Universidad Rey Juan Carlos

IDENTIFICACION DEL LOCUTOR CON K-NEAREST NEIGHBOR

TRATAMIENTO DIGITAL DEL SONIDO

Elena Mª del Río Galera
Yolanda Lillo Mata
Cristina Taboada Mayo

Grupo: 2

Curso 2020-2021
3er curso
GRADO DE INGENIERÍA EN AUDIOVISUALES Y MULTIMEDIA

1. Introducción y objetivos del trabajo.

K-Nearest-Neighbor es un algoritmo **basado en instancia**, esto quiere decir que nuestro algoritmo no aprende explícitamente un modelo. En cambio memoriza las instancias de entrenamiento que son usadas como "base de conocimiento" para la fase de predicción. Al ser un método sencillo, es ideal para introducirse en el mundo del Aprendizaje Automático. Sirve esencialmente para clasificar valores buscando los puntos de datos "más similares" (por cercanía) aprendidos en la etapa de entrenamiento y haciendo conjeturas de nuevos puntos basado en esa clasificación.

K-Nearest Neighbor es un **algoritmo supervisado,** lo que implica que la "K" se refiere al número de puntos vecinos en las cercanías que se tienen en cuenta para clasificar los "n" grupos que ya se conocen de antemano.

Este algoritmo se utiliza en la resolución de multitud de problemas, como en sistemas de recomendación, búsqueda semántica y detección de anomalías.

El objetivo final, es tener un programa funcional en Jupyter, que aplique correctamente el algoritmo K-Nearest Neighbor.

2. Metodología

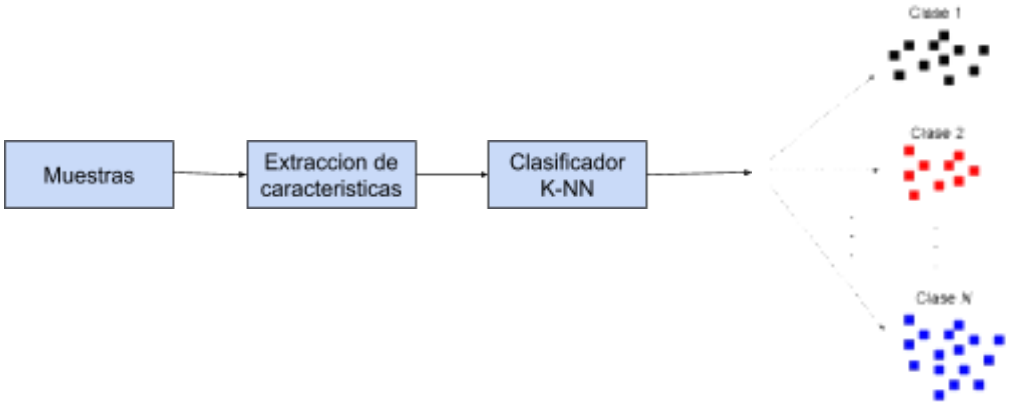


Figura. Esquema de funcionamiento del algoritmo K-NN

• Cómo se han grabado/obtenido las señales (si procede).

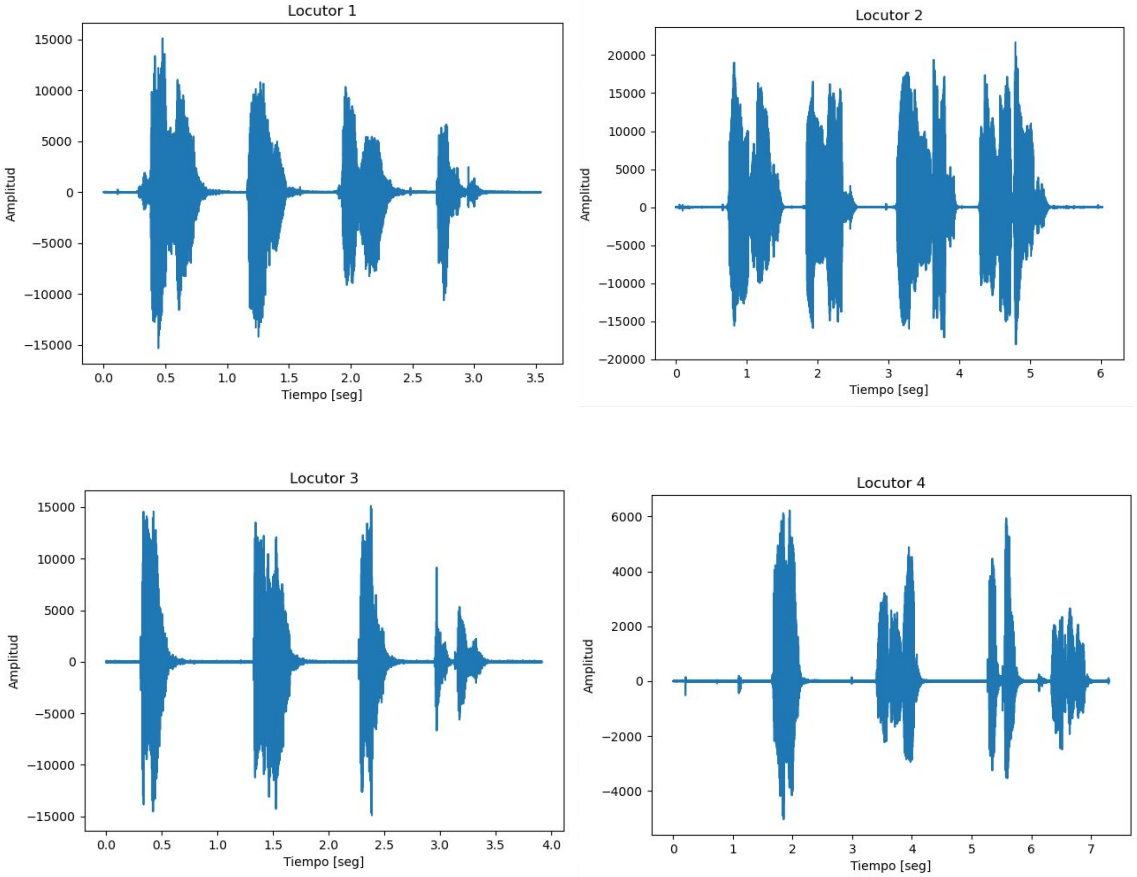
Hemos realizado grabaciones de nuestras voces y las de nuestros familiares y amigos para obtener las señales que vamos a utilizar en el grupo de entrenamiento. Las grabaciones contienen varias palabras con espacios de silencio entre ellas para facilitar su procesamiento.
Hemos utilizado el programa *Audacity* para separar los dos canales y analizar un único canal, en concreto el canal izquierdo.

Debemos tener en cuenta que a la hora de ver nuestras señales, algunas seguían mostrando dos canales y por eso mediante la línea: y1 = y1[:,0] hemos eliminado el canal que teníamos a cero y se ve gráficamente el que tiene las muestras de audio.

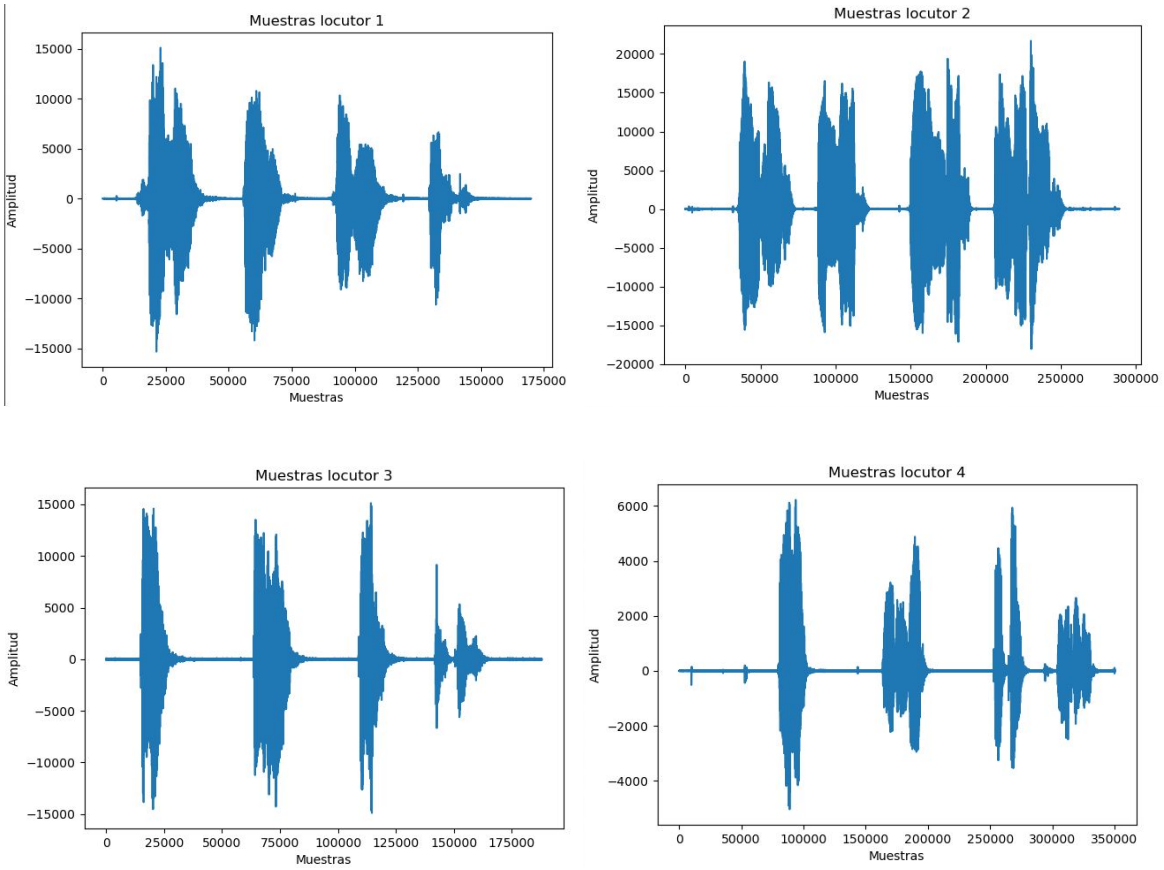
• Cómo se ha realizado la extracción de características y razones por las que se han escogido esas características.

La característica que se ha decidido extraer de cada grabación es el **pitch**, porque es la frecuencia a la que vibran las cuerdas vocales, también llamada frecuencia fundamental. Es uno de los parámetros más característicos de la voz de un locutor.
Para su obtención se han seguido los siguientes pasos:

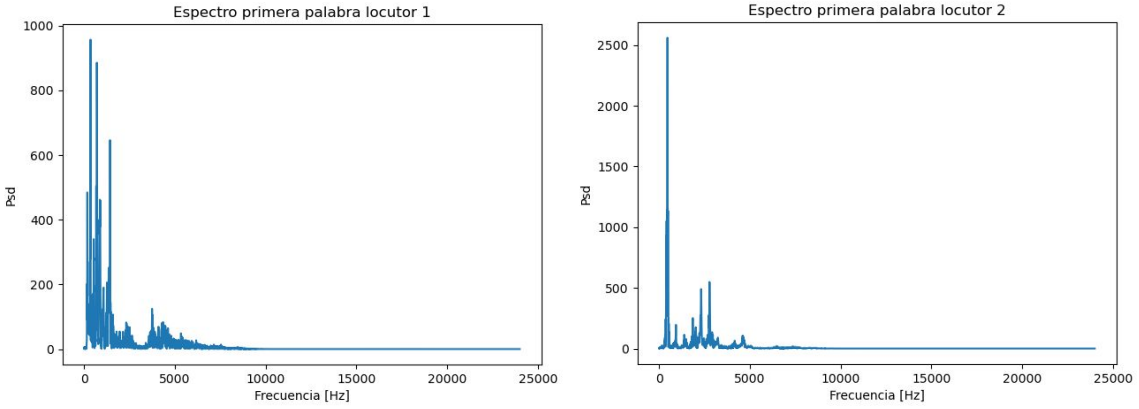
1. Primero cogemos los archivos de audio de cada uno de los locutores, **extraemos los datos de la señal (frecuencia de muestreo, tiempo para cada locutor)** y lo visualizamos por pantalla.

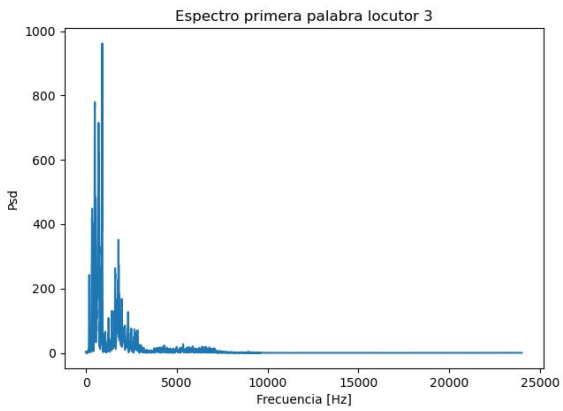


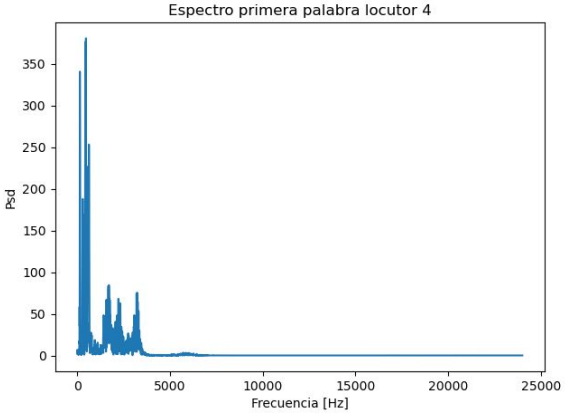
2. El segundo paso es **representar las muestras** de cada locutor.



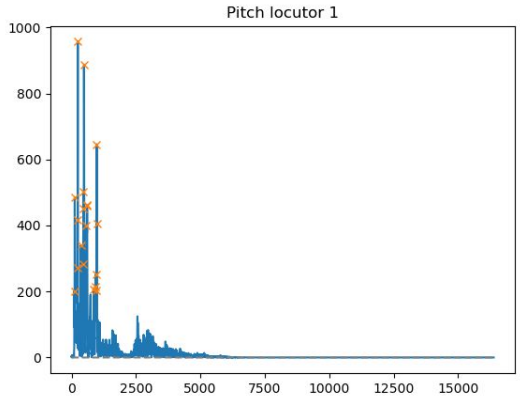
3. En tercer lugar, **se representa un segmento del espectro de la primera palabra** de cada uno de los locutores, recordamos que la frecuencia de muestreo de cada uno de ellos es 48000 y tenemos cuidado a la hora de elegir la ventana (9600 muestras) además se ajustan las marcas de inicio para cada grabación en función de la frecuencia.

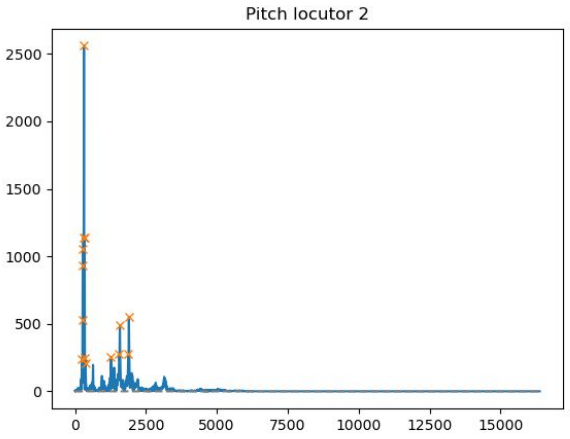


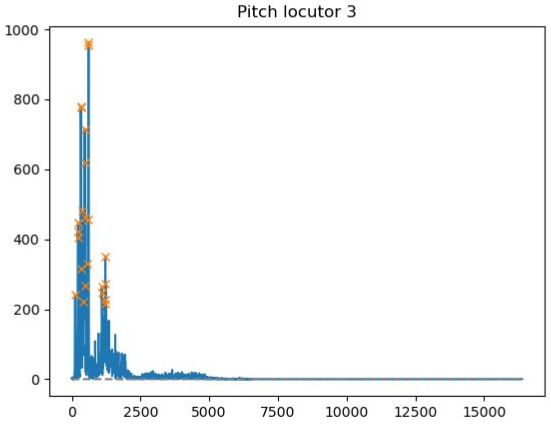


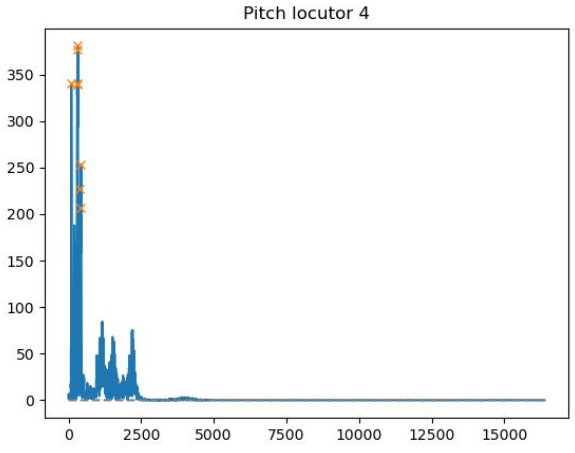


4. El cuarto y último paso para **obtener el pitch**, una vez que se tiene la representación de la primera palabra, con la función **findpeaks**, ponemos el límite de altura, esto se hace, para despreciar los picos más pequeños, que el programa podría identificar erróneamente como pitch. Una vez elegido el corte, con la línea peaks[0], se obtiene el valor en el eje x donde se encuentra el primer pico por encima de 200, es decir el pitch.









• Cómo se han formado las estructuras de datos que se han pasado como entrada a los clasificadores o al método de los umbrales.

Primero obtenemos el pitch para cada uno de los locutores de forma númerica, de este modo obtendremos el valor tanto en el eje x como en el y.

```
x11 = peaks[0]
x12 = x1[peaks[0]]
print(x11, x12)
```

En este caso sería para el locutor 1 pero debemos realizar lo mismo para cada uno de los locutores restantes.

Después de esto, creamos la matriz de entrenamiento xtrain con los valores del pitch obtenido anteriormente. Y creamos la matriz de la clase asociada a cada conjunto de datos (ytrain).

```
from sklearn.model_selection import train_test_split
```

```
xtrain = [[x11, x12],
          [x21, x22],
          [x31, x32],
          [x41, x42]
         ]
```

```
ytrain = [['loc1'],
          ['loc2'],
          ['loc3'],
          ['loc4']
         ]
```

• Cómo se han dividido los datos para entrenamiento y prueba.

Obtenemos los datos de entrenamiento a partir de los datos de prueba (aunque no sea un método muy correcto) para no alargar mucho el código del programa, ya que nuestro objetivo es comprobar el funcionamiento del algoritmo K-NN y con esto podemos visualizarlo.
Creamos una matriz de prueba a partir de los datos de las matrices de entrenamiento, utilizamos el 30 % de los datos de entrenamiento para crear la matriz de prueba y visualizamos ambas.

```
print('------TRAIN--------')
print(xtrain)
print(ytrain)
print('--------------------')
```

```
x, xtest, y, ytest = train_test_split(xtrain, ytrain, test_size=0.3)
```

```
print('------TEST-------')
print(xtest)
print(ytest)
print('-----------------')
```

• Explicar cómo se ha utilizado el clasificador correspondiente o el método de los umbrales.

Por último, cargamos todos los datos para utilizar el clasificador KNN
Se han creado varios clasificadores para ver cómo varía la precisión según el número de vecinos que estén próximos.
El código que se muestra a continuación se repite, para n=2, n=3 y n=4.

```
from sklearn.neighbors import KNeighborsClassifier
import sys

# Creamos varios clasificadores para ver con cuantos n vecinos mas proximos es mas preciso

# Creamos el clasificador para n=1
KNN_classifier = KNeighborsClassifier(n_neighbors=1)

# Entrenamiento
KNN_classifier.fit(xtrain, ytrain)

# Prediccion
y_hat_KNN = KNN_classifier.predict(xtest)
print(y_hat_KNN)

# Precision
acc = np.sum (ytest == y_hat_KNN)/len(ytest)
print('El algoritmo con n=1 tiene una precision del: ' + str(acc))
```

3. Pruebas y resultados.

• Explicar si se han realizado diferentes pruebas o se han utilizado diferentes combinaciones de características.

Según el código que se ha programado, la matriz de test varía cada vez que se ejecuta. Realizamos diferentes pruebas obteniendo los siguientes resultados:

❖ Ejecución 1

```
------TRAIN--------
[[111, 201.23033443657687], [248, 242.51883702340984], [120, 242.37056304356187], [98, 34
0.75063446067463]]
[['loc1'], ['loc2'], ['loc3'], ['loc4']]
-------------------
------TEST-------
[[98, 340.75063446067463], [248, 242.51883702340984]]
[['loc4'], ['loc2']]
-----------------

['loc4' 'loc2']
El algoritmo con n=1 tiene una precision del: 1.0
['loc3' 'loc2']
El algoritmo con n=2 tiene una precision del: 0.5
['loc1' 'loc1']
El algoritmo con n=3 tiene una precision del: 0.0
['loc1' 'loc1']
El algoritmo con n=4 tiene una precision del: 0.0
```

❖ **Ejecución 2**

```
------TRAIN--------
[[111, 201.23033443657687], [248, 242.51883702340984], [120, 242.37056304356187], [98, 34
0.75063446067463]]
[['loc1'], ['loc2'], ['loc3'], ['loc4']]
-------------------
------TEST-------
[[248, 242.51883702340984], [111, 201.23033443657687]]
[['loc2'], ['loc1']]
-----------------

['loc2' 'loc1']
El algoritmo con n=1 tiene una precision del: 1.0
['loc2' 'loc1']
El algoritmo con n=2 tiene una precision del: 1.0
['loc1' 'loc1']
El algoritmo con n=3 tiene una precision del: 1.0
['loc1' 'loc1']
El algoritmo con n=4 tiene una precision del: 1.0
```

❖ **Ejecución 3**

```
------TRAIN--------
[[111, 201.23033443657687], [248, 242.51883702340984], [120, 242.37056304356187], [98, 34
0.75063446067463]]
[['loc1'], ['loc2'], ['loc3'], ['loc4']]
-------------------
------TEST-------
[[98, 340.75063446067463], [111, 201.23033443657687]]
[['loc4'], ['loc1']]
-----------------

['loc4' 'loc1']
El algoritmo con n=1 tiene una precision del: 1.0
['loc3' 'loc1']
El algoritmo con n=2 tiene una precision del: 0.5
['loc1' 'loc1']
El algoritmo con n=3 tiene una precision del: 1.0
['loc1' 'loc1']
El algoritmo con n=4 tiene una precision del: 1.0
```

• **Errores de clasificación obtenidos en cada caso.**

❖ Vemos que cuando **n = 1** clasifica correctamente.
Esto es lógico ya que sólo tiene cerca el grupo correcto en el cual tiene que clasificar la muestra.

❖ Cuando **n = 2,** dos casos:
- ❏ Clasifica correctamente las 2 muestras, precisión = 1.0
- ❏ Clasifica bien una muestra y la otra no, precisión = 0.5

❖ Cuando **n = 3 y n = 4**, aumenta mucho el error tanto en la clasificación cómo en el cálculo de la precisión. Vemos que en la ejecución 1, no clasifica correctamente ninguna de las muestras pero calcula correctamente la precisión. Sin embargo, en

las ejecuciones 2 y 3, clasifica bien una de las muestras la otra no, pero no calcula correctamente la precisión de la clasificación, ya que en ese caso debería ser de 0.5.

4. Conclusiones y líneas de mejora.

• Explicar las dificultades encontradas, si han sido salvables o insalvables.

- Durante la realización del trabajo, hemos encontrado varios problemas a la hora de **obtener los valores del pitch.**

- A la hora de representar el espectro de la señal, aún después de haber tratado la señal con el programa "*Audacity*" para **eliminar uno de los canales**, seguía cogiendo ambos, por lo que no se mostraba correctamente. Solventamos escogiendo solo uno de los canales con python.

- A la hora de **representar el espectro**, en un principio intentamos obtener el espectro de la primera palabra de cada locutor, ya que no estábamos teniendo en cuenta que debíamos obtener las muestras de una ventana con un tamaño determinado por la frecuencia de muestreo de cada una de las señales. Pudimos solucionarlo teniendo en cuenta lo mencionado y utilizando la ventana obtenida en la función "*my_spectra*".

- Por último, a la hora de **crear la matriz xtest**, al principio creamos una matriz a "mano", sin embargo al final nos decantamos por crear una matriz, aleatoria en cada ejecución, a partir del grupo de entrenamiento. Aunque esta no sea una solución correcta, consideramos que para ver el funcionamiento de este algoritmo, que es el objetivo de nuestro trabajo, era una opción válida.

• ¿Qué se podría mejorar para obtener mejores resultados? ¿qué probaría si se dispusiese de horas ilimitadas para resolver el problema planteado?

Se podrían mejorar los resultados y la precisión si aparte de extraer el pitch de cada grabación se extrajeran **más características** como por ejemplo la energía.
También sería conveniente **aumentar el número de locutores** con los que crear las matrices de entrenamiento y nuevas grabaciones de los locutores para crear el **grupo de test**, ya que no es conveniente utilizar muestras del grupo de entrenamiento para el grupo de test.

5. Bibliografía

- https://www.aprendemachinelearning.com/clasificar-con-k-nearest-neighbor-ejemplo-en-python/
- https://www.merkleinc.com/es/es/blog/algoritmo-knn-modelado-datos
- Apuntes Tema 5, Tratamiento Digital del Sonido.
- https://academica-e.unavarra.es/bitstream/handle/2454/29228/memoria%20TFG%20merino.100363.pdf?sequence=1&isAllowed=y