CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN
INSTITUTO POLITÉCNICO NACIONAL

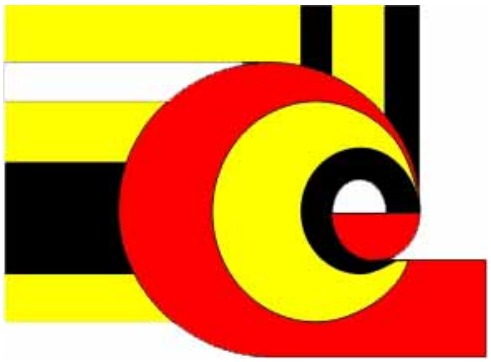
MAESTRÍA EN CIENCIAS EN INGENIERÍA DE CÓMPUTO

PROCESAMIENTO DIGITAL DE SEÑALES

Documentación de los filtros MFCC, Gammatone, Modelo Coclear y Composición.

Autor
LUIS JUVENTINO VELASQUEZ HIDALGO

Profesor
Dr. JOSÉ LUIS OROPEZA RODRÍGUEZ



8 de febrero de 2025

1. Estructura del proyecto

```
Proyecto_Filtros/
├── Caracteristicas_Extraidas/
│   ├── Composicion/
│   ├── Gammatone/
│   ├── MFCC/
│   ├── Modelo_coclear/
│   ├── Composicion.csv
│   ├── Gammatone.csv
│   ├── MFCC.csv
│   └── Modelo_coclear.csv
├── dataset_separado/
│   ├── Train/
│   └── Test/
├── libreria/
│   ├── Caracteristicas_Gammatone.py/
│   ├── Caracteristicas_MFCC.py/
│   ├── Caracteristicas_Mel_Coclea.py/
│   ├── Caracteristicas_Modelo_coclear.py/
│   ├── preprocesamiento.py/
│   └── utileria.py/
├── Reporte/
├── dataset/
├── Comparacion_de_filtros.py
├── Convertir_npy_2_csv.py
├── Entrenar_clusters.py
├── Extraer_caracteristicas.py
├── Partir_dataset.py
└── requerimientos.txt
```

Donde cada carpeta cumple con el siguiente propósito :

- **dataset** : Conjunto de datos con los que se trabajó. Se trabajó con 200 archivos de audio. 20 archivos por cada dígito del 0 al 9.
- **dataset_separado** : Contiene una referencia a los archivos de audio separados en dos carpetas, una para entrenamiento y otra para pruebas.
  - **Train** : 80 % de los archivos de audio, estratificados por dígito. El porcentaje se puede cambiar en el archivo **Partir_dataset.py**.
  - **Test** : Complemento de los archivos de audio que no se usaron en la carpeta Train.
- **Caracteristicas_Extraidas** Directorio que contiene las características por cada tipo de filtro. En formato .npy y .csv.
  - **MFCC** : Contiene un archivo por cada elemento del dataset. Cada archivo contiene las características del audio correspondiente usando el filtro MFCC. El archivo esta en formato .npy

  - **Gammatone** : Idéntico a MFCC pero con las características del filtro Gammatone.
  - **Composicion** : Idéntico a MFCC pero con las características del filtro de la composición de MFCC con el Modelo de la Coclea.
  - **Modelo_coclear** : Idéntico a MFCC pero con las características del filtro Modelo coclear.

- **libreria** : Código con la logica principal de los filtros.

  - **Caracteristicas_MFCC.py** : Funciones necesarias para obtener las características MFCC
  - **Caracteristicas_Gammatone.py** : Funciones necesarias para obtener las características Gammatone
  - **Caracteristicas_Mel_Coclea.py** : Funciones necesarias para extraer las características del filtro resultante de la composición de MFCC con el Modelo de la Coclea.
  - **Caracteristicas_Modelo_coclear.py** : Funciones para extraer las características del filtro Modelo coclear.
  - **preprocesamiento.py** : pre-procesamiento aplicado a todos los archivos
  - **utileria.py** : Funciones auxiliares para : manejar archivos, procesamiento de señales y funciones estadísticas.

Los archivos .py que se encuentran al nivel del directorio *Proyecto_Filtros* están pensado para ser ejecutados desde consola.Cada uno de estos archivos contiene una sección de *Constantes* que se pueden modificar para cambiar el comportamiento del programa, el resto del código no esta pensado para ser modificado.

Uno de las constantes recurrentes es *DIR_BASE* la cual es la ruta donde se encuentra la carpeta *Proyecto_Filtros*.

También se incluye un archivo de requerimientos para instalar las librerías necesarias para ejecutar el código.

2. Libreria

La descripción del pre-procesamiento de las señales y el procedimiento para extraer las características esta descrito en los siguientes archivos:

Preprocesamiento

La función principal de este archivo es *Preprocesamiento*, la cual esta descrita a continuación:

Listing 1: Preprocesamiento.py

```
1 import numpy as np
2
3 def Preprocesamiento(tiempo, señal, SEG_TAMAÑO = 256, SEG_TRASLAPE =
    0.5, PE_ALPHA = 0.97 ):
```

```
4       señal_pre_enfasis       = Filtro_pre_enfasis(señal, PE_ALPHA)
5       segmentos               = Segmentar_señal(señal_pre_enfasis,
            SEG_TAMAÑO, SEG_TRASLAPE)
6       segmentos_ventaneados   = Ventaneo(segmentos, SEG_TAMAÑO)
7
8       return  Segmentar_señal(tiempo), segmentos_ventaneados
```

Como se puede ver en el fragmento de código anterior, el procesamiento consta de 3 etapas, pre-énfasis, segmentado y ventaneo (usando una ventana de HAMING). La implementación de las funciones anteriores se encuentran en el mismo archivo. Se omitió el código porque la implementación es directa.

Caracteristicas_MFCC

La función principal es *Caracteristicas_MFCC*. La función *Matriz_MFCC* sirve como utileria para aplicar la función principal a todo un conjunto de segmentos.

Listing 2: Caracteristicas_MFCC.py

```
1  import matplotlib.pyplot as plt
2  from numpy import ndarray
3  import numpy as np
4
5  from libreria.utileria import Banco_filtros_no_eq,  Energia_salida , FFT
       , TDC
6
7  # Extrae las características MFCC de un conjunto de segmentos
8  def Matriz_MFCC(segmentos : ndarray[ndarray], n_coef : int, fs : int,
       fs_i : int, verbose = 0) -> ndarray[ndarray]:
9
10
11 # Función principal. Extracción de características para un segmento
12 def Caracteristicas_MFCC(segmento, n_coef : int, fs : int, fs_i : int,
       verbose = False):
13      bins, espectro, fs      = FFT(segmento, fs)
14      tam_espectro            = len(espectro)
15
16      frec_centrales          = Centros_MFCC(fs_i, fs, n_coef)
17      banco_filtros_MFCC      = Banco_filtros_no_eq(frec_centrales,
            tam_espectro, fs )
18
19      Y                       = Energia_salida(espectro,
            banco_filtros_MFCC)
20      S                       = np.log(Y)
21
22      caracteristicas_mfcc    = TDC(S, len(S) , n_coef)
23
24      if(verbose):
25          _plot_resumen(segmento,espectro,banco_filtros_MFCC,Y,S,
                caracteristicas_mfcc, bins)
26
27      return caracteristicas_mfcc
```

El proceso de la extracción de características consta de cuatro pasos: El calculo de la FFT, calculo de la energía de salida del espectro multiplicado por el banco de filtros, logaritmo de la señal y finalmente la Transformada Discreta Cosenoidal.

El banco de filtros tiene las frecuencias centrales de cada filtro distribuidas de forma equidistantes en la escala de Mel. Para construir el banco de filtro primero se calculan las frecuencias centrales usando la función de *Centros_MFCC*.

Listing 3: Caracteristicas_MFCC.py

```
def Centros_MFCC(fs_l : int, fs_h : int, n_coef : int) -> list[float]:
    def Hz_to_Mel(hz):
        return 1125 * np.log(1 + hz / 700)

    def Mel_to_hs(ml):
        return 700 * (np.exp(ml / 1125) - 1)

    B       = Hz_to_Mel
    B_1     = Mel_to_hs

    Delta_B = B(fs_h) - B(fs_l)
    B_fl    = B(fs_l)

    puntos  = np.arange(0,n_coef+2)

    coef    = [ B_1( B_fl + punto * (Delta_B / (n_coef + 1)) ) for punto
        in puntos]
    return coef
```

La variable de *puntos* es un arreglo de puntos equidistantes, el cual se utiliza para calcular la n-esima escala de Mel y posteriormente se pasa al dominio de la frecuencia. Se calculan $n + 2$ para que las frecuencias centrales abarquen la frecuencia inicial y la final.

Caracteristicas_Gammatone

Caracteristicas_Gammatone es la función principal del archivo. *Matriz_Gammatone* cumple la función de aplicar la función principal a todo un conjunto de segmentos.

Listing 4: Caracteristicas_gammatone.py

```
import matplotlib.pyplot as plt
from numpy import ndarray
import numpy as np

from libreria.utileria import Banco_filtros_no_eq,  Energia_salida , FFT
    , TDC

def Matriz_Gammatone(segmentos : ndarray[ndarray], n_coef : int, fs :
    int, fs_i : int, verbose = 0) -> ndarray[ndarray]:
    matriz_Gammatone = []

    for segmento in segmentos:
        matriz_Gammatone.append(Caracteristicas_Gammatone(segmento,
            n_coef, fs, fs_i, verbose))
    return np.array(matriz_Gammatone)

def Caracteristicas_Gammatone(segmento, n_coef : int, fs : int, fs_i :
    int, verbose = False):
```

```
15      señal_banco_gammaton    = Aplicar_banco_gammaton(segmento, fs,
            n_coef, fs_i)
16      señal_banco_gammaton    = señal_banco_gammaton[::-1]
17
18      len_s                   = len(segmento)
19      mit_s                   = len_s//2 if len_s%2 == 0 else (len_s+1)//2
20
21      espectro_banco          = [ np.abs(np.fft.fft(señal)[:mit_s]) for
            señal in señal_banco_gammaton]
22      energia_banco           = [ np.sum(np.pow( señal, 2 )) for señal in
            espectro_banco ]
23      log_energia             = np.log(energia_banco)
24
25      caracteristicas_gamma  = TDC(log_energia, len(log_energia) , n_coef)
26
27      if(verbose):
28          _plot_resumen(segmento,señal_banco_gammaton,espectro_banco,
                energia_banco,log_energia, caracteristicas_gamma )
29
30      return caracteristicas_gamma
```

Para calcular las características de un segmento primero se calcula la salida del filtro Gammatone aplicada a la señal en el dominio del tiempo. El primer elemento de la salida del filtro corresponde a la frecuencia más alta, para corregir esto los indices del banco se corrigen en la linea 16. Posteriormente se aplica la transformada de Fourier para pasar la salida al dominio de la frecuencia.

La función de *Aplicar_banco_gammatone* es la traducción del filtro propuesto en el paper *An Efficient Implementation of the Patterson-Holdsworth Auditoty Filter Bank* en la sección 4.2. La función queda de la siguiente manera:

Listing 5: Caracteristicas_gammatone.py

```
1  def Aplicar_banco_gammaton(señal, fs, n_coef, fs_inicial):
2      forward, feedback = make_erb_filters(fs, n_coef, fs_inicial)
3      y = erb_filter_bank(forward, feedback, señal)
4      return y
5
6  def make_erb_filters(fs, num_channels, low_freq):
7      T       = 1 / fs
8      EarQ    = 9.26449
9      minBW   = 24.7
10     order   = 1
11
12     E       = np.exp
13     ln      = np.log
14     PI      = np.pi
15     COS     = np.cos
16     SEN     = np.sin
17     sqrt    = np.sqrt
18
19     cf = -(EarQ * minBW) + E(np.arange(1, num_channels + 1).reshape(-1,
           1) * (-ln(fs / 2 + EarQ * minBW) + ln(low_freq + EarQ * minBW)) /
            num_channels) * (fs / 2 + EarQ * minBW)
20     cf = cf.flatten()
21
22     ERB = ((cf / EarQ) ** order + minBW ** order) ** (1 / order)
23     B = 1.019 * 2 * PI * ERB
24
```

```
    gain = np.abs(
            (-2 * E(4 * 1j * cf * PI * T) * T +\
              2 * E(-B * T + 2 * 1j * cf * PI * T) * T *\
                (COS(2 * cf * PI * T) - sqrt(3 - 2 ** (3 / 2)) *\
                  SEN(2 * cf * PI * T))) *\
            (-2 * E(4 * 1j * cf * PI * T) * T +\
              2 * E(-B * T + 2 * 1j * cf * PI * T) * T *\
                (COS(2 * cf * PI * T) + sqrt(3 - 2 ** (3 / 2)) *\
                  SEN(2 * cf * PI * T))) *\
            (-2 * E(4 * 1j * cf * PI * T) * T +\
              2 * E(-B * T + 2 * 1j * cf * PI * T) * T *\
                (COS(2 * cf * PI * T) - sqrt(3 + 2 ** (3 / 2)) *\
                  SEN(2 * cf * PI * T))) *\
            (-2 * E(4 * 1j * cf * PI * T) * T +\
              2 * E(-B * T + 2 * 1j * cf * PI * T) * T *\
                (COS(2 * cf * PI * T) + sqrt(3 + 2 ** (3 / 2)) *\
                  SEN(2 * cf * PI * T))) /\
            (-2 / E(2 * B * T) - 2 * E(4 * 1j * cf * PI * T) +\
                2 * (1 + E(4 * 1j * cf * PI * T)) / E(B * T)) ** 4)

    feedback = np.zeros((len(cf), 9))
    forward = np.zeros((len(cf), 5))

    forward[:, 0] = (T ** 4) / gain
    forward[:, 1] = -4 * T ** 4 * COS(2 * cf * PI * T) / E(B * T) / gain
    forward[:, 2] = 6 * T ** 4 * COS(4 * cf * PI * T) / E(2 * B * T) /
       gain
    forward[:, 3] = -4 * T ** 4 * COS(6 * cf * PI * T) / E(3 * B * T) /
       gain
    forward[:, 4] = T ** 4 * COS(8 * cf * PI * T) / E(4 * B * T) / gain

    feedback[:, 0] = np.ones(len(cf))
    feedback[:, 1] = -8 * COS(2 * cf * PI * T) / E(B * T)
    feedback[:, 2] = 4 * (4 + 3 * COS(4 * cf * PI * T)) / E(2 * B * T)
    feedback[:, 3] = -8 * (6 * COS(2 * cf * PI * T) + COS(6 * cf * PI *
       T)) / E(3 * B * T)
    feedback[:, 4] = 2 * (18 + 16 * COS(4 * cf * PI * T) + COS(8 * cf *
       PI * T)) / E(4 * B * T)
    feedback[:, 5] = -8 * (6 * COS(2 * cf * PI * T) + COS(6 * cf * PI *
       T)) / E(5 * B * T)
    feedback[:, 6] = 4 * (4 + 3 * COS(4 * cf * PI * T)) / E(6 * B * T)
    feedback[:, 7] = -8 * COS(2 * cf * PI * T) / E(7 * B * T)
    feedback[:, 8] = E(-8 * B * T)

    return forward, feedback
```

Caracteristicas_Modelo_Coclear

El archivo sigue la misma estructura que el archivo de *Caracteristicas_MFCC*. La función principal es *Caracteristicas_Coclear* y *Matriz_Modelo_Coclear* aplica la función principal a todo un conjunto de segmentos.

Listing 6: Caracteristicas_Modelo_coclear.py

```
import matplotlib.pyplot as plt
from numpy import ndarray
```

```
3  import numpy as np
4
5  from libreria.utileria import Banco_filtros_no_eq,  Energia_salida , FFT
       , TDC
6
7  # Extrae las características caracteristicas usando el modelo de la
       coclea resultado del laboratorio
8  def Matriz_Modelo_coclear(segmentos : ndarray[ndarray], n_coef : int, fs
       : int, fs_i : int, verbose = 0) -> ndarray[ndarray]:
9      matriz_coclea = []
10
11     for segmento in segmentos:
12         matriz_coclea.append(Caracteristicas_Coclear(segmento, n_coef,
               fs, fs_i, verbose))
13     return np.array(matriz_coclea)
14
15
16 # Función principal. Extracción de características para un segmento
17 def Caracteristicas_Coclear(segmento, n_coef : int, fs : int, fs_i : int
       , verbose = False):
18     bins, espectro, fs      = FFT(segmento, fs)
19     tam_espectro            = len(espectro)
20
21     frec_centrales          = Centros_Coclea(fs_i, fs, n_coef)
22     banco_filtros_coclea    = Banco_filtros_no_eq(frec_centrales,
           tam_espectro, fs )
23
24     Y                       = Energia_salida(espectro,
           banco_filtros_coclea)
25
26     # Los primeros centros estan muy cerca y con 128 puntos hay una
           ventana que tiene todos los valores en 0. Esto se arregla cuando
           aumenta el numero de segmentos, pero para que no de error con el
           logaritmo se implemento el siguiente código
27     for i in range(len(Y)):
28         if Y[i] == 0:
29             Y[i] = (Y[i-1] + Y[i+1]) / 2
30
31     S                       = np.log(np.abs(Y))
32
33
34     caracteristicas_coclea  = TDC(S, len(S) , n_coef)
35
36     if(verbose):
37         _plot_resumen(segmento,espectro,banco_filtros_coclea,Y,S,
               caracteristicas_coclea, bins)
38
39     return caracteristicas_coclea
```

También sigue los mismos pasos al momento de obtener las características del modelo coclear. Solo difiere al momento de calcular las frecuencias centrales, las cuales se calculan en la función de *Centros_Coclea*.

Listing 7: Caracteristicas_Modelo_coclear.py

```
1  # Genera centros del banco de filtro formando puntos equidistantes en la
       escala de Mel
2  def Centros_Coclea(fs_l : int, fs_h : int, n_coef : int) -> list[float]:
3      PI  = np.pi
```

```
    E   = np.exp
    Ln  = np.log
    SQRT= np.sqrt

    def Hz_to_M(Hz):
        return Ln( (Hz) / (2.003 * 10**4) ) / -141.2

    def M_to_Hz(M):
        return 2.003 * (10**4) * E(-141.2 * (M) )

    B       = Hz_to_M
    B_1     = M_to_Hz

    Delta_B = B(fs_h) - B(fs_l)
    B_fl    = B(fs_l)

    puntos  = np.arange(0,n_coef+2)
    coef    = [  B_1(B_fl + punto * (Delta_B / (n_coef + 1)))  for punto
        in puntos]

    return coef
```

El procedimiento es similar al de *Centros_MFCC* pero difiere en las función *B* y *B_1* las cuales ahora son funciones que van del dominio de la frecuencia a la escala del modelo coclear y viceversa.

Caracteristicas_Mel_Coclea

Como se puede ver a continuación, el archivo sigue la misma estructura que los anteriores.

Listing 8: Caracteristicas_Modelo_coclear.py

```
import matplotlib.pyplot as plt
from numpy import ndarray
import numpy as np

from libreria.utileria import Banco_filtros_no_eq,  Energia_salida , FFT
    , TDC

# Extrae las características de un conjunto de segmentos
def Matriz_Composicion(segmentos : ndarray[ndarray], n_coef : int, fs :
    int, fs_i : int, verbose = 0) -> ndarray[ndarray]:
    matriz_Comp = []

    for segmento in segmentos:
        matriz_Comp.append(Caracteristicas_Comp(segmento, n_coef, fs,
            fs_i, verbose))
    return np.array(matriz_Comp)

# Función principal. Extracción de características para un segmento
def Caracteristicas_Comp(segmento, n_coef : int, fs : int, fs_i : int,
    verbose = False):
    bins, espectro, fs      = FFT(segmento, fs)
    tam_espectro            = len(espectro)

    frec_centrales_m        = Centros_Mel(fs_i, fs, n_coef)
```

```
21      frec_centrales_c        = Centros_mel2coc(frec_centrales_m)
22      frec_centrales_hz       = Coclea2Hz(frec_centrales_c)
23
24      banco_filtros_Comp      = Banco_filtros_no_eq(frec_centrales_hz,
            tam_espectro, fs )
25
26      Y                       = Energia_salida(espectro,
            banco_filtros_Comp)
27      S                       = np.log(Y)
28
29      caracteristicas_comp    = TDC(S, len(S) , n_coef)
30
31      if(verbose):
32          _plot_resumen(segmento,espectro,banco_filtros_Comp,Y,S,
                caracteristicas_comp, bins)
33
34      return caracteristicas_comp
```

Siendo nuevamente la única diferencia la función que calcula las frecuencias centrales del banco de filtros.

Primero se calculan los centros en la escala de Mel y posteriormente se llevan a la escala del modelo coclear y finalmente se pasan al dominio de la frecuencia. La implementación de las funciones anteriores es la siguiente:

Listing 9: Caracteristicas_Modelo_coclear.py

```
1  # Genera centros equidistantes en la escala de Mel
2  def Centros_Mel(fs_l : int, fs_h : int, n_coef : int) -> list[float]:
3      def Hz_to_Mel(hz):
4          return 1125 * np.log(1 + hz / 700)
5          #return np.log( (hz) / (2.003 * 10**4) ) / -141.2
6
7      B       = Hz_to_Mel
8
9      Delta_B = B(fs_h) - B(fs_l)
10     B_fl    = B(fs_l)
11
12     puntos  = np.arange(0,n_coef+2)
13
14     coef    = [ B_fl + punto * (Delta_B / (n_coef + 1)) for punto in
           puntos]
15     return coef
16
17 # Genera centros del banco de filtro formandos por llevar la escala de
       mel a la escala de la coclea
18 def Centros_Composicion(fs_l : int, fs_h : int, n_coef : int) -> list[
       float]:
19     f_centrales_mel = Centros_Mel(fs_l, fs_h, n_coef)
20     f_centrales_coc = Centros_mel2coc(f_centrales_mel)
21     f_centrales_hz  = Coclea2Hz(f_centrales_coc)
22
23     return f_centrales_hz
24
25 # Convierte de Mel a la escala de la coclea
26 def Centros_mel2coc(centros_mel):
27     a           = 2.003 * (10**4)
28     b           = -1.4142
29     ln          = np.log
```

```
30      E           = np.exp
31      mel2cocl    = lambda f_mel : ( ln( E( f_mel / 1125 ) - 1 ) + ln(700/
            a) )/b
32
33
34      return [mel2cocl(f_mel) for f_mel in centros_mel]
35
36  # Convierte de la escala de la coclea a Hz
37  def Coclea2Hz(f_coclea):
38      E           = np.exp
39      Ln          = np.log
40
41      coclea2Hz   = lambda cm: 2.003 * (10**4) * E(-141.2 * (cm/100) )
42      return [coclea2Hz(f_coclea) for f_coclea in f_coclea]
```

3. Comparación_de_filtros

El archivo *Comparacion_de_filtros* muestra cuatro comparativas.

- Una comparativa entre los filtros MFCC calculados manualmente y los calculados con la librería *librosa*.
- Gráfica los cuatro bancos de filtros incluidos en la librería.
- Muestra una señal en le dominio del tiempo, la frecuencia y la salida de cada uno de los filtros.
- Gráfica cada uno de los centros incluido en los 4 bancos de filtros.

4. Partir_dataset.py

Es el primer archivo que se debe de ejecutar. Se encarga de separar el dataset en dos carpetas, una para entrenamiento y otra para pruebas. El porcentaje de los archivos que se usan para entrenamiento se puede modificar en la sección de constantes.

El código crea un enlace simbólico de cada archivo en la carpeta original a la carpeta de destino. Antes de crear los enlaces se limpian las carpetas de train y test.

5. Extraer_caracteristicas

Segundo archivo a ejecutar. Borra el contenido de las carpetas de *Caracteristicas_Extraidas* y a cada uno de los archivos en la carpeta de *dataset* le aplica la siguiente función:

Listing 10: Extraer_caracteristicas.py

```
1  def Procesar_archivo(registro):
2      tiempo, señal, fs       = Leer_archivo(registro)
3      tiempo_seg, segmentos   = Preprocesamiento(tiempo, señal)
4
5      parametros = {
6          "n_coef" : 36,
7          "fs" : fs,
8          "fs_i" : 100
9      }
```

```
10
11      # Ejemplo de como se pude testear un solo segmento
12      #car_coclea              = Matriz_Modelo_coclear([segmentos[2]], **
            parametros, verbose = True)
13      #return
14
15      car_MFCC                 = Matriz_MFCC(segmentos, **parametros)
16      car_coclea               = Matriz_Modelo_coclear(segmentos, **
            parametros)
17      car_Gammatone            = Matriz_Gammatone(segmentos, **parametros)
18      car_comp                 = Matriz_Composicion(segmentos, **parametros
            )
19
20      Guardar_caracteristicas(registro, car_MFCC, CAR_DEST_MFCC)
21      Guardar_caracteristicas(registro, car_coclea, CAR_DEST_COCLEA)
22      Guardar_caracteristicas(registro, car_Gammatone, CAR_DEST_GAMMATON)
23      Guardar_caracteristicas(registro, car_comp, CAR_DEST_COMP)
```

La función aplica el pre-procesamiento al audio, extrae las características y guarda el resultado en formato .npy.

6. Convertir_npy_2_csv

Tercer archivo a ejecutar. Lee todos los archivos .npy en la carpeta de *Caracteristicas_Extraidas* y crea un archivo .csv por cada tipo de característica. Cada .csv contiene como renglón los segmentos de cada archivo de audio.

Como columnas contiene : el nombre del archivo e indice del segmento, el número que representa el dígito contenido y una columna por cada coeficiente.

7. Entrenar_clusters

Cuarto archivo a ejecutar. Lee los archivos .csv en la carpeta de *Caracteristicas_Extraidas* y entrena un modelo de KMeans con los datos. La función principal es la siguiente:

Listing 11: Entrenar_clusters.py

```
1  def main( verbose = False ):
2      ## Constantes
3      DIR_BASE        = Path.home() / "Maestria" / "Primer_semestre" / "
           PDS" / "Proyecto_Filtros"
4
5      PATH_train      = DIR_BASE / "dataset_separado" / "train"
6      PATH_test       = DIR_BASE / "dataset_separado" / "test"
7
8      CAR_DEST_CSV    = DIR_BASE / "Caracteristicas_extraidas"
9
10     REGISTROS_train = list(PATH_train.iterdir())
11     REGISTROS_test  = list(PATH_test.iterdir())
12
13     DIRECTORIOS_CAR     = [
14         'MFCC',
15         'Modelo_coclear',
16         'Gammatone',
17         'Composicion'
18     ]
19     ## Se guardan los nombres de los archivos de train y test
```

```
20      train_etiquetas     = [ Discriminar_archivo(reg) for reg in
            REGISTROS_train]
21      test_etiquetas      = [ Discriminar_archivo(reg) for reg in
            REGISTROS_test]
22
23      Nombre_train        = [ reg.name for reg in REGISTROS_train]
24      Nombre_test         = [ reg.name for reg in REGISTROS_test]
25
26      ## Se cargan las caracteristicas y etiquetas de las caracteristicas
27      for tipo_car, nombre_car in enumerate(DIRECTORIOS_CAR):
28          print(f"Caracteristicas: {nombre_car}")
29          # Se abre el archivo con las caracteristicas correspondientes
30          csv_dir         = CAR_DEST_CSV / str(nombre_car + ".csv")
31          caract          = pd.read_csv(csv_dir)
32
33          # Para el entrenamiento solo se toman las caracteristicas del
                conjunto de entrenamiento
34          segmentos_train = []
35
36          for segmento in caract.nombre_segmento:
37              archivo_seg = segmento.split("_")[0]
38              if archivo_seg in Nombre_train:
39                  segmentos_train.append(True)
40              else :
41                  segmentos_train.append(False)
42
43
44          # Se eliminan las columnas de etiquetas y nombre de segmento
                para quedarse solo con las caracteristicas
45          train_x         = caract[segmentos_train]
46          train_x         = caract.drop(columns = ['nombre_segmento'])
47          train_x         = train_x.drop(columns = ['etiqueta'])
48
49
50      ## Se cargan las caracteristicas y etiquetas de las caracteristicas
51
52          # Entrenamiento del clasificador K-Means
53          model_KM        = KMeans( 10, random_state = 42 )
54          model_KM.fit(train_x)
55
56          # Predicciones de los archivos de entrenamiento
57          predic_archivo  = [ Predecir_archivo(archivo, model_KM, caract)
                for archivo in REGISTROS_train]
58
59          # Diccionario que mapea el cluster a la etiqueta
60          cluster_2_label = Imprimir_modas_de_clusters(train_etiquetas,
                predic_archivo)
61
62          # Si verbose es verdadero, se imprimen las metricas del modelo
                tambine para los archivos de entrenamiento
63          if verbose:
64              Evaluar_metricas_del_modelo(REGISTROS_train, train_etiquetas
                    , model_KM, cluster_2_label, caract)
65          Evaluar_metricas_del_modelo(REGISTROS_test, test_etiquetas,
                model_KM, cluster_2_label, caract)
```

Se entrena un cluster por cada tipo de característica. Para clasificar el cluster se sigue el siguiente procedimiento:

1. Se carga el archivo .csv correspondiente.
2. Se conservan solo los segmentos que pertenecen a los archivos en la carpeta de entrenamiento.
3. Se elimina la columna con el nombre del segmento y el dígito que representa.
4. Se entrena el modelo de KMeans.
5. Se clasifican clasifican los segmentos del archivo de test. Usando la función *Predecir_archivo*.
6. La función *cluster_2_label* asigna a cada cluster el dígito que más se repite en el cluster.
7. Finalmente se evalúa el modelo con el conjunto de entrenamiento

La función *Predecir_archivo* filtra todos los segmentos que pertenecen al archivo pasado como argumento y le clasifica todos los segmentos. Para decidir que cluster representa al archivo se selecciona el cluster que más se repite en el archivo (la moda).

Para decidir que dígito intenta representar cada cluster se usa la función *cluster_2_label*. Dicha función recorre los clusters y compara la etiqueta asignada por el modelo con la etiqueta real. La etiqueta que más se repite en el cluster es la que se asigna al cluster (la moda).

Para evaluar el modelo se usa nuevamente la función *Predecir_archivo* pero ahora con el conjunto de test.