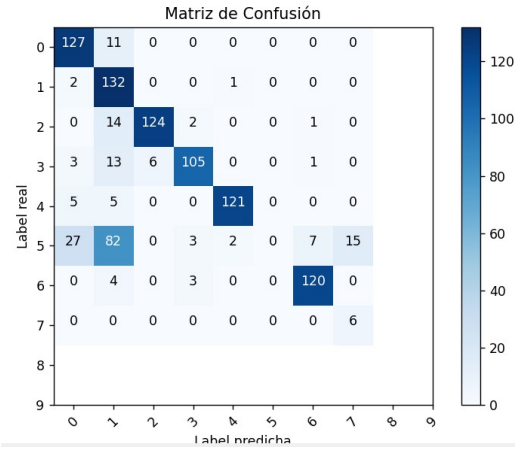
- Accuracy Test (0.81): _____
- Media de precisión (0.7): _____
- Media de recall (0.81): _____
- Media de F1-score (0.75): _____
- support(120): _____

d) Gráfico de la matriz de confusión del test

e) Fichero del mejor modelo generado (*.pkl*).

A mí me ha ido mejor de esta manera (pero ya se sabe: nuevos dato...):

- Usando el Gausiano Mixture.
- Maximizando los datos de entrenamiento: 90% para train (bueno si hay pocos datos).
- Comenzar con pocos estados ocultos (a más estados ocultos, más datos necesita y más iteraciones del algoritmo *Viterbi*) anotar resultados y pegarle un subidón a algo a ver que pasa, luego afinas a la mitad del subidón y vas tanteando resultados.
- Cambiar algo de valor y mantenerlo mientras pruebas varios valores del resto.



Informe de clasificación:

	precision	recall	f1-score	support
0	0.77	0.92	0.84	138
1	0.51	0.98	0.67	135
2	0.95	0.88	0.92	141
3	0.93	0.82	0.87	128
4	0.98	0.92	0.95	131
5	0.00	0.00	0.00	136
6	0.93	0.94	0.94	127
7	0.29	1.00	0.44	6
accuracy			0.78	942
macro avg	0.67	0.81	0.70	942
weighted avg	0.72	0.78	0.74	942

Accuracy: 0.7802547770700637

	precision	recall	f1-score	support
0	0.63	1.00	0.77	12
1	0.61	0.93	0.74	15
2	1.00	1.00	1.00	9
3	0.88	1.00	0.94	22
4	0.95	0.95	0.95	19
5	0.00	0.00	0.00	19
6	0.91	0.91	0.91	23
7	0.50	1.00	0.67	1
accuracy			0.81	120
macro avg	0.69	0.85	0.75	120
weighted avg	0.70	0.81	0.75	120

Accuracy: 0.8083333333333333

########## TRAIN ACCURACY ##########
0.7802547770700637
########## TEST ACCURACY ##########
0.8083333333333333