Práctica 4: guía para el desarrollo del código fuente

Profesores de la asignatura

noviembre de 2022

Índice

Como en el caso de la P3, el código a desarrollar en la P4 consiste en localizar y completar los \TODOs repartidos por el proyecto. Tenemos dos tipos principales de \TODOs: los presentes en los scripts (`run_spkid.sh` y `wav_lp.sh`) y los presentes en el código C++. De estos últimos hay siete repartidos en cuatro ficheros de dos directorios.

1. Entrenamiento de los modelos GMM.

Una vez compilados en instalados los programas de la práctica, los tres programas principales (`gmm_train`, `gmm_classify` y `gmm_verify`) están disponibles. Sin embargo, ninguno de ellos hace nada de provecho...

El primer programa que se tiene que completar para poder seguir adelante es el que entrena los GMM, `gmm_train`. Podemos ver el modo de empleo invocando el programa sin argumentos:

```
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        LogProbability threshold for the EM iterations in initialization  ···
↪  (def. 0.001)
```

Podemos hacer un entrenamiento de prueba usando las señales de un locutor y proporcionado los valores adecuados a cada una de las opciones del programa. Entrenaremos el modelo del locutor SES000 cuyas señales de entrenamiento están listadas en el fichero `lists/class/SES000.train`. La parametrización que usaremos en primera instancia es la `LP`, cuyos ficheros están en el directorio `work/lp` y tienen la extension `.lp`. Llamaremos al modelo generado `prueba.gmm`. El resto de opciones, por ahora, las podemos dejar en sus valores por defecto:

```
usuario:~/PAV/P4/$ gmm_train -d work/lp -e lp -g prueba.gmm lists/class/SES000.train
DATA: 3075 x 9
```

El programa sólo nos informa de que el material de entrenamiento está formado por 3075 vectores de dimensión 9. En el directorio de trabajo podremos encontrar el fichero con el modelo, `prueba.gmm`. Para visualizar su contenido hemos de usar el programa `gmm_show`:

```
usuario:~/PAV/P4/$ gmm_show prueba.gmm
GMM: prueba.gmm
GMM: nmix=0; vector_size=0
```

O sea, el GMM no contiene ninguna gaussiana (que, igualmente, sería de dimensión cero). Esto es debido a que faltan varios \TODOs por completar: la inicialización del GMM, la invocación del algoritmo de *expectation maximization* (EM), y la propia implementación del EM a partir de las fases de *expectation* (E) y *maximization* (M).

1.1. 1er \TODO: inicialización de los modelos GMM.

En primer lugar, hemos de construir el modelo inicial, que se corresponde al primer \TODO en el fichero `src/gmm/gmm_train.cpp`. En éste \TODO debemos inicializar el objeto `GMM gmm` usando uno de los métodos disponibles. En primera instancia, `case 0:`, usaremos la inicialización aleatoria usando el método `gmm.random_init()` con los argumentos adecuados (un editor potente, como Visual Studio Code o similar, nos ayudará enormemente a seleccionar estos argumentos). Evidentemente, uno de los argumentos de la inicialización será las señales de entrenamiento almacenadas en la `fmatrix data`. El resto de argumentos los deberemos vincular a los valores proporcionados en la invocación en línea de comandos y analizados en la función `read_options()`.

Usando la inicialización aleatoria, ya es posible obtener modelos GMM, aunque sean aleatorios:

```
usuario:~/PAV/P4/$ gmm_train -d work/lp -e lp -g prueba.gmm lists/class/SES000.train
DATA: 3075 x 9
usuario:~/PAV/P4/$ gmm_show prueba.gmm
GMM: prueba.gmm
GMM: nmix=5; vector_size=9
w[0]=   0.206504
mu[0]=  378.601   -1.01326     1.0416     -0.756865     0.576065      ...
sig[0]= 434.352    0.63651     0.567521    0.700258     0.600669      ...

w[1]=   0.188943
mu[1]=  390.564   -0.996871    1.04301    -0.752192     0.574105      ...
sig[1]= 441.004    0.642671    0.59161     0.700946     0.603452      ...

w[2]=   0.204553
mu[2]=  415.112   -1.02632     1.05193    -0.770152     0.560667      ...
sig[2]= 455.574    0.608472    0.598393    0.704525     0.642725      ...

w[3]=   0.202927
mu[3]=  375.416   -1.05223     1.08558    -0.817056     0.60373       ...
sig[3]= 411.574    0.601718    0.593291    0.728626     0.693162      ...

w[4]=   0.197073
mu[4]=  391.666   -1.01852     1.03838    -0.713415     0.524287      ...
sig[4]= 426.217    0.63594     0.561228    0.677126     0.571668      ...
```

La información que nos proporciona `gmm_show` es el peso, `w[i]`, el vector con la media, `mu[i]`, y el vector con la varianza, `sig[i]`, de cada una de las gaussianas que forman el GMM.

1.2. 2o \TODO: invocación del algoritmo EM.

Para que los modelos GMM *aprendan* de las señales de entrenamiento es necesario completar el segundo \TODO del fichero `src/gmm/gmm_train.cpp`, añadiendo la llamada al método `gmm.em()` con los argumentos adecuados. Nuevamente, además de las señales de entrenamiento, almacenadas en la `fmatrix data`, deberemos determinar el nombre de los argumentos a partir de las variables proporcionadas por la función `read_params()`, invocada con anterioridad.

1.3. 3er \TODO: implementación del algoritmo EM.

En cualquier caso, la invocación del método `gmm.em()` por sí sola no cambiará nada, ya que el propio método está por construir... La reestimación de modelos de mezcla de gaussianas usando el algoritmo EM se basa en dos etapas independientes, cada una de las cuales mejoran el modelado:

***Expectation* (E):**
: En esta fase se determina el mejor reparto de cada trama de la señal entre las distintas gaussianas que forman el GMM. La fase E está implementada en el método `GMM::em_expectation()`, definido en el fichero `src/pav/gmm.cpp`. El método calcula el mejor reparto de cada trama entre las gaussianas del modelo y lo devuelve en el argumento `weights`. Además, devuelve el logaritmo de la verosimilitud del conjunto de tramas dado el modelo.

***Maximization* (M):**
: En esta fase se usa el reparto calculado en la fase E para reestimar los parámetros del GMM. La fase M está implementada en el método `GMM::em_maximization()`, también definido en el fichero `src/pav/gmm.cpp`.

Las dos fases son independientes: dado un modelo, calculamos el mejor reparto en la fase E, pero ese reparto permite calcular un nuevo GMM en la fase M que será mejor que el original, pero ese modelo nuevo tendrá un reparto óptimo distinto, que dará lugar a un nuevo modelo óptimo, y así sucesivamente. De este modo, mediante iteración de las dos fases, es posible alcanzar un óptimo de la verosimilitud, al menos localmente. En principio, suponiendo una precisión de los cálculos infinita, el óptimo local sólo se alcanza después de infinitas iteraciones del algoritmo. A efectos prácticos, suele ser conveniente parar cuando la mejora entre iteraciones sucesivas es despreciable. Por este motivo es habitual usar un umbral de mejora como criterio de parada.

Tanto `GMM::em_expectation()` como `GMM::em_maximization()` están ya implementados. Lo que falta, en el segundo \TODO del fichero `src/pav/gmm.cpp`, es la combinación de los dos para implementar el método `GMM::em()`.

1.4. Primer entrenamiento auténtico de los GMM.

Una vez resueltos los tres primeros \TODOs, ya es posible realizar un entrenamiento de los GMM de verdad:

```
usuario:~/PAV/P4/$ gmm_train -d work/lp -e lp -g prueba.gmm lists/class/SES000.train
DATA: 3075 x 9
GMM nmix=5      ite=0   log(prob)=-12.6469      inc=1e+34
GMM nmix=5      ite=1   log(prob)=-11.5015      inc=1.14541
GMM nmix=5      ite=2   log(prob)=-10.6587      inc=0.842779
GMM nmix=5      ite=3   log(prob)=-10.3575      inc=0.301168
GMM nmix=5      ite=4   log(prob)=-10.1422      inc=0.215381
GMM nmix=5      ite=5   log(prob)=-9.97706      inc=0.165098
GMM nmix=5      ite=6   log(prob)=-9.87999      inc=0.097065
GMM nmix=5      ite=7   log(prob)=-9.81184      inc=0.0681543
GMM nmix=5      ite=8   log(prob)=-9.73903      inc=0.0728064
GMM nmix=5      ite=9   log(prob)=-9.67009      inc=0.0689411
GMM nmix=5      ite=10  log(prob)=-9.62126      inc=0.04883
GMM nmix=5      ite=11  log(prob)=-9.58401      inc=0.0372496
GMM nmix=5      ite=12  log(prob)=-9.55252      inc=0.0314913
GMM nmix=5      ite=13  log(prob)=-9.52331      inc=0.029213
GMM nmix=5      ite=14  log(prob)=-9.50029      inc=0.0230141
```

```
GMM nmix=5      ite=15  log(prob)=-9.48363      inc=0.0166616
GMM nmix=5      ite=16  log(prob)=-9.4714       inc=0.0122318
GMM nmix=5      ite=17  log(prob)=-9.46152      inc=0.00988483
GMM nmix=5      ite=18  log(prob)=-9.45343      inc=0.00808048
GMM nmix=5      ite=19  log(prob)=-9.44732      inc=0.00611782
```

Puede observarse como la verosimilitud (`log(prob)`) aumenta en cada iteración, mientras que el incremento de la misma (`inc`) es cada vez menor. Como el incremento de la verosimilitud no baja por debajo del umbral de parada, fijado por defecto al valor 0.001, el algoritmo sólo acaba cuando se alcanza el número máximo de iteraciones, fijado por defecto a 20.

En este momento, ya estamos en condiciones de entrenar los modelos de los 157 locutores que constituyen el conjunto de usuarios legítimos de los sistemas de reconocimiento y verificación del locutor. Esto lo podemos hacer repitiendo el proceso seguido más arriba para cada uno de los locutores, o, más convenientemente, usando el script `run_spkid` con el comando `train`.

Como no estamos interesados en volver a parametrizar toda la base de datos, hemos de informar al script de qué parametrización queremos usar:

```
usuario:~/PAV/P4/$ FEAT=lp run_spkid train
Sun Feb 30 08:03:30 CET 2022: train ---
SES000 ----
gmm_train -v 1 -T 0.001 -N 5 -m 1 -d work/lp -e lp -g work/gmm/lp/SES000.gmm      ...
 ↪  lists/class/SES000.train
DATA: 3075 x 9
GMM nmix=1      ite=0   log(prob)=-12.7629      inc=1e+34

SES001 ----
gmm_train -v 1 -T 0.001 -N 5 -m 1 -d work/lp -e lp -g work/gmm/lp/SES001.gmm      ...
 ↪  lists/class/SES001.train
DATA: 2840 x 9
GMM nmix=1      ite=0   log(prob)=-11.5276      inc=1e+34

SES002 ----
gmm_train -v 1 -T 0.001 -N 5 -m 1 -d work/lp -e lp -g work/gmm/lp/SES002.gmm      ...
 ↪  lists/class/SES002.train
DATA: 2835 x 9
GMM nmix=1      ite=0   log(prob)=-12.9097      inc=1e+34

SES003 ----
gmm_train -v 1 -T 0.001 -N 5 -m 1 -d work/lp -e lp -g work/gmm/lp/SES003.gmm      ...
 ↪  lists/class/SES003.train
DATA: 2845 x 9
GMM nmix=1      ite=0   log(prob)=-11.4628      inc=1e+34

...
```

Vemos que, para cada locutor, se lanza una ejecución de `gmm_train`. Sin embargo, el algoritmo siempre para después de una sola iteración. Es posible que se haya cometido algún error en la implementación, pero, aunque no fuera así, el algoritmo seguiría parando después de una sola iteración. ¿Sabe por qué?

2. Clasificación del locutor.

2.1. 4o \TODO: cálculo de la verosimiltud de una secuencia dado el modelo.

Podemos usar el mismo script `run_spkid` para reconocer los locutores de la base de datos usando el comando `test`. A su vez, podemos calcular la tasa de error con el comando `classerr`. Aprovechamos la estructura del script para llamar a ambos comandos en una sola orden:

```
usuario:~/PAV/P4/$ FEAT=lp run_spkid test classerr
Sat Feb 30 17:53:41 CET 2022: test ---
gmm_classify -d work/lp -e lp -D work/gmm/lp -E gmm lists/gmm.list                ...
↪  lists/class/all.test
BLOCK00/SES000/SA000S13 SES000  -1e+38
BLOCK00/SES000/SA000S14 SES000  -1e+38
BLOCK00/SES000/SA000S22 SES000  -1e+38
BLOCK00/SES000/SA000S24 SES000  -1e+38

...

BLOCK29/SES294/SA294S14 SES000  -1e+38
BLOCK29/SES294/SA294S23 SES000  -1e+38
BLOCK29/SES294/SA294S25 SES000  -1e+38
BLOCK29/SES294/SA294S28 SES000  -1e+38
Sat Feb  30 18:39:02 CET 2022
nerr=780        ntot=785        error_rate=99.36%
Sun Feb  31 08:13:56 CET 2022
```

Vemos que todas las señales son asignadas al mismo locutor, y con la misma verosimilitud (algo así como $-\infty$). La tasa de error no alcanza el 100 % simplemente porque hasta un reloj parado acierta la hora dos veces al día... El problema es que nos faltan dos nuevos \TODOs: uno, el que permite calcular la verosimilitud de una secuencia dado cada uno de los modelos; el otro, el que selecciona el modelo de verosimilitud más alta para la secuencia.

El primero de ellos se utiliza tanto en la clasificación como en la verificación y lo encontramos en el primer \TODO del fichero `src/pav/gmm.cpp`. Para el cálculo de la verosimilitud de una secuencia $\mathbf{X} = x_0\, x_1 \cdots x_{N-1}$ consideraremos que cada trama x_i es independiente del resto. Por tanto, su probabilidad dado el modelo λ_k será el producto para las N tramas de la verosimilitud de cada una de ellas:

$$p\left(\mathbf{X}|\lambda_k\right) = \prod_{i=0}^{N-1} p\left(\mathbf{x_i}|\lambda_k\right) \tag{1}$$

Sacando logaritmos para prevenir problemas de margen dinámico al multiplicar muchas probabilidades, el producto deviene un sumatorio:

$$\log p\left(\mathbf{X}|\lambda_k\right) = \sum_{i=0}^{N-1} \log p\left(\mathbf{x_i}|\lambda_k\right) \tag{2}$$

El método `GMM::gmm_logprob`, en el fichero `src/pav/gmm.cpp`, nos proporciona el logaritmo de la verosimilitud de una trama, $\log p\left(\mathbf{x_i}|\lambda_k\right)$, usándolo se ha terminar el método `GMM::logprob` para obtener el logaritmo de la verosimilitud de la señal completa, $\log p\left(\mathbf{X}|\lambda_k\right)$.

2.2. 5o \TODO: obtención del modelo (locutor) de verosimilitud más alta.

Una vez tenemos un procedimiento para el cálculo de la verosimilitud de la secuencia dado un modelo, el reconocimiento consiste en seleccionar el modelo para el que este valor es más alto. Eso también está esbozado, pero no implementado, en el 5o \TODO del código, en la función `classify()` del fichero `src/gmm/gmm_classify.cpp`.

2.3. Primer experimento de clasificación del locutor.

Una vez resueltos los dos \TODOs anteriores, ya estamos en condiciones de efectuar un reconocimiento (que realmente reconozca algo):

```
usuario:~/PAV/P4/$ FEAT=lp run_spkid test classerr
Sun Feb 31 18:13:40 CET 2022: test ---
gmm_classify -d work/lp -e lp -D work/gmm/lp -E gmm lists/gmm.list                ...
↪  lists/class/all.test
BLOCK00/SES000/SA000S13 SES003  -11.5718
BLOCK00/SES000/SA000S14 SES027  -11.3781
BLOCK00/SES000/SA000S22 SES013  -12.8079
BLOCK00/SES000/SA000S24 SES000  -12.7119

...

BLOCK29/SES294/SA294S14 SES170  -11.6485
BLOCK29/SES294/SA294S23 SES294  -12.5191
BLOCK29/SES294/SA294S25 SES140  -12.2616
BLOCK29/SES294/SA294S28 SES130  -12.6012
Sun Feb 31 18:12:34 CET 2022: classerr ---
nerr=536        ntot=785        error_rate=68.28%
Sun Feb 31 18:11:24 CET 2022
```

El resultado es un completo desastre, pero eso es debido a que tanto los coeficientes LP como los GMM de una sola gaussiana son muy poco adecuados para el reconocimiento del habla. Con una parametrización mejor y un número de gaussianas suficiente, el resultado debe bajar con claridad del 1 %.

3. Verificación del locutor.

3.1. 6o \TODO: verificación del locutor usando sólo su verosimilitud.

Un primer experimento de verificación del locutor puede realizarse usando únicamente el modelo del locutor candidato. Se corresponde con el primer \TODO en el fichero `src/pav/gmm_verify.cpp`, en la primera versión de la función `verify()`. Esta función tiene que devolver como *score* el logaritmo de la verosimilitud de la señal dado el modelo del candidato.

Haciéndolo, podemos realizar la verificación usando el programa `gmm_verify`, para lo cual tenemos que dar valores adecuados a sus opciones (que podemos consultar ejecutando el programa sin argumentos):

```
usuario:~/PAV/P4/$ gmm_verify
Usage: gmm_verify [options] list_gmm list_of_test_files list_of_candidates
```

```
Options can be:
  -d dir        Directory of the feature files (def. ".")
  -e ext        Extension of the feature files (def. "mcp")
  -D dir        Directory of the gmm files (def. ".")
  -E ext        Extension of the gmm files (def. "gmm")
  -w name       Name of the "background" GMM (def. do not use world model)
                name does not include directory and extension:
                the dir option (-D) and ext (-e) will be added

Each "trial" is defined by the speech files and the candidate (pretended user)
The number of items in both files has to be the same.

For each input sentence, different feature files (and different GMMs)
can be provided using several times the options -d -e -D and -E
```

Miramos valores adecuados para cada uno de estas opciones:

`-d dir` Debemos proporcionar el directorio de las señales parametrizadas. Por ahora, `work/lp`.

`-e ext` Extensión de los ficheros de señal parametrizada (`lp`).

`-D dir` Directorio de los GMM (`work/gmm/lp`).

`-E ext` Extensión de los ficheros de GMM (`gmm`)

`-w name` Nombre del modelo del mundo (*universal background model*). En este primer experimento de verificación no se usa, pero en el apartado siguiente deberá indicar un modelo entrenado a partir de muchos locutores que proporcione una verosimilitud de referencia.

Por su parte, los argumentos del programa son:

`list_gmm`
: Lista de locutores legítimos del sistema. Son los mismos que en el caso de la clasificación del locutor (`lists/gmm.list`).

`list_of_test_files`
: Lista de señales cuyo locutor se desea verificar. De cara al desarrollo del sistema se dispone de varias listas de señales en el directorio `lists/verif` con extensión `.test`. A cada una de ellas le corresponde una lista de usuarios que pueden ser legítimos o impostores, con la extensión `.test.candidates`. Las posibilidades son tres:

  `users` Lista de señales cuyo locutor afirma ser quien realmente es.

  `impostors4` Lista de señales cuyo locutor es un impostor. Cada impostor, y hay muchos, intenta colarse como cuatro usuarios legítimos distintos, de ahí el 4 del nombre.

  `all` Combinación de las dos listas anteriores, con una mezcla de usuarios legítimos e impostores.

  Durante el desarrollo del sistema se usará esta última lista (`lists/verif/all.test`).

`list_of_candidates`
: Usuario que pretende ser el locutor de cada señal (`lists/verif/all.test.candidates`).

Ejecutamos `gmm_verify` con las opciones y argumentos de más arriba:

```
usuario:~/PAV/P4/$ gmm_verify -d work/lp/ -e lp -D work/gmm/lp/ -E gmm ...
↪  lists/gmm.list lists/verif/all.test lists/verif/all.test.candidates
BLOCK00/SES002/SA002S12 SES002  -13.7068
BLOCK00/SES002/SA002S13 SES002  -13.7786
BLOCK00/SES002/SA002S23 SES002  -12.4063
BLOCK00/SES002/SA002S26 SES002  -14.8238

...

BLOCK29/SES291/SA291S27 SES158  -12.853
BLOCK29/SES291/SA291S27 SES172  -12.9683
BLOCK29/SES291/SA291S27 SES275  -12.6749
BLOCK29/SES291/SA291S27 SES292  -12.815
```

Realmente, el programa `gmm_verify` no verifica nada... sólo escribe en pantalla la información necesaria para realizar la verificación. La salida tiene tres columnas: en la primera aparece el nombre de la señal, y es idéntica a la lista de señales; en la segunda aparece el nombre del teórico locutor de la señal, y es idéntica a la lista de candidatos; finalmente, la tercera es el logaritmo de la verosimilitud de la señal dado el modelo del candidato.

Existe un script en Perl, `spk_verif_score`, que usa un umbral para determinar si el sistema considera al locutor como legítimo o impostor. La regla de decisión consiste en que, si el logaritmo de la verosimilitud supera el umbral, consideraremos al locutor como legítimo. En caso contrario, lo consideraremos impostor. El script aplica el umbral a cada señal, compara el nombre de la misma (en el que aparece el nombre del locutor real) con el del locutor teórico, y decide si se ha producido un error de pérdida (un locutor legítimo considerado como impostor) o un error de falsa alarma (un impostor se nos cuela como legítimo). A partir de esta información, el script calcula una tasa de detección.

Para poder ejecutar `spk_verif_score` hemos de ejecutar `gmm_verify` redirigiendo la salida a un fichero. En los ejemplos siguientes se supone que se ha redirigido al fichero `prueba.verif`, y probamos distintos valores del umbral:

```
usuario:~/PAV/P4/$ spk_verif_score -12 prueba.verif
============================================
THR: -12
Missed:     195/250=0.7800
FalseAlarm: 165/1000=0.1650
--------------------------------------------
==> CostDetection: 226.5
============================================
```

Vemos que un umbral igual a -12 produce un 78 % de pérdidas y un 16.5 % de falsas alarmas. El coste de detección (que penaliza más las falsas alarmas que las pérdidas), es de 226.5.

Como el problema con este experimento es que hemos dejado pasar a muchos impostores, subimos el umbral para evitar que se nos cuelen tantos. Probamos con un umbral igual a -10:

```
usuario:~/PAV/P4/$ spk_verif_score -10 prueba.verif
============================================
THR: -10
Missed:     249/250=0.9960
FalseAlarm: 1/1000=0.0010
```

```
-------------------------------------------
==> CostDetection: 100.5
===========================================
```

Vemos que, ahora, sólo se nos cuela un impostor, pero eso es a costa de no dejar pasar a casi ningún usuario legítimo...

El script `spk_verif_score` también permite calcular el umbral que optimiza el coste de detección. Para ello calcula 100 valores posibles del umbral, distribuidos uniformemente entre las verosimilitudes extremas para las señales a evaluar. Para ello, sólo hay que invocar el programa sin ningún valor del umbral:

```
usuario:~/PAV/P4/$ spk_verif_score prueba.verif
Threshold search:
THR     -22.14191216369 9       0(0)    1(1000)
THR     -22.0202751420531       8.991   0(0)    0.999(999)
THR     -21.8986381204162       8.991   0(0)    0.999(999)
THR     -21.7770010987793       8.991   0(0)    0.999(999)
THR     -21.6553640771424       8.991   0(0)    0.999(999)

...

THR     -10.3431210649107       1.019   0.992(248)      0.003(3)
THR     -10.2214840432738       1.005   0.996(249)      0.001(1)
THR     -10.0998470216369       1.005   0.996(249)      0.001(1)
THR     -9.97820999999999       1       1(250)  0(0)

===========================================
THR: -10.7080321298214
Missed:     241/250=0.9640
FalseAlarm: 4/1000=0.0040
-------------------------------------------
==> CostDetection: 100.0
===========================================
```

Vemos que inicia la búsqueda para un valor del umbral que deja pasar a todo el mundo, legítimo o no, y la finaliza para uno en el que no pasa ni Dios. El valor óptimo se produce para un umbral `THR: -10.708032129814`. Es conveniente realizar este tipo de búsqueda porque este valor del umbral es el que será adecuado cuando se haga la evaluación *ciega* del sistema, en la que no dispondremos de información acerca del usuario real que pronunció la frase.

3.2. 7o \TODO: verificación del locutor usando su verosimilitud y un modelo del *mundo*.

Una manera de mejorar los resultados en verificación del locutor consiste en no utilizar directamente el logaritmo de la verosimilitud del modelo, sino la diferencia entre ella y un valor de referencia determinado a partir de un modelo entrenado con muchos locutores distintos. Es lo que se llama *Universal Background Model* o, más comúnmente, *modelo del mundo*.

Para poder usar este modo de verificación es necesario completar el 7o y último \TODO del código fuente (hay más \TODOs entre los scripts...). En él se debe programar la segunda versión de la función `verify()`

en el fichero `src/pav/gmm_verify.cpp`. Esta función tiene que devolver como *score* la diferencia entre el logaritmo de la verosimilitud de la señal dado el modelo del candidato y dado un modelo del mundo.

Para poder usar esta versión de la función es necesario pasar el modelo del mundo como argumento a la opción `-w` de `gmm_verify`. Por tanto, lo primero que se ha de hacer es entrenar este modelo del mundo usando `gmm_train` y una lista de señales pronunciadas por muchos locutores que representen una variedad importante de los mismos. Para ello, se dispone en el directorio `lists/verif` de tres ficheros guía con señales provenientes de muchos locutores distintos (extensión `.train`):

`users` Son señales de 50 locutores seleccionados aleatoriamente entre los 157 locutores legítimos del sistema.

`others` Son señales de 57 locutores de la misma base de datos, pero que no están entre los 157 legítimos ni entre los 50 impostores.

`users_and_others`
Es la combinación de los dos conjuntos anteriores.

Entrenamos el modelo del mundo con el primero de estos conjuntos:

```
usuario:~/PAV/P4/$ gmm_train -d work/lp -e lp -g work/gmm/lp/users.gmm ...
↪  lists/verif/users.train
DATA: 188665 x 9
GMM nmix=5      ite=0   log(prob)=-12.9949      inc=1e+34
GMM nmix=5      ite=1   log(prob)=-12.9401      inc=0.054781
GMM nmix=5      ite=2   log(prob)=-12.047       inc=0.893135
GMM nmix=5      ite=3   log(prob)=-10.7956      inc=1.25139
GMM nmix=5      ite=4   log(prob)=-10.3241      inc=0.471456
GMM nmix=5      ite=5   log(prob)=-10.0567      inc=0.267457
GMM nmix=5      ite=6   log(prob)=-9.86978      inc=0.18691
GMM nmix=5      ite=7   log(prob)=-9.75399      inc=0.115786
GMM nmix=5      ite=8   log(prob)=-9.69273      inc=0.0612631
GMM nmix=5      ite=9   log(prob)=-9.66417      inc=0.0285587
GMM nmix=5      ite=10  log(prob)=-9.64428      inc=0.0198889
GMM nmix=5      ite=11  log(prob)=-9.62802      inc=0.016263
GMM nmix=5      ite=12  log(prob)=-9.61386      inc=0.0141611
GMM nmix=5      ite=13  log(prob)=-9.60129      inc=0.0125675
GMM nmix=5      ite=14  log(prob)=-9.58955      inc=0.0117435
GMM nmix=5      ite=15  log(prob)=-9.57859      inc=0.0109549
GMM nmix=5      ite=16  log(prob)=-9.56815      inc=0.0104389
GMM nmix=5      ite=17  log(prob)=-9.5589       inc=0.00924683
GMM nmix=5      ite=18  log(prob)=-9.55108      inc=0.00782299
GMM nmix=5      ite=19  log(prob)=-9.54461      inc=0.00647259
```

Ahora ya podemos ejecutar `gmm_verify` con el modelo del mundo generado:

```
usuario:~/PAV/P4/$ gmm_verify -d work/lp/ -e lp -D work/gmm/lp/ -E gmm -w users ...
↪  lists/gmm.list lists/verif/all.test lists/verif/all.test.candidates > ...
↪  prueba.verif
usuario:~/PAV/P4/$ spk_verif_score prueba.verif
Threshold search:
THR     -11.495711384187        9       0(0)    1(1000)
THR     -11.3818694003451       8.991   0(0)    0.999(999)
```

```
THR     -11.2680274165033       8.991   0(0)    0.999(999)
THR     -11.1541854326614       8.991   0(0)    0.999(999)

...

THR     -0.339196967683761      1.009   1(250)  0.001(1)
THR     -0.225354983841891      1.009   1(250)  0.001(1)
THR     -0.111513000000021      1.009   1(250)  0.001(1)
THR     0.00232898384184935     1       1(250)  0(0)


============================================
THR: 0.00232898384184935
Missed:     250/250=1.0000
FalseAlarm: 0/1000=0.0000
--------------------------------------------
==> CostDetection: 100.0
============================================
```

El resultado es un desastre, pero eso es debido a todo está pensado para que así sea... Usando una estrategia adecuada es posible alcanzar costes de detección por debajo de 10.