Reconeixement de la parla

Professors de l'assignatura

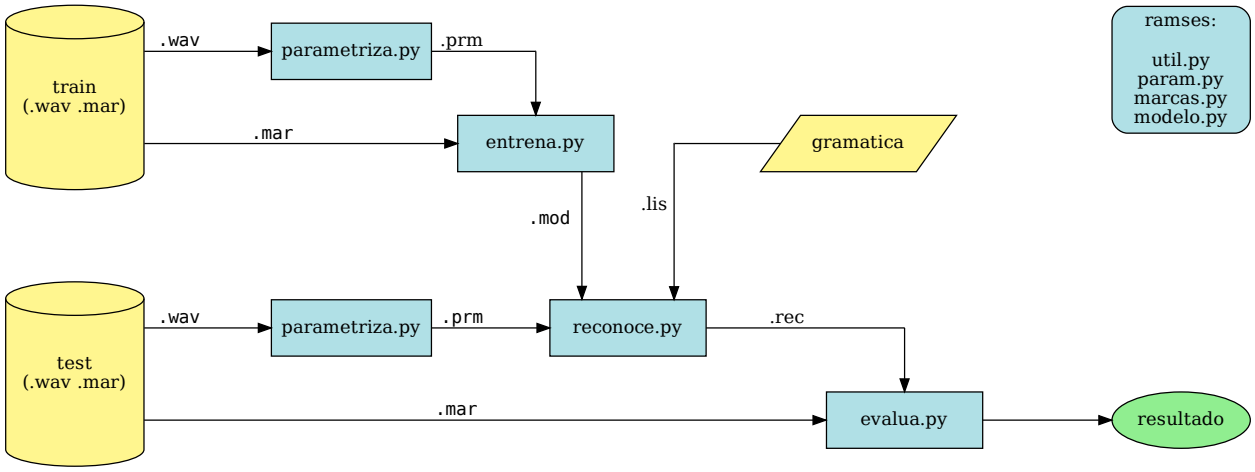
diciembre de 2025

Índice general

Introducción

Desde tiempo inmemorial, la construcción de máquinas que fueran capaces de entender lo que se les dice ha constituido el sueño de muchos, la pesadilla de otros y un reto para los científicos dedicados al tema. Probablemente, el primer sistema de reconocimiento automático del habla documentado sea el usado por los cuarenta ladrones para proteger sus tesoros. Desde entonces, se han sucedido las propuestas, alcanzándose, a principios del siglo XXI, un alto grado de perfeccionamiento. Aunque el objetivo de construir sistemas que, como HAL 9000, no sólo sean capaces de entender lo que se les dice, sino también de comprenderlo, sigue estando lejano.

El esquema general de un sistema de reconocimiento del habla es el siguiente:



Las entradas principales del sistema son las dos bases de datos que aparecen a la izquierda y con forma de tambor amarillo. La de entrenamiento (**train**) es la que se usará para *aprender* las características del vocabulario a reconocer. La de evaluación (**test**) se usará evaluar las prestaciones del sistema. En ambas hemos de disponer de las señales temporales (ficheros de audio, en este caso con formato WAVE de Microsoft, extensión `.wav`) y de los contenidos fonéticos de cada una (usaremos el formato SAM 6.0, extensión `.mar`).

El sistema de reconocimiento también requiere como entrada la gramática de la tarea a reconocer. Ésta consiste en la descripción de las frases admitidas como resultado del reconocimiento. Por ejemplo, en un sistema de reconocimiento de dígitos basado en modelos de fonemas, la gramática nos indicaría que el dígito **cuatro** se forma como concatenación de las unidades `/k/`, `/w/`, `/a/`, `/t/`, `/4/` y `/o/`. En el sistema que se implementará en este curso la gramática es la más sencilla posible: una lista (extensión `.lis`) de las unidades que se pueden reconocer de manera aislada, es decir, una unidad por señal.

Los tres (o cuatro o cinco) elementos fundamentales de un sistema de reconocimiento aparecen como rectángulos de color azulado:

Parametrización:
: También conocida como *extracción de características*, consiste en la representación de la señal de voz de modo que se optimice la información de lo que se quiere reconocer, minimizando el resto de informaciones que, para nuestra tarea, serán consideradas ruidosas.

Entrenamiento:
: También llamado *modelado acústico*, en el cual se construye un modelo matemático de la señal parametrizada correspondiente a cada una de las palabras que forman el vocabulario a reconocer.

Reconocimiento:
: El reconocimiento propiamente dicho es la fase en la que la señal a reconocer es comparada con los modelos de cada una de las palabras del vocabulario y se selecciona la palabra que mejor la representa.

Adicionalmente, durante el desarrollo del sistema de reconocimiento, es necesaria un cuarto elemento:

Evaluación:
: Consistente en la comparación del resultado del reconocimiento con lo realmente presente en la frase a reconocer, y usada para calibrar la calidad del sistema.

  El resultado de la evaluación, el oval verde a la derecha del gráfico, consiste básicamente en mostrar por pantalla dos elementos: una matriz de confusión, que nos permite analizar el comportamiento del sistema a partir de las veces que las distintas unidades se confunden entre sí; y una medida simple de calidad, en la forma de tasa de exactitud, que nos permite comparar sistemas o configuraciones distintas.

Finalmente, hay toda una serie de funciones y clases que son utilizadas desde distintos programas del sistema, y que agrupamos en la biblioteca `Ramses`:

Biblioteca `Ramses`:
: Compuesta, inicialmente, por los módulos siguientes:

  `util.py`: Utilidades de propósito general para distintas funciones, como la gestión de ficheros y directorios.

  `mar.py`: Manejo de los ficheros de marcas. En principio, sólo incluye una función para extraer la transcripción fonética de una señal.

  `prm.py`: Manejo de los ficheros de señal parametrizada. Incluye las funciones de lectura y escritura.

mod.py: Definición de las clases usadas para el modelado acústico.

El código de los distintos módulos que componen el sistema se irá presentando conforme se explique su cometido.

Capítulo 1

Tarea de reconocimiento: las vocales del castellano

En este proyecto, el trabajo se centrará en el reconocimiento de las cinco vocales del castellano (`/a/`, `/e/`, `/i/`, `/o/` y `/u/`). Cada señal a reconocer contiene una única vocal, por lo que se trata de un sistema de reconocimiento de los denominados de *palabras aisladas*. Otros tipos de sistema de reconocimiento, más complejos, son los de palabras conectadas o los de habla continua. El reconocimiento de las vocales, aunque sencilla, es una tarea bastante representativa de las prestaciones de los sistemas de reconocimiento del habla más complejos, ya que se trata de un vocabulario con alta probabilidad de confusión.

Un factor determinante en todo sistema de reconocimiento automático del habla es el de la calidad de las señales a reconocer. Resulta mucho más sencillo reconocer el habla cuando ésta está grabada con alta calidad y en un ambiente silencios, que cuando la calidad de grabación es baja y/o el ambiente es ruidoso. En este proyecto nos centraremos en un caso de especial relevancia: la señal grabada por teléfono fijo, con una frecuencia de muestreo $f_m = 8\,\text{kHz}$ y codificación PCM ley-μ de 8 bits.

1.1. Base de datos `Sen/vocales`

Las señales que se usarán en el proyecto han sido extraídas de la base de datos Spanish SpeechDa grabada a principios del milenio en el marco de un proyecto multinacional impulsado y parcialmente financiado por la Comisión Europea. En este proyecto, distintas frases pronunciadas por un total de 1000 locutores fueron grabadas usando telefonía fija.

Utilizando un sistema de reconocimiento previamente entrenado, se determinaron los instantes de inicio y final de cada fonema. De los fonemas vocálicos, se seleccionaron los de duración mayor de 100 ms y se extrajeron sus 512 muestras centrales (a 8 kHz, eso representa 64 ms). Finalmente, se escogieron aleatoriamente los segmentos producidos para obtener 6000 segmentos vocálicos para cada conjunto, a razón de 1200 por cada vocal. El resultado es la base de datos que usaremos en el proyecto, `Sen/vocales`.

Por motivos de eficiencia en el acceso a los directorios, los ficheros correspondientes a cada segmento se han distribuido en *bloques*. Cada bloque es un directorio de nombre formado por la cadena `'block'` seguida de un número entero de dos dígitos.

1.1.1. Conjuntos de entrenamiento, desarrollo y evaluación

La base de todo esquema de reconocimiento del habla es la extracción de la información subyacente que permite distinguir unas palabras (o, en general, unidades acústica) de otras. Piénsese, por ejemplo, en el reconocimiento de los dígitos. Cada vez que pronunciamos uno, como por ejemplo el **siete**, generamos una señal acústica diferente. Aunque intentemos producir dos realizaciones idénticas, las señales temporales nunca lo serán. Y aún serán más diferentes si las producimos en instantes distintos, o si las comparamos con las producidas por otro locutor. Sin embargo, en general seremos capaces de determinar que se trata de un **siete** y no de un **tres**, un **cuatro** o cualquier otro dígito.

Por tanto, para poder obtener un modelo de la palabra a reconocer, será necesario disponer de muchas realizaciones distintas de la misma, de manera que seamos capaces de capturar lo que la diferencia del resto de palabras del vocabulario, con independencia de cualquier otro condicionante. Denominamos *base de datos de entrenamiento* al conjunto de señales usadas para este cometido.

Esta base de datos, no obstante, no puede ser utilizada para valorar la calidad del sistema de reconocimiento, ya que, al haber participado en la generación de los modelos acústicos, las señales que la forman serán reconocidas más fácilmente que aquéllas que no lo hubieran hecho. Es preciso, por tanto, disponer de una segunda base de datos, que denominaremos de *evaluación*, que nos permita determinar las prestaciones del sistema a la hora de reconocer señales de origen desconocido. Idealmente, esta base de datos no sólo debería estar compuesta por señales que no participan en el entrenamiento, sino que también los locutores que las producen deberían ser diferentes. Así calibraremos la calidad del sistema en el peor de los casos posibles: cuando también el locutor a reconocer es desconocido.

Finalmente, no basta con una única base de datos de evaluación porque, durante el ajuste de los parámetros que definen el sistema, estaremos adaptándolo a ese conjunto concreto de señales y no tendremos la certeza de que el sistema se comporte de igual manera al reconocer señales distintas. Por este motivo, se usará una tercera base de datos, la de *desarrollo*, que será la única que se usará durante el desarrollo y ajuste del sistema.

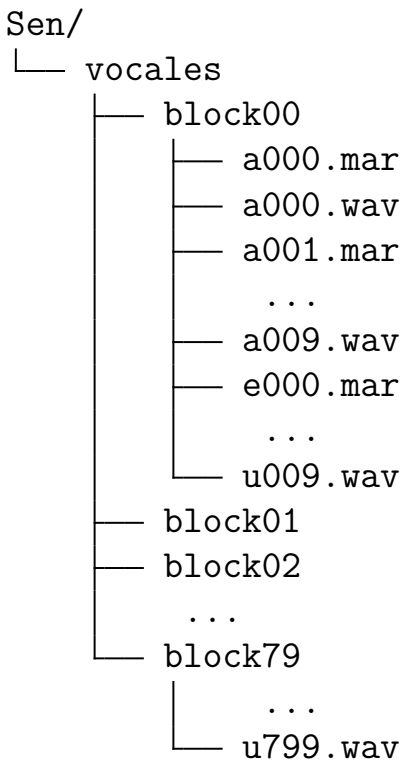
Los 1000 locutores de la base de datos SpeechDat se repartieron en tres grupos de unos 333 locutores cada uno, balanceados de manera que la distribución de sexos, edades, entornos de grabación y origen geográfico estén lo más equilibrados posible entre los tres grupos. Los tres grupos serán los que usaremos como conjuntos de entrenamiento, desarrollo y evaluación, y las señales que los forman están incluidas en los ficheros `Gui/train.gui`, `Gui/devel.gui`, `Gui/eval.gui`, respectivamente.

De cada grupo de locutores se han seleccionado 400 muestras de cada vocal (en total, $3 \times 400 \times 5 = 6000$), que son las que se usarán en los experimentos de este curso.

Estructura de los conjuntos de entrenamiento y desarrollo

Para las señales correspondientes a las bases de datos de entrenamiento y desarrollo se proporciona tanto el fichero con la señal temporal (extensión `.wav`) como el fichero de texto con la transcripción del contenido (ficheros de marcas, con la extensión `.mar`). El nombre de ambos ficheros se forma encadenando la vocal correspondiente con un número entero correlativo entre 000 y 799, y representado por tres dígitos justificados con cero. El número de bloque en el que se almacena cada señal coincide con los dos primeros dígitos del número de señal. Las 400 primeras realizaciones de cada vocal forman el conjunto de entrenamiento (bloques del 00 al 39), las 400 restantes (bloques del 40 al 79), el de desarrollo.

El esquema siguiente muestra la organización resultante:

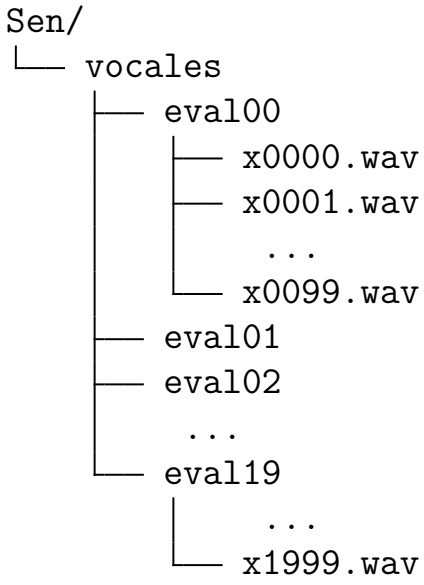


Estructura del conjunto de evaluación

La parte de evaluación de la base de datos tiene dos diferencias principales respecto a las de entrenamiento y desarrollo: no se dispone de fichero de marcas y el nombre no aporta información de la vocal a la que corresponde la señal. De este modo se garantiza que no puede ser usada en la fase de desarrollo del sistema, y que sólo algunos escogidos (en este caso, los profesores de la asignatura) están preparados para realizar la evaluación final.

Los bloques de este conjunto empiezan por la cadena `'eval'` seguida de dos dígitos del 00 al 19. Cada uno de los 20 bloques incluye los ficheros de señal de cien vocales desconocidas. El nombre de los ficheros es el carácter `'x'` seguido de cuatro dígitos, los dos primeros de los cuales coinciden con el numero de bloque.

La estructura resultante es:



1.2. Nombre de las señales y ficheros guía

De cada señal de la bases datos se van a tener múltiples ficheros. Por ahora, ya tenemos dos: el fichero con la señal de audio (extensión `.wav`) y el de marcas (extensión `.mar`). Durante la construcción y ejecución del sistema de reconocimiento se usarán varios ficheros más. Por ejemplo, los ficheros de señal parametrizada (extensión `.prm`), los de señal cuantificada vectorialmente (extensión `.pvq`), etc. Como todos ellos hacen referencia a la misma señal, será conveniente que en su nombre aparezca el de la señal; pero, como cada uno proporciona una información distinta, es necesario distinguir unos de otros, lo cual hacemos con la extensión.

Por otro lado, meter todos los ficheros de una misma señal en el mismo directorio, como se ha hecho con los de señal y marcas, no resulta conveniente por dos motivos: por un lado, no queremos meter ficheros fundamentalmente temporales dentro de los directorios de la base de datos, que es de carácter permanente; por otro lado, podemos desear realizar múltiples experimentos en los que, por ejemplo, se use una parametrización distinta. La solución evidente es colocar los distintos tipos de fichero en directorios distintos.

Todo junto, cada fichero asociado a una misma señal tendrá un nombre completo que se compone de tres partes: el directorio raíz, el nombre de la señal y la extensión. El directorio raíz depende del tipo de fichero y del experimento, el nombre es único para cada señal y la extensión depende del propósito del fichero.

Por ejemplo, podríamos tener los siguientes ficheros:

Directorio	Nombre de la señal	Extensión	Significado
Sen	vocales/block12/a126	.wav	Señal temporal
		.mar	Contenido fonético
PrmUno		.prm	Parametrización
PvqUno		.pvq	Cuantificación vectorial
PvqDos			
PvqTres			
RecUno		.rec	Resultado del reconocimiento
RecDos			

En el ejemplo de la tabla, la misma señal, `vocales/block12/a126`, aparece ocho veces: una vez como señal temporal y de marcas en el directorio `Sen`; otra vez como señal parametrizada en el directorio `PrmUno`; tres veces como señal cuantificada vectorialmente en los directorios `PvqUno`, `PvqDos` y `PvqTres`; y dos veces como resultado del reconocimiento en los directorios `RecUno` y `RecDos`.

Salvo los ficheros de la base de datos, que tenemos en el directorio `Sen`, el resto de ficheros se organizan en directorios de la elección del usuario. Por otro lado, las extensiones están asociadas al tipo de fichero y serán las propias aplicaciones las que las gestionen. Sin embargo, el nombre de las señales se mantiene durante todo el proceso.

Para gestionar esta estructura de ficheros y directorios se usarán *ficheros guía* (extensión `.gui`). Un fichero guía es, simplemente, un fichero en el que se almacenan los nombre de las señales, sin el directorio raíz (pero sí el path relativo dentro de éste) ni la extensión.

Por ejemplo, en el directorio `Gui` se puede encontrar el fichero guía `Gui/train.gui` que incluye los nombres de los ficheros que participarán en el entrenamiento de los modelos acústicos, y cuyas primeras líneas son:

```
usuario:~/TecParla$ head -6 Gui/train.gui
vocales/block00/a000
vocales/block00/a001
vocales/block00/a002
vocales/block00/a003
vocales/block00/a004
vocales/block00/a005
```

En el directorio `Gui` se dispone de tres ficheros guía que serán útiles a lo largo del proyecto:

`Gui/devel.gui`:
Conjunto de desarrollo que se usará para evaluar las prestaciones del sistema durante su construcción.

`Gui/eval.gui`: Conjunto de evaluación que se usará para evaluar las prestaciones del sistema final con señales de contenido desconocido (para el diseñador).

`Gui/train.gui`:
Conjunto de entrenamiento que se usará para modelar acústicamente las unidades fonéticas del sistema de reconocimiento.

1.2.1. Lectura de ficheros guía (y listas de palabras): función `leeLis()`

Dada la estructura de grupos de señales almacenados en ficheros guía, la primera función de propósito general que necesitamos es una que lea el contenido de estos ficheros y devuelva un iterable con los distintos nombres almacenados en ellos. Como en ocasiones podemos estar interesados en leer el contenido de más de un fichero, la función podrá ser invocada con un número variable de argumentos, devolviéndose el resultado de encadenar todos ellos.

La función se usará tanto para leer los ficheros guía con nombres de señales, como para leer las listas de palabras que forman el vocabulario de la tarea a reconocer.

Llamamos a la función que lee las palabras contenidas en ficheros de texto `leeLis()` y, dada su utilidad general, la escribimos en el fichero `ramses/util.py`:

ramses/util.py

```
from pathlib import Path

def leeLis(*ficLis):
    """
    Lee el contenido de uno o más ficheros de texto, devolviendo las palabras en ellos
    contenidas en la forma de lista de cadenas de texto.
    """

    lista = []
    for fic in ficLis:
        with open(fic, 'rt') as fpLis:
            lista += [pal for linea in fpLis for pal in linea.split()]

    return lista
```

En el fichero `Lis/vocales.lis` tenemos la lista de las cinco vocales que se usarán en la tarea:

Lis/vocales.lis

```
a
e
i
o
u
```

Comprobamos el correcto funcionamiento de la función que la lee:

```
>>> leeLis(['Lis/vocales.lis', 'Lis/vocales.lis'])
['a', 'e', 'i', 'o', 'u', 'a', 'e', 'i', 'o', 'u']
```

1.2.2. Manejo de nombres de fichero y directorios: `pathName()` y `chkPathName()`

En los programas de Ramses será necesario construir los ficheros de resultado a partir de la concatenación del directorio raíz, el nombre de la señal y la extensión.

La función `pathName()` realiza esta misión usando la biblioteca `pathlib` y devuelve un objeto de la clase `Path` con el path completo del fichero:

ramses/util.py

```
def pathName(dirFic, nomFic, extFic):
    """
    Construye el path completo del fichero a partir de su directorio raíz 'dirFic', su
    nombre de señal 'nomFic' y su extensión 'extFic'.

    El resultado es un objeto de la clase 'Path'.
    """

    if extFic and not extFic.startswith('.'): extFic = '.' + extFic

    return Path(dirFic).joinpath(nomFic).with_suffix(extFic)
```

Para poder escribir el resultado en un path construido por la función anterior es necesario que el directorio del fichero exista previamente. La función `chkPathName()` se encarga de que ese directorio exista:

ramses/util.py

```
def chkPathName(pathName):
    """
    Crea, en el caso de que no exista previamente, el directorio del fichero ...
    ↪  'pathName'.
    """

    Path(pathName).parent.mkdir(parents=True, exist_ok=True)
```

1.3. Ficheros de marcas y extracción de su contenido fonético

Los ficheros de *marcas* (extensión `.mar`) que indican el contenido de cada señal son ficheros de texto en los que cada línea está formada por una *etiqueta* de tres caracteres seguidos de dos puntos y una serie de *campos* separados por coma.

Por ejemplo:

```
usuario:~/TecParla$ cat Sen/vocales/block53/a531.mar
LHD: SAM, 6.0
SRC: test/a,131
```

```
LBO: 0,,511,a
ELF:
```

Este es un caso muy sencillo de fichero de marcas. En los ficheros usados en la base de datos SpeechDat hay más de treinta etiquetas distintas que proporcionan información acerca de la señal, las condiciones de grabación, el locutor, etc.

El significado de las etiquetas de los ficheros que usaremos en este proyecto es el siguiente:

Etiq.	**Significado**	**Campos**
`LHD`	Tipo y versión del fichero	El tipo es `SAM`, y la versión actual es la 6.0
`SRC`	Origen del fichero	El fichero es la 131 secuencia de test de la vocal `/a/`.
`LBO`	Contenido ortográfico	Principio, centro y final de la señal, seguidos de la ortog
`ELF`	Final del fichero de marcas	No tiene campos

De estas etiquetas, la única que nos interesa es la `LBO`, que, en su cuarto campo, proporciona el contenido ortográfico de la señal. En el fichero `ramses/mar.py` escribimos la función `cogeTrn()` que extrae este contenido ortográfico de un fichero de marcas:

ramses/mar.py

```python
import re

def cogeTrn(ficMar):
    """
    Devuelve el contenido del cuarto campo de la primera etiqueta LBO presente en el
    fichero de Marcas 'ficMar'.
    """
    reLBO = re.compile(r'LBO:(\s*\d*[.]?\d*,){3}(?P<trn>\w+)')

    with open(ficMar) as fpMar:
        for linea in fpMar:
            if (lbo := reLBO.match(linea)): return lbo['trn']
```

Capítulo 2

Sistema trivial de reconocimiento del habla (*no acierta ni una, pero ya tiene la estructura definitiva*).

En una primera fase se va a construir un sistema de reconocimiento con los elementos mínimos. Apenas será capaz de superar el resultado de un dado de cinco caras, pero servirá para poner en marcha la arquitectura global del sistema vista en la Introducción y de las utilidades de la biblioteca `util.py`.

Por motivos históricos, denominaremos *Ramses* al sistema de reconocimiento del habla que se va a desarrollar. El nombre proviene del sistema desarrollado a principios de los años 90 del siglo XX por los miembros del *Grup de Tractament de la Parla* del Departament de Teoria del Senyal i Comunicacions de la UPC. Originalmente, el nombre es el acrónimo de *reconocimiento automático mediante semisílabas*. Aunque el uso de las semisílabas tuvo una corta duración, el nombre gustó mucho y se ha mantenido hasta nuestros días.

2.1. Extracción de características: función `parametriza()`

La función `parametriza()` debe leer cada una de las señales especificadas en la su invocación, parametrizarlas y escribir el resultado en el fichero correspondiente del directorio de salida. En el sistema definitivo, la parametrización propiamente dicha se realizará usando el paradigma de la programación funcional, pasándosele como argumento la función que efectivamente calculará el vector de coeficientes a partir del vector de señal temporal. Por ahora, en el sistema trivial `parametriza()` simplemente copiará la señal temporal en la parametrizada.

A lo largo de este proyecto, la señal parametrizada siempre tendrá la forma `ndarray` de NumPy.

Los ficheros de señal temporal de entrada (extensión `.wav`), se especifican conforme a lo explicado en los apartados 1.2 y 1.2.1, y se leen usando la función `read()` de la biblioteca PySoundFile

La escritura de los ficheros de señal parametrizada (extensión `.prm`), cuyo path completo se determina siguiendo los mismos apartados indicados para la lectura de las señales, se realiza usando la función `save()` de la biblioteca `NumPy`, lo que permitirá su posterior lectura usando la función `load()` de la misma biblioteca.

Dado que las señal parametrizada se escribe en un módulo, pero se lee en otros dos (el de entrenamiento y el de reconocimiento), ubicaremos las funciones de escritura (`escrPrm()`) y lectura (`leePrm()`) en un módulo específico, el `ramses/prm.py`:

ramses/prm.py

```
import numpy as np

def escrPrm(pathPrm, prm):
    """
    Escribe la señal parametrizada 'prm' en el fichero 'pathPrm'.
    """

    with open(pathPrm, 'wb') as fpPrm:
        np.save(fpPrm, prm)


def leePrm(pathPrm):
    """
    Devuelve la señal parametrizada contenida en el fichero 'pathPrm'.
    """

    with open(pathPrm, 'rb') as fpPrm:
        return np.load(fpPrm)
```

Incluimos el código que efectivamente realiza la parametrización en el fichero `ramses/parametriza.py`:

ramses/parametriza.py

```
#! /usr/bin/python3

import soundfile as sf

#from util import *
#from prm import *
exec(open('ramses/util.py').read())
exec(open('ramses/prm.py').read())

def parametriza(dirPrm, dirSen, *guiSen):
    """
    Lee las señales indicadas por 'dirSen', 'guiSen' y 'extSen', y escribe la señal
    parametrizada en el directorio 'dirPrm'.

    En la versión trivial, la señal parametrizada es igual a la señal temporal.
    """

    for nomSen in leeLis(*guiSen):
```

```
        pathSen = pathName(dirSen, nomSen, "wav")
        sen, fm = sf.read(pathSen)

        prm = np.array(sen)

        pathPrm = pathName(dirPrm, nomSen, ".prm")
        chkPathName(pathPrm)
        escrPrm(pathPrm, prm)
```

Ejecutamos la función para parametrizar las señales de los conjuntos de entrenamiento y desarrollo indicadas por los ficheros guía `Gui/train.gui` y `Gui/devel.gui`. Tomamos las señales de audio del directorio `Sen` y escribimos las señales parametrizadas en `Prm/Uno`:

```
>>> run ramses/parametriza.py
>>> parametriza(dirPrm='Prm/Uno', dirSen='Sen', 'Gui/train.gui', 'Gui/devel.gui')
```

2.2. Modelado acústico: función `entrena()`

El modelado acústico consiste en construir una estructura de datos que permita representar la información relevante de cada una de las unidades acústicas a reconocer. Existen distintos planteamientos para abordar este problema, pero todos se basan en suponer que las realizaciones de cada unidad ocupa un lugar geométrico en el espacio de características separado del ocupado por el resto de unidades.

Los tres planteamientos más usados en reconocimiento del habla son:

Distancia:

Consiste en llevar la presunción de localidad de las realizaciones de las distintas unidades a sus últimas consecuencias: la distancia (euclídea, de Mahalanobis, de Itakura-Saito u otra) entre realizaciones de una misma unidad será menor que entre realizaciones de unidades distintas.

La principal ventaja de esta aproximación es su sencillez y facilidad de implementación. Sus principales limitaciones son dos: por un lado, la propia aproximación es a menudo incorrecta, con lo que realizaciones de una cierta unidad pueden estar más cerca de algunas realizaciones de unidades distintas que de otras de la misma.

Por otro lado, los sistemas de reconocimiento basados en distancia presentan fronteras entre unidades cuya forma viene determinada, fundamentalmente, por la elección de la medida de distancia. Por ejemplo, la distancia euclídea impone fronteras planas, mientras que la distancia de Mahalanobis las impone elípticas.

Probabilidad:

Los sistemas basados en probabilidad no intentan responder a la pregunta de qué

unidad está *más cerca* de la señal a reconocer, sino de qué unidad es más probable dada la señal a reconocer. En general, obligan a suponer que la señal responde a un modelo estadístico del que debemos calcular los parámetros.

La gran ventaja de estos sistemas es que permiten adaptarse mucho mejor a las características de las unidades, definiendo las fronteras entre unidades de una manera más versátil. El problema radica en la propia validez del modelo estadístico elegido y el cálculo de sus parámetros que, en general, será más complicado y costoso que para el caso de los sistemas basados en distancia.

Redes neuronales:

En los últimos años se ha impuesto el uso de técnicas de *Deep Learning* basadas en redes neuronales. Estos sistemas no hacen ninguna suposición acerca de las señales a reconocer, sino que sólo lo hacen acerca del propio sistema: una red neuronal suficientemente compleja y entrenada con la cantidad necesaria de muestras de cada unidad será capaz de realizar cualquier tarea de reconocimiento.

Aunque los resultados obtenidos con redes neuronales pueden superar los alcanzados con los otros dos planteamientos, esta técnica no está exenta de problemas: ¿qué complejidad (y, ya puestos, topología) tiene que tener la red?, ¿cómo calculamos sus parámetros?, ¿cuánto material de entrenamiento es necesario?...

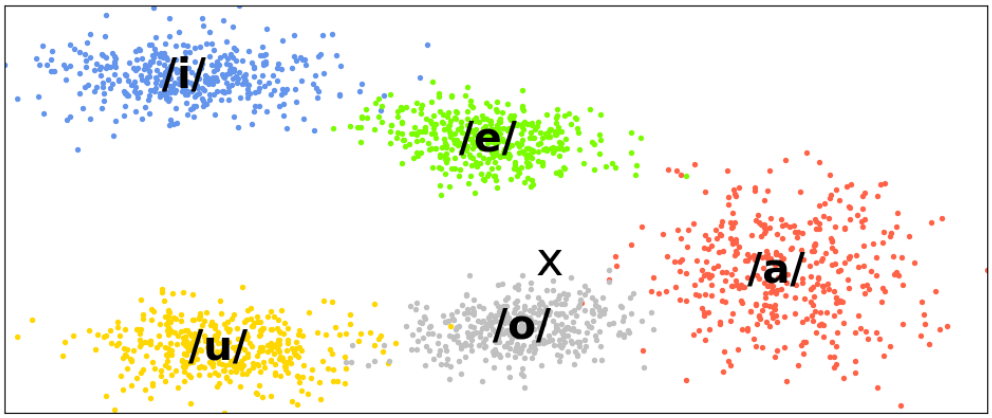
En este proyecto se presentarán los dos primeros planteamientos: en el sistema trivial inicial se usará la distancia euclídea, mientras que más adelante se usarán sistemas probabilísticos basados en funciones de densidad gaussianas y de mezcla de gaussianas. La extensión del sistema al uso de redes neuronales es inmediata, pero las limitaciones del presente curso impiden desarrollar su parte teórica con la necesaria profundidad.

2.2.1. Modelado acústico basado en distancia euclídea

El modelado acústico basado en distancia euclídea consiste en suponer que sonidos semejantes entre sí estarán parametrizados con vectores cercanos. Esta suposición, si la parametrización de la señal es adecuada, es en general cierta.

Por ejemplo, en el problema representado en la figura siguiente, cada vocal ocupa una zona del espacio bidimensional claramente diferenciada. Aunque la señal a reconocer (marcada con la letra `x`) no coincide con ninguna de estas zonas, parece razonable suponer que se trata de un sonido *asimilable* a una `/o/`, ya que es la clase cuyas realizaciones son más cercanas.

Ejemplo de clases bastante separables



Desgraciadamente, en la vida real las clases no son tan fácilmente separables. En la figura, las clases ocupan aproximadamente la posición indicada por los dos primeros formantes de las respectivas vocales (aunque son señales sintéticas y no reales). Además, podría parecer apropiado usar la posición de las formantes para realizar el reconocimiento, pero no existe ningún método suficientemente fiable de hacerlo. Por otro lado, la señal a reconocer ocuparía una posición correspondiente a una de las múltiples vocales que los angloparlantes se atreven a denominar `[a]` (aunque ninguna de ellas se corresponda con lo que existe en los distintos dialectos del latín, u otras muchas lenguas como el eusquera, el japonés o el aimara),

Usando distancia euclídea, el problema del reconocimiento se reduce a determinar la palabra del vocabulario cuyas realizaciones están más cerca de la señal a reconocer. Pero, ¿qué quiere decir *estar más cerca*? y, sobre todo, ¿cómo lo expresamos matemáticamente de manera que se permita su implementación algorítmica?

Hay distintas soluciones a estos dos problemas. Por ejemplo, podríamos calcular la distancia de la señal a todas las señales disponibles para el entrenamiento, y tomar como resultado del reconocimiento la clase a la que perteneciera la más cercana de todas ellas. Otra posibilidad sería escoger la clase cuya distancia media, para todas sus realizaciones, fuera más pequeña.

El problema de estos planteamientos es que exige calcular un montón de distancias; 2000 en nuestra tarea, pero varios millones en según cuál. Una alternativa es tomar uno o varios representantes de cada clase y calcular la distancia únicamente a éstos. Si la elección de los representantes es acertada, el método no sólo resultará mucho más asequible computacionalmente, sino que también será más robusto, ya que evitará que la decisión del reconocimiento pueda recaer en señales de entrenamiento mal condicionadas o, directamente, erróneas (y, desgraciadamente, en todas las bases de datos hay señales de estos tipos).

También para la elección del representante o representantes de cada clase a reconocer existen distintos criterios. Por ejemplo, podríamos tomar aquella señal cuya distancia al elemento más lejano de la clase es mínima. Es el criterio *min-max*, que se puede expresar matemáticamente del modo siguiente:

$$x^* = \operatorname{argmin} \left\{ \lim_{p\to\infty} \left(\frac{1}{K} \sum_k |x_k - x|^p \right)^{1/p} \right\} \tag{2.1}$$

Donde x^* es el representante óptimo de la clase cuyos K elementos son los distintos x_k.

El valor minimizado en la expresión (2.1) es el valor esperado de la *distancia en norma-∞*, ya que es el límite para $p \to \infty$ de la más general *distancia en norma-p*:

$$E\{||x - x^*||_p\} = \left(\frac{1}{K} \sum_k |x_k - x^*|^p \right)^{1/p} \tag{2.2}$$

Un caso particular de especial interés se da para $p = 2$. En ese caso, la distancia norma-2 es idéntica a la distancia euclídea:

$$E\{||x - x^*||_2\} = \sqrt{\frac{1}{K} \sum_k |x_k - x^*|^2} = \sqrt{E\{|x - x^*|^2\}} \tag{2.3}$$

Seleccionamos el representante x^* minimizando el argumento de la expresión (2.3) o, lo que es lo mismo, el radicando de la raíz cuadrada en ella. Para ello, derivamos e igualamos a cero:

$$\frac{d\left[\frac{1}{K}\sum_k (x_k - x^*)^2\right]}{dx^*} = 0 \quad \Rightarrow \quad \frac{1}{K}\left(-2\sum_k x_k + 2Kx^*\right) \quad \Rightarrow \quad x^* = \frac{1}{K}\sum_k x_k \equiv \bar{x} \tag{2.4}$$

Es decir, el representante óptimo x^* es, simplemente, la media de la población, $x^* = \bar{x}$.

2.2.2. Entrenamiento de los modelos usando distancia euclídea

En una primera versión del sistema, gestionaremos los modelos y su estimación desde la propia función. Más adelante se verá que, para poder adaptar el sistema a otros esquemas de modelado, como los basados en probabilidad o redes neuronales, será más conveniente usar un planteamiento orientado a objeto.

Usando distancia euclídea, el modelo se reduce a un vector, que implementaremos como una `ndarray` de `numpy`. Por otro lado, para la estimación de este vector debemos acumular todas las realizaciones correspondientes a cada unidad, así como el número de veces que aparecen en la base de datos. Para ello se usarán sendos diccionarios que iremos rellenando conforme recorremos la lista de señales.

ramses/entrena.py

```
import numpy as np

from util import *
from prm import *
```

```python
from mar import *


def entrena(dirPrm, dirMar, lisUni, ficMod, *figGui):
    """
    Entrena el modelo acústico de las unidades indicadas por el fichero 'lisUni' usando
    las señales de entrenamiento indicadas en los ficheros guía 'ficGui'. El modelo
    resultante se escribe en el fichero 'ficMod'.

    Los ficheros de señal parametrizada se leen del directorio 'dirPrm' y el contenido
    fonético se extrae del cuarto campo de la etiqueta LBO de los ficheros de marcas
    ubicados en 'dirMar'.
    """

    unidades = leeLis(lisUni)

    # Inicializamos el modelo
    modelo = {}

    # Inicializamos el entrenamioento
    total = {unidad : 0 for unidad in unidades}
    numUni = {unidad : 0 for unidad in unidades}

    # Bucle para todas las señales de entrenamiento
    for senyal in leeLis(*figGui):
        # Leemos la señal y el contenido del fichero de marcas
        pathPrm = pathName(dirPrm, senyal, 'prm')
        prm = leePrm(pathPrm)
        pathMar = pathName(dirMar, senyal, 'mar')
        unidad = cogeTRN(pathMar)

        # Actualizamos la información del entrenamiento
        total[unidad] += prm
        numUni[unidad] += 1

    # Recalculamos el modelo
    for unidad in unidades:
        modelo[unidad] = total[unidad] / numUni[unidad]

    # Escribimos el modelo resultante
    chkPathName(ficMod)
    with open(ficMod, 'wb') as fpMod:
        np.save(fpMod, modelo)
```

Entrenamos los modelos de las vocales a partir de las señales del conjunto de entrenamiento indicadas en el fichero `Gui/train.gui`. Obtenemos las señales parametrizadas de entrenamiento del directorio `Prm/Uno`, los contenidos fonéticos de los ficheros de marcas del directorio `Sen` y escribimos los modelos resultantes en `Mod/Uno`:

```
>>> run ramses/entrena.py
>>> entrena(guiSen='Gui/train.gui', dirMar='Sen', dirPrm='Prm/Uno', dirMod='Mod/Uno')
```

2.3. Reconocimiento de las señales: función `reconoce()`

ramses/reconoce.py

```
import numpy as np

from util import *
from prm import *

def reconoce(dirRec, dirPrm, ficMod, *guiSen):
    """
    Determina la unidad cuyo modelo se ajusta mejor a cada señal a reconocer y escribe
    su nombre en el cuarto campo de una etiqueta LBO de un fichero de marcas ubicado
    en el directorio 'dirRec' y del mismo nombre que la señal, pero con extensión
    ↪  '.rec'.
    """

    modelos = np.load(ficMod, allow_pickle=True).item()

    for sen in tqdm.tqdm(leeLis(*guiSen)):
        pathPrm = pathName(dirPrm, sen, '.prm')
        prm = leePrm(pathPrm)

        minDist = np.inf
        for mod in modelos:
            dist = sum(abs(prm - modelos[mod]) ** 2)
            if dist < minDist:
                minDist = dist
                rec = mod

    pathRec =  pathName(dirRec, sen, '.rec')
    chkPathName(pathRec)
    with open(pathRec, 'wt') as fpRec:
        fpRec.write(f'LBO: ,,,{rec}\n')
```

Ejecutamos `reconoce()` para comprobar su correcto funcionamiento:

```
>>> run ramses/reconoce.py
>>> reconoce(dirRec='Rec/Uno', dirPrm='Prm/Uno', dirMod='Mod/Uno',
↪  ficLisMod='Lis/vocales.lis', guiSen='Gui/devel.gui')
```

Podemos comprobar el resultado del reconocimiento para un fichero de la base de datos (es más cómodo hacerlo desde la consola Bash):

```
usuario:~/TecParla$ cat Rec/Uno/vocales/block40/a400.rec
LBO: ,,,o
```

Como podemos ver, el sistema ha reconocido una `/o/` en una señal en la que el locutor pronunciaba una `/a/`. Pero no todo está perdido... Si miramos el resultado para todas las realizaciones de la vocal `/a/` del bloque `block40`, el resultado es:

```
usuario:~/TecParla$ fgrep LBO Rec/Uno/vocales/block40/a40*.rec
Rec/Uno/vocales/block40/a400.rec:LBO: ,,,o
Rec/Uno/vocales/block40/a401.rec:LBO: ,,,a
Rec/Uno/vocales/block40/a402.rec:LBO: ,,,e
Rec/Uno/vocales/block40/a403.rec:LBO: ,,,a
Rec/Uno/vocales/block40/a404.rec:LBO: ,,,a
Rec/Uno/vocales/block40/a405.rec:LBO: ,,,a
Rec/Uno/vocales/block40/a406.rec:LBO: ,,,u
Rec/Uno/vocales/block40/a407.rec:LBO: ,,,o
Rec/Uno/vocales/block40/a408.rec:LBO: ,,,e
Rec/Uno/vocales/block40/a409.rec:LBO: ,,,i
```

Se observa que se han reconocido correctamente cuatro señales; de las otras seis, dos se han reconocido como `/e/`, otras dos como `/o/`, y la `/i/` y la `/u/` se han reconocido una vez cada una.

Desgraciadamente, analizar de este modo el resultado para las 2000 señales del conjunto de desarrollo seria una tarea inhumana. Sobre todo si se quiere realizar muchas veces.

Para evaluar el resultado para todos los fichero del conjunto de desarrollo, podemos sistematizar el análisis y que los cómputos los realice directamente programas del shell. Para ello vamos a echar mano de una serie de utilidades muy usadas en entornos UNIX denominadas *filtros*.

Filtros en UNIX

Un filtro en UNIX es un programa que habitualmente lee la entrada estándar (aunque también se le puede indicar que lea de un fichero de texto), procesa una a una las líneas escritas en ella y escribe el resultado en la salida estándar (la pantalla, aunque podemos redireccionar la salida para escribir el resultado en un fichero).

Cada filtro realiza una única función, que además es sumamente sencilla (*cada uno sólo hace una cosa, pero la hace muy bien*). La idea es combinar distintos filtros haciendo que la salida de uno se envíe a la entrada del siguiente. Son los llamados *pipelines*, que se indican separando los distintos comandos mediante la barra vertical (`|`). Escribiendo pipelines complejos es posible realizar tareas que, como veremos a continuación, pueden acabar siendo muy sofisticadas.

Algunos filtros que usaremos para analizar los resultados del reconocimiento son:

`fgrep`:
: Variante obsoleta, aunque muy usada, del programa `grep`. `grep`, acrónimo de *globally search for a regular expression and print matching lines*, escribe en pantalla las líneas que coinciden con la expresión regular de su argumento. La variante `fgrep` sólo permite expresiones regulares *fijas*, también llamadas *literales*.

`cut`:
: Programa que *corta* las líneas de entrada, seleccionando caracteres y/o campos

concretos de cada una. En el caso de optarse por la selección de campos (opción `-f`), el carácter usado para separarlos puede indicarse con la opción `-d`, que por defecto es el tabulador.

`sort`: Permite ordenar el resultado según distintos criterios (numérico, alfabético, mensual, etc.). Como en el caso de `cut`, también puede manejar campos y caracteres, usando la opción `-t` para indicar el carácter delimitador de los campos (por defecto, cualquier carácter espaciador).

`uniq`: Elimina las líneas duplicadas de la entrada. Si dos o más líneas consecutivas son iguales, sólo escribe en pantalla una. Con la opción `-c` cuenta el número de repeticiones.

Por ejemplo, usando estos cuatro filtros podemos determinar cómo se reconocen las distintas realizaciones de la vocal `/a/`:

```
usuario:~/TecParla$ fgrep LBO Rec/Uno/vocales/block*/a* | cut -d, -f4 | sort ...
↪  | uniq -c
    117 a
     68 e
     55 i
     91 o
     69 u
```

Existen muchos otros filtros en UNIX (`cat`, `sed`, `join`, `head`, `tail`, etc.). Se recomienda consultar la abundante documentación disponible en internet para conocer mejor cuáles son y cómo funcionan (por ejemplo, esta página de la Universidad Estatal de Kansas, o la propia Wikipedia). También se recomienda, muy encarecidamente, consultar el funcionamiento y las opciones de estos y otros programas de UNIX usando la utilidad `man` o la opción `-help` del propio programa.

2.4. Evaluación de los resultados; función `evalua()`

Aunque se podrían utilizar los filtros y resto de comandos de UNIX para analizar los resultados del reconocimiento, vamos a implementar una función Python que lo haga siguiendo el mismo interfaz usado para el resto de programas de Ramses.

La función `evalua()` recorre los ficheros reconocidos y su contenido original, y calcula la *matriz de confusión*, esto es: el número de veces que cada realización es reconocida como cada una de las palabras del vocabulario.

Además, y para permitir la comparación de las prestaciones de sistemas distintos, se calcula la *tasa de exactitud*, definida como Exac = corr/total, donde `corr` representa el número de veces en que la palabra reconocida coincide con la pronunciada, y `total` es el número total de palabras reconocidas.

ramses/evalua.py

```
#from util import *
#from mar import *
exec(open('ramses/util.py').read())
exec(open('ramses/mar.py').read())

def evalua(dirRec, dirMar, *guiSen):
    """
    Calcula la tasa de exactitud en el reconocimiento de las señales indicadas por el
    fichero guía 'guiSen' o la lista de nombres de fichero 'ficRec' y la escribe en
    pantalla. La tasa de exactitud se define como el número de aciertos (corr)
    dividido por el número total de señales (total).

                                    Exac = corr / total

    También escribe la matriz de confusión correspondiente, con cada fila indicando la
    unidad que debería ser reconocida y cada columna la efectivamente reconocida.
    """

    matCnf = {}
    lisPal = set()

    for sen in leeLis(*guiSen):
        pathRec = pathName(dirRec, sen, '.rec')
        rec = cogeTrn(pathRec)

        pathMar = pathName(dirMar, sen, '.mar')
        mar = cogeTrn(pathMar)

        if not mar in matCnf:
            matCnf[mar] = {rec: 1}
        elif not rec in matCnf[mar]:
            matCnf[mar][rec] = 1
        else:
            matCnf[mar][rec] += 1

        lisPal |= {rec, mar}


    for rec in sorted(lisPal):
        print(f'\t{rec}', end='')
    print()
    for mar in sorted(lisPal):
        print(f'{mar}', end='')
        for rec in sorted(lisPal):
            conf = matCnf[mar][rec] if mar in matCnf and rec in matCnf[mar] else 0

            print(f'\t{conf}', end='')
        print()
    print()

    total, corr = 0, 0
    for mar in matCnf:
        for rec in matCnf[mar]:
            conf = matCnf[mar][rec]

            total += conf
```

```
            if rec == mar: corr += conf

    print(f'Exac = {corr / total:.2%}')
```

2.5. Ejecución del sistema trivial de reconocimiento

Ejecutamos las cuatro funciones fundamentales del sistema de reconocimiento:

`parametriza()`: Usamos la versión trivial de `parametriza` y escribimos los ficheros de resultado en el directorio `Prm/Uno`. Se parametriza tanto el conjunto de entrenamiento como el de desarrollo.

`entrena()`: Entrenamos los modelos acústicos usando el conjunto de entrenamiento y escribimos los ficheros de resultado en el directorio `Mod/Uno`.

`reconoce()`: Reconocemos las señales del conjunto de desarrollo y guardamos el resultado en el directorio `Rec/Uno`.

`evalua()`: Evaluamos la exactitud del reconocimiento comparando el resultado almacenado en directorio `Rec/Uno` con los contenidos reales del directorio `Sen`.

```
>>> parametriza('Prm/Uno', 'Sen', 'Gui/train.gui', 'Gui/devel.gui')
>>> entrena('Mod/Uno', 'Sen', 'Prm/Uno', 'Gui/train.gui')
>>> reconoce('Rec/Uno', 'Prm/Uno', 'Mod/Uno', 'Lis/vocales.lis', 'Gui/devel.gui')
>>> evalua('Rec/Uno', 'Sen', 'Gui/devel.gui')
Exac = 25.00%

        a       e       i       o       u
a       117     68      55      91      69
e       63      99      65      74      99
i       48      90      95      40      127
o       72      103     63      79      83
u       57      78      105     50      110
```

2.6. Barra de progreso para el seguimiento de la ejecución; biblioteca `tqdm`

Aunque no es el caso en la tarea reconocida en este curso, a menudo los procesos involucrados en un sistema de reconocimiento del habla pueden durar mucho tiempo; días, semanas e incluso meses. Tal y como se han implementado hasta el momento, las funciones

de Ramses no indican de ningún modo el trabajo realizado o que resta por realizar. Esta falta de información no pasa de ser un cierto engorro cuando los procesos duran unos pocos segundos, pero resulta inaceptable incluso cuando el proceso sólo dura unos pocos minutos.

Dado que la mayor parte de las veces el tiempo se emplea en recorrer una lista de señales, una posible solución es la de sacar el nombre de cada una de las señales conforme son usadas por el algoritmo. Pero esa solución puede generar cientos de miles o millones de líneas de texto en la salida estándar que no aportan prácticamente ninguna información. Lo que es peor, los mensajes que sí puedan ser de interés quedarán enterrados entre ese mogollón de líneas superfluas.

La solución habitualmente empleada en situaciones como ésta son las *barras de progreso*: una representación gráfica que avanza de izquierda a derecha conforme avanza el proceso. Existen distintos paquetes que proporcionan barras de progreso chulísimas; una de las mejores es la desarrollada por Casper O. da Costa-Luis y llamada `tqdm`. `tqdm` es la transliteración latina de la palabra تقدم (*taqadum*, progreso en árabe), y también es el acrónimo de la expresión castellana *te quiero demasiado*.

`tqdm` tiene varios modos de funcionamiento. Por ejemplo, el comando `tqdm.trange()` es un sustituto del comando `range()` que automáticamente muestra el progreso de un bucle que recorra sus valores.

```
>>> import tqdm
>>> from time import sleep
>>> for i in tqdm.trange(1000):
...     sleep(0.1)
...
 63%|                                   | 631/1000 [01:06<00:35, 9.87it/s]
```

Además de la barra gráfica, `tqdm` nos informa del tanto por ciento completado (63 %), el número de iteraciones (631 de 1000), el tiempo transcurrido (01:06) y la estimación del que falta para completar el bucle (00:35), y el número iteraciones por segundo (9.87it/s).

Por su parte, el comando `tqdm.tqdm()` permite recorrer directamente los elementos de un iterable; por ejemplo, los nombres contenidos en una lista de nombres de señal. Para incorporar la barra de progreso de `tqdm`, todo lo que tenemos que hacer es sustituir los bucles de las cuatro funciones, `parametriza()`, `entrena()`, `reconoce()` y `evalua()`, de manera que, en lugar de recorrer una lista de nombres de señal, `lisSen`, recorra el iterador devuelto por `tqdm.tqdm(lisSen)`.

Por ejemplo, en el caso de `parametriza()`, sustituiremos la línea:

ramses/parametriza.py

```
for nomSen in lisSen:
```

Por el bucle:

ramses/parametriza.py

```
for nomSen in tqdm.tqdm(lisSen):
```

Al ejecutar las cuatro funciones, el resultado no varía, pero su ejecución es menos aburrida:

```
>>> parametriza('Prm/Uno', 'Sen', 'Gui/train.gui', 'Gui/devel.gui')
100%|████████████████████████████████████████| 4000/4000 [00:20<00:00, 2141.99it/s]
>>> entrena('Mod/Uno', 'Sen', 'Prm/Uno', 'Gui/train.gui')
100%|████████████████████████████████████████| 2000/2000 [00:16<00:00, 2546.53it/s]
>>> reconoce('Rec/Uno', 'Prm/Uno', 'Mod/Uno', 'Lis/vocales.lis', 'Gui/devel.gui')
100%|████████████████████████████████████████| 2000/2000 [00:20<00:00, 892.55it/s]
>>> evalua('Rec/Uno', 'Sen', 'Gui/devel.gui')
100%|████████████████████████████████████████| 2000/2000 [00:00<00:00, 11152.05it/s]
Exac = 25.00%

        a       e       i       o       u
a       117     68      55      91      69
e       63      99      65      74      99
i       48      90      95      40      127
o       72      103     63      79      83
u       57      78      105     50      110
```

Capítulo 3

Ejecución de las funciones desde línea de comandos y mediante *script*; `todo.py`

Habitualmente, no desearemos ejecutar las funciones desde la consola de Python, sino desde la línea de comandos del Bash, como si se tratara de un programa más. No sólo tendremos más versatilidad a la hora de ejecutar las funciones, sino que, además, podremos planificar su ejecución desde scripts del sistema operativo.

La manera más inmediata de ejecutar un script de Python desde la línea de comandos del sistema operativo es invocar el intérprete `python3` con el nombre del script como argumento:

```
usuario:~/TecParla$ python3 ramses/parametriza.py
```

El problema es que, al hacerlo, lo único que se consigue es que se definan las variables y funciones incluidas en el script, pero éstas no se ejecutan. Realmente, la orden anterior ejecuta todo lo contenido en el script. Lo que pasa es que en él sólo tenemos definiciones, no invocaciones.

Podemos resolver este problema añadiendo la invocación de la función al final del script:

ramses/parametriza.py

```
if __name__ == '__main__':
    parametriza('Prm/Uno', 'Sen', 'Gui/train.gui', 'Gui/devel.gui')
```

La condición `__name__ == '__main__'` se coloca para evitar que la orden se ejecute cuando importamos el archivo. `__name__` es una variable especial de Python que toma el valor `'__main__'` cuando el fichero es ejecutado como programa principal, mientras que indica el nombre del módulo cuando es importado. Al colocar esta comprobación, las funciones definidas en el script están disponibles para otros módulos que lo importen, sin que se ejecute nada. Ahora bien, las órdenes siguientes se ejecutarán siempre que el script sea ejecutado y no importado. Eso incluye cuando se ejecute desde la línea de

comandos, pero también cuando se ejecute desde la consola de Python con la orden `exec(open('ramses/parametriza.py').read())` o desde `ipython3` con el comando `run`.

Ahora, cuando ejecutamos el script desde la línea de comandos, sí se ejecuta la función definida en él:

```
usuario:~/TecParla$ python3 ramses/parametriza.py
100%|████████████████████████████████████████| 4000/4000 [00:20<00:00, 1636.24it/s]
```

3.1. Permisos de ejecución, el *hashbang* y las variables de entorno `PATH` y `PYTHONPATH`

Para construir un auténtico programa ejecutable a partir de una función Python definida en un script, es necesario realizar una serie de modificaciones, tanto en el propio script como en el entorno de desarrollo Bash.

3.1.1. Uso de `chmod` para proporcionar permisos de ejecución

Aunque añadamos las líneas que ejecutan las órdenes dentro del script Python, éste sigue sin ser un programa ejecutable: es necesario invocar al intérprete y dar la ruta completa del script para poderlo ejecutar.

Así, invocando directamente `ramses/parametriza.py`, obtenemos el mensaje siguiente:

```
usuario:~/TecParla$ ramses/parametriza.py
-bash: ramses/parametriza.py: Permission denied
```

Eso es así porque el script no tiene permisos de ejecución. Para modificar los permisos del fichero, hemos de ejecutar el comando `chmod`. Puede consultar el modo de empleo de este comando invocando `man chmod` o `chmod --help`, pero, simplificando, hacemos que un fichero sea ejecutable ejecutando `chmod +x fichero`.

No obstante, aún haciendo ejecutable `ramses/parametriza.py`, el resultado no es el esperado:

```
usuario:~/TecParla$ chmod +x ramses/parametriza.py
usuario:~/TecParla$ ramses/parametriza.py
```

Después de un buen rato, el programa sigue sin dar signos de vida. El problema es que el intérprete de comandos (Bash, en este caso) intenta ejecutar el fichero como si fuera un script en su propio lenguaje, y la primera línea del fichero contiene una orden, `import`, que

Bash interpreta como un programa del sistema operativo. Este programa sirve para salvar en un fichero de imagen el contenido de cualquier servidor X en marcha en el sistema. Como no tenemos ningún servidor X en marcha, el programa se queda bloqueado esperando a que haya uno (el `VcXsrv` no cuenta porque se está ejecutando en la máquina Windows, no en la Ubuntu).

3.1.2. Uso de *hashbang* para usar el intérprete correcto

Para que el shell interprete el fichero como un script de Python, y no como uno de Bash, podemos usar un shebang apropiado. El *shebang* (también conocido como *hashbang*, *sharp-bang*, *hashpling* y unos cuantos nombres más) es una línea al inicio del fichero que empieza por los caracteres `#!` y que sirve para indicarle al shell el programa que debe emplear para interpretar el script.

La sintaxis del hashbang es sencilla: la cadena `#!` debe aparecer al principio de la primera línea del fichero que queremos hacer ejecutable. A continuación, y en la misma línea, debemos proporcionar el **path completo** del intérprete que queremos que ejecute el script.

En nuestro caso queremos que el fichero sea interpretado por el programa `/usr/bin/python3`. Así pues, el hashbang que debemos colocar al principio del script es:

```
#! /usr/bin/python3
```

Al introducir esta línea al principio del fichero, el script ya se comporta como un programa ejecutable:

```
usuario:~/TecParla$ ramses/parametriza.py
100%|                                                  | 4000/4000 [00:20<00:00, 1636.24it/s]
```

3.1.3. Variables de entorno `PATH` y `PYTHONPATH`

Aunque `ramses/parametriza.py` ya es un fichero plenamente ejecutable, sigue habiendo un pequeño inconveniente: hemos de indicar su ruta para poderlo ejecutar (`ramses/parametriza.py`). Ello es así aunque nos movamos al directorio donde se ubica el fichero (deberemos escribir `./parametriza.py`, donde el punto indica el directorio actual).

El motivo por el que es necesario indicar el directorio del fichero ejecutable es que Linux no encuentra el programa, a no ser que su directorio esté incluido entre los indicados por la variable de entorno `PATH`.

PATH

La variable `PATH` es una lista de directorios separados por el carácter dos puntos (`':'`). Cuando se invoca el nombre de un programa, sin indicar explícitamente su ruta, los sistemas

del estilo de UNIX recorren esta lista buscando un archivo ejecutable de ese nombre. Si lo encuentra, lo ejecuta; si no lo hace, da error de `Command not found`.

Puede ver el valor actual de la variable ejecutando el comando `echo $PATH`. Obtendrá una lista más o menos larga en función de la versión del sistema operativo y del número de programas que tenga instalado. A menudo, cuando instala un paquete, el sistema actualizará esta variable para que localice los programas instalados.

Evidentemente, el directorio `ramses` no está inicialmente incluido en la variable `PATH`. Si queremos que esté incluida hemos de ejecutar el comando `export PATH=$PATH:directorio-de-ramses`. Deberíamos ejecutar este comando cada vez que deseemos acceder a los programas de Ramses. Para evitarlo, podemos colocar la orden en el fichero `~/.profile`, que es un fichero de comandos que se ejecuta cada vez que iniciamos una sesión.

Ya de paso, es conveniente añadir también el directorio actual (un punto) en la variable PATH, de manera que sea posible ejecutar cualquier programa presente en el directorio de trabajo sin la necesidad de anteponer el engorroso `'./'`.

PYTHONPATH

Finalmente, también es recomendable añadir la variable `PYTHONPATH`, que informa a Python de dónde debe ir a buscar las bibliotecas importadas con `import`. De no hacerlo, el comportamiento de la inclusión de bibliotecas en los scripts de Python es perfectamente impredecible.

Todo junto, editamos el fichero `~/.profile` para que incluya las siguientes dos líneas:

~/.profile

```
export PATH=$PATH:directorio-de-ramses:.
export PYTHONPATH=$PYTHONPATH:directorio-de-ramses
```

Estos cambios tendrán efecto en cuanto iniciemos una nueva sesión de Bash. Para que también lo tengan en una sesión ya abierta es necesario ejecutar el fichero sin crear un nuevo intérprete de comandos. Es decir, debemos ejecutar las líneas del fichero como si las introdujéramos desde el teclado. Si no lo hacemos así, los cambios se perderán en cuanto finalice la ejecución del fichero.

Para ejecutar el fichero sin crear un nuevo intérprete, se debe usar el comando `source`. A partir de ese momento, el fichero `ramses/parametriza.py` será ejecutable sin necesidad de indicar el directorio en el que se encuentra.

```
usuario:~/TecParla$ source ~/.profile                    # Después de modificar ...
↪  ~/.profile
usuario:~/TecParla$ parametriza.py
100%|████████████████████████████████████████| 4000/4000 [00:20<00:00, 1636.24it/s]
```

Sustitución de los `exec()` por `import`.

Una vez convertidos los scripts en programas ejecutables desde el shell, ya no es necesario ejecutar los módulos de `ramses` con el comando `exec()`. Esto es así porque, cada vez que ejecutemos uno de los programas, se importarán los módulos necesarios de nuevo. Si hacemos alguna modificación entre ejecución y ejecución, ésta quedará reflejada de manera inmediata.

Además, al incluir el directorio `ramses` en la variable `PYTHONPATH`, el intérprete siempre será capaz de encontrar los módulos, sin necesidad de ejecutar el programa desde un directorio concreto. Si siguiéramos usando el comando `exec()` sería necesario ejecutar siempre los programas desde el directorio del proyecto.

El uso de `import` tiene varias ventajas frente al de `exec()`. La primera ya se ha comentado: en combinación con el uso de `PATH` y `PYTHONPATH`, nos permitirá usar el programa desde cualquier directorio. Otra es, justamente, la que ha constituido su principal inconveniente hasta ahora: los módulos se importarán una sola vez. Esto no sólo redundará en una mayor eficiencia, sino que además evitará la posibilidad de recursiones infinitas si, directa o indirectamente, se importa un módulo que a su vez importa al primero. Una situación así, en absoluto descabellada, provocaría un error garrafal en la ejecución del programa si se usa `exec()`. Hasta el punto de que podría dejar *temblando* todo el ordenador.

Por todo ello, se recomienda eliminar o comentar en el código de los distintos scripts las órdenes `exec()`, dejando únicamente las `import`.

3.2. Análisis de los argumentos en línea de comandos usando la biblioteca `docopt`

Con las modificaciones del apartado anterior, el script `parametriza.py` ya es un programa ejecutable desde cualquier directorio del sistema. Ahora bien, siempre realiza la operación siguiente:

```
parametriza('Prm/Uno', 'Sen', 'Gui/train.gui', 'Gui/devel.gui')
```

Evidentemente, deseamos que nuestro programa tenga un comportamiento más general, permitiendo trabajar con otras bases de datos o con otras funciones de parametrización. El modo de hacer esto es pasar los argumentos y opciones del programa a través de la línea de comandos. En Python, de manera análoga a como se hace en C/C++, Bash y otros lenguajes de programación, el acceso a los argumentos del programa se realiza a través de la variable `sys.argv`, que es una lista de cadenas de texto en la que `sys.argv[0]` es el nombre del programa, y los distintos `sys.argv[1:]` son los argumentos con los que se le invocó.

El análisis de estos argumentos es más complejo de lo que parece a simple vista. El modo más sencillo sería tratar los argumentos de manera *posicional*. Así, en el ejemplo anterior, `sys.argv[1]` podría el directorio de las señales parametrizadas; `sys.argv[2]`, el de las señales temporales; etc. El problema de este planteamiento es doble: por un lado, es muy rígido, con lo que cualquier modificación en la funcionalidad del programa puede desbaratar la asignación entre argumentos y elementos de `sys.argv`. Por otro, no aporta ninguna información al usuario del significado de los argumentos que proporciona. Cuando el número de éstos es pequeño, la falta de información no resulta especialmente grave, pero en una invocación con una docena de argumentos, el galimatías puede resultar excesivo.

Tres son los principales aspectos a tener en cuenta a la hora de diseñar el interfaz de usuario en línea de comandos:

- La estrategia de asignación de los argumentos en línea de comandos a las variables del programa.
- El modo de informar al usuario de cuáles son los argumentos del programa y cómo se le pasan.
- Tan importante o más que los dos anteriores: la correcta correspondencia entre la información que se le pasa al usuario y lo que realmente hace el programa.

Estos tres aspectos han sido largamente tratados desde los inicios de la programación informática. El problema es que la solución no es única, y han sido muchas las alternativas propuestas, muchas veces incompatibles entre ellas. Para poner un poco de orden en este y otros muchos aspectos de los sistemas operativos y lenguajes de programación, el IEEE lanzó en la década de 1980 el estándar POSIX, luego ampliado por GNU, basado en el uso de opciones nombradas y argumentos posicionales.

El estándar define cómo deben ser las opciones y argumentos con que se invoca un programa, pero deja en manos del programador cómo se gestionan y la información que se proporciona al usuario. En este curso se usará la biblioteca `docopt`, que resuelve, de un modo *que le hará sonreir*, estos dos aspectos.

En el documento Gestión de opciones y argumentos usando docopt dispone de una pequeña introducción al estándar POSIX, con las ampliaciones de GNU, y de la biblioteca `docopt`, que será lo que usaremos en este curso para esta tarea. Se recomienda su lectura y, sobre todo, la de los enlaces suministrados en el documento.

3.2.1. Mensaje de sinopsis para analizar las opciones de `parametriza`

El punto de partida para gestionar las opciones de un programa usando `docopt` es escribir la sinopsis del programa, que es el mensaje que se le mostrará al usuario cuando ejecute el programa con argumentos erróneos.

La sinopsis debe empezar con una línea explicativa del cometido del programa, seguida de una sección encabezada por la cadena `'Usage:'` y formada por tantas líneas como modos de

empleo diferentes tenga el programa. En estas líneas, deberemos indicar todas las opciones y argumentos correspondientes, aunque también podemos agrupar las distintas opciones con la cadena `'[options]'`, postergando su descripción a la sección `'Opciones:'`.

En el caso de `parametriza.py`, usaremos las opciones con nombre para indicar los directorios de las señales, reservado el resto de argumentos, posicionales, para indicar el nombre de los ficheros guía con los nombres de las señales a parametrizar.

Aunque en el programa sólo habrá un modo de empleo principal, ilustraremos la posibilidad de tener más de uno usando las opciones especiales `--help` y `--versión`. La primera, que también admite la forma corta `-h`, sirve para mostrar ayuda detallada del modo de empleo del programa (será el propio mensaje de sinopsis). La segunda muestra la versión del programa. En ambos casos el programa finaliza su ejecución tras procesarlas, de lo que se encarga el propio `docopt`.

Inicializamos la parte del script dedicada a su ejecución como programa con los elementos mencionados:

ramses/parametriza.py

```
################################################################################
# Invocación en línea de comandos
################################################################################

if __name__ == '__main__':
    from docopt import docopt
    import sys

    Sinopsis = f"""
Parametriza una base de datos de señal.

Usage:
    {sys.argv[0]} [options] <guiSen>...
    {sys.argv[0]} -h | --help
    {sys.argv[0]} --version
```

Algunos detalles de esta primera parte:

- Usamos una *f*-string como mensaje de sinopsis para poder usar `sys.argv[0]` como nombre del programa. De este modo se garantiza que el nombre mostrado se corresponde con el del programa ejecutado.
    - También podríamos usar la *f*-string para vincular el valor por defecto de un argumento con una variable del programa.
- Se elimina el sangrado del mensaje de sinopsis para evitar que aparezca al mostrar el mensaje por pantalla.
- Encerramos entre corchetes oblicuos (`'<>'`) el nombre del fichero guía `guiSen`. Esto es necesario para hacer que `docopt` interprete la cadena `'<guiSen>'` como un argumento y no como un comando literal. Alternativamente, también se hubiera podido usar una expresión formada enteramente por mayúsculas y subrayados (por ejemplo, `'GUI_SEN'`).

- Los tres puntos al final de `'<guiSen>...'` indican que se puede pasar uno o más ficheros guía. El o los argumentos correspondientes se pasarán como una lista de cadenas de texto, en lugar de como una única cadena.

Sección `Opciones`; directorios de entrada/salida

Para `docopt`, todo lo que no forme parte de la sección `Usage:` y que empiece por guion es considerado la descripción de una opción del programa. Aunque, por tanto, `docopt` no requiere una sección específica para describir las opciones, es recomendable hacerlo. En este caso, lo haremos bajo el epígrafe `Opciones:`, en castellano, para resaltar su carácter no *oficial*. Sin embargo, téngase en cuenta que la sección `Usage:` sí es oficial, y debe aparecer de manera literal en inglés.

Las opciones del programa son los directorios y la función de parametrizar. Los primeros no tienen mayor complicación, y los enumeramos en la sección `'Opciones'`. En cada caso definimos una opción *corta*, con un guión y un único carácter, y una opción *larga*, con dos guiones y más de un carácter. El nombre dado al argumento de estas opciones es accesorio. Usaremos un nombre, en mayúsculas, explicativo del tipo de argumento esperado (en este caso, `'PATH'`).

ramses/parametriza.py

```
Opciones:
    -s PATH, --dirSen=PATH  Directorio con las señales temporales [default: .]
    -p PATH, --dirPrm=PATH  Directorio con las señales parametrizadas [default: .]
```

Fijémonos en que entre las opciones cortas y largas se coloca una coma (`,`) y que su descripción se separa con un mínimo de dos espacios en blanco.

Secciones `Argumentos` y `Parametrización Trivial`

El resto de la sinopsis es de formato libre. Suele ser conveniente aprovecharla para explicar el resto de argumentos del programa, su modo de funcionamiento, etc. En la versión trivial de la parametrización sólo tenemos un argumento: el nombre del fichero o ficheros guía con los nombres de las señales a parametrizar, `<guiSen>...`.

Incluimos la descripción de los argumentos en la sección (de nombre libre) `'Argumentos'`:

ramses/parametriza.py

```
Argumentos:
    <guiSen>  Nombre del fichero guía con los nombres de las señales a parametrizar.
              Pueden especificarse tantos ficheros guía como sea necesario.
```

En la sección `Parametrización Trivial` explicamos el funcionamiento de la parametrización:

ramses/parametriza.py

```
Parametrización trivial:
    En la versión trivial del sistema, la parametrización simplemente copia la señal
    temporal en la salida.
```

3.2.1.1. Sinopsis de `parametriza.py`

Todo junto, el mensaje de ayuda de la parametrización es:

ramses/parametriza.py

```
    Sinopsis = f"""
Parametriza una base de datos de señal.

Usage:
    /home/albino/TecParla/apuntes/ramses/parametriza.py [options] <guiSen>...
    /home/albino/TecParla/apuntes/ramses/parametriza.py -h | --help
    /home/albino/TecParla/apuntes/ramses/parametriza.py --version

Opciones:
    -s PATH, --dirSen=PATH  Directorio con las señales temporales [default: .]
    -p PATH, --dirPrm=PATH  Directorio con las señales parametrizadas [default: .]

Argumentos:
    <guiSen>  Nombre del fichero guía con los nombres de las señales a parametrizar.
              Pueden especificarse tantos ficheros guía como sea necesario.

Parametrización trivial:
    En la versión trivial del sistema, la parametrización simplemente copia la señal
    temporal en la salida.
"""
```

3.2.2. Análisis de argumentos y ejecución del script `parametriza.py`

Una vez construido el mensaje de sinopsis, se lo pasamos a la función `docopt()` para que lo analice y nos devuelva los argumentos en forma de diccionario:

ramses/parametriza.py

```
args = docopt(Sinopsis, version=f'{sys.argv[0]}: Ramses v3.4 (2020)')
```

En el caso de los directorios de señal temporal y parametrizada y de los ficheros guía, el formato almacenado en el diccionario ya es el deseado: cadenas de texto, en los dos primeros, y una lista de cadenas de texto, para los últimos. Así pues, no es necesaria ninguna conversión y podemos usar directamente los valores del diccionario:

ramses/parametriza.py

```
dirSen = args['--dirSen']
dirPrm = args['--dirPrm']

guiSen = args['<guiSen>']
```

Es importante remarcar el hecho de que hay que escribir los nombres de los argumentos exactamente igual a como aparecen en la sinopsis, y el que las opciones largas tienen preponderancia frente a las cortas.

Finalmente, ya tenemos todos los argumentos necesarios para invocar la función `parametriza()`:

ramses/parametriza.py

```
parametriza(dirPrm, dirSen, *guiSen)
```

Fijémonos en que los nombres de los ficheros guía, que `docopt` pasa como lista de cadenas de texto, debe *desplegarse* usando el asterisco para que aparezcan como argumentos independientes en la invocación de la función.

Comprobamos el funcionamiento del programa. En primera instancia, sin argumentos (que es un modo de invocación incorrecto bastante habitual), para que nos informe de los modos correctos de invocación del programa:

```
usuario:~/TecParla$ parametriza.py
Usage:
    /home/albino/TecParla/apuntes/ramses/parametriza.py [options] <guiSen>...
    /home/albino/TecParla/apuntes/ramses/parametriza.py -h | --help
    /home/albino/TecParla/apuntes/ramses/parametriza.py --version
```

Si ejecutamos `parametriza.py --help`, se nos muestra el mensaje de sinopsis:

```
usuario:~/TecParla$ parametriza.py --help
Parametriza una base de datos de señal.

Usage:
    /home/albino/TecParla/apuntes/ramses/parametriza.py [options] <guiSen>...
    /home/albino/TecParla/apuntes/ramses/parametriza.py -h | --help
    /home/albino/TecParla/apuntes/ramses/parametriza.py --version

Opciones:
    -s PATH, --dirSen=PATH  Directorio con las señales temporales [default: .]
    -p PATH, --dirPrm=PATH  Directorio con las señales parametrizadas [default: .]

Argumentos:
    <guiSen>  Nombre del fichero guía con los nombres de las señales a parametrizar.
              Pueden especificarse tantos ficheros guía como sea necesario.

Parametrización trivial:
    En la versión trivial del sistema, la parametrización simplemente copia la señal
    temporal en la salida.
```

Finalmente, parametrizamos las señales de la base de datos con la parametrización trivial:

```
usuario:~/TecParla$ parametriza.py -s Sen/ -p Prm/Uno Gui/train.gui Gui/devel.gui
100%|████████████████████████████████████████| 4000/4000 [00:04<00:00, 917.49it/s]
```

3.2.3. Análisis de argumentos y ejecución del resto de scripts

Visto lo realizado para el programa `parametriza.py`, la gestión de argumentos y ejecución del resto de programas es relativamente más sencilla. Básicamente, lo que se ha de hacer es construir un mensaje de sinopsis que incluya los argumentos de la función, analizarla con `docopt` para extraer los argumentos e invocar con ellos a la función.

Script `entrena.py`

Añadimos, al final del script, las órdenes siguientes:

ramses/entrena.py

```
##############################################################################
# Invocación en línea de comandos
##############################################################################

if __name__ == '__main__':
    from docopt import docopt
    import sys

    Sinopsis = f"""
Entrena los modelos acústicos a partir de una base de datos de entrenamiento

Usage:
    {sys.argv[0]} [options] <guiSen>...
    {sys.argv[0]} -h | --help
    {sys.argv[0]} --version

Opciones:
    -p PATH, --dirPrm=PATH  Directorio con las señales parametrizadas [default: .]
    -a PATH, --dirMar=PATH  Directorio con los ficheros de marcas [default: .]
    -m PATH, --dirMod=PATH  Directorio con los modelos generados [default: .]

Argumentos:
    <guiSen>   Nombre del fichero guía con los nombres de las señales usadas en el
               entrenamiento. Pueden especificarse tantos ficheros guía como sea
               necesario.

Entrenamiento:
    El programa lee los contenidos fonéticos de los ficheros de marcas y entrena los
    modelos de las unidades fonéticas encontradas en ellos.
"""

    args = docopt(Sinopsis, version=f'{sys.argv[0]}: Ramses v3.4 (2020)')

    dirPrm = args['--dirPrm']
    dirMar = args['--dirMar']
    dirMod = args['--dirMod']

    guiSen = args['<guiSen>']

    entrena(dirMod, dirMar, dirPrm, *guiSen)
```

Script `reconoce.py`

Añadimos, al final del script, las órdenes siguientes:

ramses/reconoce.py

```
##############################################################################
# Invocación en línea de comandos
##############################################################################

if __name__ == '__main__':
    from docopt import docopt
    import sys

    Sinopsis = f"""
Reconoce una base de datos de señales parametrizadas

Usage:
    {sys.argv[0]} [options] --lisMod=FILE <guiSen>...
    {sys.argv[0]} -h | --help
    {sys.argv[0]} --version

Opciones:
    -r PATH, --dirRec=PATH  Directorio con los ficheros del resultado [default: .]
    -p PATH, --dirPrm=PATH  Directorio con las señales parametrizadas [default: .]
    -m PATH, --dirMod=PATH  Directorio con los modelos acústicos [default: .]
    -l FILE, --lisMod=FILE  Fichero con la lista de unidades a reconocer

Argumentos:
    <guiSen>  Nombre del fichero guía con los nombres de las señales a reconocer.
              Pueden especificarse tantos ficheros guía como sea necesario.
"""

    args = docopt(Sinopsis, version=f'{sys.argv[0]}: Ramses v3.4 (2020)')

    dirRec = args['--dirRec']
    dirPrm = args['--dirPrm']
    dirMod = args['--dirMod']
    ficLisMod = args['--lisMod']

    guiSen = args['<guiSen>']

    reconoce(dirRec, dirPrm, dirMod, ficLisMod, *guiSen)
```

Script `evalua.py`

Añadimos, al final del script, las órdenes siguientes:

ramses/evalua.py

```
##############################################################################
# Invocación en línea de comandos
##############################################################################

if __name__ == '__main__':
    from docopt import docopt
```

```
    import sys

    Sinopsis = f"""
Evalua el resultado de un reconocimiento

Usage:
    {sys.argv[0]} [options] <guiSen>...
    {sys.argv[0]} -h | --help
    {sys.argv[0]} --version

Opciones:
    -r PATH, --dirRec=PATH  Directorio con los ficheros del resultado [default: .]
    -a PATH, --dirMar=PATH  Directorio con los ficheros de marcas [default: .]

Argumentos:
    <guiSen>  Nombre del fichero guía con los nombres de las señales reconocidas.
              Pueden especificarse tantos ficheros guía como sea necesario.

Evaluación:
    Siendo OK el número de unidades reconocidas correctamente y KO el de errores,
    el programa saca por pantalla la exactitud calculada como OK / (OK + KO)
"""

    args = docopt(Sinopsis, version=f'{sys.argv[0]}: Ramses v3.4 (2020)')

    dirRec = args['--dirRec']
    dirMar = args['--dirMar']

    guiSen = args['<guiSen>']

    evalua(dirRec, dirMar, *guiSen)
```

3.3. Construcción de un script Bash con el sistema completo: `todo.sh`

Con los cuatro scripts del sistema de reconocimiento reconvertidos a programas plenamente operativo, ya podemos ejecutar todos los pasos desde el intérprete de comandos:

```
usuario:~/TecParla$ parametriza.py -s Sen -p Prm/Uno Gui/train.gui Gui/devel.gui
100%|████████████████████████████████████████| 4000/4000 [00:02<00:00, 1617.33it/s]
usuario:~/TecParla$ entrena.py -p Prm/Uno -a Sen -m Mod/Uno Gui/train.gui
100%|████████████████████████████████████████| 2000/2000 [00:00<00:00, 2711.10it/s]
usuario:~/TecParla$ reconoce.py -p Prm/Uno -r Rec/Uno -m Mod/Uno -l Lis/vocales.lis ...
↪  Gui/devel.gui
100%|████████████████████████████████████████| 2000/2000 [00:02<00:00, 872.01it/s]
usuario:~/TecParla$ evalua.py -r Rec/Uno -a Sen Gui/devel.gui
100%|████████████████████████████████████████| 2000/2000 [00:00<00:00, 11812.10it/s]
Exac = 25.00%

        a       e       i       o       u
```

```
a       117     68      55      91      69
e       63      99      65      74      99
i       48      90      95      40      127
o       72      103     63      79      83
u       57      78      105     50      110
```

De todos modos, aún hay varios aspectos que se podrían mejorar:

- Cada comando ha de escribirse entero, con todos sus argumentos, en la línea de comandos.
  - Escribir tantos comandos puede resultar farragoso y propenso a errores; sobre todo con los más largos.
  - Algunos argumentos se repiten, y tienen que hacerlo, en varios comandos. Por ejemplo, los directorios de las señales temporales o parametrizadas.
  - Otros argumentos son prácticamente fijos y sólo cambiarán puntualmente. Por ejemplo, los ficheros guía o la lista de unidades fonéticas.
  - Podríamos plantear un experimento nuevo, en el que cambiaríamos el `Uno` por `Dos`, pero eso nos obligaría a modificar todos los comandos, uno por uno, lo cual no sólo es muy propenso a errores, sino también a desastres.
- No se guarda ningún registro de lo que se ha ejecutado o del resultado obtenido.

Para mejorar la *experiencia de usuario* en el manejo del sistema de reconocimiento, vamos a escribir un script de Bash que automatice todas las tareas y facilite su gestión.

El script debe reunir las siguientes características:

- Debe invocar secuencialmente los distintos programas...
  - ... pero debe interrumpir la ejecución si alguno de ellos falla.

  Si no tenemos cuidado con finalizar prematuramente el script cuando se produce un error en alguno de los pasos, podemos acabar obteniendo resultados absurdos, incorrectos o que no reflejen lo que realmente queríamos ejecutar.
- Debe proporcionar, y guardar en un fichero de registro, la información más completa posible de la ejecución:
  - Máquina en la que se realiza, fecha de inicio y final, etc.
  - Comandos ejecutados con todas sus opciones y argumentos, y en un formato que facilite su reproducción; por ejemplo, mediante *copia y pega*.
  - Mensajes proporcionados por cada comando; especialmente los de error.
- Debe ser fácil de entender cómo hace lo que hace, y de modificarlo.

- Tiene que tener todos los comentarios necesarios; pero, si no es necesario ninguno, mejor todavía.

  Los comentarios se ponen, o no; y se modifican cuando se modifica el código, o no. Un comentario erróneo o caduco es mucho peor que su ausencia.

- La legibilidad y mantenibilidad del código no mejora por añadir comentarios a instrucciones *oscuras*, sino evitándolas.

  Eso no quiere decir que no se usen construcciones avanzadas del lenguaje cuando éstas permitan simplificar el código. Tampoco quiere decir que no se usen construcciones oscuras, cuando no quede más remedio (lo cual, en Bash, es harto habitual). En estos casos, un buen comentario sí puede resultar de utilidad, incluso para el autor del código.

3.3.1. Cabecera del script, y otros elementos comunes

En la cabecera del script se realizan buena parte de las tareas necesarias para cumplir los requisitos mencionados en la sección anterior, así como otras necesarias para facilitar el uso del script. Muchas de estas tareas pueden ser compartidas por otros scripts para esta u otra aplicación.

todo.sh

```
#! /bin/bash

NOM=Uno

DIR_WRK=.                  # Es mejor $PWD, pero . es más conciso

DIR_LOG=$DIR_WRK/Log
FIC_LOG=$DIR_LOG/$(basename $0 .sh).$NOM.log
[ -d $DIR_LOG ] || mkdir -p $DIR_LOG

exec > >(tee $FIC_LOG) 2>&1

hostname
pwd
date
```

1. En la primera línea del script debemos escribir el hashbang (ya comentado en el apartado 3.1.2). En este caso, será el correspondiente al intérprete de comandos Bash.

2. Definición del nombre del experimento `NOM=Uno`. Este nombre permite particularizar los ficheros y directorios usados por el script, de manera que se puede realizar un experimento completamente nuevo con sólo cambiar el valor de esta variable.

3. Definición del directorio de trabajo `DIR_WRK`. Usado para localizar los ficheros y directorios usados en el script. En principio, este directorio coincide con el directorio

actual, con lo que su valor es `DIR_WRK=$PWD`. De este modo, los comandos que se escribirán en pantalla puede ser reproducidos desde cualquier otro directorio del sistema. No obstante, como se supone que siempre ejecutaremos este script desde el mismo directorio, y añadir el path completo provocará la aparición de líneas de longitud muy larga en el fichero de registro, optamos por la más escueta `DIR_WRK=.`.

4. Creación del fichero de registro o *log*. En este fichero se escribirá toda la información referida a la ejecución del script: desde los comandos ejecutados hasta los mensajes proporcionados por ellos.

   El fichero de log se crea en el directorio `DIR_LOG` e incluye en su nombre el del script (sin extensión), `$(basename $0 .sh)`, y el del experimento `$NOM`.

   - La expresión `$(comando)` ejecuta el comando encerrado entre los paréntesis y sustituye su salida estándar en la línea de comandos. Es una alternativa más cómoda, clara y potente a las tradicionales comillas invertidas del UNIX (`` `comando` ``).
   - `basename nom_fic [ext_fic]` es un comando del shell que devuelve el nombre del fichero sin el directorio. Si se especifica la extensión `ext_fic`, y el nombre del fichero acaba en ella, la suprime.
   - Las variables posicionales `$0 ... $9` indican los argumentos de la línea de comandos: `$0` es el nombre del comando ejecutado; el resto, `$1 ... $9`, sus argumentos.
   - Así pues, `$(basename $0 .sh)` se sustituye por `'todo'`, y el nombre completo del fichero de log es `FIC_LOG=./Log/todo.Uno.log`.

5. Si el directorio `DIR_LOG` no existe, lo creamos con la orden `mkdir -p $DIR_LOG`, dentro del *cortocircuito* `[ -d $DIR_LOG ] || mkdir -p $DIR_LOG`.

   - El operador `||` es el operador 'o lógico' en Bash.
   - El cortocircuito `cmd1 || cmd2` comprueba el código de retorno del comando `cmd1`. Si es cero, el conjunto es cierto sin necesidad de comprobar el valor de retorno de `cmd2`; si no lo es, debemos ejecutar `cmd2` para comprobar su valor de retorno.
     - **Atención**: recuerde que, en Bash, el valor de retorno 0 representa que el comando se ha ejecutado con éxito, y es análogo al valor lógico `true`; mientras que un valor de retorno distinto de 0 indica que ha habido un problema en la ejecución del comando, y es análogo al valor lógico `false`.
     - Por tanto, la expresión `cmd1 || cmd2` es equivalente a:

       ```
       if ! cmd1 ; then cmd2; fi
       ```

     - Análogamente, el operador `&&` se usa como atajo de la estructura condicional no negada: el segundo comando sólo se ejecuta si el primero da como resultado verdadero.
       Es decir, la expresión `cmd1 && cmd2` es equivalente a:

```
if cmd1 ; then cmd2; fi
```

- Los corchetes son un sinónimo elegante del comando `test`. Es decir, `[ -d path ]` es equivalente a `test -d path`.
  - `test` realiza tests sobre expresiones, ficheros, valores, etc.
    - En concreto, `test -d path` comprueba si `path` existe y es un directorio.
    - `test` devuelve el valor 0 (`true`), si se verifica con éxito la condición del test, y 1 (`false`) si no lo hace.
    - Es sumamente recomendable consultar el funcionamiento de `test` con la orden `man test`.
  - Hay alternativas a `[ cond ]`, pero no son tan estándar (aunque sí muy recomendables):
    - `[[ cond ]]` extiende `[ cond ]` de varias maneras; particularmente, permitiendo el uso de expresiones regulares.
    - `(( cond ))` permite operaciones aritméticas sobre enteros.
    - `( cond )` ejecuta `cond` en un subshell, evitando efectos colaterales en el shell de la invocación.

6. Redirigimos la salida del script al fichero de log con el comando `exec > >(tee $FIC_LOG) 2>&1`.

   - `exec [comando]` sustituye la ejecución actual con el `comando` indicado, pero lo hace en el propio proceso actual; es decir, no se crea un proceso nuevo. Si no se indica el `comando` a ejecutar, lo que se pasa a ejecutar es el proceso actual. Es decir, sustituye el proceso actual por el proceso actual sin crear un proceso nuevo. Por tanto, no hace nada...
     - Pero sí es útil, porque permite redirigir la salida del proceso actual a partir del punto en el que se invoca.
   - `>(pipeline)` introduce la denominada sustitución de proceso. Ésta consiste en ejecutar el *pipeline* en un subshell que se comporta como un fichero abierto para escritura. Lo que escribamos en él es leído por el pipeline desde su entrada estándar.

     Esta ciertamente oscura capacidad del Bash exige precisión en la sintaxis. En concreto, debe haber un espacio antes de `>(` y no puede haberlo entre `>` y `(`.

     En nuestro script, la sustitución de proceso es fundamental porque es la herramienta que permitirá que todo lo que tendría que aparecer por pantalla efectivamente lo haga y, además, se escriba en un fichero de registro.
   - El comando `tee fichero` es una derivación en T del flujo de entrada/salida: toma su entrada estándar y la envía a su salida estándar, y, a su vez, la escribe en el `fichero` indicado por su argumento.

- La partícula `2>&1` envía la salida de error estándar a la salida estándar, de manera que ambas se escriben en el mismo sitio.
- Todo junto, el efecto de `exec > >(tee $FIC_LOG) 2>&1` es escribir tanto en pantalla como en el fichero `$FIC_LOG` la salida, tanto la estándar como la de error, de todos los comandos que se ejecuten en el script.

7. Finalmente, escribimos, en pantalla y en el fichero de registro, el nombre de la máquina en la que se está ejecutando el script, el directorio de trabajo y la fecha de comienzo de su ejecución.

3.3.2. Selección de los elementos del proceso a ejecutar

Las operaciones fundamentales que debe realizar el script son cuatro:

PRM: Parametrizar las señales de entrenamiento y test.

ENT: Entrenamiento de los modelos acústicos.

REC: Reconocimiento de las señales de test.

EVA: Evaluación de los resultados.

A menudo no deseamos realizar las cuatro operaciones, sino sólo alguna de ellas. Para gestionar cuáles se van a ejecutar y cuáles no, definimos cuatro variables con el valor `true` o `false`, en función de si deseamos realizar la operación correspondiente o no.

todo.sh
```
PRM=true
ENT=true
REC=true
EVA=true
```

En verdad, `true` y `false` son dos programas que siempre devuelven el mismo valor de retorno: *verdad* (0) o *mentira* (1), respectivamente. En este script, se usarán en cortocircuitos para decidir si un comando ha de ejecutarse o no.

3.3.3. Variables globales del sistema

Algunas variables del sistema de reconocimiento, como los directorios donde se almacenan los distintos tipos de fichero, los ficheros guía o la lista de unidades a reconocer, suelen tener un nombre predeterminado y/o son usados por más de una de las etapas. Así, por ejemplo, almacenaremos la señal parametrizada en el directorio `DIR_PRM=$DIR_WRK/Prm/$NOM`, y este directorio será usado en la parametrización, el entrenamiento y el reconocimiento.

Es conveniente definir estas variables en un único punto de manera que, en caso de ser necesario modificarlas, sea sencillo acceder a ellas. Usaremos nombres en mayúscula, para

distinguirlos de los argumentos que efectivamente pasaremos a los programas, que, salvo para los ficheros guía, usarán nombres en joroba de camello.

En algún caso, convendrá generar el directorio del fichero, si este no existe al ejecutar el script. Lo haremos como se hizo para el fichero de registro (aunque realmente no es necesario), con un cortocircuito.

todo.sh

```
# Ficheros guía

DIR_GUI=$DIR_WRK/Gui
GUI_ENT=$DIR_GUI/train.gui
GUI_DEV=$DIR_GUI/devel.gui

# Directorios de las señales

DIR_SEN=$DIR_WRK/Sen
DIR_PRM=$DIR_WRK/Prm/$NOM
DIR_MOD=$DIR_WRK/Mod/$NOM
DIR_REC=$DIR_WRK/Rec/$NOM

# Lista de unidades a reconocer

LIS_MOD=$DIR_WRK/Lis/vocales.lis

# Fichero de resultados del reconocimiento

FIC_RES=$DIR_WRK/Res/$NOM.res
[ -d $(dirname $FIC_RES) ] || mkdir -p $(dirname $FIC_RES)
```

3.3.4. Invocación de los programas

Para conseguir escribir en el fichero de log el comando exacto que se va a ejecutar vamos a aplicar un pequeño truco: en lugar de ejecutar el comando directamente, definimos una variable, `EXE` con la orden que deseamos ejecutar; a continuación, escribimos en pantalla el contenido de la variable y, finalmente, la invocamos.

Por otro lado, queremos que el script finalice si se produce algún error en la ejecución de cualquiera de sus comandos. En caso contrario, el resto de comandos proseguirá su ejecución. En el mejor de los casos, los comandos que siguen al fallido también fallarán, enmarañando la salida en pantalla y lo que se escribe en el fichero de log. En el peor de los casos, alguno de los comandos proseguirá su ejecución (probablemente, usando el resultado de ejecuciones anteriores), con lo que se obtendrá un resultado erróneo que puede pasar inadvertido.

Todo junto, la ejecución de cada uno de los comandos responderá a la forma:

```
EXEC="programa argumentos_de_programa"
$CMD && echo $EXEC && $EXEC || exit 1
```

Donde se hace uso de un triple *cortocircuito*. Teniendo en cuenta que el programa `echo` siempre devuelve 0 (indicativo de que el programa ha finalizado con éxito, análogo a valor lógico `True`), el programa sólo se ejecuta si `$CMD` es igual a `true`, y el `exit` sólo se ejecutará si `$EXEC` finaliza la ejecución con fracaso, análogo a `False`.

Parametrización de las señales

La invocación de `parametriza.py` no tiene mayor complicación: simplemente hay que definir las opciones y argumentos del programa `parametriza.py`:

todo.sh

```
# Parametrización

dirSen="-s $DIR_SEN"
dirPrm="-p $DIR_PRM"

EXEC="parametriza.py $dirSen $dirPrm $GUI_ENT $GUI_DEV"
$PRM && echo $EXEC && $EXEC || exit 1
```

Creamos dos nuevas variables de entorno, `$dirSen` y `$dirPrm`, que combinan las opciones del programa con los valores correspondientes porque, mientras que los directorios `$DIR_SEN` y `$DIR_PRM` son constantes y comunes a múltiples programas, los nombres de las opciones empleadas en cada programa pueden y suelen ser diferentes. Por otro lado, al separar la definición de las opciones de la del propio comando, evitamos tener que editar ésta cada vez que se desea realizar algún cambio.

Entrenamiento de los modelos acústicos

El entrenamiento de los modelos queda como sigue:

todo.sh

```
# Entrenamiento

dirPrm="-p $DIR_PRM"
dirMar="-a $DIR_SEN"
dirMod="-m $DIR_MOD"

EXEC="entrena.py $dirPrm $dirMar $dirMod $GUI_ENT"
$ENT && echo $EXEC && $EXEC || exit 1
```

Reconocimiento de las señales de desarrollo

Análogamente, la parte de reconocimiento queda como sigue:

todo.sh

```
# Reconocimiento

dirRec="-r $DIR_REC"
dirPrm="-p $DIR_PRM"
dirMod="-m $DIR_MOD"
lisMod="-l $LIS_MOD"

EXEC="reconoce.py $dirRec $dirPrm $dirMod $lisMod $GUI_DEV"
$REC && echo $EXEC && $EXEC || exit 1
```

Evaluación del resultado

Finalmente, para la evaluación realizamos algo semejante, pero almacenando el resultado de la misma en el fichero `$FIC_RES`:

todo.sh

```
# Evaluación

dirRec="-r $DIR_REC"
dirMar="-a $DIR_SEN"

EXEC="evalua.py $dirRec $dirMar $GUI_DEV"
$REC && echo $EXEC && $EXEC | tee $FIC_RES || exit 1
```

3.3.5. Despedida y cierre

Es importante que el script dé información de cuándo acaba. En ejecuciones en la línea de comandos, esta información puede ser irrelevante, porque estamos presentes para saber cuándo lo hace; pero, en ejecuciones en segundo plano (*background*), si no hay un mensaje de despedida puede ser complicado saber si ya se ha acabado o no. Y hay que tener en cuenta que, como ya se ha mencionado otras veces, en reconocimiento del habla no es extraño que algún experimento tarde semanas, incluso meses, en completarse.

Así pues, las dos últimas órdenes del script escriben la fecha y hora de la finalización y un mensaje de despedida:

todo.sh

```
# Despedida y cierre

date
echo s\'acabao
```

3.3.6. Ejecución del script `todo.sh`

Al ejecutar el script, la información que obtenemos por pantalla es la siguiente:

```
usuario:~/TecParla$ todo.sh
ALBINO
/home/albino/TecParla/2020 Outono
Tue Dec  8 13:46:23 CET 2020
parametriza.py -s ./Sen -p ./Prm/Uno ./Gui/train.gui ./Gui/devel.gui
100%|████████████████████| 4000/4000 [00:02<00:00, 1557.72it/s]
entrena.py -p ./Prm/Uno -a ./Sen -m ./Mod/Uno ./Gui/train.gui
100%|████████████████████| 2000/2000 [00:00<00:00, 2667.37it/s]
reconoce.py -r ./Rec/Uno -p ./Prm/Uno -m ./Mod/Uno -l ./Lis/vocales.lis ./Gui/devel.gui
100%|████████████████████| 2000/2000 [00:02<00:00, 857.71it/s]
evalua.py -r ./Rec/Uno -a ./Sen ./Gui/devel.gui
100%|████████████████████| 2000/2000 [00:00<00:00, 11230.30it/s]
Exac = 25.00%

        a       e       i       o       u
a       117     68      55      91      69
e       63      99      65      74      99
i       48      90      95      40      127
o       72      103     63      79      83
u       57      78      105     50      110
Tue Dec  8 13:46:30 CET 2020
s'acabao
```

Donde el contenido del fichero `Log/todo.Uno.log` coincide plenamente con lo mostrado en pantalla (puede, por ejemplo, ejecutar `catLog/todo.Uno.log` para comprobarlo).

A su vez. el resultado del reconocimiento queda almacenado en el fichero `Res/Uno.res`:

```
usuario:~/TecParla$ cat Res/Uno.res
Exac = 25.00%

        a       e       i       o       u
a       117     68      55      91      69
e       63      99      65      74      99
i       48      90      95      40      127
o       72      103     63      79      83
u       57      78      105     50      110
```

Capítulo 4

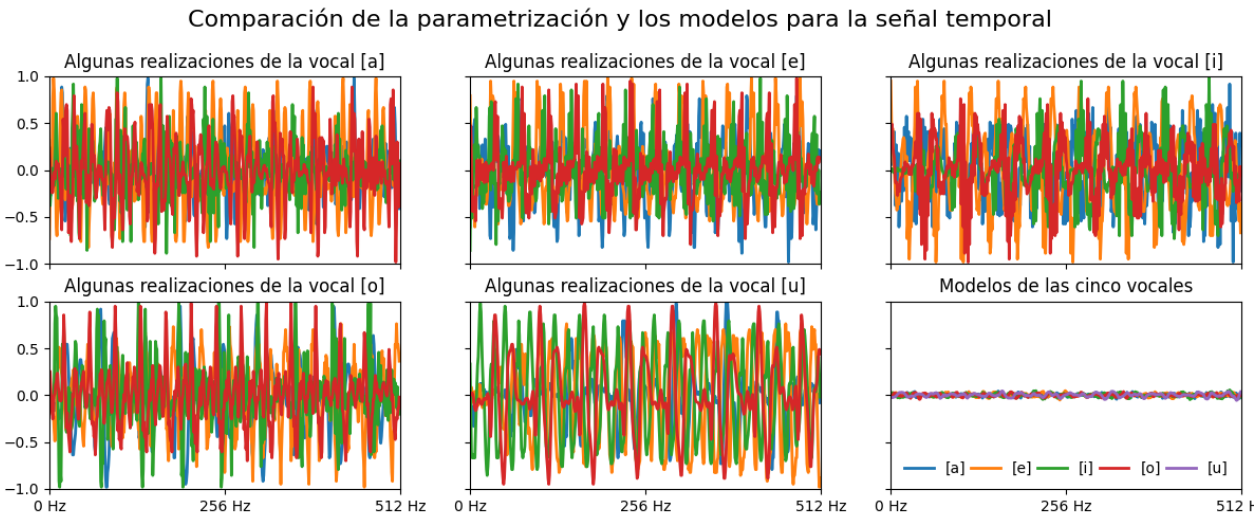
Extracción de características usando programación funcional

El resultado obtenido en el apartado 2.5 con el sistema trivial es un completo desastre. Con un vocabulario de cinco palabras, la máxima incertidumbre sería obtener una tasa de exactitud del 20 %. O sea que el sistema que tenemos es sólo un poco mejor que lanzar un dado de cinco caras y anotar el resultado.

Básicamente, sólo tenemos dos aspectos en los que el sistema puede ser mejorado: la extracción de características y el modelado acústico. En este apartado estudiaremos el primero de estos aspectos y, en el siguiente, nos centraremos en el modelado.

El principal objetivo de un esquema de parametrización para el reconocimiento del habla es capturar la información fonética presente en la señal. Evidentemente, la señal temporal presenta toda la información posible, pero, además de la fonética, también incluye otras informaciones que son irrelevantes para el reconocimiento del habla: el entorno de la grabación (ambiente ruidoso o no, espacio público, vehículo a motor, etc.), el sexo y edad del locutor, o su estado de ánimo o embriaguez, etc.

Toda esta información irrelevante para el propósito del sistema es ruido que enmascara la realmente útil. Podemos comprobar esta afirmación comparado los modelos generados con algunas señales de la base de datos:



Se observa que las distintas realizaciones de cada vocal no guardan ningún parecido con el resto. No se puede adivinar ninguna tendencia que nos permita decir que las cinco gráficas corresponden a vocales distintas. Peor aún, los modelos de las cinco vocales son casi idénticos e iguales a cero, lo cual no deja de ser lógico pues se trata del promedio de señales ergódicas de media cero.

Es fácil concluir que la señal temporal no es adecuada para plantear el sistema de reconocimiento a partir de ella. Al menos, usando como criterio de similitud fonética la distancia euclídea, pero lo mismo ocurre con otros criterios de similitud, como los probabilísticos, y no está claro que se pueda usar la señal temporal directamente en los basados en redes neuronales (aunque últimamente han aparecido arquitecturas que lo hacen con bastante éxito; por ejemplo, End-to-End Speech Recognition From the Raw Waveform). En general, se hace necesario obtener una representación alternativa, en la que señales fonéticamente semejantes se representen con parámetros igualmente semejantes.

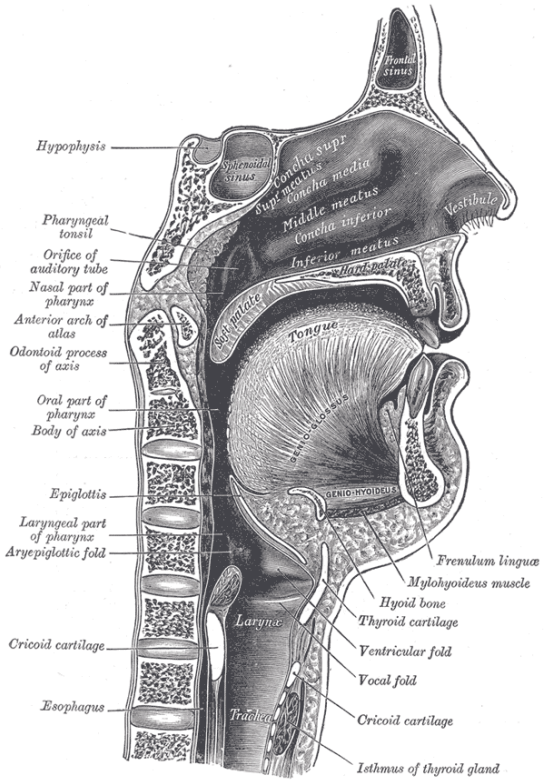
4.1. Información fonética en la señal de voz

La búsqueda de la información fonética en la señal de voz se simplifica enormemente si consideramos su fuente; es decir, el mecanismo usado cuando hablamos para producir fonemas distintos. Es el llamado *aparato fonador*.

4.1.1. El aparato fonador humano y su modelo

La señal de voz se produce al hacer pasar el aire expelido por los pulmones a través de los distintos conductos y cavidades que forman el *tracto vocal*: la tráquea, la laringe, la boca, etc., como muestra la clásica imagen publicada por Henry Gray en 1918 en su *Anatomy of the Human Body*[1]:

[1] Sí, efectivamente, esta obra es la *Gray's Anatomy* original (con *a*), un libro con más de 1200 imágenes anatómicas que ha sido, y sigue siendo, referencia obligatoria en estudios de medicina y similares.



En la respiración normal, no jadeante, el aire expelido atraviesa el tracto vocal sin encontrar apenas oposición a su paso. El resultado es un flujo de aire al exterior de la boca más o menos turbulento pero incapaz de producir sonido audible[2]. La condición indispensable para producir sonido es la existencia de alguna obstrucción o estrechamiento que incremente estas turbulencias lo suficiente para hacerse audible. Es la denominada *excitación*.

Hay dos tipos principales de excitación en función de si las cuerdas vocales están relajadas o no:

Sonidos sonoros: Los sonidos sonoros, como las vocales y buena parte de las consonantes, se producen cuando las cuerdas vocales, que son unas membranas ubicadas en la glotis, son tensionadas hasta impedir completamente el paso del aire desde la traquea al exterior. Mientras la presión del aire que sale de los pulmones no supere el umbral de resistencia de las cuerdas vocales, no se producirá ningún sonido. Pero, si la presión aumenta, llega un momento en el que se supera esta resistencia, provocando su apertura. Esto tiene dos efectos: por un lado, se permite el paso de aire al exterior; pero, por otro lado, este mismo paso de aire provoca que la presión a ambos lados de las cuerdas vocales se equilibre, cerrando de nuevo el paso del aire.

El resultado es un tren de pulsos de aire, más o menos periódico, con una frecuencia e intensidad que dependen de la presión ejercida en los pulmones y la tensión aplicada a las cuerdas vocales. Esta excitación, en un amplio margen de frecuencias, presenta un espectro aproximadamente plano, con

[2]En el fondo, la respiración siempre es audible. El lector puede comprobarlo si se ubica cerca de otra persona en un entorno suficientemente silencioso, como pueda ser un dormitorio durante la noche (y con la televisión o la radio apagadas). Aunque ese sonido natural de la respiración no puede considerarse voz.

lo que puede equipararse a un tren de deltas de Dirac. El mecanismo es semejante a cuando forzamos el paso del aire a través de la embocadura tensionada de un globo o a cuando hacemos una *pedorreta*.

Los diferentes sonidos que podemos producir dependen de la posición de los órganos que forman el tracto vocal. En este sentido, son especialmente importantes la boca, la lengua y el velo del paladar. Así por ejemplo, la mayor o menor apertura de la boca, junto con una posición más atrasada o adelantada de la lengua, permiten producir todo el inventario de vocales. Por otro lado, el hecho de que el velo del paladar permita o no el paso del aire a la cavidad nasal determina la naturaleza oral o nasal del sonido producido.

Sonidos sordos: Si las cuerdas sonoras se mantienen relajadas, el aire que las atraviesa sólo presenta unas ciertas turbulencias apenas audibles; de manera análoga a cuando dejamos salir el aire de un globo sin tensionar su boca. El resultado, si no hacemos nada más, es flujo de aire no ruidoso. Es necesario, por tanto, usar algún mecanismo adicional para producir sonido:

Oclusión: Si se introduce una oclusión completa en algún punto del tracto vocal (por ejemplo, los labios cerrados), el flujo de aire se interrumpe completamente, provocando un incremento de la presión al otro lado de la oclusión. Ahora bien, si en ese momento liberamos súbitamente la oclusión, el resultado es el paso de una pequeña cantidad de aire, hasta compensar la presión a ambos lados de la misma. Como la presión se compensa en un tiempo muy corto, el resultado es semejante a una delta de Dirac, aunque sus características espectrales dependerán de los órganos involucrados en la oclusión, y de los presentes en el resto del tracto vocal.

Así, si la oclusión se realiza entre los labios, el sonido producido es una `/p/`, mientras que, si se hace entre el parte trasera de la lengua y el velo del paladar, es una `/k/`.

Fricación: Los sonidos fricativos se producen de manera análoga a los oclusivos, pero sin cerrar completamente el paso del aire. De esta forma se consiguen unas turbulencias de gran amplitud que sí resultan audibles.

La señal acústica generada es esencialmente errática; por lo tanto, equiparable a ruido blanco, aunque la forma exacta de su espectro dependerá de los órganos involucrados en su generación, así como de los del resto del tracto vocal. De este modo, si colocamos la lengua y el velo del paladar de la misma forma que lo hacemos cuando producimos una `/k/`, pero permitiendo el paso de una pequeña cantidad de aire, el sonido producido es el de una `/j/` del castellano.

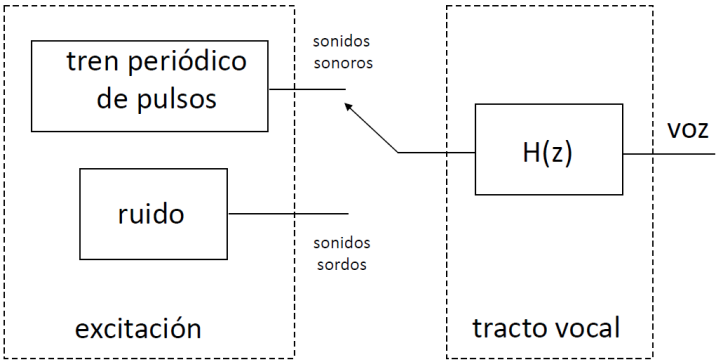
También existen sonidos que combinan los dos efectos. Son los llamados *africados*. Por ejemplo, el sonido `/ch/` de la palabra *chocolate*, se produce

encadenando una oclusiva /t/ con una fricativa /x/, lo cual justifica el uso en catalán del digrafo /tx/ para representarlo.

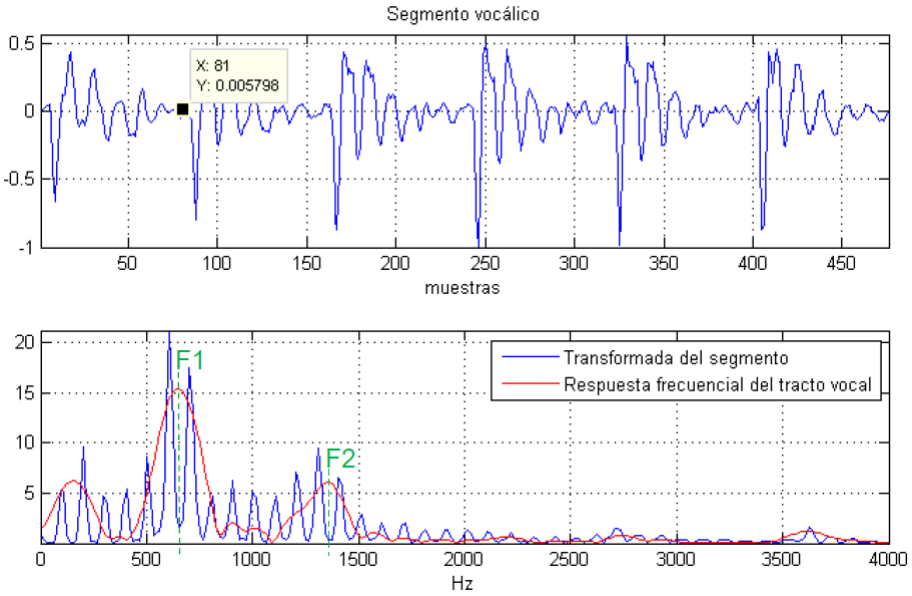
Consonantes sonoras: Cabe destacar que lo explicado para los sonidos sonoros se refiere, fundamentalmente, a las vocales (que, en la mayoría de los idiomas son exclusivamente sonoras). En el caso de las consonantes sonoras, lo anterior sigue siendo cierto, pero, además, se debe usar un mecanismo semejante a los indicados para los sonidos sordos en su generación. Así, por ejemplo, si provocamos una oclusión completa entre los labios mientras las cuerdas vocales están relajadas (sonido sordo), el sonido producido es una /p/; sin embargo, si están tensionadas (sonido sonoro), se produce una /b/.

Todos estos casos tienen una característica en común: pueden verse como una excitación de espectro plano (tren de deltas en los sonidos sonoros, delta de Dirac en los oclusivos y ruido blanco en los fricativos), cuya envolvente del espectro es modulada por la respuesta frecuencial asociada a la posición de los órganos que forman el tracto vocal. Esta respuesta frecuencial se caracteriza por la presencia de distintas frecuencias de resonancia, asociadas al conjunto de cavidades resonantes que forman el tracto vocal (cavidad bucal, faríngea, nasal, etc.). Son los denominados *formantes* que juegan un papel determinante en la diferenciación de las distintas vocales.

Todo ello nos permite representar la fonación con un sistema de procesado de señal denominado *modelo del aparato fonador humano*, en el que la información fonética está contenida, sobre todo, en la envolvente del espectro de la señal, determinada por la función de transferencia del tracto vocal, $H(z)$:



Por ejemplo, la figura siguiente muestra una realización de la vocal /a/, donde se puede apreciar que su espectro se corresponde aproximadamente con un tren de deltas de frecuencia fundamental $f_O \approx 100\,\text{Hz}$ y modulada por un filtro con frecuencias de resonancia en $F_1 \approx 700\,\text{Hz}$ y $F_2 \approx 1350\,\text{Hz}$ (además de otras, no indicadas en la figura, como las presentes a unos 2700 Hz y 3600 Hz):

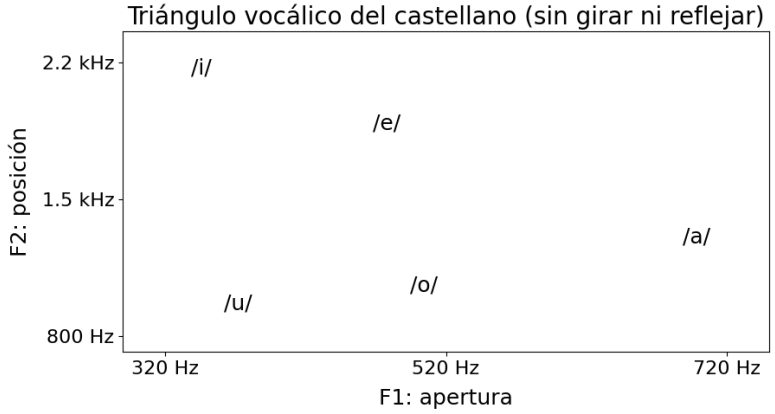


4.1.2. Información fonética de las vocales: el triángulo vocálico

En el caso concreto de las vocales, la información fonética se condensa en la posición de las resonancias del espectro o *formantes*. Así, es conocido que los dos primeros de estos formantes son suficientes para distinguir las distintas vocales (y muchos de los otros fonemas). Los otros formantes, hasta el `F6` o `F7`, proporcionan fundamentalmente naturalidad e identifican al locutor.

En las vocales, el primer formante (`F1`) está relacionado con la cercanía de la lengua al paladar, con lo que las *abiertas* (también conocidas como *bajas*, por la posición de la lengua), como la `/a/`, tienen un primer formante de frecuencia más baja que las *cerradas* (también conocidas como *altas*), como la `/i/` o `/u/`. Así mismo, el segundo formante (`F2`) está relacionado con la posición más adelantada o atrasada de la lengua, con lo que las *anteriores*, como la `/i/`, tienen un segundo formante de frecuencia más alta que las *posteriores*, como la `/u/`.

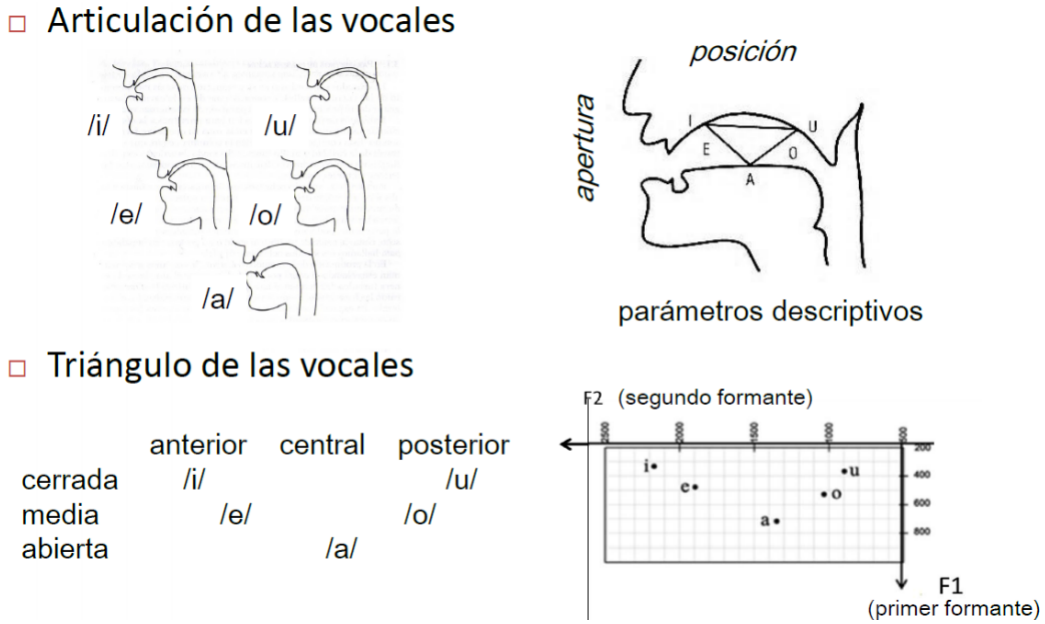
Es el llamado *triángulo vocálico*, ya que la representación gráfica de la posición de F1 y F2 para las cinco vocales se asemeja a un triángulo de vértices formados por las vocales `/a/`, `/i/` y `/u/`.



Que se corresponde con unas frecuencias centrales de los formantes aproximadamente iguales a:

	/a/	/e/	/i/	/o/	/u/
F1	700 Hz	480 Hz	350 Hz	500 Hz	370 Hz
F2	1400 Hz	1890 Hz	2170 Hz	1060 Hz	950 Hz

NOTA: A los físicos e ingenieros, la representación del triángulo vocálico en la página anterior nos parece lógica, con el primer formante en el eje de abscisas y el segundo en el de ordenadas, y con las frecuencias representadas en orden creciente. Sin embargo, entre los fonetistas produce una cierta desazón debido a que las vocales que ellos denominan *altas* figuran en la parte izquierda, mientras que las *bajas* lo hacen en la derecha; y, al mismo tiempo, las que se denominan *anteriores* figuran en la parte superior, mientras que las *posteriores* lo hacen en la inferior. A sus ojos, todo queda más *lógico* si se intercambian los ejes (F2 en abscisas y F1 en ordenadas) e invierten ambos ejes (sentido decreciente de frecuencias). Esa es la forma habitual en la que el lector encontrará esta información en los textos de fonética:



4.2. Extracción de características en `ramses` usando programación funcional

A la vista de las consideraciones fonéticas del apartado anterior, parece aconsejable abandonar la señal temporal en las tareas de reconocimiento y centrarse en la obtención de una representación de su espectro. Por desgracia, esta tarea resulta mucho más complicada de lo que parece. Es el denominado problema de la *estimación espectral*.

Podría parecer que la solución a este problema es tan sencilla como *"se hace la transformada de Fourier y ya está"*. De hecho, éste es el método más básico de hacerlo, denominado

periodograma. Por desgracia, como se verá en el apartado 4.3.1, con él sólo seremos capaces de obtener una aproximación esencialmente desviada del espectro auténtico (el problema del *sesgo* del periodograma) y profundamente ruidosa (la *varianza* del mismo).

En el caso concreto del reconocimiento del habla, además, existe el problema de que el periodograma también incorpora información del pitch de los sonidos sonoros, información que resulta irrelevante, y por tanto perniciosa, para este propósito. Así pues, es necesario obtener una buena representación de la envolvente del espectro que caracterice, con los menores sesgo y varianza posibles, la respuesta frecuencial del tracto vocal, $|H(z)|^2$.

A lo largo de este apartado se implementarán y pondrán a prueba distintos métodos de estimación espectral, pero existen muchos otros que han demostrado ser útiles en un mayor o menor grado. Para facilitar la implementación de todos estos métodos, con una modificación mínima del sistema de extracción de características, va a implementarse un sistema de parametrización basado en programación funcional, en el que el método concreto de estimación espectral se pase a la función y/o programa como un argumento más.

4.2.1. Adaptación de la función `parametriza()` a la programación funcional

La idea detrás de adaptar la extracción de características de `ramses` a la programación funcional consiste en transformar la identidad usada en la parametrización del sistema trivial a una en la cual se pase como argumento la función encargada de extraer los parámetros relevantes de la señal temporal.

Es decir, donde en el sistema trivial se usaba la asignación como método de parametrización:

ramses/parametriza.py
```
prm = np.array(sen)
```

Usar una función definida por el usuario, `funcPrm()`, que debe tomar como único argumento una `ndarray` de `numpy`, como señal temporal de entrada, y devolver otra `ndarray` con la señal parametrizada:

ramses/parametriza.py
```
prm = funcPrm(sen)
```

Esta función, `funcPrm()`, debe ser un argumento de `parametriza()`, teniendo la previsión de que, en caso de no indicarse, debe usarse la parametrización trivial; es decir, la copia de la señal temporal. Lo hacemos inicializando el argumento `funcPrm` a una función anónima que la implemente, `lambda x: x`:

ramses/parametriza.py
```
def parametriza(dirPrm, dirSen, *guiSen, funcSen=lambda x: x):
```

4.2.2. Adaptación del programa `parametriza.py` a la programación funcional

La adaptación del programa `parametriza.py` es algo más compleja que la de la función `parametriza()` debido a dos motivos:

- Los argumentos en línea de comandos son cadenas de texto. Mientras que convertir una cadena de texto a números, listas u otros tipos básicos es sencillo, hacerlo a una función no es tan inmediato.
- La función que usemos puede depender de módulos externos o definiciones no disponibles directamente. Por ejemplo, podríamos usar alguna función de la biblioteca `scipy`, pero no tenemos ninguna garantía de que `parametriza.py` la importe; y, bajo ningún concepto, queremos modificar el propio programa para que lo haga.

Uso de `eval()` para *evaluar* la función de parametrización

La función `eval()` devuelve el resultado de ejecutar su argumento, que debe ser una cadena de texto que contiene una sentencia Python válida. Hay dos opciones para que este resultado sea una función:

- La cadena de texto es, directamente, el nombre de una función definida en el momento de ejecutarse el comando.

  Por ejemplo, si sabemos que el script `parametriza.py` ha importado previamente `numpy`, la evaluación de la cadena `'numpy.abs'` da como resultado una réplica de la función `numpy.abs()`:

```
>>> import numpy
>>> funcPrm = eval('numpy.abs')
>>> funcPrm
<ufunc 'absolute'>

>>> funcPrm(-4)
4
```

- La cadena de texto es la definición de una función anónima usando la cláusula `lambda`:

```
>>> funcPrm = eval('lambda x: -x if x < 0 else x')
>>> funcPrm
<function __main__.<lambda>(x)>

>>> funcPrm(-4)
4
```

Ejecución de scripts previos; `execPre`

Sea cual sea el modo como se exprese la función de parametrización, tendremos error de ejecución si ésta depende de definiciones no disponibles previamente en el script `parametriza.py`:

```
>>> funcPrm = eval('lambda x: np.abs(fft(x))')
>>> funcPrm([5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <lambda>
NameError: name 'np' is not defined
```

La solución es permitir a `parametriza.py` el paso de uno o más scripts que se encarguen de asegurar que todas las definiciones necesarias están disponibles.

Por ejemplo, en el caso anterior, la función se generaría correctamente si, previamente, se hubieran ejecutado las órdenes siguientes, que guardamos en el fichero `script.py`:

script.py

```
import numpy as np
from numpy.fft import fft
```

Sección `Opciones` del mensaje de sinopsis

Para incorporar la orden a evaluar para obtener la función de parametrización y el nombre de los scripts a ejecutar previamente, completamos la sección `'Opciones'` de la sinopsis del programa con las dos líneas siguientes:

ramses/parametriza.py

```
-x SCRIPT..., --execPre=SCRIPT...  Scripts Python a ejecutar antes de la parametrización
-f EXPR, --funcPrm=EXPR  Expresión a evaluar para obtener la función de parametrización
```

Sección `Parametrización` del mensaje de sinopsis

En esta fase del programa, ya no tiene sentido explicar la parametrización trivial en una sección específica. Sin embargo, sí es necesario explicar al usuario los mecanismos empleados para generar la función de parametrización. Lo hacemos sustituyendo la sección `'Parametrización trivial'` por la sección `'Parametrización'` siguiente:

ramses/parametriza.py

```
Parametrización:
    La función empleada para parametrizar las señales será el resultado de ejecutar el
    argumento de la opción '--funcPrm' por el comando 'eval()'. Debe, por tanto, constar
    de una expresión Python, que puede ser el nombre de una función ya existente o una
    función anónima definida in situ con el comando 'lambda'.
```

```
    La función de parametrización debe tomar un único argumento del tipo 'ndarray' de
    'numpy' con la señal temporal, y devolver otra 'ndarray' con el resultado.

    Para posibilitar el uso de funciones definidas por el usuario, o que echen mano de
    módulos que deban ser importados con anterioridad a la definición de la función,  ...
     ↪  puede
    indicarse la ejecución de uno o más scripts Python con la opción '--execPre'. Si se
    indica más de un script, éstos deben separarse mediante comas.
```

Adaptación de la sección '__main__' del script

La nueva opción `'--execPre'` tiene como argumento una lista con los nombres, separados con coma, de uno o más scripts que deben ejecutarse antes de invocar a la función. Separamos el nombre de cada uno de ellos con el método `split()` de la clase `str` y lo ejecutamos con la notación habitual `exec(open(script).read())`:

ramses/parametriza.py

```
scripts = args['--execPre']
if scripts:
    for script in scripts.split(','):
        exec(open(script).read())
```

Por su parte, el argumento de la opción `'--funcPrm'` proporciona un comando Python que, evaluado con la función `eval()`, devuelve la función que realiza la parametrización de las señales. En caso de no usarse esta opción, la función de parametrización debe ser la trivial; esto es, la copia de la señal temporal:

ramses/parametriza.py

```
funcPrm = eval(args['--funcPrm'] if args['--funcPrm'] else 'lambda x: x')
```

4.2.3. Incorporación de la programación funcional al script `todo.sh`

La incorporación de las dos nuevas opciones de `parametriza.py` al script `todo.sh` debe realizarse manteniendo los requisitos expresados en el apartado 3.3; en concreto, los que se refieren a que debemos obtener información completa de los comandos ejecutados, permitiendo su reproducción de un modo sencillo. Para conseguir este propósito, tanto la definición de la función de parametrización, `funcPrm`, como el script previo, `execPre`, se construirán desde el propio script.

Aunque en el caso de la parametrización trivial no es necesario ejecutar ningún script previo, sí se creará uno que servirá de base para experimentos posteriores. Su

Usaremos el nombre de la parametrización para nombrar tanto la función de parametrización, como el script previo, en el cual se definirá aquélla. El script previo se mostrará en

pantalla, con lo cual también aparecerá en el fichero de registro de la ejecución, y se almacenará en el mismo directorio que las señales parametrizadas, de manera que siempre esté disponible para reproducir el experimento o para conocer los detalles de la parametrización:

todo.sh

```
# Definición de la función de parametrización

FUNC_PRM=trivial
EXEC_PRE=$DIR_PRM/$FUNC_PRM.py
[ -d $(dirname $EXEC_PRE) ] || mkdir -p $(dirname $EXEC_PRE)

echo "############################################################" | tee $EXEC_PRE
echo "#  Parametrización trivial                                 #" | tee -a $EXEC_PRE
echo "############################################################" | tee -a $EXEC_PRE
echo ""                                                            | tee -a $EXEC_PRE
echo "def $FUNC_PRM(x):"                                           | tee -a $EXEC_PRE
echo "    return x"                                              | tee -a $EXEC_PRE

# Parametrización

dirSen="-s $DIR_SEN"
dirPrm="-p $DIR_PRM"
execPre="-x $EXEC_PRE"
funcPrm="-f $FUNC_PRM"

EXEC="parametriza.py $dirSen $dirPrm $execPre $funcPrm $GUI_ENT $GUI_DEV"
$PRM && { echo $EXEC; $EXEC || exit 1; }
```

Al ejecutar el script, el resultado, como no podía ser de otro modo, es idéntico al obtenido hasta ahora con el sistema trivial:

```
usuario:~/TecParla$ todo.sh
ALBINO
/home/albino/TecParla/2021 Outono
Fri Dec 10 19:34:38 CET 2021
############################################################
#  Parametrización trivial                                 #
############################################################

def trivial(x):
    return x[:]
parametriza.py -s ./Sen -p ./Prm/Dos -x ./Prm/Dos/trivial.py -f trivial                ...
↪  ./Gui/train.gui ./Gui/devel.gui
100%|████████████████████████████████████████| 4000/4000 [00:01<00:00, 2302.54it/s]
entrena.py -p ./Prm/Dos -a ./Sen -m ./Mod/Dos ./Gui/train.gui
100%|████████████████████████████████████████| 2000/2000 [00:00<00:00, 2375.87it/s]
reconoce.py -r ./Rec/Dos -p ./Prm/Dos -m ./Mod/Dos -l ./Lis/vocales.lis ./Gui/devel.gui
100%|████████████████████████████████████████| 2000/2000 [00:01<00:00, 1070.78it/s]
evalua.py -r ./Rec/Dos -a ./Sen ./Gui/devel.gui
100%|████████████████████████████████████████| 2000/2000 [00:00<00:00, 10498.78it/s]
        a       e       i       o       u
a       117     68      55      91      69
e       63      99      65      74      99
i       48      90      95      40      127
o       72      103     63      79      83
```

```
u       57      78      105     50      110

Exac = 25.00%
Fri Dec 10 19:34:43 CET 2021
s'acabao
```

Podemos comprobar que el fichero `Prm/Dos/trivial.py` guarda toda la información acerca de la parametrización empleada:

```
usuario:~/TecParla$ cat Prm/Dos/trivial.py
#############################################################
#  Parametrización trivial                                  #
#############################################################

def trivial(x):
    return x[:]
```

4.3. Uso como parámetro de la transformada de Fourier y el periodograma

A la vista de que la información fonética radica, fundamentalmente, en la distribución frecuencial de la energía, la primera alternativa que se nos puede ocurrir es usar la representación frecuencial de la señal en lugar de la temporal.

Modificamos el script `todo.sh` para ello. En esencia, lo único necesario es crear un nuevo script previo que incorpore la biblioteca `numpy` e invoque a la función `fft()` del módulo `numpy.fft`:

todo.sh

```
FUNC_PRM=fft
EXEC_PRE=$DIR_PRM/$FUNC_PRM.py
[ -d $(dirname $EXEC_PRE) ] || mkdir -p $(dirname $EXEC_PRE)

echo "#############################################################" | tee $EXEC_PRE
echo "#  Transformada discreta de Fourier                         #" | tee -a $EXEC_PRE
echo "#############################################################" | tee -a $EXEC_PRE
echo                                                                 | tee -a $EXEC_PRE
echo "import numpy as np"                                           | tee -a $EXEC_PRE
echo                                                                 | tee -a $EXEC_PRE
echo "def $FUNC_PRM(x):"                                            | tee -a $EXEC_PRE
echo "    return np.fft.fft(x)"                                     | tee -a $EXEC_PRE
```

Al ejecutarlo, el resultado resulta sorprendente; se obtiene el mismo resultado que con la parametrización trivial:

```
usuario:~/TecParla$ todo.sh
ALBINO
/home/albino/TecParla/2021 Outono
Fri Dec 10 19:47:05 CET 2021
#########################################################
#  Transformada discreta de Fourier                     #
#########################################################

import numpy as np

def fft(x):
    return np.fft.fft(x)
parametriza.py -s ./Sen -p ./Prm/Dos -x ./Prm/Dos/fft.py -f fft ./Gui/train.gui        ...
 ↪  ./Gui/devel.gui
100%|████████████████████████████████████████| 4000/4000 [00:01<00:00, 2105.98it/s]
entrena.py -p ./Prm/Dos -a ./Sen -m ./Mod/Dos ./Gui/train.gui
100%|████████████████████████████████████████| 2000/2000 [00:00<00:00, 2224.25it/s]
reconoce.py -r ./Rec/Dos -p ./Prm/Dos -m ./Mod/Dos -l ./Lis/vocales.lis ./Gui/devel.gui
100%|████████████████████████████████████████| 2000/2000 [00:02<00:00, 950.07it/s]
evalua.py -r ./Rec/Dos -a ./Sen ./Gui/devel.gui
100%|████████████████████████████████████████| 2000/2000 [00:00<00:00, 10229.44it/s]
        a       e       i       o       u
a       117     68      55      91      69
e       63      99      65      74      99
i       48      90      95      40      127
o       72      103     63      79      83
u       57      78      105     50      110

Exac = 25.00%
Fri Dec 10 19:47:11 CET 2021
s'acabao
```

Por sorprendente que pueda resultar el hecho de obtener el mismo resultado que usando la parametrización trivial, no se trata de ningún error. De hecho, es un resultado completamente lógico ya que ambas representaciones, la temporal y la frecuencial, aportan **exactamente** la misma información. Así, el teorema de Parseval nos indica que la energía calculada en el dominio transformado de Fourier es la misma que la calculada en el dominio temporal (salvo un posible factor de escala, dependiendo de cómo se defina la transformada de Fourier):

$$\sum_n |s[n]|^2 \overset{\text{Teorema de Parseval}}{=\!=\!=} \sum_k |S[k]|^2 \tag{4.1}$$

El cuadrado de la distancia euclídea entre la señal $x[n]$ y la referencia $r[n]$ puede verse como la energía de la diferencia entre ambas, $d^2_{xr} = \sum_n |x[n] - r[n]|^2$. Por tanto, es indiferente calcular esta distancia en el dominio temporal o en el frecuencial:

$$d^2_{xr} = \sum_n |x[n] - r[n]|^2 = \sum_k |X[k] - R[k]|^2 = d^2_{XR} \tag{4.2}$$

4.3.1. Uso del periodograma como parámetro

El fracaso del uso de la transformada de Fourier puede verse como la consecuencia de que ambas representaciones aportan exactamente la misma información. Como se ha dicho con anterioridad, el objetivo de la parametrización es, justamente, eliminar toda información superflua o redundante para la tarea de reconocimiento.

La primera información que podemos eliminar fácilmente de la señal es la de fase. La fase de una señal depende de la relación temporal entre sus componentes frecuenciales. No aporta, por tanto, información de la amplitud de estos componentes, que es lo que refleja la posición de los formantes. Más aún, si se tiene en cuenta la fase de las señales, la distancia entre ellas dependerá de su alineado temporal, el cual tampoco aporta ninguna información acerca de su parecido fonético.

Una alternativa a la transformada de Fourier en la tarea de reconocimiento del habla es el periodograma, definido como el módulo al cuadrado de la transformada discreta de Fourier. El periodograma es el método más sencillo para la estimación de la *densidad espectral*, que es, justamente, la función que nos indica cómo se distribuye en frecuencia la energía de la señal.

Para usar el periodograma como parametrización en nuestro sistema de reconocimiento, reescribimos el script previo de manera que la función ejecutada devuelva el módulo al cuadrado de la transformada de Fourier:

todo.sh

```
FUNC_PRM=periodograma
EXEC_PRE=$DIR_PRM/$FUNC_PRM.py
[ -d $(dirname $EXEC_PRE) ] || mkdir -p $(dirname $EXEC_PRE)

echo "############################################################" | tee $EXEC_PRE
echo "#  Periodograma                                            #" | tee -a $EXEC_PRE
echo "############################################################" | tee -a $EXEC_PRE
echo                                                               | tee -a $EXEC_PRE
echo "import numpy as np"                                          | tee -a $EXEC_PRE
echo                                                               | tee -a $EXEC_PRE
echo "def $FUNC_PRM(x):"                                           | tee -a $EXEC_PRE
echo "    return np.abs(np.fft.fft(x)) ** 2"                       | tee -a $EXEC_PRE
```

Ahora el resultado ya es mucho mejor (aunque tampoco hay que volverse loco con él...):

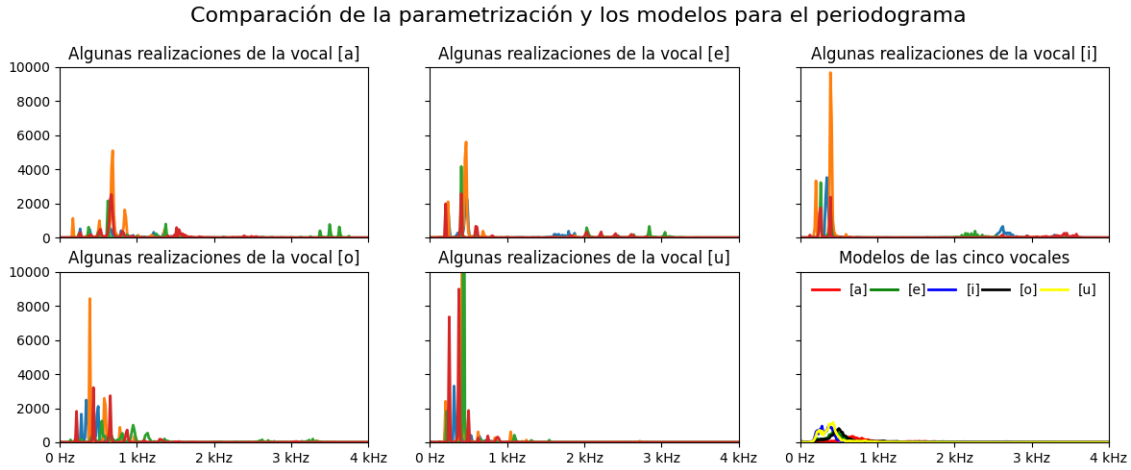
```
usuario:~/TecParla$ todo.sh
ALBINO
/home/albino/TecParla/2021 Outono
Fri Dec 10 19:57:30 CET 2021
############################################################
#  Periodograma                                            #
############################################################

import numpy as np

def periodograma(x):
    return np.abs(np.fft.fft(x)) ** 2
```

```
parametriza.py -s ./Sen -p ./Prm/Dos -x ./Prm/Dos/periodograma.py -f periodograma      ...
 ↪  ./Gui/train.gui ./Gui/devel.gui
100%|████████████████████████████████████████| 4000/4000 [00:02<00:00, 1974.19it/s]
entrena.py -p ./Prm/Dos -a ./Sen -m ./Mod/Dos ./Gui/train.gui
100%|████████████████████████████████████████| 2000/2000 [00:00<00:00, 2394.03it/s]
reconoce.py -r ./Rec/Dos -p ./Prm/Dos -m ./Mod/Dos -l ./Lis/vocales.lis ./Gui/devel.gui
100%|████████████████████████████████████████| 2000/2000 [00:01<00:00, 1095.17it/s]
evalua.py -r ./Rec/Dos -a ./Sen ./Gui/devel.gui
100%|████████████████████████████████████████| 2000/2000 [00:00<00:00, 10492.28it/s]
        a       e       i       o       u
a       380     0       3       17      0
e       107     158     13      80      42
i       68      37      213     9       73
o       120     95      8       158     19
u       49      74      142     28      107

Exac = 50.80%
Fri Dec 10 19:57:36 CET 2021
s'acabao
```

4.3.2. Análisis de los resultados usando el periodograma

El resultado no es ninguna maravilla (aún queda mucha información superflua por eliminar), pero, por lo menos, ya es bastante mejor que lanzar un dado de cinco caras...

Si realizamos una gráfica como la dibujada en la página 48, el resultado, lejos de ser ideal, sí muestra una cierta estructura en las señales parametrizadas, que se refleja en los modelos de cada vocal.

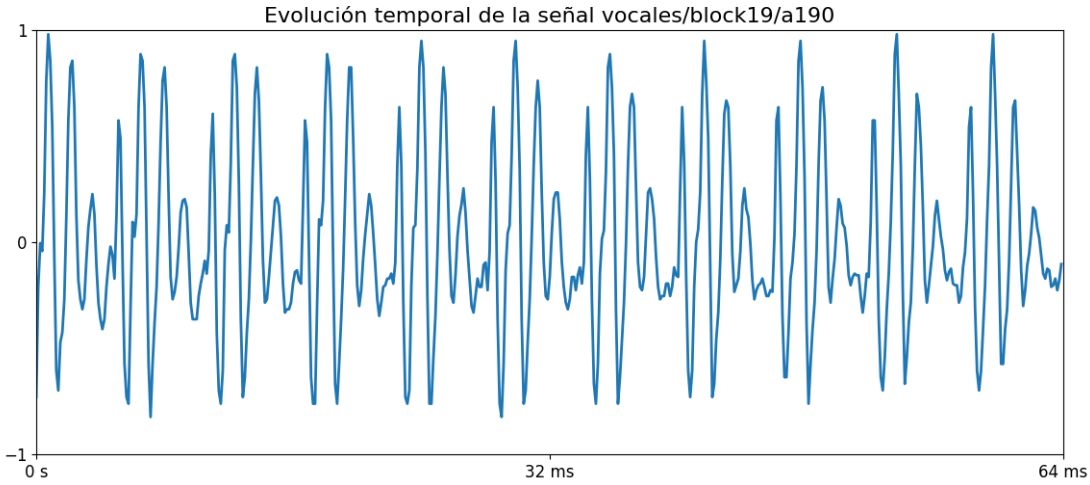


La estructura apreciada en las señales parametrizadas se corresponde, aproximadamente, con la expuesta en el triángulo vocálico del apartado 4.1.2. Por ejemplo, según se indica en ese apartado, el primer formante de frecuencia más alta se corresponde con el de la vocal /a/ (unos 700 Hz), que tiene un segundo formante en medio de los de las otras vocales (unos 1310 Hz). Sin embargo, es difícil observar esa estructura en los modelos, y éstos siguen siendo sensiblemente diferentes de las señales que representan.

Análisis del periodograma de una realización concreta de la vocal /a/

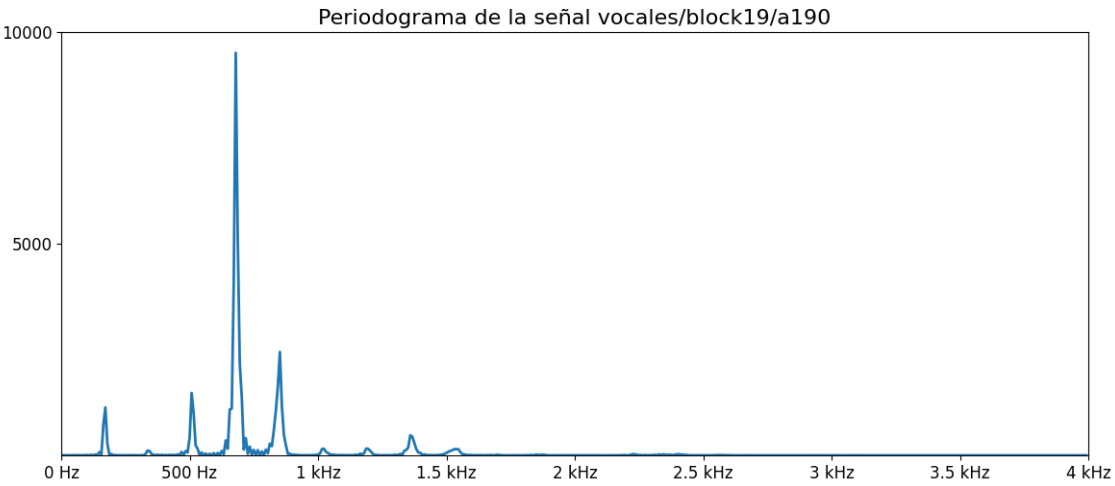
El uso del periodograma como parametrización de la señal, ya es algo mejor que lanzar un dado de cinco caras, pero sigue obteniendo un resultado bastante lamentable: en torno a un 50 % de exactitud. Podemos comprender un poco mejor las carencias de este método de estimación espectral fijándonos en un caso concreto.

La gráfica siguiente muestra la señal temporal `a190.wav`, correspondiente a una realización de la vocal /a/ de la base de datos `vocales`:



Como corresponde a una vocal, en la gráfica podemos comprobar el carácter aproximadamente periódico de la señal. Así, la misma forma se repite algo menos de 11 veces en los 64 ms de duración del segmento, con lo que el periodo viene a ser unos 6 ms. También se observa que la señal no es completamente periódica; esto es, los distintos periodos son muy parecidos, pero no idénticos. Esto es debido a que la voz es un proceso estocástico, de naturaleza en gran medida errática.

Si visualizamos el peridograma de esta misma señal, se observa como el espectro está formado fundamentalmente por contribuciones discretas, a frecuencias múltiplo entero del inverso del periodo; esto es, unos 170 Hz:



Si la señal fuera perfectamente periódica, su espectro debería estar formado por deltas de Dirac de área proporcional a la respuesta frecuencial del tracto vocal y ubicadas en las frecuencias de los armónicos. Sin embargo, la señal no es perfectamente periódica. Como ya se ha comentado antes, tiene una componente errática que hace que los distintos periodos sean ligeramente distintos. Pero, además, aunque todos fueran idénticos, seguiría sin ser periódica porque es de duración limitada, y una señal sólo puede ser completamente periódica si su duración es infinita. Las señal de la gráfica está *enventanada*; en este caso, con una ventana rectangular. Así pues, aún si todos los periodos del segmento fueran idénticos, lo que se obtendría sería la transformada de Fourier de la ventana (la función sinc(·)) centrada en la posición de los armónicos.

Al observar esta señal en concreto, se pueden destacar una serie de cuestiones:

- El segundo armónico es casi inexistente. Son cosas que pasan... Probablemente sea un rasgo característico del locutor. En cualquier caso, no afecta ni a la periodicidad de la señal, ni a la estructura de formantes.
- El acusado pico en torno a 700 Hz se corresponde con el cuarto armónico de la señal, realzado por la presencia de un primer formante de gran amplitud a esa frecuencia. Lo cual concuerda con el hecho de tratarse de una vocal /a/ (véase la gráfica de la sección 4.1.2).
- Es difícil de apreciar, pero en torno a unos 1350 Hz aparece también un pequeño pico de intensidad que, probablemente, refleje la presencia de un segundo formante a esa frecuencia. Esto también es coherente con la posición teórica del segundo armónico en las vocales /a/.

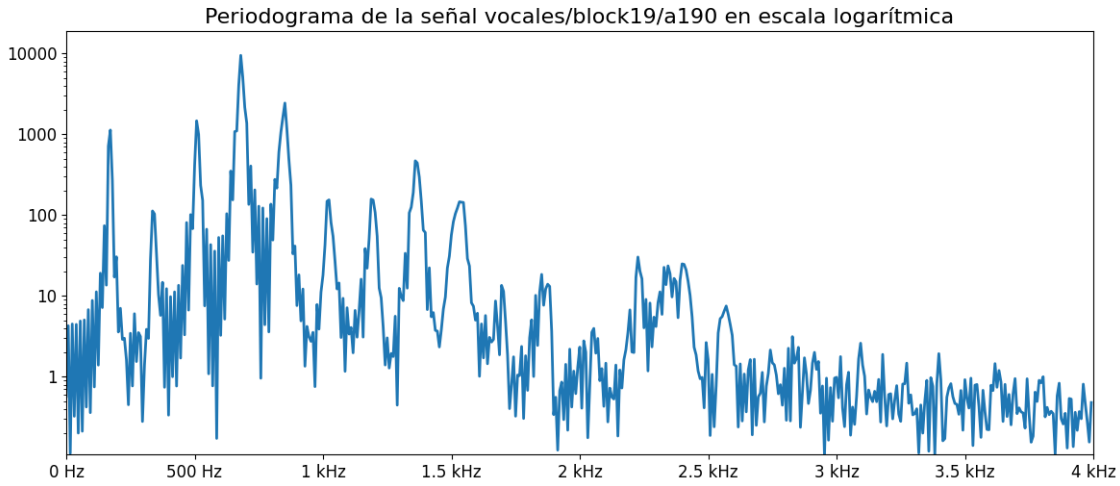
4.3.3. Uso del logaritmo del periodograma (decibelios)

Las limitaciones de esta representación de la señal para la diferenciación de las distintas vocales son evidentes. Como ya se vio en la figura de la sección 4.3.2 (de hecho, la señal analizada es una de las que se muestran en aquella gráfica), aunque es posible asociar la posición de los formantes de las distintas vocales con su posición teórica en el triangulo vocálico, podemos dejarnos la vista haciéndolo. Y es complicado encontrar semejanza matemática donde no la hay visual...

La principal limitación es debida a que la intensidad del segundo formante es muy inferior a la del primero, lo que hace que apenas sea perceptible visualmente. Sin embargo, según se ve en el triángulo vocálico, ambos formantes son igual de determinantes a la hora de distinguir una vocal de otra.

El problema de señales con componentes que aportan la misma información, pero tienen magnitudes muy diferentes es bastante habitual en el procesado de señal; especialmente en los campos de las comunicaciones y la acústica. En el caso de la acústica, además, se da el hecho de que el oído no presenta una respuesta lineal en amplitud y que el margen dinámico audible es extremadamente grande.

En estas situaciones, una solución ampliamente empleada es la representación logarítmica. Por ejemplo, en el caso de la señal `a190.wav` usada como ejemplo en esta sección, el peridograma representado en escala logarítmica es el siguiente:



Puede observarse como los detalles del espectro de la señal son mucho más visibles con la representación logarítmica que con la lineal usada en la gráfica en la página 64. Por ejemplo, ahora los dos primeros formantes, en torno a $F_1 \approx 700\,\text{Hz}$ y $F_2 \approx 1350\,\text{Hz}$ son claramente visibles, y de amplitud comparable. Es más, ahora también se manifiesta con claridad un tercer formante alrededor de $F_3 \approx 2300\,\text{Hz}$.

Incorporación del periodograma logarítmico a `Ramses`

En el sistema de reconocimiento del habla, esto se reduce a sustituir la expresión usada para calcular el peridograma por la siguiente:

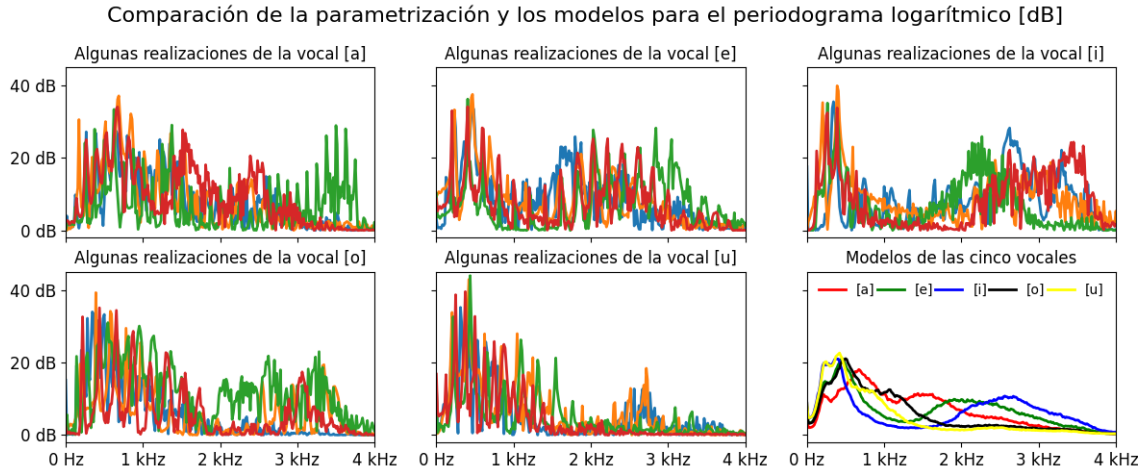
todo.sh

```
echo "############################################################"        | tee $EXEC_PRE
echo "#  Periodograma logarítmico                                 #"        | tee -a      …
 ↪  $EXEC_PRE
echo "############################################################"        | tee -a      …
 ↪  $EXEC_PRE
echo "from numpy.fft import fft"                                          | tee -a      …
 ↪  $EXEC_PRE
echo "import numpy as np"                                                 | tee -a      …
 ↪  $EXEC_PRE
echo "eps = 1"                                                            | tee -a      …
 ↪  $EXEC_PRE
echo "def $FUNC_PRM(x): return 10 * np.log10(eps + abs(fft(x)) ** 2)"     | tee -a      …
 ↪  $EXEC_PRE
```

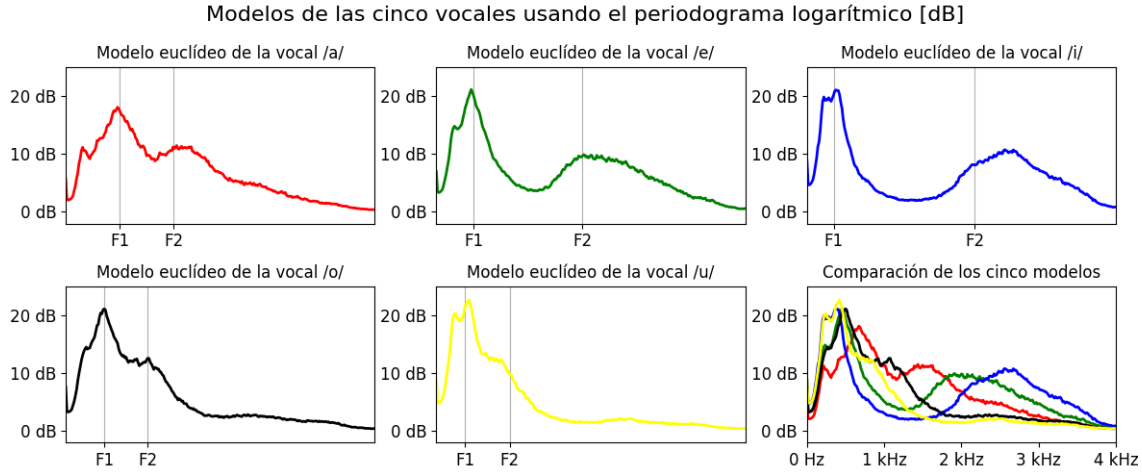
La variable `eps` es necesaria para evitar el cálculo de logaritmos de cantidades excesivamente pequeñas, incluso cero, cuyo logaritmo tiende a menos infinito.

Al ejecutar `todo.sh` con esta parametrización, el resultado obtenido es una exactitud del 82.25 %, muy superior al uso directo del periodograma, que era del 50.80 %.

La gráfica siguiente es idéntica a la mostrada en la página 63, pero usando escala logarítmica en lugar de la lineal, y ayuda a comprender la mejora obtenida en el reconocimiento:



La gráfica siguiente muestra más detalladamente los cinco modelos, indicándose la posición de los valores típicos de los dos primeros formantes según la tabla en la página 53.



Puede apreciarse la buena concordancia entre los valores de referencia para los dos formantes y la posición de los picos en las gráficas. La mayor discrepancia, para la vocal `/i/`, es debida a la presencia de un tercer formante muy próximo al segundo.

Sensibilidad al umbral del logaritmo, `eps`

Un problema con el uso de los logaritmos es que el resultado es bastante sensible al valor del umbral `eps`. De hecho, para `eps=0`, el periodograma presenta valores iguales a cero que provocan que, al calcular el logaritmo, se produzcan errores de ejecución. La tabla siguiente muestra el resultado obtenido para distintos valores del parámetro:

Dependencia de la exactitud con el umbral `eps`

eps	**0.00001**	**0.1**	**1**	**10**	**100**	**1000**
Exac	72.50 %	76.80 %	82.25 %	84.70 %	76.55 %	61.45 %

4.3.4. El cepstrum

Directamente relacionado con el uso del logaritmo del periodograma, o de cualquier otra estimación del espectro, está la función cepstrum (pronunciado habitualmente como *kepstrum*).

El cepstrum de una señal es, básicamente, la transformada inversa de Fourier del logaritmo del espectro de la señal. Existen distintas implementaciones de esta idea. En procesado de voz, la más habitual es la denominada *cepstrum real*, definida para señales discretas por la siguiente ecuación:

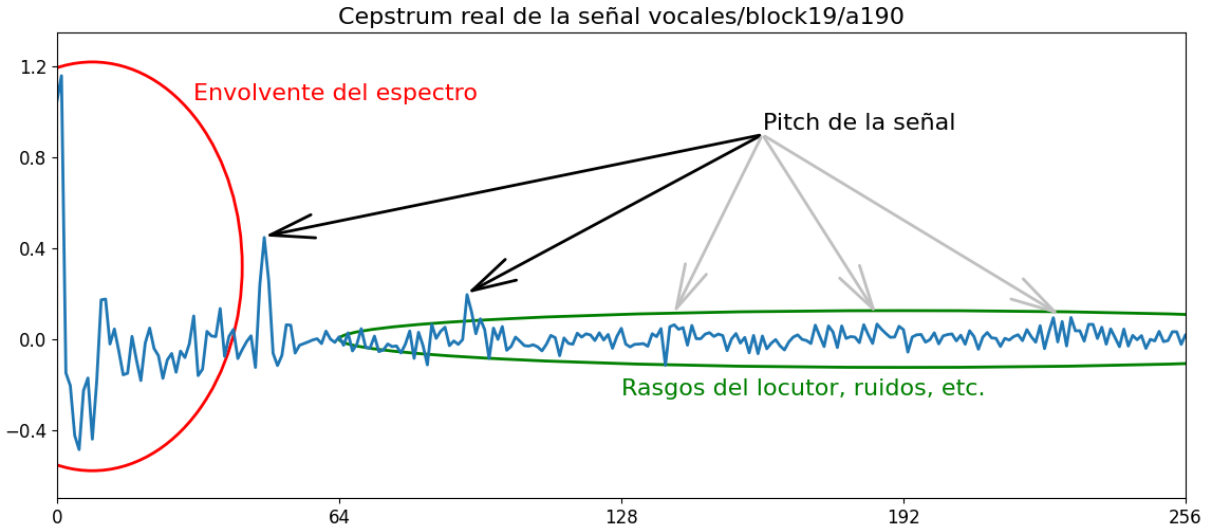
$$c[k] = \mathrm{DFT}^{-1}\left\{\log\left|\mathrm{DFT}\left\{x[n]\right\}\right|^2\right\} \tag{4.3}$$

El cepstrum fue propuesto a mediados del siglo XX por Bogert, Healy y Tukey en un artículo que enseguida se convirtió en un clásico del procesado de señal. Maravillados por las muchas aplicaciones de su invento, desarrollaron toda una teoría alrededor del mismo, en el que cada elemento del procesado clásico (*spectrum, frequency, filter, analysis, phase...*) tenía su propia correspondencia alternando el orden de las sílabas y/o las consonantes (*cepstrum, quefrency, lifter, alanysis, saphe...*).

Las aplicaciones en las que el *alanisis cepstral* resulta de mayor utilidad son aquéllas en las que aparecen señales periódicas, o en las que la señal está contaminada por ecos, reverberación o ruido convolutivo. Un ejemplo temprano de su uso fue la restauración de grabaciones musicales de principios del siglo XX, como las realizadas por Enrico Caruso, restauradas por Stockham y Cannon en 1975.

En la tarea del reconocimiento del habla, la principal utilidad del cepstrum es su capacidad de separar la envolvente del espectro, que es donde se encuentra la información fonética, de su detalle fino, que es donde encontramos la información del pitch y del locutor. Puede demostrarse que la envolvente del espectro queda representada en los coeficientes cepstrales de orden más bajo, mientras que la periodicidad y los rasgos del locutor se concentran en lo de orden elevado.

Por ejemplo, en la señal del ejemplo del apartado 4.3.2, el cepstrum sería el siguiente:



Por tanto, si queremos obtener la envolvente de la señal, eliminando la información del pitch o del locutor, así como buena parte del ruido, todo lo que tenemos que hacer es usar sólo los coeficientes de orden más bajo del cepstrum.

Mención aparte merece el coeficiente de orden cero. La DFT^{-1} particularizada en $m = 0$ es, simplemente, la media del logaritmo del periodograma de la señal:

$$c[0] = \mathrm{DFT}^{-1}\left\{\log\left|\mathrm{DFT}\left\{x[n]\right\}\right|^2\right\}\Big|_{m=0} = \frac{1}{N}\sum_k \log\left|\mathrm{DFT}\left\{x[n]\right\}\right|^2 e^{j\frac{2\pi}{N}k0} = \frac{1}{N}\sum_k \log\left|\mathrm{DFT}\left\{x[n]\right\}\right| \tag{4.4}$$

Es, por tanto, un coeficiente un tanto *raro*. Aporta más información acerca de la potencia de la señal que de su forma; aunque, al ser la media de los logaritmos, también refleja si el espectro de la señal tiene valles y picos muy acusados o no. En general, suele ser conveniente evitar el uso de este coeficiente, aunque hay aplicaciones en las que sí ha demostrado ser útil.

Uso del cepstrum en el sistema de reconocimiento

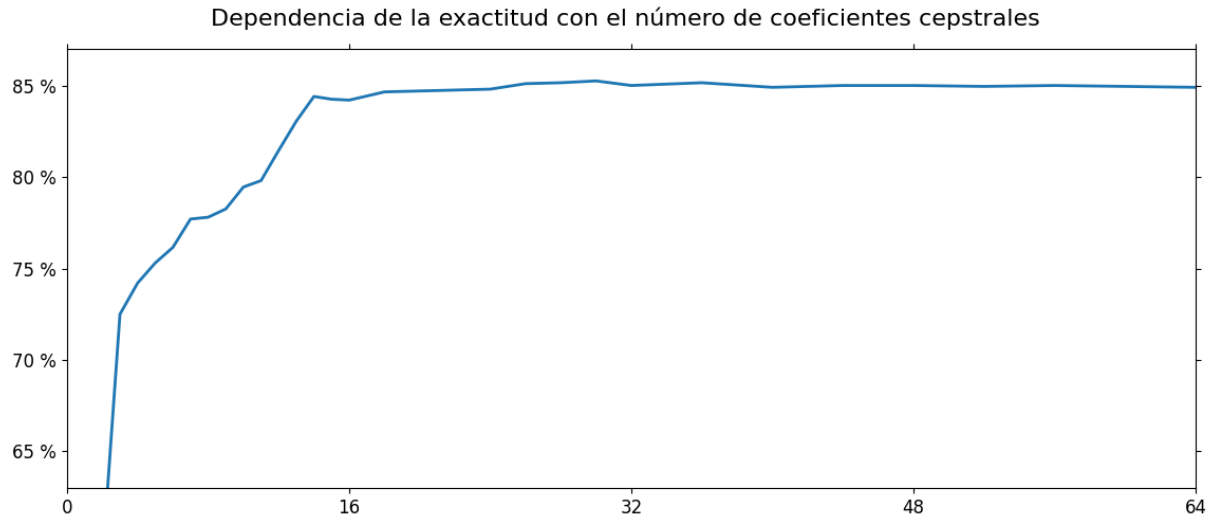
Puede incluirse en Ramses el cepstrum de varios modos. Lo hacemos en la función invocada desde `todo.sh`, teniendo en cuenta que el número de coeficientes que usemos, `numCof`, será un parámetro determinante de las prestaciones del sistema.

Añadimos a `todo.sh` la definición de la función de parametrización y del script previo para usar el cepstrum:

todo.sh

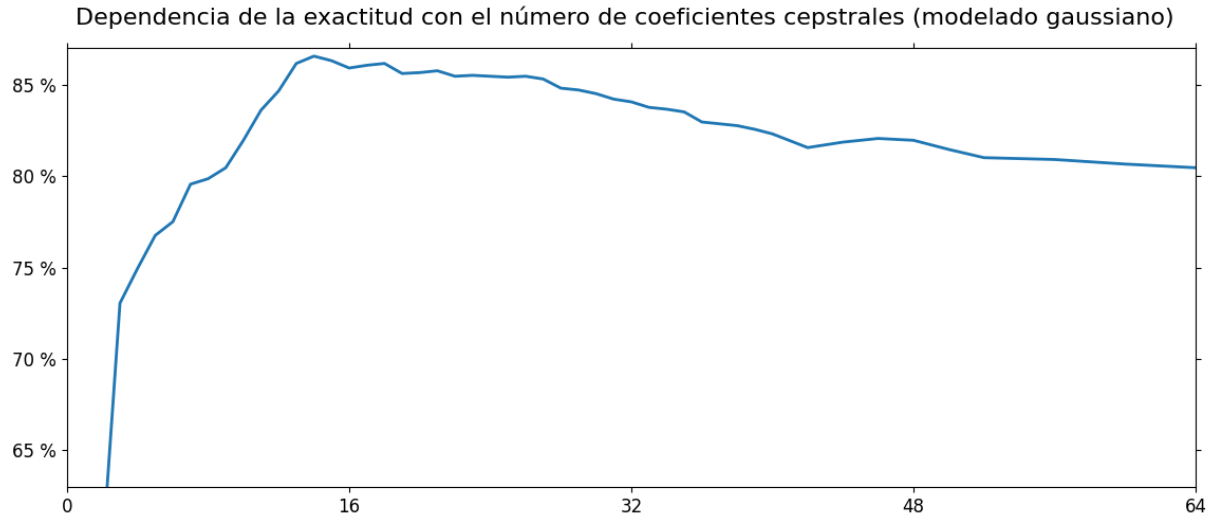
```
FUNC_PRM=cepstrum
EXEC_PRE=$DIR_PRM/$FUNC_PRM.py
[ -d $(dirname $EXEC_PRE) ] || mkdir -p $(dirname $EXEC_PRE)

echo "############################################################" | tee $EXEC_PRE
echo "#  Cepstrum real                                           #" | tee -a $EXEC_PRE
echo "############################################################" | tee -a $EXEC_PRE
echo "from numpy.fft import fft, ifft"                             | tee -a $EXEC_PRE
echo "import numpy as np"                                          | tee -a $EXEC_PRE
echo "eps = $EPS"                                                  | tee -a $EXEC_PRE
echo "numCof = $NUM_COF"                                           | tee -a $EXEC_PRE
echo "def $FUNC_PRM(x):"                                           | tee -a $EXEC_PRE
echo "    logPdgm = np.log(eps + abs(fft(x)) ** 2)"                | tee -a $EXEC_PRE
echo "    ceps = np.real(ifft(logPdgm))"                           | tee -a $EXEC_PRE
echo "    return ceps[1 : numCof + 1]"                             | tee -a $EXEC_PRE
```

La gráfica siguiente muestra la dependencia del resultado con el número de coeficientes:



Se observa que a partir de unos 20 coeficientes se obtiene un resultado sensiblemente mejor que el obtenido con el periodograma logarítmico, 85.25 % frente a 84.70 %. Sin embargo, y contrariamente a lo predicho por la teoría la tasa de exactitud no disminuye al aumentar el número de coeficientes. Este resultado no previsto puede tener que ver con el tipo de modelado, basado en distancia euclídea: los coeficientes cepstrales disminuyen de magnitud con el orden, así que los problemáticos apenas tienen repercusión en el resultado.

Utilizando modelado probabilístico, como el que se presentará en la sección 5.2.3, el comportamiento ya es el esperado. El motivo es que en ese tipo de modelado la magnitud de los coeficientes no es relevante:



Se puede observar que ahora el máximo se produce para `numCof=14`, con una exactitud `Exac=86.55%`, y que las prestaciones se degradan claramente con el aumento del número de coeficientes.

4.3.5. Limitaciones del periodograma (y del cepstrum obtenido a partir del mismo)

El periodograma es el método más inmediato para obtener una estimación del espectro de una señal, $S_x[k]$, pero presenta varios problemas que hacen su uso poco aconsejable en muchas aplicaciones, incluyendo el reconocimiento del habla.

Aunque se define el espectro de la señal como el módulo al cuadrado de su transformada de Fourier, también puede verse como la transformada de Fourier de su secuencia de autocorrelación, $r_{xx}[m]$:

$$S_x[k] = \left|\mathcal{F}\left\{x[n]\right\}\right|^2 = X[k] \cdot X^*[k] = \mathcal{F}\left\{x[n] * x^*[-n]\right\} = \mathcal{F}\left\{r_{xx}[m]\right\} \tag{4.5}$$

El objetivo del periodograma (y del resto de estimadores del espectro) es estimar este espectro a partir de un segmento de la señal de N muestras, $x_N[n]$. Al trabajar con una señal muestreada y de duración finita, podemos calcular su transformada de Fourier usando la transformada discreta de Fourier, DFT.

Por tanto, la estimación proporcionada por el periodograma, $\hat{S}_P[k]$, es el módulo al cuadrado de la DFT del segmento $x_N[n]$:

$$\hat{S}_P[k] = \left|\mathrm{DFT}\left\{x_N[n]\right\}\right|^2 = \mathrm{DFT}\left\{x_N[n] * x_N^*[-n]\right\} = \mathrm{DFT}\left\{r_{x_N x_N}[m]\right\} \tag{4.6}$$

El sesgo del periodograma

El primer problema del periodograma es que la autocorrelación de $x_N[n]$, $r_{x_N x_N}[m]$ es un estimador *sesgado* de la autocorrelación de la señal auténtica, $r_{xx}[n]$. Esto es así debido a que, mientras en el cálculo de la autocorrelación para $m = 0$ intervienen N términos auténticos de la señal, conforme $|m|$ aumenta cada vez tenemos más valores sustituidos por cero, con lo que el número de términos efectivos es sólo $N - |m|$:

$$r_{x_N x_N}[m] = r_{xx}[m]\frac{N - |m|}{N} \tag{4.7}$$

El resultado es un estimador igualmente sesgado del espectro:

$$\hat{S}_P[k] = \mathrm{DFT}\left\{r_{xx}[m]\frac{N - |m|}{N}\right\} = \mathrm{DFT}\left\{r_{xx}[m]\right\} * \mathrm{DFT}\left\{\frac{N - |m|}{N}\right\} = S_x[k] * \mathrm{sinc}^2\left(\frac{m}{N}\right) \tag{4.8}$$

Es decir, obtenemos el espectro auténtico de la señal, $S_x[k]$, pero convolucionado con una función $\mathrm{sinc}(\cdot)$ elevada al cuadrado.

El sesgo del periodograma se manifiesta de dos maneras:

- La anchura no nula del lóbulo principal de la función $\text{sinc}^2(\cdot)$ provoca una disminución de la resolución frecuencial. Por ejemplo, el peridograma no permitirá distinguir dos componentes sinusoidales cuya diferencia de frecuencias sea menor que esta anchura, ya que los lóbulos principales de las respectivas sincs se solaparán.
- Los lóbulos secundarios provocan el llamado *desparrame frecuencial* consistente en que la energía de cada frecuencial se distribuye por frecuencias distintas a la real.

Es posible controlar, hasta cierto punto, el sesgo del periodograma sustituyendo la ventana rectangular por otra de forma apropiada, como las de von Hann, Hamming, Blackmann, etc.

- La ventana rectangular tiene mucho desparrame frecuencial, pero su resolución es la máxima posible.
- Otras ventanas consiguen reducir el desparrame, a costa de disminuir también la resolución.

Es importante reseñar, finalmente, que el sesgo del periodograma disminuye conforme aumenta el número de muestras N. En el límite $N \to \infty$, el peridograma no tiene sesgo. Es lo que se denomina *asintóticamente insesgado*. No obstante, eso es una ventaja despreciable cuando se manejan señales cuasiestacionarias, como es el caso de la de voz.

La varianza del periodograma

El segundo y principal problema del periodograma es que proporciona una estimación del espectro de los procesos estocásticos con una varianza muy elevada. Esta varianza se manifiesta como variaciones aleatorias en torno al valor estimado. De alguna manera, si $S_x[k]$ es el valor auténtico del espectro, el proporcionado por el periodograma será (casi) cualquier cosa alrededor de ese valor. Para colmo, esta varianza no disminuye con el tamaño del segmento; decimos que se trata de un método *inconsistente*.

Aunque el análisis de la varianza del periodograma es bastante complicado y excede totalmente el propósito de este escrito, hay un caso en el que su valor es relativamente sencillo: si la señal es ruido blanco gaussiano de media cero y enventanado con una ventana rectangular de longitud infinita, entonces la función de densidad del valor estimado responde a una exponencial de media y desviación típica iguales al valor auténtico:

$$\text{pdf}\left\{\hat{S}_P[k]\right\} = \frac{1}{S_x[k]} e^{-\frac{\hat{S}_P[k]}{S_x[k]}} \tag{4.9}$$

Aunque el análisis se complica conforme alguna de las condiciones anteriores deja de cumplirse, el resultado obtenido para el ruido blanco de longitud infinita es bastante generalizable a señales de otro tipo.

La desviación típica de un estimador, raíz cuadrada de su varianza, nos proporciona una orientación acerca de cuál puede ser el valor real dada su estimación. En función de la forma de la función de densidad, el valor real tendrá más o menos probabilidad de estar a un cierto número de desviaciones típicas del valor estimado.

Para algún valor de k_1 y k_2, los valores más probables del estimador estarán en el margen:

$$S_x[k] - k_1\sigma_{\hat{S}_P} \leq \hat{S}_P[k] \leq S_x[k] + k_2\sigma_{\hat{S}_P} \tag{4.10}$$

Que, en el caso del periodograma, y asumiendo que su media y su varianza son iguales, se reduce a:

$$S_x[k](1 - k_1) \leq \hat{S}_P[k] \leq S_x[k](1 + k_2) \tag{4.11}$$

Extraemos logaritmos:

$$\log S_x[k] + \log(1 - k_1) \leq \log \hat{S}_P[k] \leq \log S_x[k] + \log(1 + k_2) \tag{4.12}$$

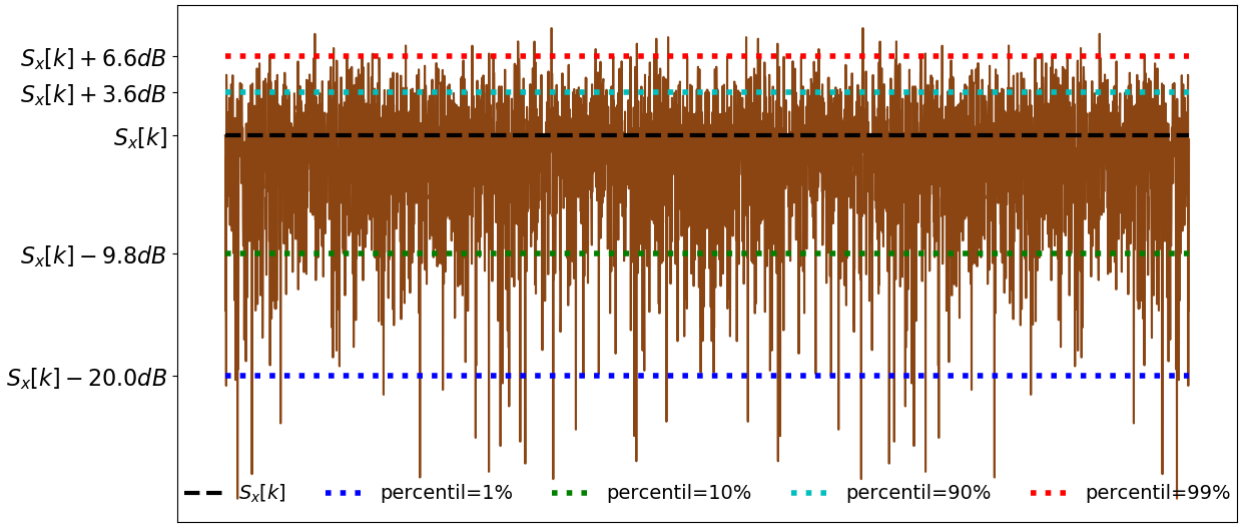
Es decir, los valores mínimo y máximo que podemos esperar de la estimación están a una distancia constante del valor real, formando una franja de anchura más o menos constante.

Podemos visualizar esta idea determinando los valores para los que hay una cierta probabilidad de que la estimación se más pequeña que el valor; lo que se denomina *percentiles*.

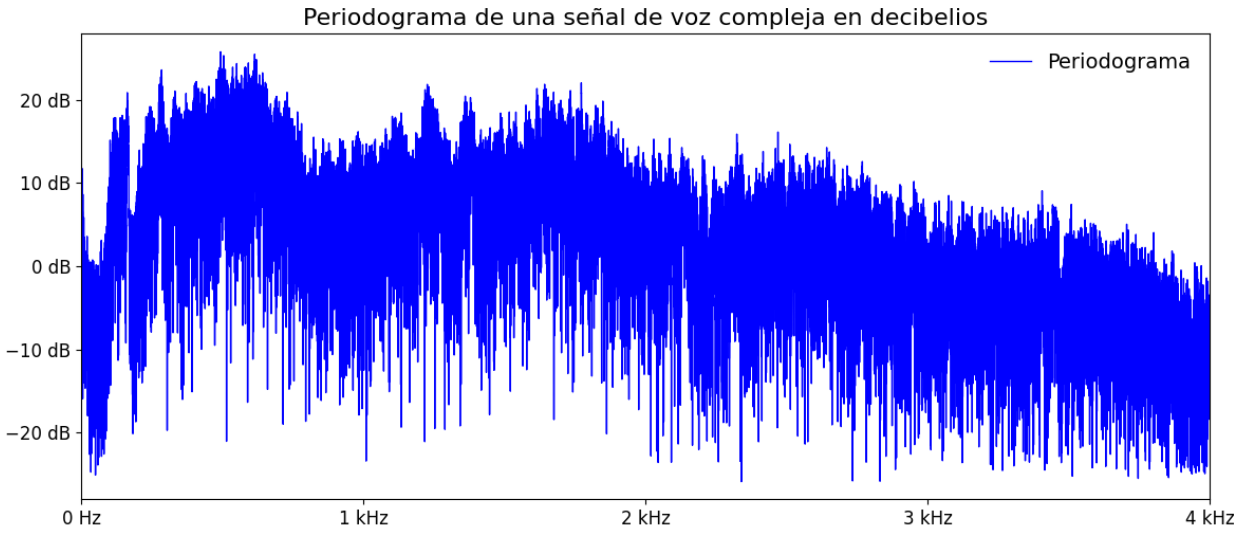
En el caso de ruido blanco gaussiano de media cero y longitud infinita, algunos percentiles significativos son:

Percentil	**Umbral**	**Margen**
1 %	$0.01\, S_x[k]$	$S_x[k] - 20.0\,\text{dB}$
10 %	$0.11\, S_x[k]$	$S_x[k] - 9.8\,\text{dB}$
90 %	$2.30\, S_x[k]$	$S_x[k] + 3.6\,\text{dB}$
99 %	$4.61\, S_x[k]$	$S_x[k] + 6.6\,\text{dB}$

Gráficamente, estos percentiles mostrados sobre una muestra de ruido blanco, son:



En señales de naturaleza distinta, en particular en segmentos de corta duración de señales de voz, como la mostrada en la página 66, esta varianza sigue estando presente, aunque es más difícil de evaluar y de visualizar. Sin embargo, al visualizar el periodograma de señales de mayor longitud, la franjas mencionadas, de más de 20 dB, se ponen claramente de manifiesto:



4.4. Otros métodos de estimación espectral

Las limitaciones del periodograma obligan a buscar soluciones en las que, sobre todo, se reduzca la varianza del estimador. Existen múltiples alternativas, las más clásicas son los estimadores de Blackman-Tukey y de máxima entropía, que se explican a continuación. En los últimos años, la opción más usada en tareas de reconocimiento del habla son los MFCC (*Mel-frequency cepstrum coefficients*). En los apartados siguientes se hace un repaso de los dos primeros. También se hace una mención al tercero, junto con enlaces para conseguir más información y/o acceder a bibliotecas que permiten su inclusión en `Ramses`.

4.4.1. Estimador de Blackman-Tukey

El modo más inmediato de abordar la varianza del periodograma consiste en *suavizar* sus valores mediante promediado, en general ponderado. Este promediado es equivalente a una convolución y análogo al realizado cuando se filtra una señal temporal, sólo que realizado sobre los valores frecuenciales.

Suavizado en el tiempo es lo mismo que filtrar paso-bajo, eliminando las componentes de señal de variación más rápida. En frecuencia, no es muy riguroso llamarlo *filtrado*, pero el efecto es el mismo: eliminar las fluctuaciones más rápidas en el valor del periodograma.

Llamamos estimador de Blackman-Tukey, $\hat{S}_{BT}[k]$ al resultado de suavizar el peridograma convolucionándolo con una función $W_c[k]$ (más adelante se verá el por qué de esta nomenclatura):

$$\hat{S}_{BT}[k] = \hat{S}_P[k] * W_c[k] \tag{4.13}$$

Ponemos el segundo término de la igualdad en función de la transformada inversa de Fourier de sus elementos:

$$\hat{S}_{BT}[k] = \mathrm{DFT}\left\{ \underbrace{\mathrm{IDFT}\left\{\hat{S}_P[k]\right\}}_{r_{xx}[m]} \right\} * \mathrm{DFT}\left\{ \underbrace{\mathrm{IDFT}\left\{W_c[k]\right\}}_{w_c[m]} \right\} = \mathrm{DFT}\left\{r_{xx}[m] \cdot w_c[m]\right\} \tag{4.14}$$

Es decir, el suavizado de Blackman-Tukey es equivalente a calcular la DFT de la secuencia de autocorrelación multiplicada por una función, $w_c[m]$, que denominaremos *ventana de autocorrelación*.

La elección de la ventana de autocorrelación es delicada: es necesario garantizar que su transformada de Fourier es no negativa, ya que, en caso contrario, podríamos obtener valores negativos del espectro, lo cual no sólo no tiene sentido, sino que invalidaría toda la estimación. Por ejemplo, no se puede simplemente truncar la secuencia de autocorrelación, porque eso es equivalente a multiplicar por una ventana rectangular, cuya transformada de Fourier es la función sinc[·], que tiene valores negativos.

Las ventanas más usualmente empleadas en el método de Blackman-Tukey son las formadas por convolución de dos ventanas simétricas e idénticas, ya que su transformada de Fourier es el cuadrado de una función real. Así, podemos usar la convolución de dos ventanas de Hamming o de van Hann, aunque lo más habitual es la de dos ventanas rectangulares, que resulta en una ventana de autocorrelación triangular nunca negativa, $\mathrm{sinc}^2[\cdot]$.

Junto con la elección de la ventana de autocorrelación, el único parámetro importante es la longitud de la ventana. Cuanto menor sea ésta, mayor será el efecto de suavizado. Es importante recordar que la secuencia de autocorrelación es una función par, con parte positiva y negativa. La energía de la señal, o autocorrelación de cero, está en el centro de la secuencia. Por tanto, al multiplicar por la ventana, tendremos que centrarla en ese punto, tomando tantos valores a la izquierda como a la derecha del mismo, pero que, como la autocorrelación es par, serán idénticos. Es habitual, por tanto, considerar el `orden` de la estimación igual al número de coeficientes distintos efectivamente usados. En ese caso, la longitud de la ventana será la suma de `orden` valores para `m<0`, el valor para `m=0` y los `orden` valores para `m>0`: `len(corr)=2*orden+1`.

Procedimiento de cálculo del estimador de Blackman-Tukey

Para obtener la estimación espectral de Blackman-Tukey, hemos de obtener primero la secuencia de autocorrelación; a continuación, su parte central se multiplica por una ventana de transformada positiva y la longitud deseada; finalmente, se obtiene $\hat{S}_{BT}[k]$ calculando la DFT del resultado.

El cálculo de la autocorrelación se realiza usando la función `correlate()` de `numpy`.

- Esta función calcula la correlación de dos secuencias `a` y `v`. Obtenemos la autocorrelación haciendo las dos iguales a la señal a analizar.
- Por defecto, `correlate()` sólo calcula los coeficientes *válidos*, es decir, aquellos que son calculados con valores de la señal y no con los ceros con que la rellenamos. Para poderla usar en esta aplicación, hemos de fijar `mode='full'`, de manera que se calculen todos los términos de la autocorrelación sesgada.
- Para una señal de longitud N, el resultado es la secuencia completa de autocorrelación, que abarca desde $m = -N + 1$ hasta $m = N - 1$. Su longitud es, por tanto, $2N - 1$, con el valor correspondiente a $m = 0$ en el índice $N - 1$.
- Invocamos la función `fft()` con el segundo argumento igual a la longitud de la señal analizada para que la estimación sea de la misma longitud que el periodograma.

Por otro lado, y debido a errores de redondeo, es muy probable que el resultado tenga una parte imaginaria residual. Para evitar obtener una estimación compleja, nos quedamos con el valor absoluto.

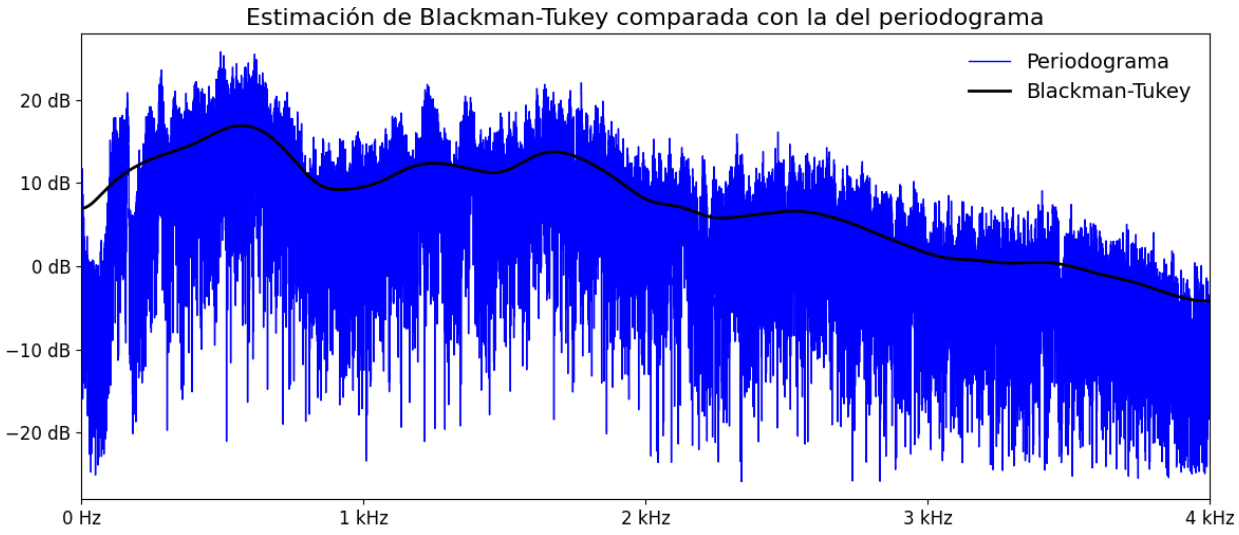
Todo junto, escribimos el código correspondiente en el script **`bkackman_tukey.py`**:

blackman_tukey.py

```
import numpy as np
from numpy.fft import fft, fftshift

def blackmanTukey(x, orden):
    N = len(x)
    corr = np.correlate(x, x, mode='full')[N - orden - 1 : N + orden]
    wCorr = corr * np.bartlett(2 * orden + 1)

    return np.abs(fft(wCorr, n=N)) ** 2
```

Nótese que el código de este estimador es suficientemente complejo como para ser escrito en un script aparte, y no directamente en `todo.py`. Esto debería ser la norma general en casos como éste.

Al usar este estimador, con `orden=40`, con la misma señal que la de la gráfica en la página 74, podemos observar que el estimador sigue la evolución de la envolvente del espectro, pero con una fuerte reducción de la varianza en la estimación: lo que antes era una franja de casi 30 dB de anchura, ahora es una línea fina y bastante suave.



Incorporación de la estimación de Blackman-Tukey a `Ramses`

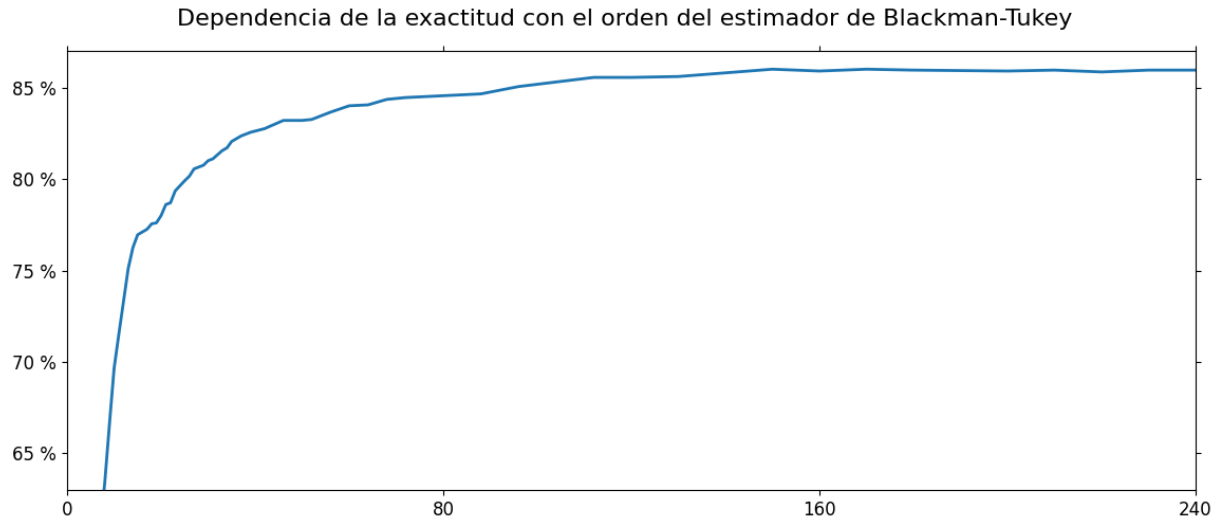
Sólo queda incorporar la estimación al script `todo.sh`. Existen varios modos de hacerlo. Por ejemplo, podemos incorporar el script `blackman_tukey.py` al argumento `--execPre` del programa `parametriza.py` y, a continuación, definir una función que invoque el estimador. Sin embargo, y para evitar modificar en exceso el script `todo.py`, ejecutaremos `blackman_tukey.py` desde la función creada en él.

Necesitaremos dos variables para controlar el algoritmo: por un lado, el número de coeficientes de autocorrelación conservados, u `orden` del análisis; por otro, como en el caso del periodograma, un umbral que evite problemas al extraer el logaritmo de valores cercanos a cero.

todo.sh

```
FUNC_PRM=BT
EXEC_PRE=$DIR_PRM/$FUNC_PRM.py
[ -d $(dirname $EXEC_PRE) ] || mkdir -p $(dirname $EXEC_PRE)

EPS=10
ORDEN=40

echo "#############################################################" | tee $EXEC_PRE
echo "#  Estimador de Blackman-Tukey                              #" | tee -a $EXEC_PRE
echo "#############################################################" | tee -a $EXEC_PRE
echo "exec(open('blackman_tukey.py').read())"                        | tee -a $EXEC_PRE
echo "import numpy as np"                                            | tee -a $EXEC_PRE
echo "orden=$ORDEN"                                                  | tee -a $EXEC_PRE
echo "eps=$EPS"                                                      | tee -a $EXEC_PRE
echo "def $FUNC_PRM(x): return np.log(eps + blackmanTukey(x, orden))"   | tee -a
↪  $EXEC_PRE
```

El resultado obtenido en el reconocimiento depende mucho del orden usado. La gráfica siguiente muestra esta dependencia:



Puede verse que la exactitud aumenta más o menos monótonamente, alcanzando un valor sensiblemente superior al obtenido con el periodograma para órdenes del análisis iguales a `orden=160` o mayores. Para `orden=240` se obtiene una `Exac=85.95%`, un punto y cuarto mejor que la obtenida con aquél.

Al usar este estimador, y de hecho todos ellos, para tareas de reconocimiento del habla, es conveniente combinarlo con el cepstrum. Lo hacemos en la función `cepstrumBT()` que incluimos en el mismo fichero `blackman_tukey.py`.

blackman_tukey.py

```
def cepstrumBT(x, orden, *, eps=0, numCof=None):
    if not numCof: numCof = orden

    bt = blackmanTukey(x, orden)

    return np.real(ifft(10 * np.log10(eps + bt)))[1: numCof + 1]
```

Con una elección adecuada de `orden`, `numCof` y `eps`, esta parametrización permite alcanzar, usando el modelado basado en distancia euclídea, exactitudes superiores al 85 %. En cualquier caso, como se vio anteriormente, llegados a este punto de exactitud, el factor principal que limita las prestaciones del sistema es antes el tipo de modelado acústico que el de parametrización de la señal.

4.4.2. Estimador de Máxima entropía

Aunque el estimador de Blackman-Tukey proporciona una ligera mejoría respecto al periodograma, el resultado no es sustancialmente mejor. Un problema de la estimación de Blackman-Tukey es que el espectro obtenido tiende a ser más plano que el de la señal. Es decir, proporciona un espectro *blanqueado*. Esto está motivado por el modo cómo se construye la estimación, atenuando e incluso anulando los términos elevados de la autocorrelación. Esta atenuación hace que la autocorrelación se asemeje más a la delta de Dirac, que se corresponde con un espectro plano.

El estimador de máxima entropía es semejante al de Blackman-Tukey, en el sentido de que la estimación se realiza a partir de los `orden` primeros coeficientes de autocorrelación; pero con dos importantes diferencias:

- Los `orden` coeficientes de la autocorrelación son conservados intactos, sin ninguna atenuación.
- Los coeficientes anulados en el método de Blackman-Tukey se sustituyen por valores calculados de manera que el error cometido en la estimación esté lo más incorrelado posible; de ahí el nombre de *máxima entropía*.

John Parker Burg demostró en su Tesis Doctoral que la estimación de máxima entropía está directamente relacionada con la predicción lineal. En un predictor lineal se usa una combinación lineal de las muestras pasadas de la señal para obtener una estimación de la muestra siguiente:

$$x_p[n] = \sum_{k=1}^{P} a_k x[n-k] \tag{4.15}$$

Los coeficientes a_k se calculan de manera que se minimice el *error de predicción*, $e_p[n]$, definido como la diferencia entre el valor previsto y el real, $e_p[n] = x[n] - x_p[n]$. Si el criterio es la minimización de la potencia de este error, podemos obtener los coeficientes aplicando las ecuaciones de Yule-Walker:

$$a_k = R_{xx}^{-1} \cdot P_{rs} \tag{4.16}$$

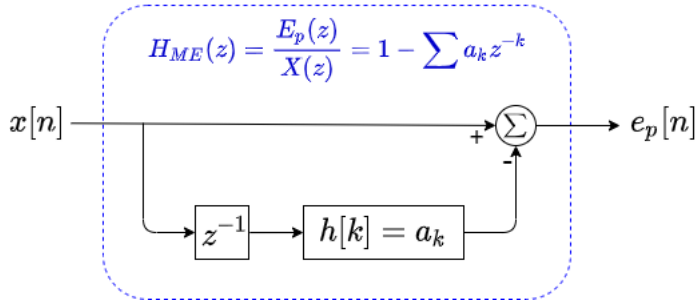
Donde R_{xx} es la matriz de autocorrelación, que es una matriz cuadrada de dimensión $P-1$, en la que los elementos en las diagonales son todos iguales (estructura de Toeplitz) a los coeficientes de la autocorrelación:

$$R_{xx} = \begin{bmatrix} r_{xx}[0] & r_{xx}[1] & r_{xx}[2] & r_{xx}[3] & \cdots & r_{xx}[P-1] \\ r_{xx}[1] & r_{xx}[0] & r_{xx}[1] & r_{xx}[2] & \cdots & r_{xx}[P-2] \\ r_{xx}[2] & r_{xx}[1] & r_{xx}[0] & r_{xx}[1] & \cdots & r_{xx}[P-3] \\ r_{xx}[3] & r_{xx}[2] & r_{xx}[1] & r_{xx}[0] & \cdots & r_{xx}[P-4] \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{xx}[P-1] & r_{xx}[P-2] & r_{xx}[P-3] & r_{xx}[P-4] & \cdots & r_{xx}[0] \end{bmatrix} \tag{4.17}$$

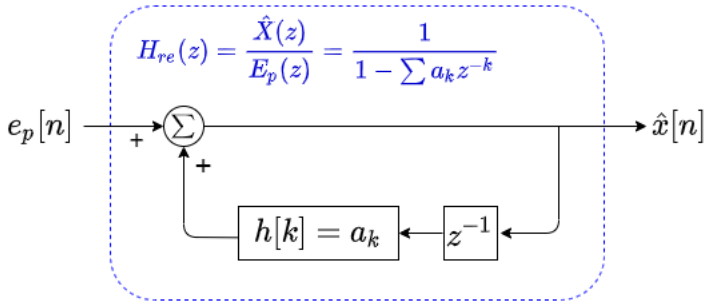
Por su parte, P_{rs} es la proyección, o correlación cruzada, entre la referencia y los datos, que en el caso que nos ocupa es la señal actual, $x[n]$, y sus muestras pasadas:

$$P_{rs} = \begin{bmatrix} r_{xx}[1] \\ r_{xx}[2] \\ r_{xx}[3] \\ \vdots \\ r_{xx}[P] \end{bmatrix} \tag{4.18}$$

El resultado es un error de predicción que no sólo es de potencia mínima, sino que además es de espectro máximamente plano. Esto quiere decir que, si construimos el sistema cuya salida es el error de predicción cuando la entrada es la señal, $H_{ME}(z)$, su respuesta frecuencial es la mejor aproximación, según el criterio de máxima entropía, del inverso del espectro de la señal.



Dado un predictor lineal, se define el filtro *reconstructor* aquél cuya salida, cuando la entrada es el error de predicción, es igual a la entrada del filtro predictor. Su función de transferencia es, simplemente, la inversa de la del predictor:



Así pues, este sistema proporciona una señal con el mismo espectro que la señal analizada, $\hat{x}[n]$, cuando a su entrada tenemos una señal de espectro máximamente plano, $e_p[n]$. Por tanto, el módulo de su función de transferencia al cuadrado, $|H_{re}(f)|^2$ proporciona una estimación del espectro de la señal, con un error que maximiza la entropía para el `orden` utilizado.

Procedimiento de cálculo de la estimación de Máxima Entropía

Los pasos necesarios para el cálculo de la estimación de Máxima Entropía son los siguientes:

- Obtención del predictor lineal óptimo:
  - Calculamos la autocorrelación de la señal, $r_{xx}[m]$. En Python, usaremos la función `correlate()` de `numpy`, de manera análoga a lo realizado en la sección 4.4.1 para el estimador de Blackman-Tukey.

- Construimos la matriz de autocorrelación, R_{xx}. Esta matriz cuadrada, de dimensión `orden`, tiene estructura de Topelitz; es decir, sus diagonales son constantes e iguales a los coeficientes $r_{xx}[m]$ para $0 \leq m <$ `orden`. Para construirla usaremos la función `toeplitz` de `scipy.linalg`.
- Construimos el vector de proyección referencia datos, P_{rs}, con los valores de $r_{xx}[m]$ en el rango $1 \leq m \leq$ `orden`.
- Los coeficientes del predictor óptimo se obtienen aplicando $a_k = R_{xx}^{-1} \cdot P_{rs}$.

- Construimos el filtro reconstructor $H_{re}(z)$. Se trata de un filtro IIR cuyo numerador es uno y su denominador es la concatenación de uno seguido de los coeficientes de predicción lineal cambiados de signo.

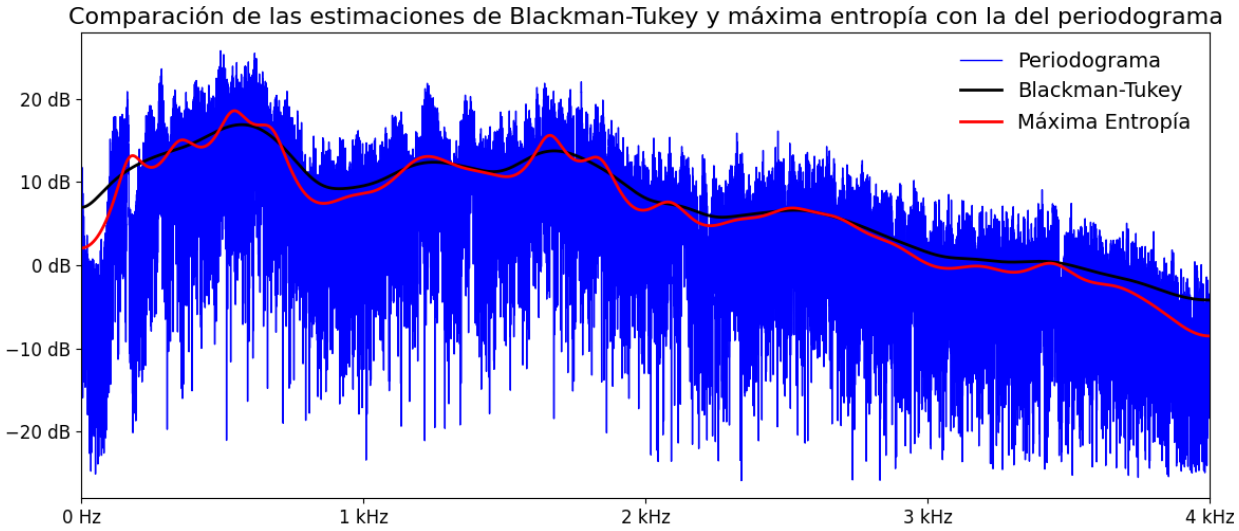
- Calculamos la respuesta impulsional del filtro reconstructor filtrando una delta de Dirac de la misma longitud que la señal a analizar.

- Ajustamos la potencia de la respuesta impulsional para que sea igual a la de la señal. Esto es necesario porque el predictor es independiente de la potencia de la señal analizada; sólo depende de la forma de su espectro.

  Aunque hay otros mecanismos para obtener este ajuste de potencias, son mucho más complicados, sobre todo su explicación, y el ajuste directo es igualmente efectivo.

- La estimación de máxima entropía del espectro de la señal es el módulo al cuadrado de la transformada de Fourier de la respuesta impulsional del filtro reconstructor.

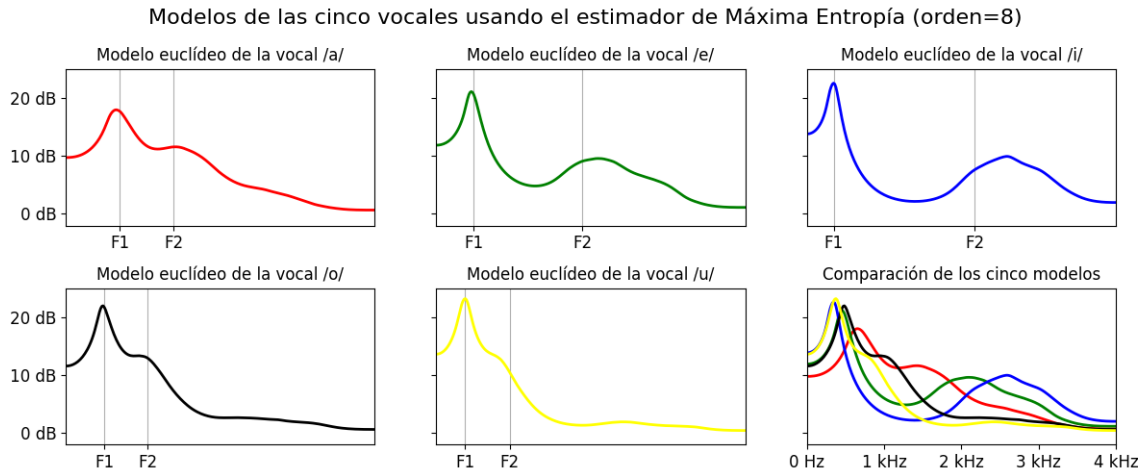
Al estimar el espectro de la señal de las secciones 4.3.5 y 4.4.1, el resultado, con el mismo `orden=40` que se usó en la estimación de Blackman-Tukey, se adapta mucho mejor a la evolución de la envolvente del espectro:



Por ejemplo, incorporando el estimador de máxima entropía en decibelios, los modelos obtenidos con un orden tan bajo como `orden=8` proporcionan un resultado mucho mejor

que las otras alternativas presentadas (cerca de `Exac=90%` usando distancia euclídea, aún más usando modelos probabilísticos).

La gráfica siguiente muestra los modelos obtenidos, en los cuales se aprecia como la captura de la posición de los formantes de las cinco vocales es casi perfecta:



Puede resultar interesante comparar esta gráfica con la mostrada en la página 63 a partir del uso del periodograma como parámetro. Mientras ambas gráficas presentan evoluciones semejantes *grosso modo*, en aquel caso aparece un cierto *tembleque* alrededor del valor medio que podemos identificar, sin lugar a dudas, como ruido (o, lo que es lo mismo, varianza) en el estimador.

En código Python, este procedimiento es:

maxima_entropia.py

```
import numpy as np
from scipy.fft import fft
from scipy.linalg import toeplitz, inv
from scipy.signal import lfilter

def maximaEntropia(x, orden):
    N = len(x)
    corr = np.correlate(x, x, mode='full')[N - 1 : N + orden]
    Rxx = toeplitz(corr[: -1])
    Prs = corr[1: ]
    hlp = inv(Rxx) @ Prs

    nume = [1]
    deno = np.concatenate(([1], -hlp))

    delta = np.zeros(N); delta[0] = 1
    hrec = lfilter(nume, deno, delta)

    hrec *= np.std(x) / np.std(hrec)

    return np.abs(fft(hrec)) ** 2
```

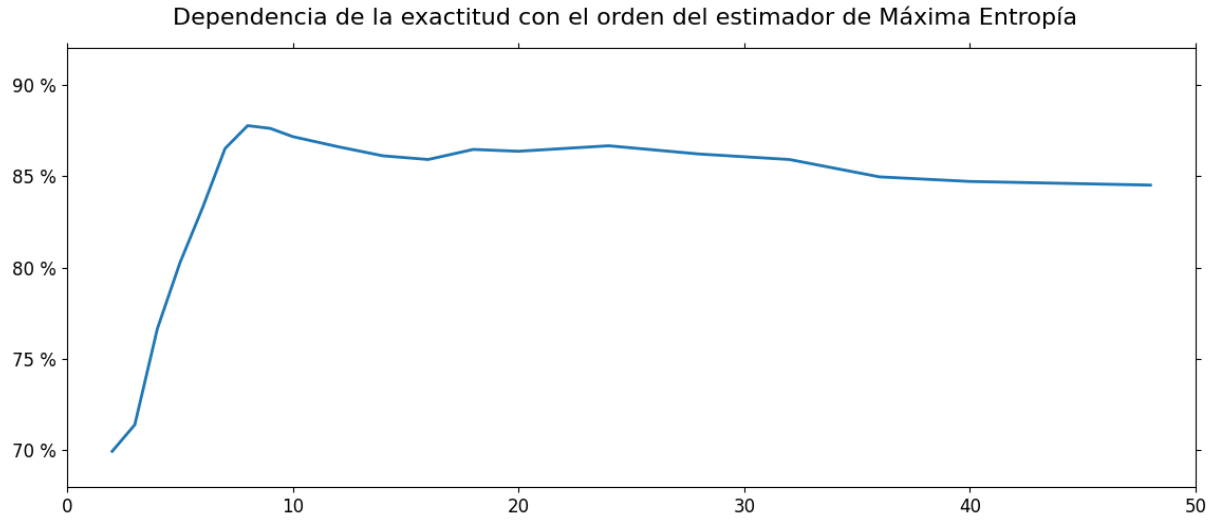
Incorporación de la estimación de Máxima Entropía a `Ramses`

Añadimos las órdenes necesarias para usar el estimador de Máxima Entropía en el script `todo.py`:

todo.sh

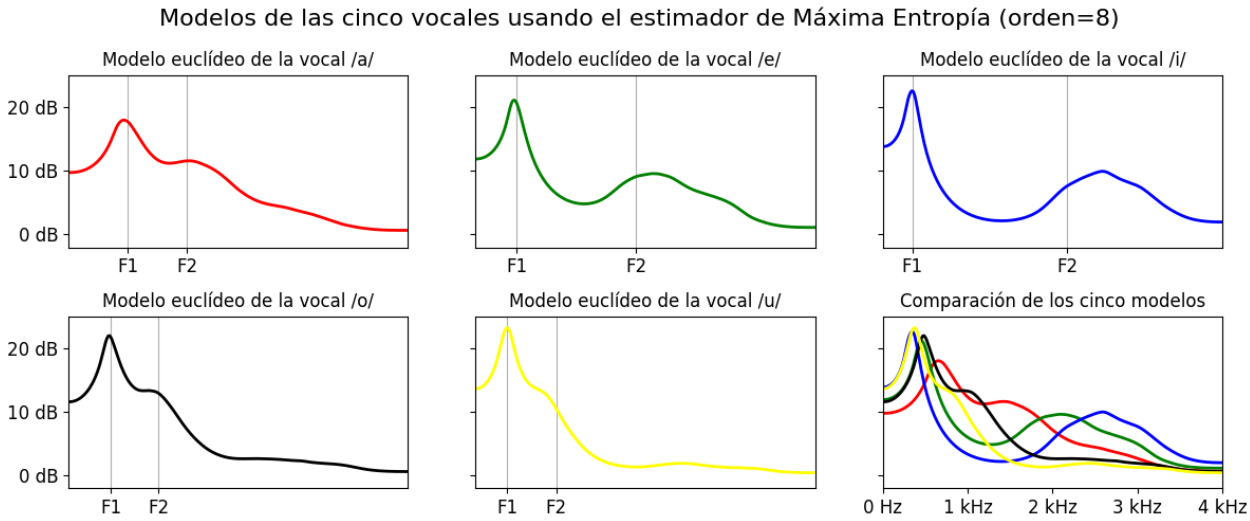
```
FUNC_PRM=ME
EXEC_PRE=$DIR_PRM/$FUNC_PRM.py
[ -d $(dirname $EXEC_PRE) ] || mkdir -p $(dirname $EXEC_PRE)

echo "#############################################################" | tee $EXEC_PRE
echo "#  Uso del estimador de Máxima Entropía                     #" | tee -a $EXEC_PRE
echo "#############################################################" | tee -a $EXEC_PRE
echo "exec(open('maxima_entropia.py').read())"                       | tee -a $EXEC_PRE
echo "import numpy as np"                                            | tee -a $EXEC_PRE
echo "orden=$ORDEN"                                                  | tee -a $EXEC_PRE
echo "eps=$EPS"                                                      | tee -a $EXEC_PRE
echo "def $FUNC_PRM(x): return np.log(eps + maximaEntropia(x, orden))"  | tee -a
↪  $EXEC_PRE
```

Como en el caso del estimador de Blackman-Tukey, el resultado en el reconocimiento depende mucho del `orden`:



Sin embargo, en este caso el valor óptimo del orden es mucho más bajo, `orden=8` que para el caso de Blackman-Tukey. Además, el resultado es mucho mejor que con el periodograma o el estimador de Blackman-Tukey, con una exactitud máxima `exac=87.75%`.

Si observamos los modelos obtenidos, se aprecia la mejor caracterización de cada una de las vocales:



Se puede ver que, ahora, los modelos capturan casi perfectamente la posición de los dos primeros formantes de cada vocal, facilitando su reconocimiento.

4.4.3. Los coeficientes cepstrales en escala Mel (MFCC)

Aunque los métodos de estimación espectral vistos en los apartados precedentes ya permiten construir sistemas de reconocimiento del habla razonablemente potentes, continúan presentando limitaciones importantes. Una de ellas es que no atienden al carácter cuasi-logarítmico de la respuesta en frecuencia del oído humano. Es decir, al menos para las frecuencias medias y altas, la información por intervalo logarítmico (década, octava, tercio de octava, etc.) es constante. Sin embargo, en una escala de frecuencias lineal, la octava más elevada (la que en las gráficas precedentes cubre el margen $2000\,\text{Hz} \leq f \leq 4000\,\text{Hz}$) abarca la mitad de la banda; la siguiente octava más elevada ($1000\,\text{Hz} \leq f \leq 2000\,\text{Hz}$), la mitad de la mitad restante; y así sucesivamente.

Aunque existen otros procedimientos de estimación espectral adaptados a un tratamiento logarítmico de la frecuencia, el modo más sencillo consiste en hacer pasar la señal por un banco de filtros paso-banda en el que la anchura de cada banda es proporcional a su frecuencia central. La densidad espectral asociada a cada banda es igual a la potencia media de la señal en ella. Este planteamiento, además, tiene la ventaja de permite fijar los límites de cada banda con absoluta flexibilidad. Así, es conocido que, a bajas frecuencias, el comportamiento del oído es más cercano al lineal que al logarítmico. La escala Mel es un modo de reflejar este comportamiento desigual a bajas y a medias y altas frecuencias.

En el último cuarto del s. XX, Mermelstein, Bridle y Brown propusieron el uso de un banco de filtros equiespaciados en frecuencia Mel como base de sus *coeficientes cepstrales en escala Mel* (MFCC).

El procedimiento de obtención de los MFCC sigue los pasos siguientes:

1. Se calcula el periodograma de la señal a partir de su transformada discreta de Fourier:

$$S_P[k] = \left|\text{DFT}\left\{x[n]\right\}\right|^2$$

2. Se obtienen los `numBnd` valores de potencia para cada banda multiplicando por filtros solapados de respuesta en frecuencia triangular, $H_m[k]$ y sumando para todas las frecuencias:

$$P[m] = \sum_{k=k^m_{\text{mín}}}^{k^m_{\text{máx}}} H_m[k] S_P[k]$$

3. Se extrae el logaritmo de los $m = 0...$`numBnd-1` valores de potencia.

4. Los coeficientes cepstrales son la transformada coseno inversa de los logaritmos de las potencias:

$$c[l] = \text{IDCT}\left\{\log P[m]\right\}$$

5. Suelen usarse únicamente los `numCof` coeficientes de orden más bajo, con `numCof` $\leq$ `numBnd`.

Puede encontrar una descripción más detallada en el MFCC Tutorial escrito por los autores de una biblioteca Python que los implementa junto con otros esquemas de parametrización, la python_speech_features. Se recomienda el uso de esta biblioteca si se desea utilizar esta parametrización (puede instalarla en su sistema con la orden `pip3 install python_speech_features`).

Usando los valores por defecto de la función `mfcc()` (salvo los necesarios para ajustar la frecuencia de muestreo y la longitud de la ventana de señal), `ramses` alcanza un 89 % de exactitud; valor que pasa del 90 % tras una mínima optimización de los argumentos de la función:

```
        a       e       i       o       u
a       361     5       0       32      2
e       2       378     19      0       1
i       0       27      366     0       7
o       22      8       0       335     35
u       3       1       2       27      367

Exac = 90.35%
```

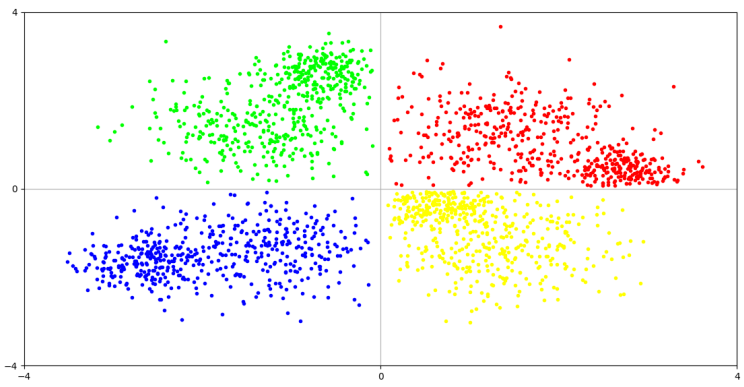
Capítulo 5

Modelado acústico usando Programación Orientada a Objeto

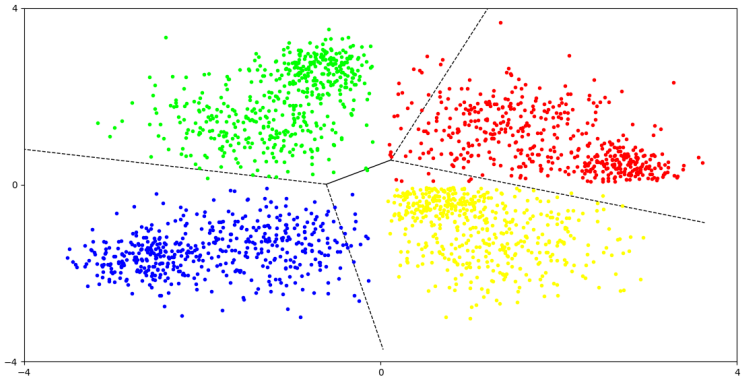
Usando el modelado acústico basado en distancia euclídea es complicado mejorar mucho más los resultados obtenidos en el apartado anterior con parámetros cepstrales basados en un estimador de espectro apropiado. El problema no está tanto en la representación de las señales como en la caracterización de las clases a reconocer; los clasificadores basados en distancia presentan fronteras entre las clases que vienen determinadas, en gran medida, por la propia elección de la distancia, y no por las características de las clases.

Por ejemplo, en el caso de la distancia euclídea, las fronteras son hiperplanos de dimensión $N-1$ perpendiculares a la recta que une las medias geométricas de las clases. Incluso en el caso de que fronteras planas permitieran clasificar correctamente las distintas clases, las obtenidas minimizando la distancia cuadrática media no tienen por qué ser adecuadas.

La gráfica siguiente muestra un caso en que hiperplanos de dimensión 1 (rectas) permiten clasificar correctamente un problema bidimensional con cuatro clases, ya que la clase verde ocupa el cuadrante superior izquierdo, la roja el superior derecho, la azul el inferior izquierdo y la amarilla el inferior derecho:



Sin embargo, las fronteras establecidas usando distancia euclídea con el criterio de mínima distancia cuadrática media no seguirían los ejes, sino que serían las siguientes:



Puede observarse que estas fronteras subóptimas provocan numerosos errores de clasificación, con muchas realizaciones rojas reconocidas como verdes o amarillas, realizaciones azules reconocidas como amarillas, etc. El problema es aún mayor cuando el espacio vectorial es de dimensión mayor.

Aunque se han realizado múltiples propuestas basadas en otros tipos de distancia y/o criterios de optimización, los resultados obtenidos no son mucho mejores que los obtenidos con distancia euclídea y una buena parametrización de las señales. Es necesario, por tanto, buscar la mejoría del sistema de reconocimiento en el propio criterio de clasificación, lo que se denomina el *modelado acústico*.

El reemplazo tradicional de los criterios basados en distancia ha sido los basados en probabilidad; en concreto, los basados en el teorema de Bayes. Más recientemente, los sistemas basados en redes neuronales han alcanzado una gran popularidad gracias a las excelentes prestaciones obtenidos con ellos.

La introducción de estos nuevos tipos de modelado acústico implica modificaciones del sistema importantes en aspectos como el entrenamiento, el reconocimiento y el almacenamiento de los modelos. Lo peor del caso es que cada tipo presente o futuro requiere soluciones y algoritmos diferentes. Para evitar tener que modificar los programas básicos de `ramses` cada vez que se quiera usar un modelado acústico distinto, se va a proceder de manera semejante a como se hizo en la fase de parametrización; es decir, el tipo concreto de modelado empleado se determinará a partir de los argumentos de los programas involucrados. Sin embargo, dado que el tipo de modelado no afecta a una sola función, sino a varias, y, además, también implica modificaciones en las estructuras de datos empleadas, en lugar de programación se optará por una solución basada en programación orientada a objetos.

5.1. La clase genérica `Modelo`

La adaptación del sistema a la programación orientada a objeto implica convertir los modelos de las unidades, que hasta ahora eran simplemente la media de sus realizaciones,

en objetos de una clase. Como cada tipo de modelado implicará una clase diferente, se definirá una clase genérica, `Modelo`, de la cual serán herederas las clases particulares de cada uno; por ejemplo, en el caso del modelado euclídeo, `ModEucl`[1].

De manera análoga a como se hacía con el paso de la función de parametrización a la función `parametriza()`, la clase concreta que se usará en el entrenamiento y reconocimiento se pasará como un argumento más de las respectivas funciones. Como valor inicial, se le asignará la clase genérica `ModEucl`.

La clase `Modelo` tendrá como atributo principal la estructura de datos que represente la unidad acústica. Por ejemplo, en el modelado euclídeo la unidad está representada por un vector con la media de las realizaciones de la unidad, mientras que en un modelado con funciones de densidad gaussianas, lo estará con un vector que represente la media de la función y una matriz que represente su covarianza.

Además, deberemos definir toda una serie de métodos para entrenar el modelo y usarlo en el reconocimiento:

`__init__(self, pathMod=None):`
: Crea un objeto de la clase. Si `pathMod` evalúa a `True`, el modelo se lee del archivo indicado; en caso contrario, el modelo inicial debe crearse desde cero.

`leeMod(self, pathMod):`
: Lee el modelo almacenado en el fichero `pathMod`.

`escrMod(self, pathMod):`
: Escribe el modelo en el fichero indicado por `pathMod`.

`inicMod(self):`
: Inicializa el modelo de manera que, en la fase de entrenamiento, podamos incorporar la información aportada por las realizaciones que participan en la misma.

`__add__(self, prm):`
: Sobrecarga del operador suma (`+`) para incorporar al modelo la contribución de la trama de señal parametrizada `prm`.

`calcMod(self):`
: Recalcula el modelo a partir de la información acumulada durante el entrenamiento.

`__call__(self, prm):`
: Sobrecarga la llamada a función sobre la clase (`()`) para obtener el *score* de la señal `prm` según el modelo. Es un *score* en el sentido de que, cuanto mayor sea su valor, más probable es que la señal pertenezca a la unidad modelada (en oposición a lo que ocurre cuando usamos distancias).

[1] Realmente, la clase genérica `Modelo` no hace absolutamente nada. Por tanto, sería perfectamente posible definir directamente los modelos específicos de cada tipo de modelado sin necesidad de heredar nada de ella. Sin embargo, el hecho de que todos los modelos específicos deban proveer, como mínimo, de una serie de métodos concretos aconseja el uso del mecanismo de la herencia por dos motivos: en primer lugar, para servir como patrón para la construcción de modelos nuevos; por otro, para proporcionar versiones iniciales a los distintos métodos necesarios.

ramses/mod.py

```
import numpy as np

from util import *

class Modelo:
    """
    Clase raíz de los modelos acústicos. No realiza ninguna misión concreta, sólo
    definir versiones 'inocuas' de los distintos métodos necesarios en los modelos
    concretos que la hereden.
    """

    def __init__(self, pathMod=None):
        """
        Crea un objeto de la clase. Como mínimo, ha de tener un argumento opcional,
        'pathMod', que indica el fichero del que se deben leer los datos para
        crearlo; si este argumento evalúa a 'False', el modelo inicial debe crearse
        desde cero.
        """

        if pathMod:
            self.leeMod(pathMod)
        else:
            pass

    def leeMod(self, pathMod):
        """
        Lee el modelo almacenado en el fichero 'pathMod'.
        """
        pass

    def escrMod(self, pathMod):
        """
        Escribe el modelo en el fichero 'pathMod'.
        """
        pass

    def inicMod(self):
        """
        Inicializa los atributos del modelo de manera que podamos acumular en él la
        información aportada por las señales de entrenamiento.
        """
        pass

    def __add__(self, prm):
        """
        Sobrecarga del operador suma (+} para incorporar al modelo la contribución
        de la trama de señal parametrizada 'prm'
        """
        return self

    def __call__(self, prm):
        """
        Sobrecarga la llamada a función sobre la clase para obtener el score de la
        señal prm según el modelo.
        """
        return np.random.rand()
```

```
    def calcMod(self):
        """
        Recalcula el modelo a partir de la información acumulada durante el
        entrenamiento.
        """
        pass
```

5.1.1. Entrenamiento de los modelos de la clase `Modelo`, función `entrena()`

Se ha de modificar la función `entrena()` para que use los objetos de la clase `Modelo` y sus métodos. Empezamos por el propio prototipo de la función, que incorporará la clase de los objetos como último argumento. Por defecto, su valor será la clase que implementa el modelado según distancia euclídea, `ModEucl`, aún por definir:

entrena.py

```
#! /usr/bin/python3

import tqdm

from util import *
from prm import *
from mar import *
from mod import *

def entrena(dirMod, dirMar, dirPrm, *guiSen, dirModIni=None, ClsMod=Modelo):
    """
    Entrena los modelos acústicos de las unidades encontradas en los ficheros de
    entrenamiento, indicados en el fichero guía 'guiSen' escribiendo el resultado en
    el directorio 'dirMod'.

    Los ficheros de señal parametrizada se leen del directorio 'dirPrm' y el contenido
    fonético se extrae del cuarto campo de la etiqueta LBO de los ficheros de marcas
    ubicados en 'dirMar'. Si se indica el argumento, 'dirModIni', los modelos ...
    ↪ iniciales
    se leen del directorio indicado por él; en caso contrario, se inicializan a los
    valores por defecto. La inicialización por modelos preexistentes puede ser útil
    para implementar algoritmos de entrenamiento iterativos.

    El tipo de modelado se determina a partir de clase de los modelos, que se pasa ...
    ↪ a la
    función mediante el argumento 'ClsMod', por defecto igual a 'Modelo'. La clase
    pasada debe definir los métodos siguientes:

    __init__(self, pathMod=None): Crea el modelo inicial o, si se usa 'pathMod', ...
    ↪ lo lee
    inicMod__(self):        Inicializa los modelos para su entrenamiento
    __add__(self, prm):     Acumula la señal correspondiente a una señal de ...
    ↪ entrenamiento
    calcMod__(self):        Recalcula los modelos a partir de los datos acumulados
```

```
    escrMod(self, pathMod): Escribe el modelo en el fichero 'pathMod'
    """
```

Nótese que, además de incorporar el argumento `ClsMod` para indicar la clase de los objetos modelo, también se ha añadido a `entrena()` la posibilidad de inicializar los modelos a partir de otros preexistentes en el directorio `dirModIni`. Esto se hace así para permitir el uso de modelados que requieren de algoritmos iterativos para entrenar los modelos, como puedan ser los GMM y las redes neuronales.

A continuación, hemos de localizar las distintas partes del código en las que intervienen los modelos de las unidades para adaptarlos al nuevo modelado:

entrena.py

```
modelos = {}
for sen in tqdm.tqdm(leeLis(*guiSen)):
    pathMar = pathName(dirMar, sen, '.mar')
    mod = cogeTrn(pathMar)

    pathPrm = pathName(dirPrm, sen, '.prm')
    prm = leePrm(pathPrm)

    if mod not in modelos:
        modIni = pathName(dirModIni, mod, 'mod') if dirModIni else None
        modelos[mod] = ClsMod(modIni)
        modelos[mod].inicMod()

    modelos[mod] += prm

for mod in modelos:
    modelos[mod].calcMod()
    pathMod = pathName(dirMod, mod, '.mod')
    modelos[mod].escrMod(pathMod)
```

Los cambios con respecto al sistema trivial se pueden resumir en los siguientes, en formato `diff` de UNIX, donde las líneas de color rojo corresponden al código original del sistema trivial y las de color verde al correspondiente a la versión basada en programación orientada a objeto:

```
7a8
> from mod import *
9c10
< def entrena(dirMod, dirMar, dirPrm, *guiSen):
---
> def entrena(dirMod, dirMar, dirPrm, *guiSen, dirModIni=None, ClsMod=Modelo):
24d34
<     numSen = {}
33,37c43,47
<             modelos[mod] = prm
<             numSen[mod] = 1
<         else:
<             modelos[mod] += prm
```

```
<              numSen[mod] += 1
---
>              modIni = pathName(dirModIni, mod, 'mod') if dirModIni else None
>              modelos[mod] = ClsMod(modIni)
>              modelos[mod].inicMod()
>
>          modelos[mod] += prm
40c50
<          modelos[mod] /= numSen[mod]
---
>          modelos[mod].calcMod()
42,44c52
<          chkPathName(pathMod)
<          with open(pathMod, 'wb') as fpMod:
<              np.save(fpMod, modelos[mod])
---
>          modelos[mod].escrMod(pathMod)
```

Invocación de `entrena.py` como script

El paso de la clase a emplear en el modelado acústico se realiza de manera análoga a como se pasa la función de parametrización a `parametriza.py`. En primer lugar, añadimos las opciones `--ClsMod` y `--execPre` al mensaje de ayuda de `entrena.py`:

entrena.py

```
    Sinopsis = f"""
Entrena los modelos acústicos a partir de una base de datos de entrenamiento

Usage:
    {sys.argv[0]} [options] <guiSen>...
    {sys.argv[0]} -h | --help
    {sys.argv[0]} --version

Opciones:
    -p PATH, --dirPrm=PATH  Directorio con las señales parametrizadas [default: .]
    -a PATH, --dirMar=PATH  Directorio con los ficheros de marcas [default: .]
    -m PATH, --dirMod=PATH  Directorio con los modelos generados [default: .]
    -i PATH, --dirModIni=PATH  Directorio con los modelos iniciales
    -x SCRIPT..., --execPre=SCRIPT...  Scripts Python a ejecutar antes del modelado
    -C EXPR, --ClsMod=EXPR  Expresión a evaluar para obtener la clase del modelado

Argumentos:
    <guiSen>   Nombre del fichero guía con los nombres de las señales usadas en el
               entrenamiento. Pueden especificarse tantos ficheros guía como sea
               necesario.

Entrenamiento:
    El programa lee los contenidos fonéticos de los ficheros de marcas y entrena los
    modelos de las unidades fonéticas encontradas en ellos.

Modelado acústico:
    El modelado acústico usa la clase indicada por el argumento de la opción
    '--ClsMod'. Para posibilitar el uso de clases externas o definidas por el usuario,
```

```
    puede indicarse la ejecución de uno o más scripts Python con la opción '--execPre'.
    Si se indica más de un script, éstos deben separarse mediante comas.
"""
```

Así mismo, el procesado de las dos opciones es semejante al empleado en `parametriza.py`:

entrena.py

```
args = docopt(Sinopsis, version=f'{sys.argv[0]}: Ramses v3.4 (2020)')

dirPrm = args['--dirPrm']
dirMar = args['--dirMar']
dirMod = args['--dirMod']
dirModIni = args['--dirModIni']

guiSen = args['<guiSen>']

scripts = args['--execPre']
if scripts:
    for script in scripts.split(','):
        exec(open(script).read())

ClsMod = eval(args['--ClsMod'] if args['--ClsMod'] else 'Modelo')

entrena(dirMod, dirMar, dirPrm, *guiSen, dirModIni=dirModIni, ClsMod=ClsMod)
```

5.1.2. Reconocimiento de las señales, función `reconoce()`

De manera análoga al caso de `entrena()`, las modificaciones en `reconoce()` son bastante triviales. Para darle más emoción, sustituimos la búsqueda del *score* máximo por una versión fuertemente orientada a la programación funcional...

ramses/reconoce.py

```
#! /usr/bin/python3

import tqdm
from functools import reduce

from util import *
from prm import *
from mod import *

def reconoce(dirRec, dirPrm, dirMod, ficLisMod, *guiSen, ClsMod=Modelo):
    """
    Determina la unidad cuyo modelo se ajusta mejor a cada señal a reconocer y escribe
    su nombre en el cuarto campo de una etiqueta LBO de un fichero de marcas ubicado
    en el directorio 'dirRec' y del mismo nombre que la señal, pero con extensión
    ↪  '.rec'.

    Las unidades a reconocer se enumeran en el fichero 'ficLisMod', y sus modelos
    deben estar en el directorio 'dirMod'. Se reconocen las señales indicadas por el
    fichero guía 'guiSen', que deben estar en el directorio 'dirPrm'.
```

```
    El tipo de modelado se determina a partir de clase de los modelos, que se pasa ...
    ↪ a la
    función mediante el argumento 'ClsMod', por defecto igual a 'Modelo'. La clase
    pasada debe definir los métodos siguientes:

    __init__(self, pathMod=None): Lee el modelo indicado por 'pathMod'
    __call__(self, prm):          Calcula la puntuación de la señal 'prm' por el ...
    ↪ modelo
    """

    modelos = {mod: ClsMod(pathName(dirMod, mod, '.mod')) for mod in leeLis(ficLisMod)}

    for sen in tqdm.tqdm(leeLis(*guiSen)):
        pathPrm = pathName(dirPrm, sen, '.prm')
        prm = leePrm(pathPrm)

        scores = {mod: modelos[mod](prm) for mod in modelos}
        rec = reduce(lambda x, y: max(x, y, key=lambda mod: scores[mod]), modelos)

        pathRec =  pathName(dirRec, sen, '.rec')
        chkPathName(pathRec)
        with open(pathRec, 'wt') as fpRec:
            fpRec.write(f'LBO: ,,,{rec}\n')
```

Donde las diferencias entre las dos versiones son las siguientes:

```
3a4
> from functools import reduce
6a8
> from mod import *
8c10
< def reconoce(dirRec, dirPrm, dirMod, ficLisMod, *guiSen):
---
> def reconoce(dirRec, dirPrm, dirMod, ficLisMod, *guiSen, ClsMod=Modelo):
18,23d19
<     modelos = {}
<     for mod in leeLis(ficLisMod):
<         pathMod = pathName(dirMod, mod, '.mod')
<         with open(pathMod, 'rb') as fpMod:
<             modelos[mod] = np.load(fpMod)
24a21,22
>     modelos = {mod: ClsMod(pathName(dirMod, mod, '.mod')) for mod in ...
↪ leeLis(ficLisMod)}
29,34c27,28
<         minDist = np.inf
<         for mod in modelos:
<             dist = sum(abs(prm - modelos[mod]) ** 2)
<             if dist < minDist:
<                 minDist = dist
<                 rec = mod
---
>         scores = {mod: modelos[mod](prm) for mod in modelos}
>         rec = reduce(lambda x, y: max(x, y, key=lambda mod: scores[mod]), modelos)
```

Invocación de `reconoce.py` como script

La introducción del modelado acústico orientado a objeto en `reconoce.py` es casi idéntico a como se hizo en `entrena.py`:

reconoce.py

```
##############################################################################
# Invocación en línea de comandos
##############################################################################

if __name__ == '__main__':
    from docopt import docopt
    import sys

    Sinopsis = f"""
Reconoce una base de datos de señales parametrizadas

Usage:
    {sys.argv[0]} [options] --lisMod=FILE <guiSen>...
    {sys.argv[0]} -h | --help
    {sys.argv[0]} --version

Opciones:
    -r PATH, --dirRec=PATH  Directorio con los ficheros del resultado [default: .]
    -p PATH, --dirPrm=PATH  Directorio con las señales parametrizadas [default: .]
    -m PATH, --dirMod=PATH  Directorio con los modelos acústicos [default: .]
    -l FILE, --lisMod=FILE  Fichero con la lista de unidades a reconocer
    -x SCRIPT..., --execPre=SCRIPT...  Scripts Python a ejecutar antes del modelado
    -C EXPR, --ClsMod=EXPR  Expresión a evaluar para obtener la clase del modelado

Argumentos:
    <guiSen>  Nombre del fichero guía con los nombres de las señales a reconocer.
              Pueden especificarse tantos ficheros guía como sea necesario.

Modelado acústico:
    El modelado acústico usa la clase indicada por el argumento de la opción
    '--ClsMod'. Para posibilitar el uso de clases externas o definidas por el usuario,
    puede indicarse la ejecución de uno o más scripts Python con la opción '--execPre'.
    Si se indica más de un script, éstos deben separarse mediante comas.
"""

    args = docopt(Sinopsis, version=f'{sys.argv[0]}: Ramses v3.4 (2020)')

    dirRec = args['--dirRec']
    dirPrm = args['--dirPrm']
    dirMod = args['--dirMod']
    ficLisMod = args['--lisMod']

    guiSen = args['<guiSen>']

    scripts = args['--execPre']
    if scripts:
        for script in scripts.split(','):
            exec(open(script).read())

    ClsMod = eval(args['--ClsMod'] if args['--ClsMod'] else 'Modelo')
```

```
    reconoce(dirRec, dirPrm, dirMod, ficLisMod, *guiSen, ClsMod=ClsMod)
```

5.1.3. Modelado trivial usando distancia euclídea; clase `ModEucl`

Una vez incorporado el modelado basado en la clase `Modelo` en `entrena()` y `reconoce()`, la introducción de los distintos tipos de modelado consiste, únicamente, en definir los métodos de la clase correspondiente. En principio, cada tipo de modelado estará definido en un fichero diferente. En el caso del modelado euclídeo lo haremos en `euclideo.py`.

En el caso modelado mediante distancia euclídea, el atributo principal es un vector `numpy` con la `media` de las señales que han participado en el entrenamiento.

euclideo.py

```
import numpy as np

from util import *
from mod import Modelo

class ModEucl(Modelo):
    """
    Clase propia del modelado usando distancia euclídea. La puntuación otorgada a
    cada señal es su distancia euclídea al representante del modelo cambiada de
    signo.

    Usando el criterio de mínima varianza, el representante óptimo de la clase es
    la media aritmética de sus señales de entrenamiento, que se almacena en el
    atributo 'self.media'.
    """
```

A continuación, definimos los distintos métodos que definen el comportamiento del modelo. La versión estándar del constructor, que devuelve un modelo vacío o, en el caso de usarse el argumento `pathMod`, lo lee de fichero, es perfectamente válida para este caso. Por tanto, no es necesario definir uno específico.

Funciones `leeMod(self, pathMod)` y `escrMod(self, pathMod)`

El método `leeMod(self, pathMod)` lee el modelo del fichero `pathMod` usando la función `load()` de `numpy`. Es un *copia-y-pega* de las líneas correspondientes de la versión original de la función `reconoce()`.

euclideo.py

```
def leeMod(self, pathMod):
    with open(pathMod, 'rb') as fpMod:
        self.media = np.load(fpMod)
```

El método `escrMod(self, pathMod)` escribe el modelo en el fichero `pathMod` usando la función `save()` de `numpy`. Es un *copia-y-pega* de las líneas correspondientes de la versión original de la función `entrena()`.

euclideo.py

```
def escrMod(self, pathMod):
    chkPathName(pathMod)
    with open(pathMod, 'wb') as fpMod:
        np.save(fpMod, self.media)
```

Funciones `inicMod(self)`, `__add__(self, prm)` y `calcMod(self)`

El entrenamiento de los modelos se basa en el uso de los tres métodos `inicMod()`, `__add__()` y `calcMod()`. El objetivo conjunto de las tres es almacenar la media aritmética de las señales de entrenamiento en el atributo `media`. Para ello, `inicMod()` inicializa a cero los atributos `sumPrm`, que almacena la suma de las señales, y `numPrm`, que almacena su número:

euclideo.py

```
def inicMod(self):
    self.sumPrm = None
    self.numSen = 0
```

La suma de cada una de las señales, `prm`, se realiza con la sobrecarga del operador suma (+). Como en el caso de la versión original de `entrena()`, debemos tener en cuenta si `sumPrm` ya está inicializado o no:

euclideo.py

```
def __add__(self, prm):
    if self.sumPrm is None:
        self.sumPrm = prm
    else:
        self.sumPrm += prm

    self.numSen += 1

    return self
```

Finalmente, obtenemos el modelo dividiendo la suma de señales por su número:

euclideo.py

```
def calcMod(self):
    self.media = self.sumPrm / self.numSen
```

Función `__call__(self, prm)`

El cálculo del *score* de una señal por el modelo se realiza sobrecargando la invocación como función del objeto con el método *mágico* `__call__()`. Es semejante al caso de la versión original de `reconoce()`, pero teniendo en cuenta que ahora usamos puntuaciones y no distancias. Por tanto, el valor devuelto por el método es la distancia cambiada de signo:

euclideo.py

```
def __call__(self, prm):
    return -sum(np.abs(self.media - prm) ** 2)
```

5.1.4. Incorporación del modelado orientado a objeto al script `todo.sh`

Como en el caso de la incorporación de la parametrización basada en programación funcional a `todo.sh`, primero escribimos, desde el propio script, el fichero de comandos previos. En este caso, se guarda el fichero en el mismo directorio que los modelos.

Definimos, para probar el sistema, una primera clase que no hace nada, sólo heredar los atributos de la previamente definida `ModEucl`, pero que renombramos como `Euclidea`. Introducimos la definición de esta clase en el fichero `$DIR_MOD/$CLS_MOD.py`, que se genera desde el propio script. También debemos ejecutar, como script previo, el fichero `euclideo.py`, donde se define la clase `ModEucl`, que implementa el modelado mediante distancia euclídea.

todo.sh

```
# Definición de la clase del modelado acústico

CLS_MOD=Euclidea
EXEC_PRE=$DIR_MOD/$CLS_MOD.py
[ -d $(dirname $EXEC_PRE) ] || mkdir -p $(dirname $EXEC_PRE)

echo "############################################################" | tee $EXEC_PRE
echo "#  Modelado por distancia euclídea                         #" | tee -a $EXEC_PRE
echo "############################################################" | tee -a $EXEC_PRE
echo                                                               | tee -a $EXEC_PRE
echo "class $CLS_MOD(ModEucl):"                                    | tee -a $EXEC_PRE
echo "    pass"                                                    | tee -a $EXEC_PRE
```

Modificamos la invocación a `entrena.py` para incorporar el modelado acústico definido por la clase `Euclidea`:

todo.sh

```
dirModIni=""
execPre="-x $EXEC_PRE,euclideo.py"
ClsMod="-C $CLS_MOD"

EXEC="entrena.py $dirPrm $dirMar $dirMod $dirModIni $execPre $ClsMod $GUI_ENT"
$ENT && { echo $EXEC; $EXEC || exit 1; }
```

Nótese que la opción `'--dirModIni'` se deja en blanco para que el script use el valor `None`, indicando que no se usan modelos iniciales.

De manera análoga, incorporamos el modelado definido por clases orientadas a objeto a la invocación de `reconoce.py`:

todo.sh

```
execPre="-x $EXEC_PRE"
ClsMod="-C $CLS_MOD,euclideo.py"

EXEC="reconoce.py $dirRec $dirPrm $dirMod $lisMod $execPre $ClsMod $GUI_DEV"
$REC && { echo $EXEC; $EXEC || exit 1; }
```

Ejecución de `todo.sh` con el modelado proporcionado por la clase Euclidea

Comprobamos el correcto funcionamiento de la clase `ModEucl` y del script `todo.sh` repitiendo una de sus ejecuciones:

```
usuario:~/TecParla$ todo.sh
ALBINO
/home/albino/TecParla/2021 Outono
Sat Dec 18 22:42:49 CET 2021
#############################################################
# Periodograma logarítmico                                  #
#############################################################

import numpy as np

eps = 10
def periodograma(x):
    return 10 * np.log10(eps + np.abs(np.fft.fft(x)) ** 2)
parametriza.py -s ./Sen -p ./Prm/Dos -x ./Prm/Dos/periodograma.py -f periodograma      ...
 ↪  ./Gui/train.gui ./Gui/devel.gui
100%|| 4000/4000 [00:02<00:00, 1954.95it/s]
#############################################################
# Modelado por distancia euclídea                           #
#############################################################

class Euclidea(ModEucl):
    pass
entrena.py -p ./Prm/Dos -a ./Sen -m ./Mod/Dos -x ./Mod/Dos/periodograma.py -C          ...
 ↪  Euclidea ./Gui/train.gui
100%|| 2000/2000 [00:00<00:00, 2391.33it/s]
reconoce.py -r ./Rec/Dos -p ./Prm/Dos -m ./Mod/Dos -l ./Lis/vocales.lis -x             ...
 ↪  ./Mod/Dos/periodograma.py -C Euclidea ./Gui/devel.gui
100%|| 2000/2000 [00:01<00:00, 1154.46it/s]
evalua.py -r ./Rec/Dos -a ./Sen ./Gui/devel.gui
100%|| 2000/2000 [00:00<00:00, 11056.23it/s]
        a       e       i       o       u
a       350     5       0       39      6
e       2       333     24      6       35
i       0       20      341     0       39
o       26      5       0       324     45
u       1       0       4       49      346

Exac = 84.70%
```

```
Sat Dec 18 22:42:56 CET 2021
s'acabao
```

5.2. Modelado bayesiano

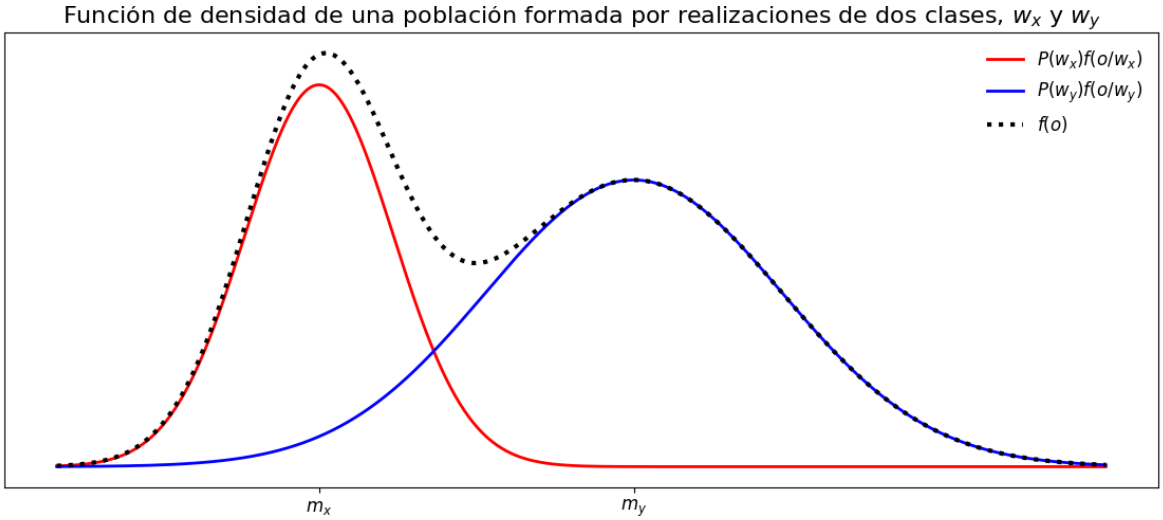
En las secciones precedentes se ha visto que, usando una parametrización adecuada, un modelado basado en la distancia euclídea puede ser suficiente para alcanzar un resultado muy superior al aleatorio. No obstante, una exactitud de menos del 90 % en la tarea de reconocimiento de las cinco vocales del castellano sigue pareciendo un resultado muy pobre.

El porqué de este mal resultado puede explicarse si consideramos cómo se distribuyen las realizaciones de las distintas unidades a reconocer. Idealmente, la parametrización debería representar la señal en un espacio vectorial en el que cada una estuviera separada de las otras. En general, este objetivo es inalcanzable, y tendremos zonas del espacio exclusivas de cada unidad, pero también zonas en las que aparecerá solapamiento entre ellas.

Por ejemplo, supóngase que el sistema debe reconocer un vocabulario formado por dos palabras, w_x y w_y. Cada una de estas palabras tiene una cierta probabilidad de aparecer en el reconocimiento, $P(w_x)$ y $P(w_y)$. Las realizaciones a reconocer tienen una función de densidad igual a:

$$f(o) = P(w_x)f(o/w_x) + P(w_y)f(o/w_y) \tag{5.1}$$

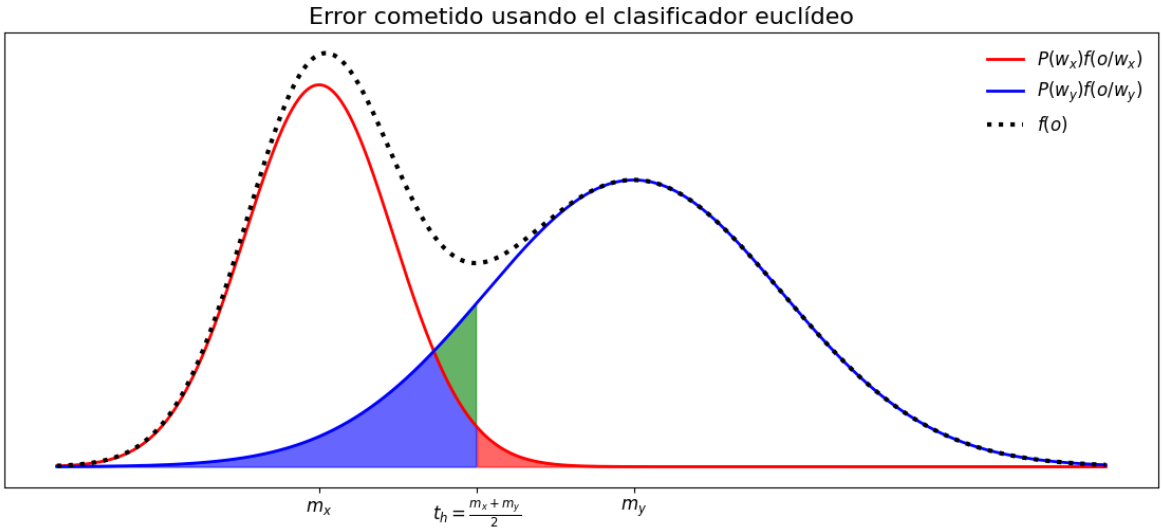
Donde $f(o/w_x)$ y (o/w_y) son las funciones de densidad de las observaciones correspondientes a cada una de las palabras. Las medias de las realizaciones de w_x y w_y son m_x y m_y, respectivamente:



En la figura se puede observar que, en general, las contribuciones de la palabra w_x tienen valores más bajos que las de w_y. Así pues, un posible esquema de reconocimiento sería

asignar los valores más bajos de o a w_x y los más altos a w_y. Sin embargo, existe un rango de valores en los que las distribuciones de ambas palabras se solapan. Si todo nuestro conocimiento de la observación es su valor, será imposible decidir con seguridad de qué palabra se trata.

En el clasificador basado en distancia euclídea, la regla de decisión consiste en asignar la observación a la clase de media más próxima. Por tanto, todas las realizaciones tales que $|o - m_x|^2 < |o - m_y|^2$ se asignarán a la palabra w_x y viceversa. En una dimensión, esto es lo mismo que ubicar un umbral t_h en el punto medio entre m_x y m_y, de tal modo que asignamos a w_x las realizaciones que cumplen $o < t_h$ y a w_y las que $o > t_h$:



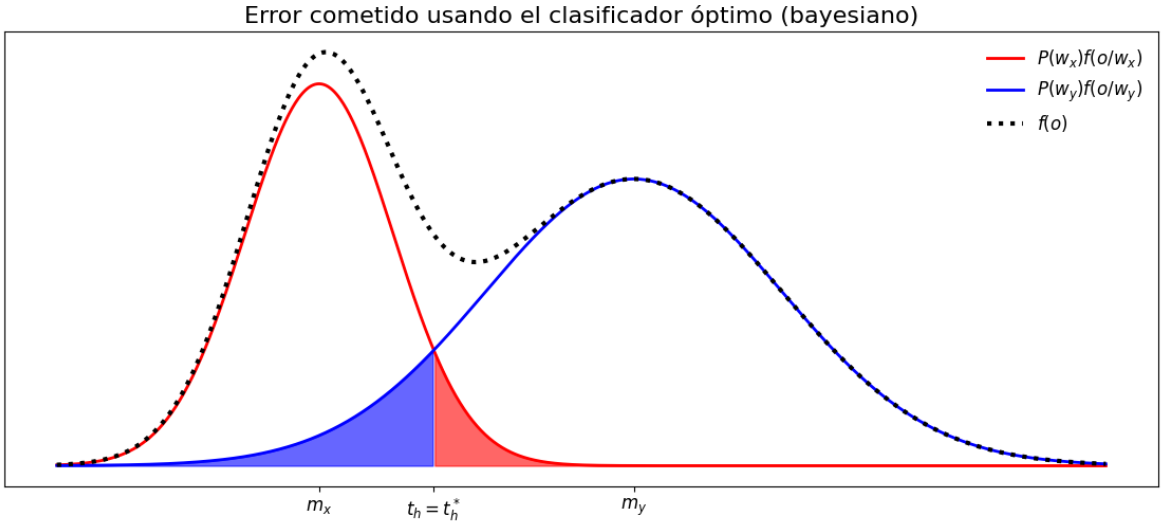
En la figura anterior distinguimos tres áreas que representan observaciones reconocidas incorrectamente (recuérdese que, en una función de densidad, el área representa la frecuencia de las observaciones):

Roja: Realizaciones de w_x, pero que, como $o > t_h$, son asignadas a w_y.

Azul: Realizaciones de w_y, pero que, como $o < t_h$, son asignadas a w_x. En esta área tenemos tantas realizaciones de w_x como de w_y. Por tanto, si en vez de asignarlas a w_x las asignáramos a w_y, el tipo de error variaría, pero su cantidad no.

Verde: También son realizaciones de w_y que se asignan a w_x, pero, en este caso, esa área sólo representa realizaciones de w_y. Por tanto, si se asignasen todas a w_y, eliminaríamos estos errores sin introducir errores nuevos.

A la vista de esta distribución de los errores en el reconocimiento de w_x y w_y, es evidente que la elección del punto medio entre m_x y m_y como umbral de decisión es subóptima. La elección óptima sería usar el umbral de decisión que elimina los errores de la zona verde. Esto es, aquél para el que $P(w_x)f(t_h/w_x) = P(w_y)f(t_h/w_y)$. Dicho de otra manera, seleccionar la clase w^* para la que $P(w^*)f(\mathbf{x}/w^*)$ es más grande:



El esquema de reconocimiento construido de este modo minimiza el número esperado de errores. Por este motivo se le denomina *clasificador de mínimo riesgo de Bayes*, o, más escuetamente, clasificador bayesiano.

5.2.1. El teorema de Bayes

En esencia, la idea subyacente al clasificador bayesiano consiste en abandonar el criterio basado en distancia para considerar otro basado en probabilidades. La cuestión puede plantearse del modo siguiente:

Dada una señal $o(t)$, ¿cuál es la unidad más probable del vocabulario a reconocer?

Planteada matemáticamente, tenemos: siendo o la señal a reconocer y w_i las M palabras posibles, el resultado óptimo del reconocimiento es $w^* = \text{argmax}\{P(w_i/o)\}$.

Las probabilidades $P(w_i/o)$ son conocidas como *a posteriori*. Su significado es la probabilidad de un evento condicionada a una evidencia. En el problema del reconocimiento, esta probabilidad es justamente lo que buscamos: la probabilidad de cada palabra del vocabulario (evento desconocido), dada la señal observada (evidencia).

Desgraciadamente, en muchos casos estas probabilidades son difíciles de estimar directamente. En muchas de estas situaciones, el teorema de Bayes proporciona un mecanismo para su cálculo a partir de otras probabilidades más fáciles de estimar, o cuya aportación al resultado final resulta irrelevante.

Dados sus sucesos independientes A y B, el teorema de Bayes establece:

$$P(A/B) = \frac{P(B/A)P(A)}{P(B)} \tag{5.2}$$

El significado de las distintas variables y probabilidades que participan en esta fórmula es el siguiente:

A y B: A es el evento cuya probabilidad queremos calcular sabiendo que se ha cumplido el evento B, denominado *evidencia*.

En el sistema de reconocimiento del habla, A es una palabra del vocabulario y B es la señal temporal que se quiere reconocer.

$P(A/B)$: Probabilidad de A condicionada a B, también conocida como probabilidad *a posteriori*.

Objetivo del sistema de reconocimiento, es la probabilidad de que la señal observada B sea una realización de la palabra a reconocer, A.

$P(B/A)$: Probabilidad de B condicionada a A, también conocida como *verosimilitud* de B dado A.

En el sistema de reconocimiento sería la probabilidad con la que los eventos A resultan en señales B.

$P(A)$: Probabilidad del evento A, también conocida como probabilidad *a priori*.

En un sistema de reconocimiento es la probabilidad con la que aparece cada palabra del vocabulario. En este contexto, suele denominarse *modelo del lenguaje*.

$P(B)$: Probabilidad de la evidencia B, llamada simplemente *evidencia*.

En el reconocimiento del habla es irrelevante, ya que representa lo probable que es la señal observada, pero esta contribución es la misma para todas las palabras del vocabulario, con lo que no afecta a cuál de ellas será más o menos probable.

Evidentemente, qué se considera *evidencia* y qué *evento* depende de la situación, y ambos pueden intercambiarse sin afectar a la validez de la ecuación (5.2).

Ejemplo de aplicación del teorema de Bayes: dados de rol.

En los juegos de rol, y similares, es habitual el uso de dados con un número de caras distinto de las seis habituales. En concreto, es corriente jugar con un número variable de dados de cuatro, seis, ocho, diez, doce y veinte caras.

Supongamos que se realiza una tirada con un único dado de éstos tomado al azar de una caja en la que hay un cierto número de ellos. Cada cara está marcada con un número entero distinto comprendido entre uno y el número de caras del dado.

En la caja tenemos los dados siguientes:

Dado	Cantidad	Caras
$\boldsymbol{D_4}$	4 dados	C_1, C_2, C_3 y C_4
$\boldsymbol{D_6}$	3 dados	C_1, C_2, C_3, C_4, C_5 y C_6
$\boldsymbol{D_8}$	2 dados	C_1, C_2, C_3, C_4, C_5, C_6, C_7 y C_8
$\boldsymbol{D_{12}}$	1 dado	C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9, C_{10}, C_{11} y C_{12}

Lanzamos un dado escogido al azar de los contenidos en la caja y conociendo el resultado, queremos saber cuál es la probabilidad de que se hubiera lanzado un dado de cada tipo.

En este problema, el evento cuya probabilidad queremos evaluar es el tipo de dado, $D4$, $D6$, $D8$ y $D12$; mientras que la evidencia observada es la cara $C_1 \dots C_{12}$. El objetivo es calcular la probabilidad de cada dado, conociendo el resultado del lanzamiento: $P(D_i/C_j)$.

Verosimilitud de las carcas para cada dado; $\boldsymbol{P(C_j/D_i)}$:

Suponiendo que los dados no están trucados, la probabilidad de cada cara es la misma. Por tanto, es igual a uno dividido por el número de caras:

Dado	$\boldsymbol{P(C_j/D_i)}$	
$\boldsymbol{D_4}$	$P(C_1/D_4) = \cdots = P(C_4/D_4) = 1/4$	$P(C_5/D_4) = \cdots = P(C_{12}/D_4) = 0$
$\boldsymbol{D_6}$	$P(C_1/D_6) = \cdots = P(C_6/D_6) = 1/6$	$P(C_7/D_6) = \cdots = P(C_{12}/D_6) = 0$
$\boldsymbol{D_8}$	$P(C_1/D_8) = \cdots = P(C_8/D_8) = 1/8$	$P(C_9/D_8) = \cdots = P(C_{12}/D_8) = 0$
$\boldsymbol{D_{12}}$	$P(C_1/D_{12}) = \cdots = P(C_{12}/D_{12}) = 1/12$	

Probabilidades *a priori* de los dados; $\boldsymbol{P(D_i)}$

Dentro de la bolsa hay un total de $4+3+2+1 = 10$ dados. Por tanto, la probabilidad a priori de cada tipo de dado es el número de ellos que hay en la bolsa dividido por diez:

$$\begin{aligned} P(D_4) &= 4/10 \\ P(D_6) &= 3/10 \\ P(D_8) &= 2/10 \\ P(D_{12}) &= 1/10 \end{aligned}$$

Probabilidades de las carcas (evidencias); $\boldsymbol{P(C_i)}$

Cada cara posible puede obtenerse lanzando dados distintos. Como los cuatro tipos de dados son sucesos disjuntos (o sale un dado de cuatro caras o sale uno de seis, o...), la probabilidad de cada cara será la suma de las probabilidades de sacarla con cada dado:

$$P(C_i) = \sum_j P(D_j)P(C_i/D_j) \tag{5.3}$$

$$
\begin{aligned}
P(C_1) = P(C_2) = P(C_3) = P(C_4) &= \frac{4}{10}\cdot\frac{1}{4} + \frac{3}{10}\cdot\frac{1}{6} + \frac{2}{10}\cdot\frac{1}{8} + \frac{1}{10}\cdot\frac{1}{12} = \frac{22}{120} \\
P(C_5) = P(C_6) &= \frac{4}{10}\cdot\frac{0}{4} + \frac{3}{10}\cdot\frac{1}{6} + \frac{2}{10}\cdot\frac{1}{8} + \frac{1}{10}\cdot\frac{1}{12} = \frac{10}{120} \\
P(C_7) = P(C_8) &= \frac{4}{10}\cdot\frac{0}{4} + \frac{3}{10}\cdot\frac{0}{6} + \frac{2}{10}\cdot\frac{1}{8} + \frac{1}{10}\cdot\frac{1}{12} = \frac{4}{120} \\
P(C_9) = P(C_{10}) = P(C_{11}) = P(C_{12}) &= \frac{4}{10}\cdot\frac{0}{4} + \frac{3}{10}\cdot\frac{0}{6} + \frac{2}{10}\cdot\frac{0}{8} + \frac{1}{10}\cdot\frac{1}{12} = \frac{1}{120}
\end{aligned}
$$

Probabilidades *a posteriori* cuando la cara observada es C_1 (o C_2, C_3 o C_4).

Usando las probabilidades de los párrafos anteriores, y aplicando el teorema de Bayes, calculamos cuál es la probabilidad de cada dado:

$$
\begin{aligned}
P(D_4/C_1) &= \frac{P(C_1/D_4)P(D_4)}{P(C_1)} = \frac{1/4\cdot 4/10}{22/120} = \frac{12}{22} \\
P(D_6/C_1) &= \frac{P(C_1/D_6)P(D_6)}{P(C_1)} = \frac{1/6\cdot 3/10}{22/120} = \frac{6}{22} \\
P(D_8/C_1) &= \frac{P(C_1/D_8)P(D_8)}{P(C_1)} = \frac{1/8\cdot 2/10}{22/120} = \frac{3}{22} \\
P(D_{12}/C_1) &= \frac{P(C_1/D_{12})P(D_{12})}{P(C_1)} = \frac{1/12\cdot 1/10}{22/120} = \frac{1}{22}
\end{aligned}
$$

Estas mismas probabilidades son las que se obtendrían si la cara observada hubiera sido C_2, C_3 o C_4.

Fijémonos en que la probabilidad conjunta de los cuatro dados, el suceso unión, de probabilidad la suma de cada uno, es igual a uno; es decir, es el suceso seguro (el dado ha de ser uno de los cuatro posibles).

En una tarea de reconocimiento de dados, si obtenemos una cara C_1, la mejor hipótesis sería el dado de cuatro caras, con más del 50 % de probabilidad. Además, es importante remarcar que la probabilidad de obtener la cara C_1, $P(C_1) = 22/120$, aparece dividiendo en todos los términos, así que su valor no afecta a la decisión del reconocimiento.

Probabilidades *a posteriori* cuando la cara observada es C_9 (o C_{10}, C_{11} o C_{12}).

Si la cara observada es una de las que son exclusivas del dado de doce caras, las probabilidades son:

$$
\begin{aligned}
P(D_4/C_9) &= \frac{P(C_9/D_4)P(D_4)}{P(C_9)} = \frac{0{\cdot}4/10}{1/120} &= 0\\
P(D_6/C_9) &= \frac{P(C_9/D_6)P(D_6)}{P(C_9)} = \frac{0{\cdot}3/10}{1/120} &= 0\\
P(D_8/C_9) &= \frac{P(C_9/D_8)P(D_8)}{P(C_9)} = \frac{0{\cdot}2/10}{1/120} &= 0\\
P(D_{12}/C_9) &= \frac{P(C_9/D_{12})P(D_{12})}{P(C_9)} = \frac{1/12{\cdot}1/10}{1/120} &= 1
\end{aligned}
$$

Como era de esperar, es imposible (probabilidad cero) que el resultado C_9 hubiera sido obtenido con cualquier dado distinto del de doce caras. Por tanto, que D_{12} es el suceso seguro (probabilidad uno).

5.2.2. Clasificadores bayesianos y el criterio de máxima verosimilitud

El clasificador bayesiano es aquél cuya regla de decisión consiste en seleccionar como resultado del reconocimiento la clase cuya probabilidad *a posteriori* es máxima:

Para un sistema de reconocimiento con $i = 1 \ldots N$ clases, w_i, esto es equivalente a:

$$w^* = \text{argmax}_i \{P(w_i/o)\} = \text{argmax}_i \left\{ \frac{P(o/w_i)P(w_i)}{P(o)} \right\} \tag{5.4}$$

La aplicación práctica de este criterio presenta un problema: aunque el cálculo de $P(w_i/o)$ puede simplificarse mucho a partir del conocimiento de $P(o/w_i)$ y $P(w_i)$[2], sigue pendiente el cómo calcular estas dos probabilidades.

La probabilidad de las palabras del vocabulario, $P(w_i)$, es lo que hemos denominado *modelo del lenguaje*, y existen distintas técnicas para su estimación, como las gramáticas de estados finitos, los n-gramas, etc. En la tarea tratada en este curso, el reconocimiento de las cinco vocales del castellano, cada palabra del vocabulario tiene la misma probabilidad, y su cálculo puede obviarse completamente porque no afectará a la decisión tomada en el reconocimiento.

Nos queda la probabilidad $P(o/w_i)$, que es la de que una realización de la palabra w_i tome el valor o. En general, la observación será una variable vectorial de carácter continuo, con lo que, en lugar de manejar probabilidades, usaremos funciones de densidad, $f(o/w_i)$. El cálculo de esta función de densidad suele ser inabordable, así que será necesario echar mano de alguna simplificación.

El modo más habitual de abordar la estimación de $f(o/w_i)$ es suponer que su estadística responde a alguna forma sencilla (gaussiana, laplaciana, etc.) y calcular sus parámetros a partir de las realizaciones de entrenamiento.

[2]En el cálculo de $P(w_i/o)$ sería necesario usar también $P(o)$; pero, como se ha visto, en problemas de clasificación, esta probabilidad es irrelevante

Siendo λ_i el modelo de la palabra w_i, hacemos la aproximación:

$$f(o/w_i) \approx f(o/\lambda_i) \tag{5.5}$$

Evidentemente, la calidad del sistema de reconocimiento dependerá de la bondad de los modelos λ_i y de nuestra capacidad de estimar correctamente sus parámetros, lo que se conoce como *entrenamiento*.

Entrenamiento de máxima verosimilitud

Queda un problema adicional: ¿cómo estimamos los parámetros de λ_i? El criterio de Bayes nos dice que lo hemos de hacer de manera que $f(o/\lambda_i)$ se parezca lo máximo posible a $f(o/w_i)$. Pero, ¿cómo medimos ese parecido? y, sobre todo, ¿cómo lo maximizamos?

Con los modelos más sencillos, la respuesta a la segunda de estas preguntas es igualmente sencilla. Por ejemplo, en un modelo gaussiano, caracterizado por sus parámetros μ y Σ, el máximo parecido se obtiene cuando hacemos μ igual a la media de la población de entrenamiento y Σ igual a su matriz de covarianza.

Con modelos más complicados, como los de mezcla de gaussianas, la solución es igualmente complicada. Sin embargo, podemos plantear el problema de un modo alternativo: en lugar de buscar una buena aproximación de la función de densidad, usaremos una función de la señal a reconocer que represente una *puntuación* de lo bien que el modelo representa las señales de entrenamiento, $\mathcal{S}_i(o)$. En este planteamiento, el objetivo del aprendizaje pasa a ser la maximización de esta puntuación; un objetivo mucho más concreto y tangible que el establecido por el teorema de Bayes.

Sin pérdida de generalidad, imponemos las restricciones siguientes, necesarias para garantizar la convergencia de la solución:

$$\begin{aligned} \mathcal{S}_i(o) &\geq 0 && (5.6)\\ \int \mathcal{S}_i(o)\,\mathrm{d}o &= 1 && (5.7) \end{aligned}$$

Definimos como objetivo del entrenamiento la maximización del valor esperado del logaritmo de la función de puntuación para toda la población de entrenamiento:

$$\mathcal{L}(O, \lambda_i) = E\left\{\log\left(\mathcal{S}_i(o)\right)\right\} \tag{5.8}$$

En estas condiciones, el valor máximo de $\mathcal{L}(O, \lambda_i)$ se alcanza cuando $\mathcal{S}_i(o)$ es igual a la función de densidad de la población:

$$\begin{aligned} E\left\{\log\left(\mathcal{S}_i(o)\right)\right\} &\leq E\left\{\log\left(f_o(o/w_i)\right)\right\} && (5.9)\\ \mathcal{S}_i(o)^* &= f_o(o/w_i) && (5.10) \end{aligned}$$

Es decir, si conseguimos alcanzar el valor máximo de $E\{\log(\mathcal{S}_i(o))\}$, el sistema cumplirá el criterio de mínimo riesgo de Bayes. Por tanto, dado un modelo de las señales, podemos plantear la estimación de sus parámetros óptimos como un problema de maximización de esta función. Dado que el óptimo se alcanza cuando $\mathcal{S}_i(o)^* = f_o(o/w_i)$ y $f_o(o/w_i)$ es la denominada *verosimilitud* de la palabra w_i dada la señal o, a este método de entrenamiento se le denomina de máxima verosimilitud, y a la función optimizada, $E\{\log(\mathcal{S}_i(o))\}$, *verosimilitud* o, con más propiedad, *log-verosimilitud*.

La ventaja de este planteamiento consiste en que podemos maximizar la verosimilitud usando distintas técnicas de optimización; tales como la búsqueda de gradiente, la *back-propagation*, el algoritmo EM, etc. El hecho de alcanzar un valor máximo de la verosimilitud no implica necesariamente que se verifique el criterio de mínimo de Bayes. Para ello es necesario que el modelo propuesto sea el correcto (o, al menos, pueda converger al mismo) y que dispongamos de una población de entrenamiento apropiada. Sin embargo, parece plausible que cuanto mayor sea su valo, menor será el riesgo de error de clasificación[3].

5.2.3. Modelado gaussiano con matriz de covarianza diagonal

Uno de los modelados acústicos más sencillos es el gaussiano, consistente en suponer que las realizaciones de las distintas clases presentan una función de densidad normal, completamente determinada a partir de su media, μ_i, y su matriz de covarianza, Σ_i:

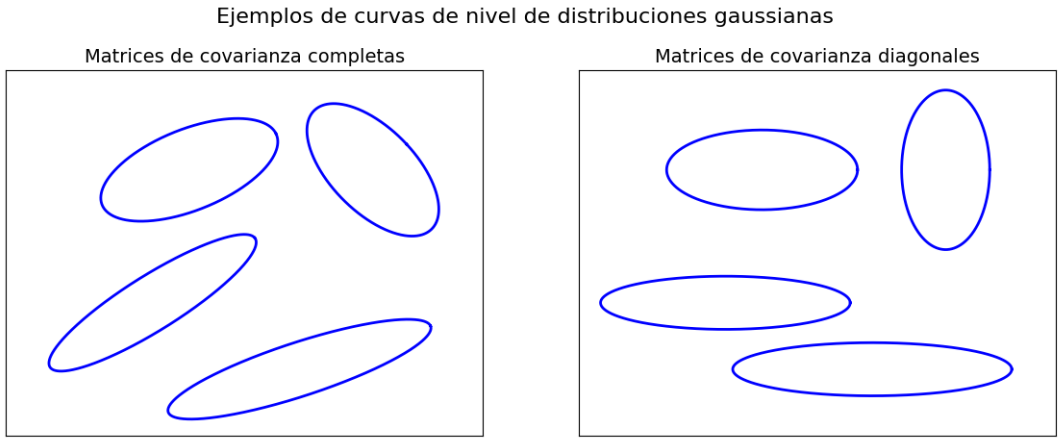
$$f(o/\lambda_i) = \mathcal{N}(o, \mu_i, \Sigma_i) = \frac{\exp\left(-\frac{1}{2}(o-\mu_i)^T\Sigma^{-1}(o-\mu_i)\right)}{\sqrt{(2\pi)^K \det\Sigma_i}} \tag{5.11}$$

Donde K es la dimensión del espacio vectorial y $\det\Sigma_i$ es el determinante de la matriz de covarianza.

Es muy habitual, e incluso necesario cuando la dimensión de los vectores K es elevada, usar una aproximación de matriz de covarianza diagonal. De este modo, el número de parámetros se reduce del orden de K^2 a sólo K. La aproximación es equivalente a suponer que las distintas componentes del vector están incorreladas.

A efectos prácticos, esta incorrelación se traduce en que la función de densidad está orientada respecto a los ejes. Podemos visualizar esta incorrelación usando un espacio vectorial de dimensión $K = 2$ y dibujando alguna curva de nivel de la función de densidad. En general, para una función de densidad gaussiana, estas curvas de nivel tienen la forma de elipses; en el caso de tener matriz de covarianza diagonal, sus dos ejes están orientados según los ejes cartesianos:

[3] De hecho, existen otros algoritmos de entrenamiento que consideran las limitaciones del modelo y del material de entrenamiento, y abordan el problema de minimizar directamente la probabilidad de error. Por ejemplo, el entrenamiento de mínimo error de clasificación (MCE), el de máxima información mutua (MMIE), etc.



Una implicación de que el modelo utilizado suponga incorrelación es que la parametrización, además de los objetivos ya comentados de extraer la información fonética de la señal, eliminando el máximo de información irrelevante o redundante, también deberá procurar que los vectores obtenidos cumplan esta incorrelación.

Siendo $\sigma_i[k]$ la desviación típica de la componente k-ésima de la i-ésima gaussiana, la ecuación (5.11) se convierte, suponiendo matriz de covarianza diagonal, en:

$$f(o/\lambda_i) = \frac{\exp\left(-\frac{1}{2}\sum_k \left(\frac{o[k]-\mu_i[k]}{\sigma_i[k]}\right)^2\right)}{\sqrt{(2\pi)^K \prod_k \sigma_i[k]}} \tag{5.12}$$

5.3. Implementación del modelado gaussiano en `Ramses`

5.3.1. Clase `multivariate_normal` de `scipy.stats`

La implementación en Python del modelado gaussiano se basará en la clase `multivariate_normal` de la biblioteca `scipy.stats`.

El constructor de la clase es:

```
multivariate_normal(mean=None, cov=1, allow_singular=False, seed=None)
```

Donde:

`mean`: Media de la distribución gaussiana. Si `mean=None`, valor por defecto, se usa un vector de ceros.

`cov`: Matriz de covarianza de la distribución. Puede ser de tres tipos:

Escalar: Por ejemplo, su valor por defecto `cov=1`. Usa una matriz de covarianza diagonal con todos sus valores iguales a `cov`. Es decir, usa simetría esférica.

Vector: Si `cov` es un vector de dimensión K, la matriz de covarianza es diagonal, donde `cov` indica el valor de la varianza para cada componente, $\sigma^2[k]$.

Matriz: Si `cov` es una matriz, debe ser cuadrada de dimensiones $K \times K$. Se supone que la matriz es simétrica y definida positiva. El resultado es impredecible si no es así.

`allow_singular`:
Determina si la matriz de covarianza puede ser singular o no. Si `allow_singular=True` y la matriz de covarianza es singular, el resultado es impredecible, pero algunos métodos de la clase darán resultados razonables (y otros no).

`seed`: Semilla del generador de números aleatorios usado en algunos de los métodos de la clase.

La clase `multivariate_normal` proporciona distintos métodos que permiten realizar cosas como la generación de variables aleatorias gaussianas, el cálculo de la entropía de una distribución normal, etc. En la aplicación de la estimación de la función de densidad que se usará en el sistema de reconocimiento sólo estamos interesados en el método `logpdf()`, que devuelve el valor del logaritmo de la función de densidad para un punto determinado del espacio, `x`.

El método `logpdf()` admite dos modos de empleo, en función de si se invoca como método de la clase o de un objeto de la misma. En el primer caso, la invocación es semejante al constructor de la clase, salvo que toma un primer argumento posicional con el valor del punto `x`. En este modo, por tanto, debemos especificar la media y covarianza en cada invocación, o usar sus valores por defecto.

El segundo modo de empleo de `logpdf()` es como método de un objeto previamente construido con su propia media y covarianza. Al realizar la invocación sólo debemos pasarle el valor del punto en el que se calcula la función de densidad. La ventaja de usar este modo de empleo es que hay distintas operaciones, como el cálculo del determinante de la matriz de covarianza, que sólo se realizan una vez, con el consiguiente aumento de la eficiencia computacional.

5.3.2. Modelos acústicos gaussianos con matriz de covarianza diagonal; clase `ModGauss`

Gestionaremos el entrenamiento y reconocimiento usando modelos gaussianos de matriz de covarianza diagonal con la clase `ModGauss`, que, como la basada en distancia euclídea `ModEucl`, es heredera de la clase `Modelo`.

Si en el caso de `ModEucl` el atributo fundamental era la `media` de las realizaciones de cada unidad, ahora usaremos como atributo principal un objeto de la clase `multivariate_normal` de nombre `gauss`. En la escritura y lectura de los ficheros de los modelos usaremos los atributos `mean` y `variance` del objeto.

Almacenamos la definición del modelo gaussiano en el fichero `gaussiano.py`, con el contenido siguiente:

gaussiano.py

```
import numpy as np
from scipy.stats import multivariate_normal

from mod import Modelo

class ModGauss(Modelo):
    """
    Clase propia del modelado usando funciones de distribución gaussianas.

    Usando funciones de distribución gaussianas, el representante óptimo de la
    clase consiste en la media y la varianza de las señales de entrenamiento,
    que se almacenan en un objeto del tipo scipy.stats.multivariate_normal.
    """

    def escrMod(self, pathMod):
        chkPathName(pathMod)
        with open(pathMod, 'wb') as fpMod:
            np.save(fpMod, self.gauss.mean)
            np.save(fpMod, self.gauss.cov)

    def leeMod(self, pathMod):
        with open(pathMod, 'rb') as fpMod:
            media = np.load(fpMod)
            varianza = np.load(fpMod)

        self.gauss = multivariate_normal(mean=media, cov=varianza, allow_singular=True)

    def inicMod(self):
        self.sumPrm = self.sumPrm2 = None
        self.numSen = 0

    def __add__(self, prm):
        if self.sumPrm is None:
            self.sumPrm = prm
            self.sumPrm2 = prm ** 2
        else:
            self.sumPrm += prm
            self.sumPrm2 += prm ** 2

        self.numSen += 1

        return self

    def calcMod(self):
        media = self.sumPrm / self.numSen
        varianza = self.sumPrm2 / self.numSen - media ** 2
        self.gauss = multivariate_normal(mean=media, cov=varianza, allow_singular=True)

    def __call__(self, prm):
        return self.gauss.logpdf(prm)
```

5.3.3. Incorporación del modelado gaussiano a `todo.sh`

La sustitución del modelado euclídeo por el gaussiano no puede ser más sencilla: basta con usar como script previo el fichero `gaussiano.py` e indicar que la clase a usar en el modelado es `ModGauss`:

todo.sh

```
CLS_MOD=ModGauss
EXEC_PRE=gaussiano.py

execPre="-x $EXEC_PRE"
ClsMod="-C $CLS_MOD"
```

Los resultados obtenidos con la distribución gaussiana, para las tres parametrizaciones vistas, mejoran los obtenidos con distancia euclídea:

Modelo	Periodograma	Cepstrum	Blackman–Tukey	Máxima
Distancia Euclídea	84.70 %	85.25 %	85.95 %	87
Distribución Gaussiana	86.20 %	86.55 %	86.65 %	87

Puede observarse que en todos los casos la tasa de exactitud aumenta al pasar del modelado acústico basado en la distancia euclídea al basado en la distribución gaussiana. Por otro lado, los resultados parecen menos dependientes del tipo de parametrización usada. En el caso de la distancia euclídea, la diferencia entre la mejor y la peor es superior a tres puntos, mientras que con la distribución gaussiana disminuye a menos de dos; prácticamente la mitad.

5.4. Modelos de mezcla de gaussianas (GMM)

Los resultados con modelado bayesiano usando funciones de densidad gaussianas permiten mejorar algo los resultados obtenidos con la distancia euclídea. Sin embargo, podemos estar casi completamente seguros de que los datos no presentan una distribución gaussiana. Por tanto, una mejora posible es sustituir la aproximación gaussiana por una más potente.

Un ejemplo de modelado estadístico más potente que el gaussiano es el de los GMM (*Gaussian Mixture Models*, o modelos de mezcla de gaussianas). Este tipo de modelado puede verse como una extensión del modelado gaussiano, en el cual, en vez de una única gaussiana, se usa una combinación lineal de varias de ellas:

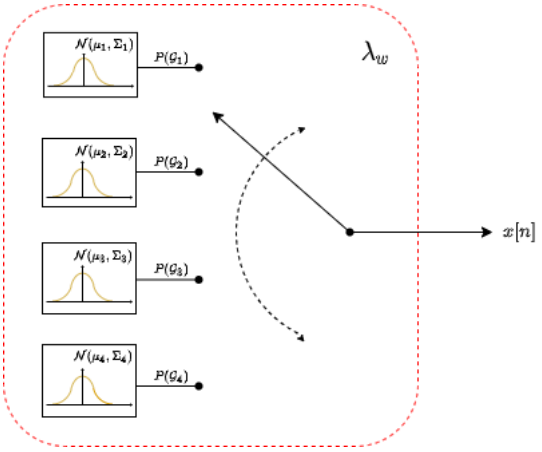
$$f_x(\mathbf{x}) = \sum_{k=0}^{K} c_k \, p(\mathbf{x}|\mathcal{G}_k) = \sum_{k=0}^{K} c_k \, \mathcal{N}(\mathbf{x}, \mu_k, \Sigma_k) \tag{5.13}$$

Donde los llamados *pesos*, c_k, y las medias, μ_k, y matrices de covarianza, Σ_k, de las funciones de densidad gaussiana se calculan aplicando el algoritmo *expectation maximization* (EM) con el objetivo de maximizar la verosimilitud (*likelihood*) de las señales dados sus modelos.

5.4.1. Algoritmo EM (*Expectation-Maximization*)

Un modo habitual y conveniente de manejar los modelos de mezcla de gaussianas consiste en suponer que el proceso estocástico gobernado por ellos incluye un parámetro oculto y discreto. En función del valor que toma este parámetro, la función de densidad del proceso responde a una u otra gaussiana de la mezcla.

El esquema siguiente muestra esta idea: la variable discreta $\mathcal{G}$ puede tomar cuatro valores distintos con probabilidad $P(\mathcal{G}_i)$. En función del valor concreto de $\mathcal{G}_i$, la función de densidad es la correspondiente gaussiana de parámetros μ_i y Σ_i:



Podemos decir, por tanto, que las señales de la clase modelada por el GMM se *reparten* entre las distintas gaussianas que lo forman. Este reparto no sólo es desconocido a priori, sino que, en principio, no nos preocupa; todo lo que realmente necesitamos es estimar los parámetros del modelo para que conjuntamente proporcionen la mejor estimación posible de la función de densidad de las señales. O, lo que es equivalente, que maximicen la verosimilitud de las señales dado el modelo.

Es inmediato que la función de densidad del proceso es la combinación de la probabilidad de la variable latente y las funciones de densidad gaussianas:

$$f(\mathbf{x}|\lambda_w) = \sum_i P(\mathcal{G}_i)\mathcal{N}(\mathbf{x}, \mu_i, \Sigma_i) \tag{5.14}$$

El algoritmo EM se basa en los dos hechos siguientes:

1. Si conocemos los parámetros del modelo, que en un GMM son los pesos de cada gaussiana (c_k) y sus medias y varianzas (μ_k y Σ_k), entonces es posible determinar el reparto que maximiza la log-verosimilitud media del modelo dada la población de entrenamiento.

2. Si conocemos el reparto de la población entre las distintas gaussianas, entonces es inmediato estimar los parámetros del modelos (c_k, μ_k y Σ_k) que maximizan la log-verosimilitud media.

Es decir, disponemos de dos mecanismos independientes que permiten maximizar la verosimilitud. Usando el primero, denominado *fase de estimación* (E), optimizamos el reparto de las señales. Pero, con este nuevo reparto, los parámetros del modelo ya no son los óptimos. Sin embargo, sí podemos optimizarlos en la denominada *fase de maximización* (M). Pero, con estos nuevos parámetros, el reparto anterior es sub-óptimo; aunque podemos calcular un nuevo reparto óptimo aplicando una nueva fase de estimación (E), que dará lugar a unos nuevos parámetros, que darán lugar a un nuevo reparto... Y así, sucesivamente, podemos aplicar fases de estimación y maximización en lo que se denomina *algoritmo EM*, que garantiza que la verosimilitud aumenta en cada iteración.

Aunque el reparto se ha presentado a nivel de señales (cada señal es asignada a una gaussiana u otra), como el objetivo es maximizar el valor esperado de la verosimilitud, podemos aplicarlo a cada una de las señales. Esto es equivalente a considerar que cada señal es dividida en cachos, cada uno de los cuales es asignado a cada una de las gaussianas que forman el modelo.

Aplicando Bayes, es inmediato determinar el reparto óptimo de la señal o entre las K gaussianas de la mezcla, $p(\mathcal{G}_k|\mathbf{x})$:

$$p(\mathcal{G}_k|\mathbf{x}) = \frac{c_k\,\mathcal{N}(\mathbf{x},\mu_i,\Sigma_i)}{\sum_{k=0}^{K} c_k\,\mathcal{N}(\mathbf{x},\mu_i,\Sigma_i)} \tag{5.15}$$

En la fase *expectation* del algoritmo EM, se calcula este reparto para las N tramas de las señales de entrenamiento, $\mathbf{x}_i$; mientras que en la fase *maximization*, se calculan los pesos, medias y matrices de covarianza atendiendo al mismo:

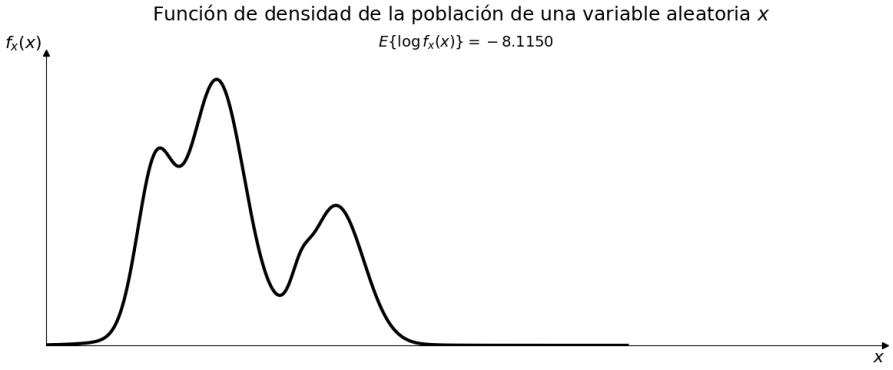
$$c_k = \frac{1}{N}\sum_i p(\mathcal{G}_k|\mathbf{x}_i) \tag{5.16}$$

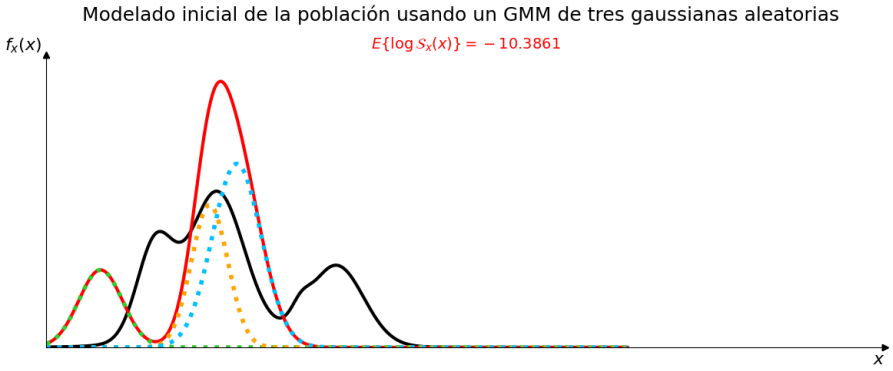
$$\mu_k = \frac{\sum_i p(\mathcal{G}_k|\mathbf{x}_i)\,\mathbf{x}_i}{\sum_i p(\mathcal{G}_k|\mathbf{x}_i)} \tag{5.17}$$

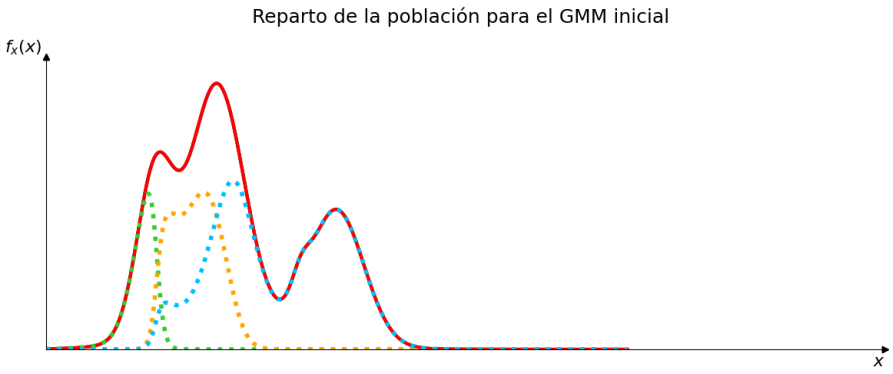
$$\Sigma_k = \frac{\sum_i p(\mathcal{G}_k|\mathbf{x}_i)\,(\mathbf{x}_i-\mu_k)(\mathbf{x}_i-\mu_k)^T}{\sum_i p(\mathcal{G}_k|\mathbf{x}_i)} \tag{5.18}$$

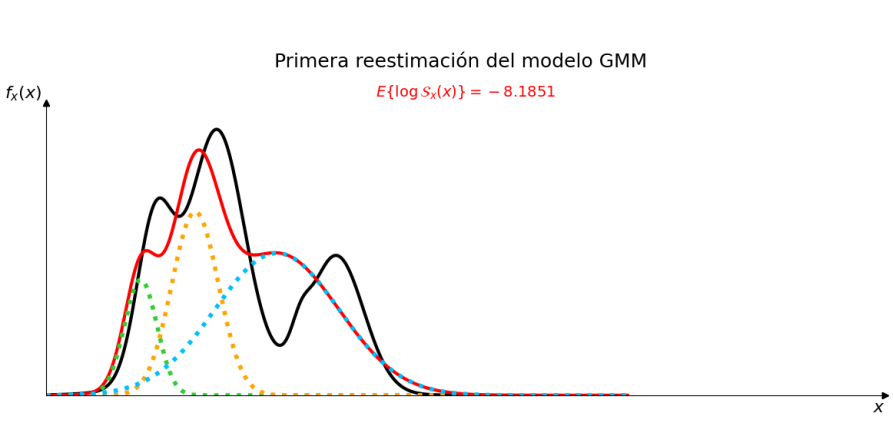
Ejemplo de entrenamiento de un GMM

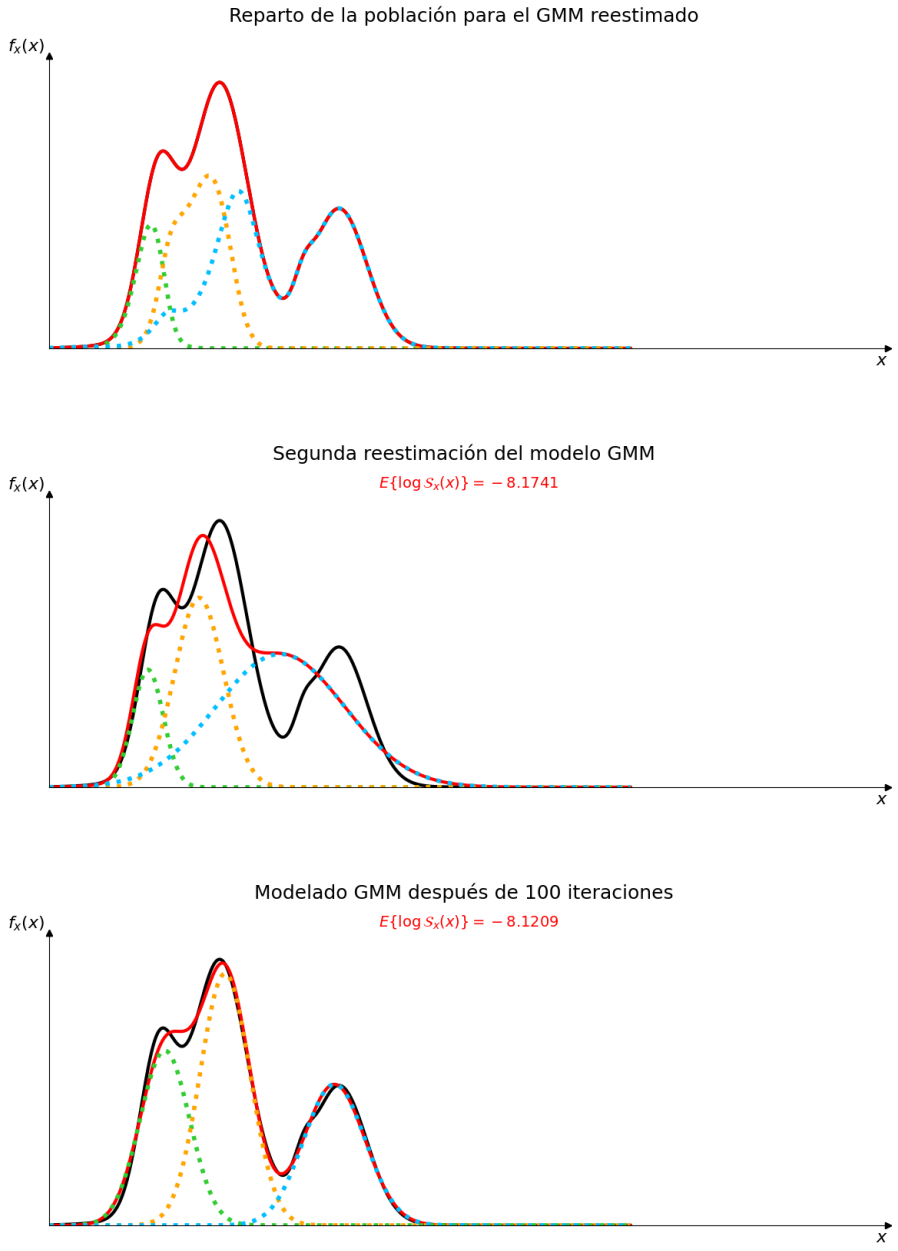
Supongamos la variable aleatoria x cuya función de densidad se muestra en la figura siguiente:


Función de densidad de la población de una variable aleatoria x
$E\{\log f_x(x)\} = -8.1150$
$f_x(x)$
x


Modelado inicial de la población usando un GMM de tres gaussianas aleatorias
$E\{\log S_x(x)\} = -10.3861$
$f_x(x)$
x


Reparto de la población para el GMM inicial
$f_x(x)$
x


Primera reestimación del modelo GMM
$E\{\log S_x(x)\} = -8.1851$
$f_x(x)$
x



5.4.2. Atributos y métodos básicos de la clase `ModGMM`

Implementaremos el modelado con GMM en la clase `GMM` definida en el fichero `gmm.py`. Los atributos propios de esta clase son tres: el número de gaussianas que forman la mezcla, que, atendiendo a la tradición, se denominará `nmix`[4]; y las medias y covarianzas de cada una de las gaussianas que forman la mezcla, `medias` y `varianzas`. También se añade el atributo `logProb` para calcular la log-probabilidad media de las secuencias de entrenamiento. Este atributo sólo se utiliza para mostrar información acerca de la convergencia del algoritmo.

El constructor de la clase debe ser algo distinto al genérico de `Modelo` porque, además de permitir el paso de un argumento opcional `pathMod`, que indica el fichero en el que se encuentra el modelo a leer, también debe permitir un argumento, `nmix` con el número deseado de gaussianas por mezcla cuando ésta debe ser inicializada desde cero:

[4] Al autor de estas líneas siempre le ha desagradado este término, ya que parece indicar *número de mezclas*, cuando a lo que se refiere es al *número de gaussianas por mezcla*.

gmm.py

```
import numpy as np
from scipy.stats import multivariate_normal

from mod import Modelo

class ModGMM(Modelo):
    """
    Clase propia del modelado usando mezcla de gaussianas
    """
    def __init__(self, pathMod=None, nmix=4):
        super().__init__(pathMod)
        if not pathMod: self.nmix = nmix
```

Los métodos `inicMod()`, `leeMod()` y `escrMod()` son equivalentes a los del modelado con una única gaussiana, con la única diferencia de que ahora tenemos `nmix` gaussianas en lugar de una sola:

gmm.py

```
def escrMod(self, pathMod):
    chkPathName(pathMod)
    with open(pathMod, 'wb') as fpMod:
        np.save(fpMod, self.nmix)
        np.save(fpMod, self.pesos)
        for mix in range(self.nmix):
            np.save(fpMod, self.gauss[mix].mean)
            np.save(fpMod, self.gauss[mix].cov)

def leeMod(self, pathMod):
    with open(pathMod, 'rb') as fpMod:
        self.nmix = int(np.load(fpMod))
        self.pesos = np.load(fpMod)
        self.gauss = [None] * self.nmix
        for mix in range(self.nmix):
            media = np.load(fpMod)
            varianza = np.load(fpMod)
            self.gauss[mix] = multivariate_normal(mean=media, cov=varianza,
             ↪  allow_singular=True)

def inicMod(self):
    self.sumPrm = None
```

Del mismo modo, la modificación principal del método `calcMod()`, consiste en la extensión a `nmix` gaussianas de lo realizado para una sola. Además, de manera un tanto chapucera, se incorpora en ella la escritura en pantalla de la log-probabilidad media de las secuencias de entrenamiento:

gmm.py

```
def calcMod(self):
    for mix in range(self.nmix):
        self.pesos = self.numSen / np.sum(self.numSen)
        media = self.sumPrm[mix] / self.numSen[mix]
        varianza = np.fmax(self.sumPrm2[mix] / self.numSen[mix] - media ** 2, 1.e-24)
        self.gauss[mix] = multivariate_normal(mean=media, cov=varianza,
         ↪  allow_singular=True)
```

```
    self.logPrb /= np.sum(self.numSen)
    print(f'{self.logPrb}')
```

Las modificaciones más importantes respecto al caso de una sola gaussiana se producen en el cálculo de la probabilidad de las señales, `__call__()` y la incorporación de la información de las señales al modelo, `__add__()`. En estas dos funciones es fundamental determinar cuál es el *reparto* de la señal entre las distintas gaussianas de la mezcla. Para ello, definimos en primer lugar el método *reparte()*, que realiza esta operación, devolviendo una dupla formada por el reparto de la señal entre las gaussianas (en la forma de `numpy.array` de `nmix` valores) y la log-probabilidad de la señal:

gmm.py

```
def reparte(self, prm):
    logGauss = np.array([gauss.logpdf(prm) for gauss in self.gauss])
    maxGauss = max(logGauss)
    logGauss -= maxGauss
    reparto = np.exp(logGauss) * self.pesos
    logGMM = maxGauss + np.log(np.sum(reparto))
    reparto /= np.sum(reparto)
    return reparto, logGMM
```

La variable auxiliar `maxGauss` se utiliza para evitar problemas con el margen dinámico de las gaussianas. Siempre es mejor manejar su valor logarítmico, `logpdf` y `logGauss`, pero el *reparto* debe realizarse según su valor lineal. El problema del margen dinámico se puede solventar dividiendo los valores de todas las gaussianas por el mismo valor; operación que, en logaritmos, se convierte en una resta (`logGauss -= maxGauss`). Pero, al calcular la log-probabilidad hemos de deshacer esta operación (`logGMM = maxGauss + ...`).

Usando este reparto, la implementación de `__call__()` y `__add__()` resulta:

gmm.py

```
def __add__(self, prm):
    if self.sumPrm is None:
        self.sumPrm = np.zeros((self.nmix,) + prm.shape)
        self.sumPrm2 = np.zeros((self.nmix,) + prm.shape)
        self.numSen = np.zeros(self.nmix,)
        self.logPrb = 0

    reparto, logGMM = self.reparte(prm)
    for mix in range(self.nmix):
        self.sumPrm[mix] += reparto[mix] * prm
        self.sumPrm2[mix] += reparto[mix] * prm ** 2
        self.numSen[mix] += reparto[mix]

    self.logPrb += logGMM

    return self

def __call__(self, prm):
```

```
    reparto, logGMM = self.reparte(prm)
    return logGMM
```

Inicialización de los modelos GMM

Un problema que surge con la definición de la función `entrena()` y la clase `ModGMM` es que, cuando pretendemos entrenar modelos desde cero, se desconoce tanto el número de gaussianas `nmix` como la dimensión de los vectores de señal. La solución trivial, y en general óptima, consiste en usar siempre modelos inicializados externamente. Pero también es deseable tener un mecanismo de inicialización que permita usar los programas ya desarrollados directamente. Para ello, se añade, de manera tal vez un poco chapucera, al método `__add__()` el código siguiente, que, si detecta que no hay modelos previos, los inicializa de manera aleatoria:

gmm.py

```
def __add__(self, prm):
    if self.sumPrm is None:
        # Si no hay modelo, hemos de inicializarlo de alguna manera... Lo hacemos
        # aleatoriamente. Es una marranada... pero algo hay que hacer
        if 'gauss' not in self.__dir__():
            self.pesos = np.ones(self.nmix) / self.nmix
            self.gauss = [None] * self.nmix
            for mix in range(self.nmix):
                self.gauss[mix] =                                                    ...
                ↪  multivariate_normal(mean=np.random.randn(*prm.shape),             ...
                ↪  allow_singular=True)
        self.sumPrm = np.zeros((self.nmix,) + prm.shape)
        self.sumPrm2 = np.zeros((self.nmix,) + prm.shape)
        self.numSen = np.zeros(self.nmix,)
        self.logPrb = 0

                ...
```