Clasificación y verificación del hablante

Antonio Bonafonte – profesores de la asignatura

mayo de 2020

Resumen

En esta práctica:

- Se implementará un sistema de extracción de características para la clasificación del hablante.
- Se completará y finalizará el algoritmo de estimación y evaluación de el modelo probabilístico GMM.
- Se utilizarán para clasificar el locutor, utilizando el corpus de desarrollo.
- Se utilizarán para una aplicación de verificación del locutor.
- Se investigarán métodos para mejorar las prestaciones de los sistemas base.
- Se utilizará la programación `bash` para realizar los experimentos.
- Participará en una evaluación formal, y presentará tanto el sistema base como las alternativas que ha analizado para obtener su propuesta de sistema.

Índice

1. Introducción teórica.

1.1. Reconocimiento y verificación del locutor.

Entre las tecnologías de la voz y el habla pueden destacarse tres como las más relevantes:

1. La codificación, consistente en la representación, transmisión y almacenamiento de la señal de voz de manera eficiente.
2. La síntesis de voz, consistente en la reproducción automática de mensajes orales.
3. El reconocimiento, consistente en la determinación del mensaje contenido en un mensaje oral, o de cualquier otra característica del mismo.

Las posibles características a reconocer por los sistemas de reconocimiento abarcan un amplio abanico de posibilidades: desde el propio mensaje contenido en ellas, denominado *reconocimiento del habla*, hasta cosas tan exóticas como el estado de somnolencia o embriaguez. Dos de los ámbitos más populares de las técnicas de reconocimiento son el reconocimiento y/o la verificación del productor del mensaje oral (locutor).

- El reconocimiento del locutor consiste en determinar qué locutor, de entre un conjunto predeterminado de los mismos, es el autor de un cierto mensaje oral. Podemos encontrar ejemplos de aplicación en tareas como la identificación del usuario por asistentes personales virtuales como Apple Siri o Amazon Alexa.
- La verificación del locutor consiste en confirmar si el locutor es quien dice ser. Podemos encontrar ejemplos de aplicación en multitud de películas o series de TV, en las que el acceso a una dependencia ultra secreta y/o ultra segura es restringido y activado por voz, como en Mission: Impossible o la serie de películas del agente secreto del MI6 Bond, James Bond.

En ambas tareas, así como en la mayoría de tareas de reconocimiento automático de voz o de cualquier otro tipo, es habitualmente posible distinguir tres tareas fundamentales:

- La parametrización (o *extracción de características*) es el conjunto de técnicas empleadas para representar la señal en un espacio vectorial en el cual las diferentes clases a reconocer sean fácilmente separables.

  Ejemplos típicos de parametrización de la señal de voz de cara a su empleo en sistemas de reconocimiento automático son los coeficientes de predicción lineal (LP) o los *Mel-frequency cepstral coefficients* (MFCC).

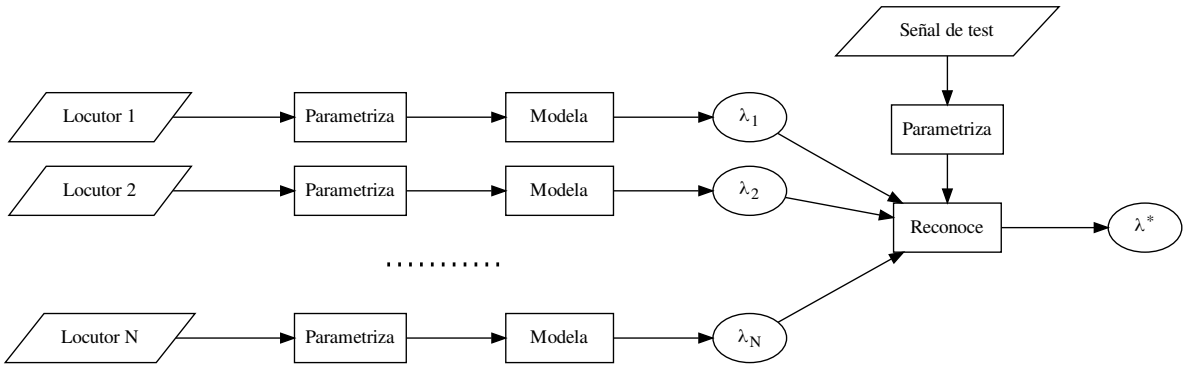
- El modelado (o entrenamiento) consiste en la construcción de un modelo matemático que capture las propiedades estadísticas de las clases a reconocer.

  Entre los modelos más usados en reconocimiento del habla encontramos las plantillas, los modelos de mezclas de gaussianas (*Gaussian Mixture Models*, GMM), los modelos ocultos de Markov (*Hidden Markov Models*, HMM) o las redes neuronales (*Neural Networks*, NN).

- El reconocimiento consiste en determinar cuál de los modelos de las clases a reconocer representa mejor la señal según algún criterio.

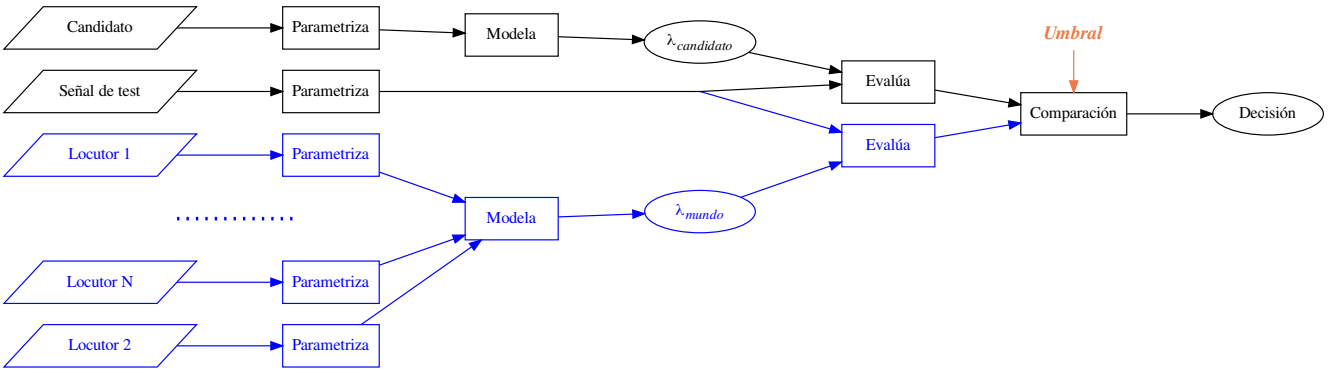
  Los criterios más habituales son distintos tipos de distancia, en el modelado mediante plantillas; la probabilidad, en GMM o HMM; o algún tipo de puntuación (o *score*), que nadie sabe bien bien qué representa (ni le importa), en las redes neuronales.

En el caso del reconocimiento del locutor, se construye un modelo para cada posible locutor y se determina qué modelo representa mejor la señal a reconocer:



En el caso de la verificación del locutor, se calcula cuán apropiado es el modelo del locutor candidato para representar la señal de test, y se compara con un umbral de referencia. En el caso de usarse la verosimilitud como criterio, si la verosimilitud del modelo del candidato para la señal de test supera el umbral, diremos que se la señal es de un usuario *legítimo*; si no lo hace, diremos que es un *impostor*.

Suele ser conveniente establecer el umbral en base a un modelo genérico, a menudo denominado *modelo del mundo* (*world*). Este modelo genérico se diseña de manera que represente a la generalidad de los posibles locutores, para lo cual es necesario entrenarlo a partir de un conjunto elevado de locutores. En función de la diferencia entre el resultado para uno y otro modelo, decidiremos si la señal ha sido producida por el locutor candidato o no:



En la gráfica, la parte inferior, representada en color azul, sólo se utiliza si fijamos el umbral de decisión usando el modelo *mundo*.

Evidentemente, la rama superior deberá repetirse tantas veces como candidatos tengamos en el sistema. Si éstos son los mismos que participan en la tarea de reconocimiento del locutor, los modelos acústicos pueden ser compartidos por ambas tareas.

1.2. Extracción de características en procesado de voz.

En las aplicaciones de reconocimiento es habitual transformar la señal de entrada de manera que se obtenga una representación de la misma en la que sea más fácil la clasificación. Es la fase denominada *extracción de características* o *parametrización*. Puede verse la parametrización como un proceso de eliminación de la información redundante o irrelevante para nuestros objetivos. Considérese la señal temporal: a partir de ella somos capaces de extraer mucha información: el mensaje oral, el sexo del locutor, su edad, estado de ánimo... incluso, como se ha comentado anteriormente, su estado de ebriedad o somnolencia. Ahora bien, si sólo estamos interesados en una de estas informaciones, el resto de ellas puede ser considerado como ruido que entorpecerá nuestro objetivo.

En aplicaciones de reconocimiento del habla, por ejemplo, la información fonética se concentra en la envolvente del espectro. Sin embargo, en el detalle fino del mismo encontramos información irrelevante fonéticamente, como el pitch o las características del locutor. Así pues, diversos esquemas de parametrización orientados al reconocimiento del habla están diseñados para reflejar esa envolvente, descartando el resto de la información presente en la señal.

$$\begin{aligned} H(z) &= \frac{1}{1 + a_1\, z^{-1} + \cdots + a_p\, z^{-p}} \\ S(F) &= \left|H(e^{j2\pi F})\right|^2 \end{aligned} \tag{1}$$

Uno de estos esquemas son los coeficientes de predicción lineal (*linear prediction*, LP), $v = (a_1 \ldots a_p)$, muy empleados en los orígenes del reconocimiento del habla debido a sus buenas prestaciones con un coste computacional muy reducido. De ellos se obtienen los coeficientes LPCC (*linear prediction cepstral coefficients*), que tienen la ventaja de presentar una mayor incorrelación que los primeros. Más adelante, en las décadas de los 80 y 90 del siglo pasado, cobraron popularidad los coeficientes cepstrales en escala Mel (MFCC, *Mel-frequency cepstral coefficients*), basados en el uso de un banco de filtros pasa banda equiespaciados en escala Mel.

En esta práctica comenzaremos comparando los LP con los LPCC, y se dejará al alumno la posibilidad de experimentar con los MFCC o similares.

1.3. Modelos de mezcla de gaussianas (*Gaussian Mixture Model*, GMM).

En esta práctica, al menos en primera instancia, se abordará el reconocimiento y verificación del locutor usando el modelado mediante mezcla de gaussianas (GMM) (ver las Transparencias de la asignatura). Este tipo de modelado puede verse como una extensión del modelado gaussiano, en cual, en vez de una única gaussiana, se usa una combinación lineal de las mismas:

$$f_x(\mathbf{x}) = \sum_{k=0}^{K} c_k\, p(\mathbf{x}|m_k) = \sum_{k=0}^{K} c_k\, \mathcal{N}(\mathbf{x}, \mu_k, \Sigma_k) \tag{2}$$

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) en aras a maximizar la verosimilitud (*likelihood*) de las señales dados sus modelos.

Puede verse el modelado mediante GMM como un reparto de la trama de señal entre cada una de las gaussianas que forman la mezcla, $p(m_k|\mathbf{x})$:

$$p(m_k|\mathbf{x}) = \frac{c_k\, p(\mathbf{x}|m_k)}{\sum_{k=0}^{K} c_k\, p(\mathbf{x}|m_k)} \tag{3}$$

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:

$$\hat{c}_k = \frac{1}{N} \sum_i p(m_k|\mathbf{x}_i) \tag{4}$$

$$\hat{\mu}_k = \frac{\sum_i p(m_k|\mathbf{x}_i)\, \mathbf{x}_i}{\sum_i p(m_k|\mathbf{x}_i)} \tag{5}$$

$$\hat{\Sigma}_k = \frac{\sum_i p(m_k|\mathbf{x}_i)\, (\mathbf{x}_i - \hat{\mu}_k)(\mathbf{x}_i - \hat{\mu}_k)^T}{\sum_i p(m_k|\mathbf{x}_i)} \tag{6}$$

2. Estructura del proyecto.

Tareas:

- Siga las instrucciones de `README.md` para realizar un fork en su directorio `PAV` del repositorio de la Práctica P4.
- Baje de Atenea el fichero `db_8mu.tgz` y descomprima su contenido en el directorio `PAV/P4` con la orden `tar xvzf db_8mu.tgz`.

Al bifurcar (*fork*) el repositorio y descomprimir el fichero con la base de datos, su directorio de trabajo de la práctica queda con el siguiente contenido:

```
P4
├── Makefile
├── README.md
├── img
├── lists
├── meson.build
├── scripts
└── src
```

En las secciones siguientes se explica el contenido de los subdirectorios de P4.

2.1. Códigos fuente de la práctica.

El directorio `src` incluye los códigos fuente de la práctica:

```
P4/src/
├── doxyfile
│   ├── Doxyfile.in
│   ├── meson.build
│   └── style.css
├── fmatrix
│   ├── fmatrix_cut.cpp
│   └── fmatrix_show.cpp
├── gmm
│   ├── gmm_classify.cpp
│   ├── gmm_show.cpp
│   ├── gmm_train.cpp
│   └── gmm_verify.cpp
├── include
│   ├── digital_filter.h
│   ├── ffft
│   ├── filename.h
│   ├── gmm.h
│   ├── keyvalue.h
│   ├── matrix.h
│   └── wavfile_mono.h
├── meson.build
├── pav
│   ├── digital_filter.cpp
│   ├── docopt_cpp
│   ├── filename.cpp
```

```
│   ├── gmm.cpp
│   ├── gmm_vq.cpp
│   ├── keyvalue.cpp
│   ├── meson.build
│   └── wavfile_mono.cpp
└── pearson
    ├── pearson.cpp
    └── readme
```

- Programas de la práctica, repartidos entre los subdirectorios `src/gmm`, `src/pearson` y `src/fmatrix`.
- Documentación del proyecto, a partir de los ficheros de configuración en `src/doxyfile`.
- Librería `libpav.a`, con las funciones y clases genéricas desarrolladas en la asignatura y que son usadas por múltiples proyectos.

Los programas principales de la práctica están en el directorio `src/gmm`, y son:

`gmm_train.cpp`:
Programa que realiza el entrenamiento de los modelos acústicos.

`gmm_classify.cpp`:
Programa que determina qué locutor, de entre un conjunto de los mismos, que llamaremos *usuarios*, es el que ha producido la señal de test.

`gmm_verify.cpp`:
Programa que determina si la señal de test ha sido producida por un locutor determinado, que llamaremos *candidato*, o bien ha sido producida por un *impostor*.

Otros programas que se generarán en la práctica y que tienen un interés más específico son:

`gmm_show.cpp`:
También en el directorio `gmm`, muestra el contenido de los GMM en modo texto.

`pearson.cpp`:
En el directorio del mismo nombre, analiza la información proporcionada por un cierto conjunto de parámetros

`fmatrix_cut.cpp`:
En el directorio `fmatrix`, selecciona columnas concretas de un fichero `fmatrix`.

`fmatrix_show.cpp`:
En el directorio `fmatrix`, muestra el contenido de un fichero `fmatrix` en modo texto.

2.2. Scripts de la práctica.

También se instalarán los scripts de la práctica, ubicados en el directorio `scripts`:

```
P4/scripts/
├── meson.build
├── plot_gmm_feat.py
├── run_spkid.sh
├── spk_verif_score.pl
└── wav2lp.sh
```

run_spkid.sh:
: Script principal de la práctica. Desde él, se invocan los distintos programas y funciones que intervienen en la práctica.

wav2lp.sh:
: Script que realiza la parametrización de una señal de voz usando coeficientes de predicción lineal. Servirá como modelo para escribir otros scripts en los que se usen esquemas de parametrización más potentes, como los LPCC o los MFCC.

spk_verif_score.pl:
: Script usado en el cálculo de la tasa de error en clasificación del locutor en función del umbral de admisión. También se puede usar para determinar el umbral óptimo.

plot_gmm_feat.py:
: Programa Python para la representación gráfica de modelos GMM y señales parametrizadas.

En esta práctica añadimos dos nuevos códigos fuente a la librería de la asignatura, `libpav.a`:

gmm.cpp:
: En este fichero, junto con el `include/gmm.h`, se define la clase `GMM`, que se usará para manejar los modelos de mezcla de gaussianas.

gmm_vq.cpp:
: Definición de las funciones usadas por la clase `GMM` para implementar la *cuantificación vectorial*, usada a menudo como método de inicialización de los modelos de mezcla de gaussianas.

2.3. Bases de datos para evaluación y listas de ficheros y locutores.

En el directorio `spk_8mu` tenemos la base de datos que se usará en la fase experimental de esta práctica:

```
P4/spk_8mu/
├── speecon
└── sr_test
    ├── spk_cls
    └── spk_ver
```

La base de datos se basa en SPEECON (*Speech Databases for Consumer Devices*). En este caso, se han convertido las señales, originalmente muestreadas a 16 kHz con PCM lineal de 16 bits, a una frecuencia de muestreo de 8 kHz, ancho de banda telefónico (300 Hz − 3400 Hz) y codificación PCM ley-μ de 8 bits.

En `spk_8mu/speecon` se dispone de las señales usadas durante el transcurso del desarrollo de los programas de la práctica. El nombre de las señales sigue el formato `BLOCK00/SES000/SA000S1.wav`, donde el nombre del subdirectorio `SES000` indica que se trata del locutor *SES000*.

En `spk_8mu/sr_test` se dispone las señales de test para clasificación `spk_8mu/sr_test/spk_cls` y verificación `spk_8mu/sr_test/spk_ver`. En este caso, el nombre esta mangoneado, de manera que es imposible saber de qué locutor se trata (por ejemplo, `spk_8mu/sr_test/spk_cls/c00/c0000.wav`).

Las listas de ficheros para clasificación y verificación están en el directorio `lists`:

```
P4/lists/
├── class
├── final
├── gmm.list
└── verif
```

lists/class: Listas necesarias para entrenar los GMM de cada locutor, por ejemplo: `P4/lists/class/SES000.t`
También se incluye la lista `P4/lists/class/all.test`, que se usará en la tarea de reconocimiento del locutor.

lists/verif: Listas usadas en verificación del locutor. Incluye las listas de señales a usar, `all.test` y los usuarios que se alegan en cada una de ellas, `all.test.candidates`. También hay listas con los ficheros de los usuarios legítimos, de los impostores, y de locutores externos, que no pertenecen ni a un grupo ni al otro.

final: Incluye las listas necesarias para la evaluación final, tanto en reconocimiento, `class`, como en verificación, `verif`.

gmm.list: El directorio `lists` también incluye el fichero `gmm.list`, con la lista de los usuarios de entrenamiento para la tarea de clasificación.

2.4. Mantenimiento de los programas y la documentación.

En esta práctica se va a realizar el mantenimiento de los programas usando un enfoque híbrido entre `make` y Meson/Ninja. Los motivos para ello, y la descripción completa del sistema de mantenimiento, pueden encontrarse en el Anexo I, pero, básicamente, consiste en usar `make` como *front-end* amigable, con Meson/Ninja realizando el trabajo duro.

Al invocar `make` sin argumentos se muestran en pantalla los posibles *targets* del fichero `Makefile`:

```
usuario:~/PAV/P4$ make
Usage:
  make release       : create "bin-&-lib" of the release version
  make debug         : create "bin-&-lib" of the debug version
  make all           : make debug and release

  make clean_release : remove the "release" intermediate files
  make clean_debug   : remove the "debug"   intermediate files
  make clean         : make clean_debug and clean_release

  make doc           : generate the documentation of the project
```

Ejecutando `make release` o `make debug`, se invocan las reglas necesaria de `meson.build` para instalar los programas en el directorio `$HOME/PAV/bin` con las opciones correspondientes a:

release: Compilación con la máxima optimización del código. Los programas se ejecutarán más rápidamente (a menudo, mucho más), pero a costa de perderse la correspondencia uno a uno entre las instrucciones en C++ y las instrucciones en ensamblador. Además, el compilador hará de todo con las variables de C++: las eliminará, si son irrelevantes; las almacenará en los registros de la CPU, si son usadas a menudo; etc.

debug: Compilación en la que se respeta al máximo el código fuente en C++: cada instrucción C++ se traduce en un bloque compacto de instrucciones en ensamblador; y se respetan las variables. Además, junto con el código ensamblador se almacena el código fuente correspondiente.

El motivo de disponer de dos versiones es evidente a partir de su propio nombre: `release` se usará cuando queramos construir la versión definitiva de los programas, que querremos que se ejecuten tan rápido como sea posible. Pero el efecto de depurar con un *debugger* código optimizado por el compilador es simplemente demencial: la ejecución va dando tumbos entre líneas y muchas variables no son accesibles. Para ello se construye la versión `debug`, que facilita el depurado, a costa de una ejecución más lenta y de programas más grandes.

2.4.1. Instalación de los scripts.

Al ejecutar `make release` o `make debug`, también se instalan en el directorio `~/PAV/bin` los scripts de la práctica, cuyo código fuente está en el directorio `scripts`. De este modo, los scripts serán también accesibles desde cualquier directorio, sin necesidad de especificar su ruta completa (siempre que tenga `~/PAV/bin` en el `$PATH`). Por coherencia con el resto de casos, en los cuales los programas ejecutables pierden la extensión de su código fuente, los scripts instalados en `~/PAV/bin` también lo hacen.

Tareas:

- Recompile los programas de la práctica con la orden `make release` y compruebe que no se produce ningún error:
  - Compruebe que se han generado los programas en el directorio `~/PAV/bin`.
  - Compruebe que este directorio está incluido en su variable `PATH`. En el caso de que no lo esté, edite el fichero `~/.profile` y añada la línea `PATH+=$HOME/PAV/bin`. Compruebe que los programas de la práctica pueden ser accedidos sin especificar su ruta. (Si ha tenido que modificar `~/.profile`, deberá ejecutar `source ~/.profile` para que los cambios tengan efecto en su sesión Bash actual).
- Genere la documentación del proyecto con la orden `make doc`.
  - Acceda al directorio `~/PAV/html/P4` y compruebe que se ha generado la documentación correctamente.
  - Use su navegador para explorar la documentación y compruebe, en la sección *Lista de TODOs*, el mucho trabajo que tiene que realizar.

3. Extracción de características usando la librería SPTK.

Para realizar la extracción de características durante esta práctica, usaremos el popular paquete de código abierto para procesado de señal de voz SPTK (*Speech Signal Processing Toolkit*), desarrollado en el Tokyo Institute of Technology y el Nagoya Institute of Technology. SPTK proporciona un conjunto de herramientas que permite realizar la mayor parte de tareas necesarias en procesado digital de la señal de voz: enventanado, estimación espectral, predicción lineal, etc. Además, proporciona un entorno simple, pero muy eficiente, que permite implementar casi cualquier técnica de procesado de voz no cubierta por el propio paquete.

SPTK permite dos tipos de interacción: en línea de comandos, como programas independientes que realizan, cada uno, tareas concretas de procesado de señal; o como librería C, `libSPTK.a`, que permite su inclusión en programas desarrollados en ese lenguaje.

3.1. Instalación de SPTK.

En entornos Debian (como es el caso de Ubuntu), la instalación puede realizarse del modo habitual con la orden `sudo apt-get install sptk`.

Instalación de SPTK en otros sistemas operativos (por ejemplo, MacOS X).

En otros entornos, muy significativamente en MacOS X, puede ser necesario descargar el código fuente de su página web en Source Forge e instalar manualmente el paquete. Para ello, baje el fichero `SPTK-3.11.tar.gz` y ejecute los comandos siguientes:

```
usuario:~/PAV$ gunzip SPTK-3.11.tar.gz
usuario:~/PAV$ tar xvf SPTK-3.11.tar
```

Al hacer esto, aparecerá en su directorio un subdirectorio llamado `SPTK-3.11`. En él encontrará el código fuente preparado para ser instalado usando las herramientas de *GNU Build System*, habitualmente conocidas como *Autotools*. Este último es un paquete para la instalación de aplicaciones basado en `make` y otras aplicaciones clásicas de UNIX (como `awk`, `yacc`, etc.), que es utilizado por infinidad de paquetes de software, y cuya complejidad de implementación (aunque no de uso) es el responsable último de la aparición de alternativas como `CMake` o `Meson/Ninja`. Para realizar la instalación de SPTK, vaya al directorio `SPTK-3.11` y ejecute las órdenes siguientes:

```
# IMPORTANTE: si va a realizar la instalación en MacOS X, debe ejecutar primero la orden:
usuario:~/PAV/SPTK-3.11$ export CFLAGS="-flax-vector-conversions"
```

```
usuario:~/PAV/SPTK-3.11$ ./configure
usuario:~/PAV/SPTK-3.11$ make
usuario:~/PAV/SPTK-3.11$ sudo make install
```

En todos los casos, **es muy importante descargar** el Manual de Referencia de SPTK. Aunque los distintos programas proporcionan información de su ejecución usando la opción `-h`, ésta es habitualmente más detallada en el manual de referencia. Además, SPTK se compone de más de 130 programas; al descargar y hojear el manual, es posible obtener una visión de conjunto rápida de sus capacidades.

3.2. Ejecución de SPTK.

Nosotros usaremos SPTK desde la línea de comandos y usando scripts escritos en Bash. Existen dos modos de invocar los programas, en función de cómo se ha instalado el paquete:

- Si se ha instalado a partir del código fuente, como es necesario en máquinas MacOS X, debe invocarse el programa por su nombre. Por ejemplo, para calcular los coeficientes MFCC invocaremos el programa correspondiente con la orden `mfcc`.
- Si se ha instalado usando `apt-get`, deberemos invocar el programa usando el programa de envoltorio (*wrapper*) `sptk`: `sptk mfcc`.

  El motivo para usar este wrapper es evitar colisiones con otros programas que pudieran tener el mismo nombre. SPTK proporciona más de 130 programas, cuyo nombre suele se de corta longitud, ante lo cual es fácil que coincida con otros programas o utilidades del sistema operativo. Por ejemplo, en sistemas UNIX, la partícula inicial `lp` se reserva para denominar programas que interactúan con el sistema de impresión (`lp`, `lpq`, `lpadmin`, etc.); pero SPTK proporciona cuatro programas cuyo nombre empieza por `lp` (`lpc`, `lpc2c`, `lpc2lsp` y `lpc2sp`), pero que no tienen nada que ver con impresoras, sino con predicción lineal.

3.2.1. Cadenas de comandos SPTK.

Casi todos los programas de SPTK tienen el formato de *filtros UNIX*. Un filtro, en el contexto de los sistemas operativos tipo UNIX, es un programa que lee la entrada de la entrada estándar (el teclado), y escribe el resultado en la salida estándar (la pantalla). Evidentemente, en el contexto del procesado de señal tiene poco sentido escribir la señal de entrada utilizando el teclado, o tener que leer la señal resultante de la pantalla. Por suerte, UNIX proporciona dos mecanismos que hacen de los filtros herramientas de gran versatilidad y potencia: la redirección y el encadenado (*pipelining*).

La redirección permite redirigir la salida de un comando a un fichero. En Bash se indica con el carácter *mayor que* (`>`). El encadenado hace que la salida estándar de un comando se utilice como la entrada estándar del siguiente. En Bash se indica con la *barra vertical* (`|`).

Por otro lado, los filtros, entre ellos los programas de SPTK, suelen permitir un argumento opcional, que indica el fichero del cual se debe leer la entrada en sustitución del teclado. Esta opción de invocación suele ser útil como primer elemento de un pipeline. Así, por ejemplo, en la orden siguiente el programa `frame` lee el fichero `señal.raw`, que suponemos que está codificado con números reales de 32 bits y en formato *crudo*, para generar ventanas de 256 muestras con un desplazamiento de 100 (opciones por defecto de `frame`); y el programa `fftr` toma esas ventanas de 256 muestras y escribe su transformada de Fourier en la salida estándar, que redireccionamos al fichero `señal.fft`.

```
usuario:~/PAV/P4$ sptk frame señal.raw | sptk fftr > señal.fft
```

3.3. Formato de la señales y ficheros en SPTK.

Todas las señales en SPTK, así como los ficheros que SPTK es capaz de leer, tienen el mismo formato: sucesión de reales en coma flotante de 32 bits (el tipo `float` de C) sin ningún tipo de cabecera o formato adicional. Puede usarse el programa `sox` para generar una señal del formato adecuado a partir de una señal con otro formato (por ejemplo, WAVE). Indicando como fichero de salida un guion (`-`) hacemos que `sox` escriba la salida en la salida estándar. Así, la orden siguiente tiene el mismo resultado que la indicada en al apartado anterior, pero sin la necesidad de generar un fichero en formato crudo:

```
usuario:~/PAV/P4$ sox señal.wav -t raw - | sptk frame | sptk fftr > señal.fft
```

La ausencia de todo tipo de cabecera implica que es responsabilidad del usuario el que la estructura de las señales que se pasan a través de un pipeline sea la adecuada. Por ejemplo, el programa `frame` toma una señal a su entrada y devuelve a su salida la misma señal dividida en tramas de l muestras tomadas con un desplazamiento de p muestras. Si fijamos $l = 5$ y $p = 2$, y siendo $x[n]$ la señal de entrada, a la salida obtenemos:

$x[0]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	$x[6]$	$x[4]$	$x[5]$	$x[6]$	$x[7]$	$x[8]$	$\cdots$
trama 0					*trama 1*					*trama 2*					$\cdots$

Puede observarse que, debido a que las tramas están solapadas entre ellas, hay muestras que aparecen en más de una y, por tanto, aparecen duplicadas en la secuencia de salida. Si esta señal va a ser pasada a otro programa de SPTK, el segundo deberá ser coherente con esta estructura de los datos de entrada. Por ejemplo, y siguiendo con el ejemplo anterior, si queremos calcular la DFT en ventanas de 5 muestras solapadas 3 (desplazamiento de ventana $p = 5 - 3 = 2$), la cadena de órdenes será:

```
usuario:~/PAV/P4$ sox señal.wav -t raw - | sptk frame -l5 -p2 | sptk fftr -l5 > ...
↪  señal.fft
```

Fíjese en que el desplazamiento de ventana es irrelevante para `fftr`, lo único relevante es que tiene que coger bloques de $l = 5$ muestras, calcular su DFT y escribir el resultado en la salida.

3.3.1. El tipo de datos `fmatrix` y su almacenamiento en fichero.

En los programas de PAV, el formato de las señales parametrizadas es el de matriz de `float` (`fmatrix`), definido en la cabecera `include/matrix.h`. En él los datos se almacenan en `nrow` filas de `ncol` columnas, en los que cada fila corresponde a una trama de señal, y cada columna a cada uno de los coeficientes con los se parametriza la trama. Al escribirse en fichero, este formato requiere que primero se indique el número de filas y columnas, lo cuales se almacenan como enteros de cuatro bytes (tipo `int` de C) al principio del fichero. A continuación, se escriben los datos, como reales de cuatro bytes (formato `float` de C), organizados en filas por columnas.

Los programas `fmatrix_show` y `fmatrix_cut`, cuyo código fuente está en el directorio `src/fmatrix`, permiten mostrar el contenido de estos ficheros, o seleccionar columnas concretas de los mismos, respectivamente.

3.3.2. Script `wav2lp.sh`.

El script `wav2lp.sh` es un ejemplo de cómo combinar los programas de SPTK para confeccionar un programa que implementa una parametrización en concreto. En este caso, los coeficientes de predicción lineal. Para ello, ejecuta un pipeline que realiza, encadenadamente, las siguientes tareas:

- Convierte la señal de entrada a reales en coma flotante de 32 bits sin cabecera (*raw*), y escribe el resultado en la salida estándar:

  ```
  sox $inputfile -t raw - | sptk x2x +sf
  ```

  - Hacemos esta operación en dos pasos porque `x2x` no permite leer ficheros con formato WAVE (o cualquier otro, sólo permite el *raw*), y, cuando `sox` convierte el formato a real, lo hace normalizando la señal en el margen $-1 < x < 1$, lo cual no nos interesa.
  - `x2x` es el programa de SPTK que permite la conversión entre distintos formatos de datos. Ejecute `sptk help x2x` o `x2x -h`, o lea el manual de referencia, para conocer todas las opciones que soporta, que son muchas.

- Divide la señal de entrada en tramas de 200 muestras (25 ms) con desplazamiento de ventana de 40 muestras (5 ms) (tenga en cuenta que, en esta práctica, la frecuencia de muestreo es 8 kHz):

  ```
  sptk frame -l 200 -p 40
  ```

- Multiplica cada trama por la ventana de Blackman (opción por defecto):

  ```
  sptk window -l 200
  ```

- Calcula los `lpc_order` primeros coeficientes de predicción lineal, precedidos por el factor de ganancia del predictor:

  ```
  sptk lpc -l 200 -m $lpc_order
  ```

- El resultado del pipeline se redirecciona a un fichero temporal, ubicado en el directorio `/tmp`, y cuyo nombre es el mismo que el del script seguido del identificador del proceso (de este modo se consigue un fichero temporal único para cada ejecución).

Una vez almacenado el resultado de la parametrización en un fichero temporal, hemos de almacenar la información en un fichero `fmatrix`, esto es: el número de filas y de columnas, seguidos por los datos. El número de columnas (igual al número de coeficientes) es fácil de calcular a partir del orden del predictor lineal: es igual a uno más el orden, ya que en el primer elemento del vector se almacena la ganancia de predicción. La obtención del número de filas (igual al número de tramas) es un poco más complicada, ya que depende de la longitud de la señal, la longitud y desplazamiento de la ventana, y la cadena de comandos que se ejecutan para obtener la parametrización. Por todo ello, es mejor, simplemente,

extraer esa información del fichero obtenido. Lo hacemos convirtiendo la señal parametrizada a texto, usando `sox +fa`, y contando el número de líneas, con el comando de UNIX `wc -l`.

Durante el proceso se generan ficheros temporales que deben ser cerrados al finalizar el programa. Si todo va bien, el programa se finaliza al final del script, pero, si hay algún error durante la ejecución, ésta finalizará antes de tiempo. Para garantizar que siempre se borran los ficheros temporales, creamos una función `cleanup()`, que es la que se encargará de realizar la limpieza, y utilizamos el comando `trap` para que se invoque la función siempre que se termine la ejecución, sea desde el punto que sea.

Como la señal parametrizada es un fichero `fmatrix`, podemos ver su contenido con el programa `fmatrix_show`:

```
usuario:~/PAV/P4$ fmatrix_show hola.lp
FMATRIX: hola.lp
[0]    3.98184  -1.71495  1.33577  -1.13519  0.88587  -0.64250  0.39266          ...
↪  -0.20401  0.09050
[1]    2.90145  -1.47158  0.92759  -0.79457  0.50280  -0.41136  0.30113          ...
↪  -0.08756  0.03795

...

[1639] 2.46503  -1.09709  0.09954  -0.19883  0.09305  -0.02337  0.07747          ...
↪  -0.00937  0.06214
```

3.4. Parametrización de una base de datos.

El script `wav2lp.sh` (y los scripts `wav2lpcc.sh` y `wav2mfcc.sh`, cuando los desarrolle) permite parametrizar una única señal. Sin embargo, en la mayor parte de las aplicaciones de reconocimiento del habla es necesario trabajar con bases de datos, a menudo de gran tamaño. Por ejemplo, la base de datos `spk_8mu/speecon`, que se usará a lo largo de esta práctica, incluye más de 3000 ficheros. Evidentemente, procesar todos estos ficheros uno a uno no resulta práctico.

Vamos a usar el script `run_spkid.sh` para automatizar la tarea de parametrización de las bases de datos utilizadas en la práctica, así como otras tareas relacionadas con el entrenamiento y el reconocimiento.

Si ejecutamos `run_spkid` sin argumentos, el programa escribe en pantalla su modo de empleo:

```
usuario:~/PAV/P4$ run_spkid
Empleo: /home/albino/PAV/bin/run_spkid command...

Where command can be one or more of the following (in this order):

     FEAT: where FEAT is the name of a feature (eg. lp, lpcc or mcp).
           - A function with the name compute_FEAT() must be defined.
           - Initially, only compute_lp() exists and can be used.
           - Edit this file to add your own features.

     train: train GMM for speaker recognition and/or verification
      test: test GMM in speaker recognition
  classerr: count errors in speaker recognition
 finaltest: reserved for final test
trainworld: estimate world model for speaker verification
    verify: test gmm in verification task
 verifyerr: count errors of verify
```

Podemos ver que `run_spkid` se utilizará para casi todas las tareas relacionadas con el reconocimiento y verificación del locutor. No se trata de un script particularmente largo (unas 150 líneas, de las que menos de 50 son realmente operativas), aunque puede abrumar un poco al usuario poco experto.

El script se invoca con uno o más comandos (manteniendo el orden en que aparecen). En todos los casos, salvo en la extracción de características, el nombre del comando es directamente el argumento que se ha de dar al programa. Dentro del script, una cláusula `if ... elif ... else` determina las órdenes que se han de ejecutar.

En el caso de la extracción de características, el funcionamiento es ligeramente distinto: debe invocarse con el nombre del parámetro (por ejemplo: `lpcc` o `mfcc`) y, dentro de `run_spkid.sh` se ejecutará la función `compute_FEAT()`, donde `FEAT` es el nombre del parámetro. Esta función debe existir antes de invocarse al programa. En su versión original, sólo está implementada la parametrización `lp` (*linear prediction*), dentro de la función `compute_lp()`:

```
77 compute_lp() {
78     for filename in $(cat $w/lists/class/all.train $w/lists/class/all.test); do
79         mkdir -p `dirname $w/$FEAT/$filename.$FEAT`
80         EXEC="wav2lp 8 $db/$filename.wav $w/$FEAT/$filename.$FEAT"
81         echo $EXEC && $EXEC || exit 1
82     done
83 }
```

Para ejecutar el script, primero hay que editarlo para que las variables `w`, `name_exp`, `lists` y `db` tengan el valor adecuado:

`w`: Directorio de trabajo de los experimentos. En él se almacenarán los resultados intermedios y finales (señales parametrizadas, modelos GMM, resultados del reconocimiento, etc.).

En su versión inicial es `w=work`, pero puede modificar este valor para tener experimentos con distintas parametrizaciones. Además, dependiendo de dónde haya instalado los directorios de la práctica y de desde dónde ejecute el script, puede tener que modificarlo.

`name_exp`: Nombre del experimento. Este nombre *personaliza* el directorio de los GMM y el nombre de los ficheros de resultados, de manera que, cambiándolo, puede tener distintos experimentos que usen el mismo directorio de trabajo y la misma parametrización.

`lists`: Directorio con las listas de señales para entrenamiento y reconocimiento. Inicialmente, `lists=lists`, y, probablemente, no tenga que cambiar este valor, salvo que modifique la estructura de directorios de la práctica, o desee ejecutar los programas desde un directorio distinto a `PAV/P4`.

`db`: Directorio con la base de datos de señales de voz.

En su versión inicial es `db=spk_8mu`, pero, como en el caso de `lists`, es posible que tenga que modificar este valor para que se adapte a su configuración.

La función extrae el nombre de las señales de los ficheros `$lists/class/all.train` y `$lists/class/all.test` y, para cada uno de ellos, invoca al script `wav2lp` con el número de coeficientes y el nombre de los ficheros de entrada y de salida.

Al ejecutar el script con el argumento `lp`, `run_spkid` calcula la parametrización de la base de datos, mostrando el comando ejecutado para cada fichero:

```
usuario:~/PAV/P4$ run_spkid lp
Sun Nov 3 19:30:04 CET 2019: lp ---
wav2lp 8 spk_8mu/speecon/BLOCK00/SES000/SA000S01.wav                    ...
 ↪  work/lp/BLOCK00/SES000/SA000S01.lp
```

```
wav2lp 8 spk_8mu/speecon/BLOCK00/SES000/SA000S02.wav ...
↪  work/lp/BLOCK00/SES000/SA000S02.lp
...

wav2lp 8 spk_8mu/speecon/BLOCK29/SES294/SA294S28.wav ...
↪  work/lp/BLOCK29/SES294/SA294S28.lp
Sun Nov 3 19:39:41 CET 2019
```

Como puede observarse, incluso con una parametrización sencilla como los coeficientes LP de orden 8, el script tarda un ratito en completar la tarea para toda la base dates (entorno a 15 minutos, en la *potente* máquina del profesor).

3.5. Información proporcionada por la parametrización.

Siempre que se propone una parametrización distinta hemos de considerar cuánta información proporcionan sus coeficientes. Aunque este análisis es bastante complejo, sí podemos extraer algunas conclusiones rápidamente de manera sencilla. Por ejemplo, que dos coeficientes estén muy correlados (o sea, sus valores forman una recta), implica que conociendo uno de los dos podemos determinar el valor del otro. Así que, con sólo uno de los dos, obtendríamos la misma información que con ambos. Ahora bien, lo mismo podemos decir de cualquier caso en que los valores formen una línea estrecha, aunque no sea recta.

El programa `pearson` calcula el coeficiente de correlación *Pearson* entre componentes de un vector de características, $p[i]$ y $p[j]$, utilizando la fórmula siguiente:

$$\rho_{ij} = \frac{E\left\{\left(p[i] - \overline{p[i]}\right)\left(p[j] - \overline{p[j]}\right)\right\}}{\sigma_{p[i]}\sigma_{p[j]}} \tag{7}$$

El coeficiente Pearson está acotado $-1 <= \rho_{ij} <= 1$. Un valor cercano a ± 1 implica una alta correlación entre componentes; es decir, el valor de una de ellas es fácilmente estimable a partir el valor de la otra. Por tanto, la información conjunta proporcionada por las dos componentes es prácticamente la misma que la proporcionada por sólo una.

Por el contrario, si $\rho_{ij} \approx 0$, entonces las dos componentes están poco correladas, y la información conjunta proporcionada por ambas es el *doble* de la proporcionada por sólo una de ellas.

Tareas:

- Estudie el fichero `scripts/wav2lp.sh` y comprenda cómo funciona.
- Aproveche `scripts/wav2lp.sh` como modelo para implementar dos parametrizaciones alternativas:

  LPCC: Coeficientes cepstrales de predicción lineal, que puede obtener usando el programa de SPTK `lpc2c` a partir de los ya obtenidos LP.

  MFCC: Coeficientes cepstrales en escala Mel (MFCC), que puede obtener usando el programa de SPTK `mfcc`.

  - Ambas parametrizaciones admiten y/o requieren más opciones que la predicción lineal, así que deberá construir los scripts de manera que acepten estas opciones como argumentos del programa.
    - Se recomienda la lectura del tutorial de Bimbot et al., *A Tutorial on Text Independent Speaker Verification*, o de las Transparencias de clase para seleccionar valores adecuados para los parámetros de la parametrización.

- También se recomienda leer el manual de SPTK para ver cómo pasarle estas opciones a los programas correspondientes.

- Edite el script `run_spkid.sh` y estudie su funcionamiento; defina los valores adecuados de `w`, `name_exp`, `lists` y `db` (probablemete, en esta primera ejecución no necesite modificar los valores iniciales).
- Construya las funciones `compute_lpcc()` y `compute_mfcc()` y úselas, junto a la proporcionada `compute_lp()`, para parametrizar la base de datos de reconocimiento del locutor.
- Represente gráficamente en el plano los coeficientes 2 y 3 de las tres parametrizaciones: LP, LPCC y MFCC, calculados para las señales de un par de locutores. Compare la distribución de los puntos y analice en qué caso le parece que tenemos más información, y en qué caso menos.
  - Puede serle conveniente convertir a texto el fichero de parámetros, con el programa `fmatrix_show` y seleccionar las columnas adecuadas usando el comando de UNIX `cut`. Por ejemplo, para guardar en un fichero los coeficientes 2 y 3 de los ficheros del locutor `SES017`, puede ejecutar:

    ```
    usuario:~/PAV/P4$ fmatrix_show work/lp/BLOCK01/SES017/*.lp |      ···
    ↪  egrep '^\[' | cut -f2,3 > lp_2_3.txt
    ```

    A continuación, puede representar los valores usando Python, MATLAB/Octave o GNUplot.
- Utilice el programa suministrado `pearson` y contraste los resultados que proporciona con los observados en el punto anterior.

4. Entrenamiento de los modelos acústicos.

Una vez seleccionadas los parámetros con los que se va a caracterizar cada señal, hemos de construir un modelo matemático que nos permita separar las distintas clases a reconocer, en este caso, los locutores. En un clasificador *Bayesiano*, esto es equivalente a estimar la función de densidad de cada clase.

Los modelos GMM, que pueden verse como una extensión de la función de densidad gaussiana, proporcionan un mecanismo muy potente para representar la función de densidad de poblaciones complejas como las debidas a las tramas de voz de los distintos locutores. En ellas, y debido a la propia naturaleza de la voz, compuesta de sonidos diferentes, podemos prever la presencia de zonas del espacio de mayor probabilidad y separadas unas de otras. Por este motivo, un modelo más sencillo, como podría ser la función de densidad gaussiana, que presenta un único máximo (*unimodal*) no resulta apropiado.

Para el manejo de los GMM se proporciona la clase `GMM`, definida en el fichero `gmm.h` e implementada *casi completamente* en los ficheros `gmm.cpp` y `gmm_vq.cpp`; este último, sólo se usará, en una segunda fase, para la inicialización de los GMM mediante cuantificación vectorial (VQ).

Tareas:

- Lea la documentación sobre la clase GMM. Entienda qué son las variables miembro y también las distintas funciones. Puede estudiar directamente la declaración `gmm.h` o la documentación que se genera en `~/PAV/html/P4/index.html`, al ejecutar `make doc`).
- En la definición del GMM en `gmm.cpp` complete el código que falta (busque los comandos Doxygen `\TODO`, y recuerde añadir un `\HECHO` comentando el trabajo realizado).

- Cálculo del logaritmo de la función de densidad estimada por el GMM para una señal.

```
float GMM::logprob(const fmatrix &data) const;
```

  Esta función debe calcular el logaritmo de la probabilidad del GMM para la secuencia de datos `data`, que es un matriz con formato `fmatrix`, con tantas filas como tramas de voz (`data.nrow()`), y tantas columnas como coeficientes tenga el vector de características (`data.ncol()`). La expresión `data[i]` devuelve el vector `C`, es decir `float *`, con los coeficientes de la trama `i`.

- Implementación del algoritmo *Expectation Maximization*:

```
int GMM::em(const fmatrix &data, unsigned int max_it, float
↪  inc_threshold, int verbose);
```

  Esta función debe realizar la estimación mediante el método *EM: expectation-maximization*. Básicamente, debe repetir, hasta la convergencia, los dos pasos, *E* (expectation) y *M* (maximization).

  E: El algoritmo calcula la probabilidad de las secuencias dado el modelo y el *reparto* de cada trama entre las distintas gaussianas del GMM (variable `weights`).

  M: Utilizando el reparto de las tramas, `weights`, el algoritmo recalcula los parámetros del modelo GMM: pesos, medias y varianzas.

  Este procedimiento garantiza que la verosimilitud aumenta en cada iteración, de manera que se alcanza un mínimo local con seguridad, pero en un número infinito de iteraciones. Utilizaremos dos criterios para considerar que estamos suficientemente cerca de la convergencia y finalizar el bucle: alcanzar un número máximo de iteraciones, o que el incremento en la función de verosimilitud, *logprob*, baje de un cierto umbral. Deberá implementar este último criterio.

- Complete el código del programa principal de entrenamiento, `gmm_train.cpp`.
  - Efectúe la estimación EM de los modelos GMM.
  - Inicialice los GMM de manera, en principio, aleatoria.
- Verá que en el programa `gmm_train`, y todos los demás de esta práctica, salvo `plot_gmm_feat.py`, las opciones y argumentos en línea de comandos no se gestionan usando `docopt`, sino usando el método *clásico*: la función `getopt()` de la librería estándar de C `stdlib`.
  - Estudie el modo de gestionar las opciones y el mensaje de sinopsis usados en el programa, y alégrese de haber conocido `docopt`.
- Ejecute el programa de entrenamiento y compruebe la convergencia. Puede comprobar y representar la evolución de la *verosimilitud* de los datos de entrenamiento en función de la iteración, cambiando los parámetros que regulan la convergencia del algoritmo *EM* (número de iteraciones, umbral de mejora), el método de inicialización, número de gausianas.

4.1. Uso de `gmm_train` y análisis de los modelos GMM.

Una vez completado y compilado el programa `gmm_train`, podemos ver su modo de empleo invocándolo sin opciones:

```
usuario:~/PAV/P4$ gmm_train
ERROR no list of files provided

Usage: gmm_train [options] list_of_train_files
Usage: gmm_train [options] -F train_file1 ...

Options can be:
  -d dir        Directory of the input files (def. ".")
  -e ext        Extension of the input files (def. "mcp")
  -g name       Name of output GMM file  (def. output.gmc)
  -m mix        Number of mixtures (def. 5)
  -N ite        Number of final iterations of EM (def. 20)
  -T thr        LogProbability threshold of final EM iterations (def. 0.001)
  -i init       Initialization method: 0=random, 1=VQ, 2=EM split (def. 0)
  -v int        Bit code to control "verbosity"; eg: 5 => 00000101)

In case you use initialization by VQ or EM split, the following options also apply:
  -n ite        Number of iterations in the initialization of the GMM (def. 20)
  -t thr        LogProb threshold for the EM iterations in initialization (def.  ...
↪  0.001)
```

Se debe invocar `gmm_train` para entrenar cada uno de los GMM que se van a usar en el sistema. Aparte de las opciones que gobiernan el algoritmo EM, que, en general, tienen valores por defecto razonables, deberemos proporcionar los argumentos que determinan los ficheros de entrada y salida:

- El entrenamiento siempre utiliza un conjunto de ficheros de señal parametrizada como entrada. El nombre de estos ficheros puede especificarse de dos modos:

  `list_of_train_files`:
  Es un fichero de texto con el nombre, sin extensión, de cada uno de las señales de entrenamiento. En las bases de datos habituales, este *nombre*, suele incluir la ubicación de la señal dentro de la base; por ejemplo, `BLOCK00/SES000/SA000S01`, y se combina con los argumentos `-d dir` y `-e ext` para formar el nombre completo del fichero: `dir/`*`nombre`*`.ext`.

  `-F train_file...`:
  Nombre de las señales de entrenamiento, igual que los contenidos en `list_of_train_files`, pero indicado directamente en la línea de comandos. El motivo de este modo de operación es que, a menudo, la fase de entrenamiento es costosa desde el punto de vista computacional. Por ello, suele ser interesante depurar los scripts usando un número limitado de señales. Esta opción permite realizarlo, sin afectar a las variables del entrenamiento, con una orden del tipo: `gmm_train ... -F $(head list_of_train_files)`.

- Como se ha indicado más arriba, el nombre completo de fichero conteniendo la señal parametrizada de entrada se construye encadenando el directorio, proporcionado por la opción `-d dir`, el nombre de la señal y la extensión, proporcionada por la opción `-e ext`. Así, por ejemplo, el proceso de parametrización realizado en la sección anterior debería haber producido, entre otros, el fichero `work/lp/BLOCK00/SES000/SA000S01.lp`.

- El resultado del entrenamiento es un fichero cuyo nombre viene dado por la opción `-g name`. En el caso del reconocimiento/verificación del locutor, este nombre debería reflejar el del locutor empleado para entrenar el modelo. Por ejemplo, `SES000.gmm`.

Por ejemplo, para entrenar los modelos del locutor `SES000`, ejecutaríamos la orden:

```
usuario:~/PAV/P4$ gmm_train -d work/lp -e lp -g SES000.gmm lists/class/SES000.train
DATA: 8290 x 9
GMM nmix=5      ite=0   log(prob)=-14.8628      inc=1e+34
GMM nmix=5      ite=1   log(prob)=-13.298       inc=1.56481
GMM nmix=5      ite=2   log(prob)=-10.7919      inc=2.50619
GMM nmix=5      ite=3   log(prob)=-9.53015      inc=1.26171
GMM nmix=5      ite=4   log(prob)=-8.76726      inc=0.762885
GMM nmix=5      ite=5   log(prob)=-8.37261      inc=0.394651
GMM nmix=5      ite=6   log(prob)=-8.17425      inc=0.198361
GMM nmix=5      ite=7   log(prob)=-8.06083      inc=0.11342
GMM nmix=5      ite=8   log(prob)=-7.99206      inc=0.0687671
GMM nmix=5      ite=9   log(prob)=-7.9424       inc=0.0496569
GMM nmix=5      ite=10  log(prob)=-7.91874      inc=0.0236683
GMM nmix=5      ite=11  log(prob)=-7.9052       inc=0.0135355
GMM nmix=5      ite=12  log(prob)=-7.89617      inc=0.0090313
GMM nmix=5      ite=13  log(prob)=-7.88729      inc=0.00888252
GMM nmix=5      ite=14  log(prob)=-7.8794       inc=0.00788498
GMM nmix=5      ite=15  log(prob)=-7.87602      inc=0.00338173
GMM nmix=5      ite=16  log(prob)=-7.87402      inc=0.00199842
GMM nmix=5      ite=17  log(prob)=-7.87273      inc=0.00128841
GMM nmix=5      ite=18  log(prob)=-7.87184      inc=0.0008955
```

Podemos observar que, aunque el número de iteraciones por defecto es 20, el algoritmo ha parado una antes, porque el incremento en `log(prob)` es menor al umbral 0.001 en la iteración `ite=18`.

4.2. Entrenamiento de los GMM del sistema de reconocimiento del locutor.

El script `run_spkid` permite automatizar la generación de los GMM para todos los locutores del sistema. Para ello, sólo hay que invocarlo con el comando `train`. Es necesario que la vriable de entorno `FEAT` indique el nombre de los parámetros a usar, lo cual podemos conseguir anteponiendo la definición de la variable a la invocación del script:

```
usuario:~/PAV/P4$ FEAT=lp run_spkid train
Sat Nov 9 10:47:17 CET 2019: train ---
SES000 ----
DATA: 8290 x 9
GMM nmix=4      ite=0   log(prob)=-14.8798      inc=1e+34
GMM nmix=4      ite=1   log(prob)=-13.8255      inc=1.05435
GMM nmix=4      ite=2   log(prob)=-10.7292      inc=3.09631

...

GMM nmix=4      ite=22  log(prob)=-6.6373       inc=4.05312e-05
GMM nmix=4      ite=23  log(prob)=-6.63726      inc=4.05312e-05

Sat Nov  9 10:48:02 CET 2019
```

Tareas:

- Localice la parte del script `run_spkid.sh` en la que se realiza el entrenamiento de los GMM y comprenda su funcionamiento.

- Entrene los modelos de los locutores de la base de datos usando las tres parametrizaciones disponibles.

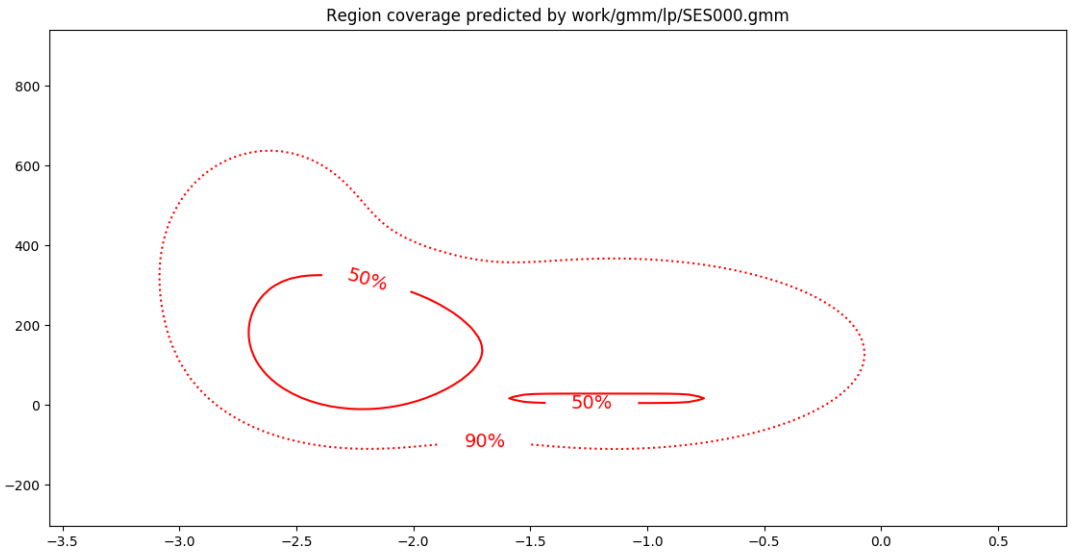
4.3. Visualización de los GMM.

El script Python `plot_gmm_feat.py` permite visualizar la función de densidad de probabilidad modelado por los GMM. Al ejecutarlo con la opción `-help`, nos muestra su modo de empleo:

```
usuario:~/PAV/P4$ plot_gmm_feat --help
Draws the regions in space covered with a certain probability by a GMM.

Usage:
    plotGMM [--help|-h] [options] <file-gmm> [<file-feat>...]

Options:
    --yDim INT, -x INT               'x' dim. from GMM and feature vectors       ...
↪   [default: 0]
    --xDim INT, -y INT               'y' dim. from GMM and feature vectors       ...
↪   [default: 1]
    --percents FLOAT..., -p FLOAT...  Percentages covered by the regions         ...
↪   [default: 90,50]
    --colorGMM STR, -g STR            Color of the GMM regions boundaries        ...
↪   [default: red]
    --colorFEAT STR, -f STR           Color of the feature population [default:  ...
↪   red]
    --limits xyLimits -l xyLimits     xyLimits are xMin,xMax,yMin,yMax           ...
↪   [default: auto]

    --help, -h                        Shows this message
```

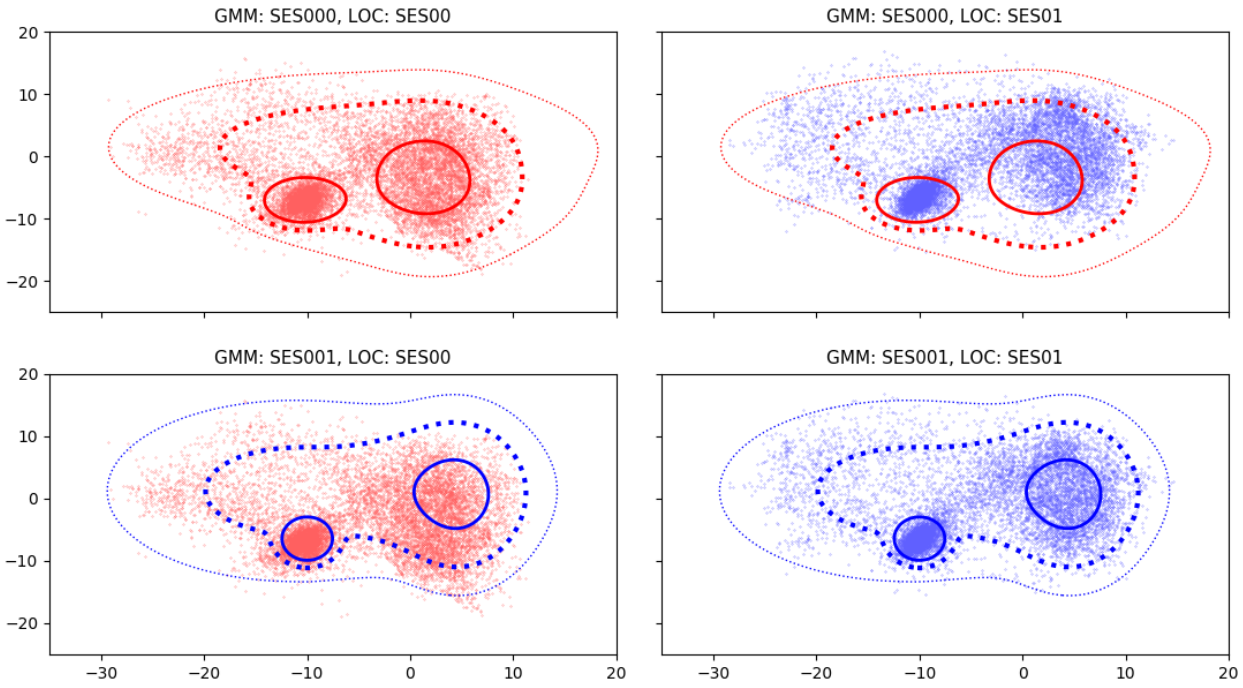
Lo ejecutamos para visualizar el modelo del locutor `SES000`:

```
usuario:~/PAV/P4$ plot_gmm_feat work/gmm/lp/SES000.gmm
```



En esta gráfica se muestran las regiones que abarcan un cierto tanto por ciento de la masa de probabilidad modelada por el GMM. Así, la curva punteada exterior encierra la región con el 90 % de las tramas con mayor valor de su densidad de probabilidad; mientras que las dos curvas continuas encierran la región con el 50 % de las tramas (fijémonos que, en este caso, la región está formada por dos zonas separadas, lo que muestra el carácter *multimodal* tanto de los GMM como de la propia población que modelan).

`plot_gmm_feat` también permite visualizar una población de señales, que puede coincidir o no con la que ha generado el modelo. Por ejemplo, la gráfica siguiente, obtenida con una versión modificada de `plot_gmm_feat` para poder usar *subplots*, permite analizar la adecuación de los modelos a los datos y su capacidad de discriminar entre locutores distintos:



En la gráfica podemos ver las regiones con el 99 %, 90 % y 50 % de la masa de probabilidad para los GMM de los locutores SES000 (en rojo, arriba) y SES001 (en azul, abajo); también se muestra la población del usuario SES000 (en rojo, izquierda) y SES001 (en azul, derecha).

Aunque sutilmente, se puede apreciar que el modelo de cada locutor se adapta mejor a sus datos que a los del otro. Cabe destacar que la zona de mayor densidad de probabilidad en la parte central inferior de las gráficas, y que es prácticamente idéntica para los dos locutores, se corresponde con las tramas de silencio, que, evidentemente, no pertenecen a ninguno de los dos locutores. Es la sutil diferencia, en las áreas distintas del ruido de fondo, la que permitirá usar los modelos para determinar si una señal pertenece a uno u otro locutor.

Tareas:

- Utilice el programa `plot_gmm_feat.py` para visualizar la función de densidad de probabilidad modelada por el GMM de un locutor cualquiera (para garantizar variedad, escoja uno que contenga el número de su puesto en el laboratorio). Represente los dos primeros coeficientes MFCC.
- Genere una gráfica como la de la página 20, que permite comparar los modelos y poblaciones de dos locutores distintos. Puede utilizar el programa `plot_gmm_feat.py`, o usarlo como modelo para implementar una versión más bonita (y que se valorará mejor).

  Como segundo locutor, escoja uno en el que el número de su puesto de laboratorio aparezca

en una posición distinta.

- Comente el resultado obtenido y discuta la utilidad de los GMM para determinar si una cierta señal pertenece a uno u otro locutor.

5. Reconocimiento y verificación del locutor usando GMM.

5.1. Reconocimiento del locutor.

En esta parte de la práctica se debe construir, optimizar y evaluar un sistema de reconocimiento del locutor basado en Modelos de Mezcla de Gaussianas (GMM).

Tareas:

- Complete el código necesario para reconocer el locutor en el fichero `src/gmm/gmm_classify.cpp`.
- Edite el fichero `run_spkid.sh` para efectuar el reconocimiento del locutor (comando `test` y la evaluación del error de clasificación (`classerr`).
- Reconozca y evalúe el resultado con la base de datos SPEECON.
- Analice los resultados y optimice los parámetros del sistema.
  - Tipo de características y parámetros que las gobiernan.
  - Número de componentes de los GMM.
  - Cualquier otro aspecto del sistema que considere oportuno mejorar o añadir, como la inclusión de características dinámicas, el análisis de componentes principales, la detección de actividad vocal, etc.
  - Métodos alternativos de estimación del GMM, como la adaptación del modelo universal.

Se recomienda leer A Tutorial on Text Independent Speaker Verification que, aunque trata de verificación del locutor, aporta ideas interesantes que son también útiles en reconocimiento.

5.2. Verificación del locutor.

En esta parte de la práctica, deberá aplicar los conocimientos adquiridos a desarrollar una de las aplicaciones más útiles de la identificación del locutor: la *verificación* del hablante que se utiliza en el control de acceso. En primer lugar, desarrollará un sistema de referencia, basado en la probabilidad de la señal de entrada mediante el GMM del (pretendido) usuario. Veremos que este *score* debe ser normalizado para que sea fácil utilizarlo en la decisión. Una vez finalizado el sistema base, queda a su propia iniciativa encontrar métodos para mejorar estas prestaciones. El objetivo es obtener el mínimo error en verificación y para ello debe ser capaz de elegir las estrategias más prometedoras considerando el compromiso esfuerzo/beneficio.

En verificación del locutor mediante habla, se utiliza su voz para comprobar si la voz es del supuesto usuario. Una aplicación típica es la seguridad en el control de acceso, como un *password* vocal.

El sistema, analizando la voz a su entrada y la información sobre el usuario, debe decidir si es el usuario legítimo o un *impostor*. Por tanto, los posibles errores pueden ser:

- Fallo de detección (*miss*): el habla correspondía al usuario pero se ha denegado el acceso.

- Falsa alarma: la señal de test no correspondía al candidato, sino que era un *impostor*, pero se ha autorizado el acceso.

Normalmente, los sistema generan para cada fichero de test, x un *score* s, que se utiliza para autorizar o no el acceso según el *score* supere o no un umbral t_h.

$$\begin{aligned} s < t_h &\quad \rightarrow \texttt{ACCESS\_DENIED} \\ s \geq t_h &\quad \rightarrow \texttt{ACCESS\_GRANTED} \end{aligned}$$

Si el umbral es muy pequeño, todos acceden, incluso los impostores. Si el umbral es muy alto, nadie puede acceder, ni los impostores ni los usuarios legítimos. El umbral debe ajustarse según la importancia que tenga cada error en la aplicación concreta.

Hay varias formas de evaluar y comparar sistemas (véase el *paper* proporcionado). En este caso, vamos a utilizar un coste definido como:

$$C = \frac{1}{\min(\rho, 1-\rho)}\big(\rho \cdot p_m + (1-\rho)\cdot p_{fa}\big)\cdot 100$$

Donde:

p_m	porcentaje de fallos del sistema (bloqueo) frente a usuarios legítimos
p_{fa}	porcentaje de fallos del sistema (acceso) frente a impostores
ρ	parámetro de definición del objetivo (*target*). En este caso, se ha elegido $\rho = 0.01$, que penaliza mucho los accesos sin autorización (una falsa alarma cuenta como 99 bloqueos a usuarios legítimos).

Por ejemplo, (si $\rho < 0.5$)

- Un sistema que siempre deniega el acceso ($p_m = 1, p_{fa} = 0$), tendrá un coste $C = 100$.
- Un sistema perfecto ($p_m = 0, p_{fa} = 0$) tendría coste nulo.

El objetivo de esta parte de la práctica es diseñar un sistema con el coste lo más bajo posible.

La primera idea sería que el *score* fuera la probabilidad del GMM del usuario, que en principio será más alta para el propio usuario que para el impostor. Sin embargo, este valor es muy variable con la señal y el hablante por lo que se suele normalizar. Una forma típica es comparar con la probabilidad de la señal respecto un modelo general, que llamaremos modelo del *mundo* o *background*:

$$s = \frac{f_{\theta_u}(x)}{f_{\theta_w}(x)}$$

Donde:

s	*score*, magnitud utilizada para decidir si se permite el acceso
$f_{\theta_u}(x)$	modelo mediante GMM de las características acústicas del usuario
$f_{\theta_w}(x)$	modelo mediante GMM de las características acústicas del conjunto de usuarios, *world*

Tareas:

1. El programa `gmm_verify.cpp` en el directorio `gmm_verify` está *casi* finalizado para una implementación básica, que utiliza un GMM tanto para modelar de cada usuario legítimo como también para el modelo del *mundo*. Debe finalizar la función `verify` para producir

el `score` s que permitirá, comparándolo con un *umbral*, si se permite el acceso o no.

Puede generar en una primera versión un *score* basado únicamente en el GMM del usuario para, más adelante, normalizar utilizando la probabilidad del GMM del *mundo* y así validar si es o no importante la normalización del *score*.

2. Modifique adecuadamente `run_spkid.sh` para realizar la verificación del locutor.

   - Añada un comando para estimar el modelo general. Puede utilizar los datos de entrenamiento que considere conveniente: `users.train`, `others.train` o `users_and_others.train`.
   - Añada también un comando para la evaluación, que ejecute `gmm_verify`. Para la evaluación deberá utilizar el fichero `all.test` (ficheros de test de usuarios e impostores) así como `all.test.candidates`, que son las supuestas declaraciones que realizan los usuarios legítimos o los impostores.
   - `gmm_verify` debe calcular el logaritmo de la verosimilitud de la señal de test, tanto para el modelo del supuesto locutor como para el modelo de referencia *mundo*, y escribir la diferencia entre ambos en pantalla.

     En el script `run_spkid` debe redirigir esta salida a un fichero que se le pasará al programa Perl `spk_verif_score.pl` para calcular la precisión de la verificación. Este script puede invocarse con un argumento adicional, el umbral de admisión, en este caso, informará de la tasa de error alcanzada suponiendo ese valor del umbral. En caso de no especificarse el umbral, `spk_verif_score.pl` prueba distintos valores del umbral y selecciona el valor con menor función de coste. Este valor del umbral es que deberá usar en la evaluación final.

5.3. Ampliación y mejoras.

Una vez finalizado el sistema de referencia, la parte final consiste en implementar y evaluar variantes de este sistema de referencia. Queda a su iniciativa las propuestas de mejora, aunque se propondrán opciones que típicamente han utilizado estos sistemas.

Se recomienda que consulte A Tutorial on Text Independent Speaker Verification. Las mejoras pueden venir de múltiples aspectos, desde el llamado *front-end* (características dinámicas, pitch, detección de silencio, etc.), la normalización del *score*, el mejor modelado del *mundo* y de los *usuarios*, etc.

Por ejemplo, con las herramientas y el conocimiento que dispone podría desarrollar un nuevo programa en `C++`, para implementar la adaptación de GMM siguiendo las directrices de la sección 3.3 del mencionado *paper*. Esto le permitirá afianzar el conocimiento de desarrollo, realizando una herramienta que es parte de un sistema de cierta complejidad.

6. Ejercicios y entrega.

En esta práctica no deberá entregar memoria. En lugar de ello, deberá completar los ejercicios indicados en el documento `README.md`, escribiendo los resultados en el mismo. Se valorará el uso adecuado del formato *markdown* y la calidad visual del documento generado. También deberá dar respuesta en los distintos códigos fuente de la práctica a cada comando Doxygen `\TODO` con su correspondiente `\HECHO` (o, si lo prefiere, `\DONE`).

Mediante *pull-request* deberá subir a GitHub el repositorio de la práctica, que deberá contener los códigos fuente necesarios para reproducir los resultados publicados en el documento `README.md`. Éstos deberán estar preparados para su compilación y ejecución sin mayor intervención por parte del profesor que adaptar los directorios de instalación y de las bases de datos orales. En concreto, el sistema deberá proporcionar los resultados publicados ejecutando las órdenes siguientes:

```
usuario:~/PAV/P4/fulano$ git checkout fulano-mengano
usuario:~/PAV/P4/fulano$ make release
usuario:~/PAV/P4/fulano$ run_spkid mfcc train test classerr trainworld verify verifyerr
```

El repositorio deberá contener, además, los dos ficheros siguientes:

class_test.log:
Con el resultado del reconocimiento de los ficheros de la base de datos de reconocimiento `sr_test/spk_cls`.

verif_test.log:
Con el resultado de la verificación del locutor para los ficheros de la base de datos de verificación `sr_test/spk_ver`.

- En este fichero, el formato es ligeramente distinto al usado hasta ahora. En él deberá escribir, en columnas separadas por tabulador, el nombre del fichero, el nombre del usuario alegado por el locutor, y el valor 0 ó 1 en función de si el locutor es un impostor (0) o es el usuario legítimo (1):

  ```
  spk_ver/v00/v0000 SES146 1
  spk_ver/v00/v0001 SES002 0
  ...
  ```

  Para producir este tipo de fichero puede tener que modificar el programa `gmm_verify.cpp`. Alternativamente, puede dejar inalterado `gmm_verify.cpp` y ejecutar el siguiente script de Perl (en el que suponemos que el umbral de admisión es -3.214):

  ```
  perl -ane 'print "$F[0]\t$F[1]\t";
          if ($F[2] > -3.214) {print "1\n"}
          else {print "0\n"}' verification.log > verif_test.log
  ```

En ambos casos, las bases de datos son desconocidas para el alumno. La evaluación la realizarán los profesores de la asignatura y su resultado tendrá un peso importante en la nota de la práctica.

6.1. Trabajo de ampliación.

El alumno deberá presentar los resultados de un trabajo de ampliación en el cual deberá realizar, de **manera autónoma**, una profundización o extensión de las tareas realizadas durante el desarrollo de la práctica. La elección de la temática de este trabajo queda a elección del alumno. A continuación se enumeran distintas tareas que pueden realizarse o servir de referencia para otras. En caso de realizarse una tarea distinta a las enumeradas, se ruega que el alumno se ponga en contacto con los profesores de la asignatura para que estos valoren su viabilidad e interés, y den su visto bueno.

Las tareas están ordenadas de menor a mayor grado de dificultad; entendiéndose que la repercusión en la nota del trabajo estará altamente correlada con la dificultad del trabajo realizado. Cabe señalar que, aunque conceptualmente más sencillas, las primeras tareas propuestas pueden representar una carga computacional elevada.

- Optimización de los parámetros de un sistema basado en LPCC (cepstrum LPC): orden LPC, número de coeficientes cepstrales, preénfasis, duración y desplazamiento del tramo, tipo de ventana...
- Optimización de los parámetros de un sistema basado en MFCC y/o MCEP.

- Análisis y estudio del modelado GMM:
  - Influencia en la verosimilitud del método de inicialización, número de gaussianas, número de iteraciones, umbral de convergencia, etc.
  - Validación cruzada durante el entrenamiento, apartando del mismo una parte del material de entrenamiento. Este material no participará en el entrenamiento, pero sí que se usará para, justamente, estimar la evolución de la verosimilitud en un conjunto de datos independiente.

    También puede usarse la validación cruzada para seleccionar el número de gaussianas por GMM, aumentando éste en tanto la verosimilitud en el conjunto de validación siga creciendo.
  - Estudio estadístico de la variabilidad en las prestaciones obtenidas por el modelado GMM, realizando múltiples experimentos con los distintos tipos de inicialización, pero distintas semillas del generador de números aleatorios.

- Incorporación de las características dinámicas (Δ y Δ^2). Implementar su cálculo usando SPTK y estudiar las prestaciones obtenidas al usar estas características de forma conjunta, usando vectores de características de mayor tamaño, o separada, construyendo sistemas independientes para cada característica y combinando las verosimilitudes de cada uno para tomar la decisión final del sistema de reconocimiento y/o verificación.

  En el caso de los sistemas independientes para la característica estática y las dinámicas, estudio del método de combinación de los resultados de cada uno. Las alternativas típicas son la suma de los logaritmos de la verosimilitud de cada información, o su promedio ponderado. En este último caso, los coeficientes de ponderación se pueden ajustar manualmente, a partir de los resultados de la experimentación, o usando regresión lineal.

- Empleo de los resultados de prácticas anteriores para analizar efecto de prescindir de los segmentos de silencio, o de los segmentos o sonoros.
  - En este último caso, estudio de la inclusión del pitch en las características modeladas. Suele utilizarse el $\log(F0)$ debido a que su función de densidad de probabilidad es aproximadamente gaussiana.

    Como en el caso de las características dinámicas más arriba, esta inclusión puede realizarse suponiendo independencia entre las informaciones o ampliando el vector de características con la nueva información.

- Análisis discriminativo lineal (LDA). Este método se emplea para capturar la máxima información de los vectores de características con el mínimo número de componentes posible. El modo de empleo suele consistir en ampliar los vectores de características con sus vecinos, reduciendo a continuación la dimensionalidad de los vectores ampliados usando como criterio de selección de las componentes su capacidad de discriminación lineal de las clases a reconocer.

- Clasificación utilizando DeepLearning. Se parte de un sistema básico para clasificación del locutor, escrito en Python utilizando la librería PyTorch, pav_spkid_pytorch. En primera instancia, deberán optimizarse los parámetros que lo gobiernan: número de neuronas, número y tamaño de las capas ocultas, longitud del contexto, función activación, etc. En el propio repositorio del sistema puede encontrar información detallada de su funcionamiento y de las posibilidades de experimentación con él.

  En el caso de optarse por esta ampliación, será imprescindible extender el sistema para que realice verificación del locutor. Por este motivo, no se recomienda salvo que se tengan conocimientos previos de Python (aunque el nivel necesario puede alcanzarse en poco tiempo).

- También es posible realizar como trabajo de ampliación la realización de una tarea distinta a las de reconocimiento/verificación del locutor, pero estrechamente relacionada con éstas: la agrupación (*clustering*) de locutores. En esta tarea se parte de una señal en la que hay distintos locutores (se partiría de una ordenación aleatoria de las señales de la base de datos),

de los que se desconoce su número e identidad. El objetivo es determinar cuántos locutores hay y a cuál de ellos corresponde cada trama de señal. En los apartados 4 y 2 del paper de S. Chen y P.S. Gopalakrishnan *"Speaker, environment and channel change detection and clustering via the Bayesian information criterion"*, podrá encontrar más información acerca de este tipo de tarea utilizando el criterio BIC.

El trabajo de programación desarrollado durante la fase de ampliación se incluirá en una rama distinta del repositorio, de manera que los sistemas puedan ser evaluados de manera independiente a los de la práctica básica (cuya rama tendrá un nombre formado a partir de los primeros apellidos de los integrantes del grupo de laboratorio). Escoja un nombre descriptivo del objeto de experimentación en el nombre de la rama; por ejemplo, `optimiza-params`, o `prueba-inic`, etc.

6.1.1. Entrega de los trabajos de ampliación.

Los códigos fuente correspondientes a los trabajos de ampliación deberán ir en una o más ramas de su rama principal del repositorio (la que se nombra a partir de los primeros apellidos de los integrantes del grupo de prácticas). El nombre de las ramas de ampliación deberá ser descriptivo del trabajo contenido en ellas y estar referenciado en la memoria.

Deberá subir a Atenea un fichero comprimido en formato zip o tgz con los elementos siguientes:

- Memoria, del estilo de las realizadas en las prácticas anteriores, en la que explique los trabajos realizados en la ampliación y los resultados obtenidos.
- Ficheros `class_ampl.log` y/o `verif_ampl.log` con los resultados de la evaluación *ciega* realizada con los sistemas desarrollados en la ampliación. Si desarrolla más de un trabajo de ampliación, dele un nombre descriptivo adecuado; por ejemplo: `class_LDA.log`, `verif_DNN.log`, etc.

ANEXOS.

I. Mantenimiento de la práctica usando `meson` y `make`.

La estrategia seguida para mantener las prácticas precedentes, consistente en un único fichero `meson.build` que construye todos los elementos cada vez que invocamos `ninja`, es válida para proyectos pequeños como la práctica P2 y, en menor medida, la P3. Pero en proyectos de mayor envergadura, como esta P4 o la siguiente P5, resulta ineficiente y difícil de mantener.

Para simplificar el mantenimiento del proyecto, así como para aumentar su eficiencia, se han introducido una serie de modificaciones que se detallan en las secciones siguientes.

I.A. Empleo de `make`.

Aunque la herramienta fundamental que se usará en el mantenimiento de la práctica seguirá siendo Meson/Ninja, se va a emplear `make` debido a que esta herramienta permite o simplifica toda una serie de cuestiones:

- Desde `make` es posible acceder a las variables de entorno del sistema, algo que Meson no permite, al menos de un modo sencillo.

  Al poder acceder, por ejemplo, a la variable de entorno `$HOME`, seremos capaces de instalar los programas, librerías y demás en el directorio `$HOME/PAV`, sin necesidad de editar ningún fichero para que el usuario lo especifique explícitamente.

- `make` simplifica mucho la construcción parcial del proyecto. Esto es: seleccionar qué partes se compilan y con qué opciones. Esto es así por dos motivos: por un lado, si no se especifica ningún objetivo concreto, `make` sólo construye el primero que encuentra en el fichero `Makefile`; por el contrario, Meson/Ninja construye todos los objetivos presentes en el fichero `meson.build`. Por otro lado, en `make` es muy sencillo definir objetivos que siempre se actualizarán (siempre están desfasados) con el comando `.PHONY`.

- `make` permite simplificar mucho la invocación de comandos Meson/Ninja complejos, que, usando Meson/Ninja directamente, deberían recordarse y escribir sin error.

  Por ejemplo, usando `make`, la orden fundamental para construir los programas del proyecto es `make release`; mientras que, usándolos directamente, la misma tarea, utilizando Meson/Ninja, se debe invocar como:

```
usuario:~/PAV/P4$ meson --buildtype=release --prefix=$HOME/PAV --libdir=lib bin/release
usuario:~/PAV/P4$ ninja install bin/release
```

Así pues, usaremos `make` como *front-end* para Meson/Ninja con el siguiente fichero de reglas:

Makefile

```
# PREFIX overrides the 'prefix' option of Meson's function project() on 'ninja install'.
PREFIX = ${HOME}/PAV

BUILD_RELEASE = bin/release
BUILD_DEBUG   = bin/debug
FILE_RELEASE  = ${BUILD_RELEASE}/build.ninja
```

```
FILE_DEBUG    = ${BUILD_DEBUG}/build.ninja

.PHONY: help release debug all clean_release clean_debug clean doc

help:
    @echo '-------------------------------------------------------------------'
    @echo 'Usage:'
    @echo '  make release       : create "bin-&-lib" of the release version'
    @echo '  make debug         : create "bin-&-lib" of the debug version '
    @echo '  make all           : make debug and release'
    @echo ' '
    @echo '  make clean_release : remove the "release" intermediate files'
    @echo '  make clean_debug   : remove the "debug"   intermediate files'
    @echo '  make clean         : make clean_debug and clean_release'
    @echo ' '
    @echo '  make doc           : generate the documentation of the project'
    @echo '-------------------------------------------------------------------'

release: ${FILE_RELEASE}
        ninja install -C ${BUILD_RELEASE}

${FILE_RELEASE}:
        meson --buildtype=release --prefix=${PREFIX} --libdir=lib ${BUILD_RELEASE}

clean_release:
        \rm -rf ${BUILD_RELEASE}

debug: ${FILE_DEBUG}
        ninja install -C ${BUILD_DEBUG}

${FILE_DEBUG}:
        meson --buildtype=debug --prefix=${PREFIX} --libdir=lib ${BUILD_DEBUG}

clean_debug:
        \rm -rf ${BUILD_DEBUG}

all: release debug

clean: clean_release clean_debug

doc:
        ninja doc -C ${BUILD_RELEASE}
```

I.B. Estructura de subdirectorios.

En este proyecto, Meson/Ninja se usa de manera distribuida entre los subdirectorios que forman el código fuente. Un fichero `meson.build` en el directorio principal del proyecto se encarga de importar, mediante el comando `subdir()`, los ficheros de dependencias correspondientes a:

- Librería `libpav.a`, con las funciones y clases genéricas desarrolladas en la asignatura y que son usadas por múltiples proyectos.

- Programas de la práctica, repartidos entre los subdirectorios `src/gmm`, `src/pearson` y `src/fmatrix`.
- Scripts de la práctica, almacenados en el directorio `src/scripts`.
- Documentación del proyecto, a partir de los ficheros de configuración en `src/doxyfile`.

meson.build

```
project(
    'Práctica 4 de PAV - reconocimiento y verificación del locutor', 'cpp',
    default_options: 'cpp_std=c++11',
    version: 'v2.0'
)

inc = include_directories(['src/include', 'src/pav/docopt_cpp'])

# Librería PAV
subdir('src/pav')

# Programas del proyecto
subdir('src')

# Scripts de la práctica
subdir('scripts')

# Documentación del proyecto
subdir('src/doxyfile')
```

La función `subdir(`*`directorio`*`)` es, a casi todos los efectos, equivalente a insertar el contenido del fichero *`directorio`*`/meson.build` en el punto de su invocación por el `meson.build` principal. Por este motivo, los diferentes ficheros *`directorio`*`/meson.build` no deben incluir definición de proyecto (usarán la definición del principal). Por otro lado, todas la variables y dependencias definidas previamente por el fichero `meson.build` son accesibles desde los ficheros `directorio/meson.build` subordinados; y todas las variables y dependencias definidas en éstos son accesibles por el `meson.build` principal, y sus subordinados, a partir del punto de la invocación de `subdir()`.

La principal diferencia entre invocar `subdir(`*`directorio`*`)` o incluir el contenido de *`directorio`*`/meson.build` directamente en el fichero `meson.build` es que `ninja` será consciente de que el fichero `meson.build` está en un subdirectorio de `src`, en lugar de en el directorio principal del proyecto, y generará los programas y librerías en `bin/src` en lugar de en `bin`.

I.B.A. Programas de la práctica.

Como tenemos ficheros en distintos subdirectorios, el fichero `meson.build` que los generará estará en el directorio raíz de todos ellos (`src`), y su contenido es:

src/meson.build

```
# Programas del proyecto

sources = [
    'gmm/gmm_classify.cpp',
    'gmm/gmm_show.cpp',
    'gmm/gmm_train.cpp',
    'gmm/gmm_verify.cpp',
    'pearson/pearson.cpp',
    'fmatrix/fmatrix_cut.cpp',
```

```
    'fmatrix/fmatrix_show.cpp',
]

foreach src: sources
    executable(
        src.split('/')[-1].split('.')[0],      # Nombre sin directorio ni extensión
        sources: src,
        include_directories: inc,
        link_args: ['-lm', '-lsndfile'],
        link_with: lib_pav,
        install: true,
    )
endforeach
```

I.B.B. Scripts de la práctica.

En el caso de los scripts de la práctica, al ejecutar `make release` o `make debug`, los scripts se enlazan simbólicamente en `~/PAV/bin` con el mismo nombre que el script original, pero sin su extensión.

Como son enlaces simbólicos, cualquier cambio en los códigos fuente en el directorio `scripts` se reflejan automáticamente en los ejecutables en `~/PAV/bin`, así que no hace falta volver a ejecutar `ninja`.

scripts/meson.build

```
# Scripts del proyecto

scripts = [
    'run_spkid.sh',
    'plot_gmm_feat.py',
    'spk_verif_score.pl',
    'wav2lp.sh',
]

foreach script: scripts
    source = join_paths(meson.source_root(), 'scripts', script)      # Ruta completa
    prog = script.split('.')[0]                                      # Nombre sin extensión
    dest = join_paths(get_option('prefix'), get_option('bindir'), prog)
    custom_target(script,
        input : source,
        output : prog,
        command : ['ln', '-sf', source, dest],
        build_by_default: true,
    )
endforeach
```

I.C. Generación de la documentación con Doxygen en `~/PAV/html/P4`.

Como en el caso de la tercera práctica, se usará la documentación generada por Doxygen para gestionar las modificaciones a realizar en los programas de la práctica (lista de `TODO`s y `HECHO`s). En este caso, sin embargo, no se generará documentación nueva cada vez que se modifica un programa del proyecto, porque esto representaba una carga computacional excesiva durante la fase de desarrollo. En vez de

ello, la documentación sólo se generará al usar la opción `doc` de `ninja`, en cuyo caso se instalará en el directorio `~/PAV/html/P4`[1].

src/doxygen/meson.build

```
# Documentación usando Doxygen

items = [
    'README.md',
    'src',
    'scripts',
]

inputs = ''
foreach item : items
    inputs += '"' + join_paths(meson.source_root(), item) + '" '
endforeach

doxygen = find_program('doxygen')
practica = meson.source_root().split('/')[-1]
html = join_paths(get_option('prefix'), 'html')

config_DG = configuration_data()
config_DG.set('project_name', meson.project_name())
config_DG.set('project_version', meson.project_version())
config_DG.set('inputs', inputs)
config_DG.set('install_dir', html)
config_DG.set('dir_html', practica)
config_DG.set('style_sheet', join_paths(meson.source_root(), 'src/doxyfile', ...
↪  'style.css'))

doxyfile = configure_file(
    input: 'Doxyfile.in',
    output: 'Doxyfile',
    configuration: config_DG,
)

doc = run_target('doc', command: [doxygen, doxyfile])
```

Nótese que, mientras que en la tercera práctica se usaba el comando `custom_target()` para generar la documentación, en ésta utilizamos el comando `run_target()`. La diferencia entre ambos es que ahora se evita generar la documentación salvo que se pida expresamente el objetivo `doc` al invocar `ninja`.

I.D. Librería `libpav.a`.

Finalmente, el `meson.build` para reconstruir la librería `libpav.a` es idéntico al de la práctica P3 con el añadido de los dos nuevos códigos fuente introducidos en esta práctica: `gmm.cpp` y `gmm_vq.cpp`.

src/pav/meson.build

```
# Librería PAV

lib = 'pav'
src = [
```

[1] Hubiera estado bien hacerlo de este modo ya en aquella práctica; de este modo, en el directorio `~/PAV/html` dispondríamos de la documentación de todas las prácticas (a partir de la P3).

```
    'digital_filter.cpp',
    'filename.cpp',
    'wavfile_mono.cpp',
    'keyvalue.cpp',
    'gmm.cpp',
    'gmm_vq.cpp',
    'docopt_cpp/docopt.cpp',
]

lib_pav = static_library(
    lib,
    sources: src,
    include_directories: inc,
    install: true,
)
```