Clasificación y verificación del hablante

Antonio Bonafonte

Resumen

En esta práctica:

- Se implementará un sistema de extracción de características para la clasificación del hablante (P4-S1)
- Se completará y finalizará el algoritmo de estimación y evaluación de el modelo probabilístico GMM.(P4-S2)
- Se utilizarán para clasificar el locutor, utilizando el corpus de desarrollo (P4-S3)
- Se utilizarán para una aplicación de verificación del locutor,(P4-S4)
- Se investigarán métodos para mejorar las prestaciones de los sistemas base (P4-S5)
- 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.

1. Extracción de características *features*

En esta primera parte de práctica se diseñará un sistema de extracción de características para reconocimiento del locutor. Se empezará comparando los coeficientes de predicción linear, *LP*, $v = (a_1 \dots a_p)$, que representan la envolvente espectral

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

y los coeficientes cepstrales *LPCC*, que son una representación equivalente pero que son más incorrelados por lo que son características más indicadas para la clasificación.

Finalmente, consultando el trabajo A Tutorial on Text-Independent Speaker Verification, de Fréderic Bimbot et al., se diseñará e implementará un sistema para extraer las características más indicadas para esta aplicación.

Un sistema de reconocimiento del hablante es un sistema de cierta complejidad, por lo que se van a organizar los programas, separandolos en librerias y programas. Además utilizaremos la programación en `bash`, que permite agrupar las ordenes que pondríamos en la línea de comandos en unos *programas*.

Instalación del entorno de trabajo *PAV*

Instale los programas proporcionados en la misma estructura que en la práctica anterior. Los programas nuevos, serán un nuevo directorio dentro de `prj`. Añada los nuevos *módulos* al `makefile` de `prj`

Añada a la variable de entorno `PATH` (en el fichero .bashrc) el directorio `scripts` (con la ruta completa, `/home/...`).

Instalación de la libreria SPTK, *Speech Toolkit*

Esta librería incluye varias utilidades prácticas para el procesado de voz y audio, tales como el cálculo de correlación, fft, etc. La vamos a utilizar para implementar la extracción de características. En las versiones más nuevas de ubuntu, se encuentra en el repositorio (pruebe `sudo apt-get install sptk`, pero sino, siempre puede descargarse los fuentes e instalarlos.[1]

Descargue también el manual de referencia de SPTK.

Si instala directamente de los fuentes, que le permitirá ver la implementación, encontrará el fichero INSTALL con las instrucciones de compilación (típicamente *i* `./configure`; *ii* `make`; *iii* `make install`). Por defecto se instala en los directorios del sistema (en ese caso, ejecute `sudo make install`, o puede elegir un directorio del usuario (`./configure -help` le indicará la opción adecuada).

Si no se instala en el directorio del sistema, modifique la variable `PATH` (en `.bashrc`) para incluir en el `PATH` el directorio con los ejecutables. Recuerde ejecutar `source ~/.bashrc` si el terminal se ha abierto antes de cambiar el `.bashrc`.

Extracción de características

En el directorio `prj/scripts` verá dos pequeños *scripts* (`wav2lp.sh, wav2lpcc.sh` que utilizan SPTK para calcular los coeficientes LP o los cepstrales LPC. Examínelos y consulte el manual de referencia de SPTK para entender que hace cada uno de ellos.

Las *características* pueden verse com una matriz, con tantas columnas como coeficientes, y una fila por cada tramo analizado. En esta práctica se han definido ficheros binarios, asociados a la clase `fmatrix`. En estos ficheros se almacenan primero dos enteros con el número de filas y columnas, y a continuación números los valores reales de la *matriz*. Estos ficheros pueden visualizarse mediante el programa `fmatrix_show`.

Calcule los coeficientes *LP*, de orden 8, para el fichero `ona.wav` así como los coeficientes *LPCC* (orden 8, 12 coeficientes). Represente mediante *gnuplot* el coeficiente '2' frente el '3'. Para ello, transforme el fichero a texto (`fmatrix_show`), seleccione las columnas de interés (`cut -f ...`), –consulte el manual de SPTK para seleccionar las columnas en las que se encuentran estos

[1] Si instala este paquete para ejecutar alguno de sus programas, como por ejemplo `window` deberá precedirlo con `sptk window`

coeficientes LPC– Ejecute `gnuplot` y desde allí, `plot "your_file_of_data"`. Discuta cuáles de estas características son más adecuadas para la clasificación.

Consulte el paper mencionado y estudie los dos métodos que proponen para los sistemas de reconocimiento y verificación del hablante, así como los valores típicos (por ejemplo, duración de tramos, número de coeficientes, etc.).

Cree un nuevo script para calcular las características de un sistema de reconocimiento del hablante por el otro método indicado en el *pasper* utilizando la librería SPTK. Consulte para ello el manual de referencia. Puede representar también las características '2' y '3' de su sistema de extracción de características.

Para las distintas características, y para algunos ficheros, estudie el coeficiente de correlación entre características utilizando el programa proporcionado `pearson`

$$\rho = \frac{\mathbf{E}\{(X - \mu_x)(Y - \mu_y)\}}{\sigma_x \cdot \sigma_y}$$

2. Estimación *Gaussian Mixture Models* (GMM) y Reconocimiento del Hablante.

Una vez seleccionadas las características, para realizar un clasificador *Bayesiano*, necesitamos modelar la función de probabilidad de cada clase, en este caso, de las características escogidas de cada hablante. Los modelos GMM ofrecen buenas prestaciones en muchos problemas de clasificación de señales de voz y audio que son complejos, pues al representar sonidos distintos tienen gran dispersión.

Se proporciona la clase `GMM`, definida en el fichero `gmm.h` e implementada *casi completamente* en los fichero `gmm.cpp` y `gmm_vq.cpp` (este último únicamente para la inicialización mediante VQ).

Además, hay tres programas que utilizan esta clase:

- `gmm_train.cpp` para estimar un GMM a partir de datos (características, en nuestro caso, acústicas).
- `gmm_show.cpp` para visualizar el fichero binario creado.
- `gmm_classify.cpp` para decidir la clase a la que están asociados unos datos (utilizando los GMM estimados anteriormente).

1. Lea la documentación sobre la clase GMM. Entienda que son las variables miembro y también las distintas funciones. Puede estudiar directamente la declaración `gmm.h` o la documentación que se genera en `prj/html/index.html`, al ejecutar `make doc`, desde el directorio `prj`).
2. En la definición del GMM `gmm.cpp` complete el código que falta (busque la palabra `TODO`, *"to do"*.
   - `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 como las que ha generado en la primera sesión, con tantas filas como tramos de voz (`data.row()`) 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 del tramo `i`.

- `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* y *M*. Los parámetros que regulan la convergencia son tanto el número máximo de iteraciones, como el mínimo incremento exigido en la función de verosimilitud, *logprob*. La matriz `weigths[i][j]` se utilizará para guardar la probabilidad de que el tramo `i` pertenezca a la gausiana `j`.

3. Revise el programa principal de entrenamiento, `gmm_train.cpp`. Verá que se gestionan las opciones al ejecutar desde la línea de comandos (funcion de `stdlib getopt`). El programa llama al algoritmos de estimación *EM*, pero deberá implementar las distintas inicializaciones.

4. Edite el programa de clasificación, `gmm_classify.cpp`. Implemente la función `int classify(const vector<GMM> &vgmm, const fmatrix &dat, float &maxlprob)`. Esta función recibe un vector de *GMM*, uno por cada hablante, y debe devolver el índice del hablante identificado, así como el logaritmo de la probabilidad, como información complementaria.

5. Para estudiar la convergencia del algoritmo, encuentre las características de dos personas. Para facilitar la visualización, tome características con únicamente 2 parámetros.

6. 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.

7. Ya disponemos de los ingredientes necesarios para realizar un sistema básico de clasificación del hablante. Ahora debe crear un GMM por hablante, reservar una parte de los datos para test, analizar los resultados, optimizar el sistema, etc.

   Para realizar este tipo de tareas, se suelen agrupar los comandos en *scripts*, por ejemplo en `bash`. Por ejemplo, puede utilizar el *script* `run_spkid.sh`. Examine este fichero y entienda su propósito. Deberá modificar el directorio *de trabajo*, `(w)` (un directorio donde se guardarán ficheros temporales), y también el directorio de la base de datos (`db`). Revise cómo se realizan las distintas tareas y adaptelo según sus preferencias, por ejemplo, tipo de características, parámetros de estas características, número de componentes del GMM, etc.

   Ejecute este *script* en bash, para realizar los distintos experimentos y optimizar el sistema.

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.

3. Verificación del Locutor utilizando modelos GMM

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.

$$s < t_h \quad \rightarrow \texttt{ACCESS\_DENIED}$$
$$s \geq t_h \quad \rightarrow \texttt{ACCESS\_GRANTED}$$

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:

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

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*

Se recomienda leer el *paper tutorial* sobre verificación del locutor.

Sistema de Referencia

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 más adelante, 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. Para compilar deberá generar un fichero `makefile`. Puede utilizar `make_wizard -p1`.

3. Utilice `run_spkid.sh`, modificándolo adecuadamente, para experimentar el sistema de verificación.

   a) Ejecute el comando `listverif`. Al ejecutarlo, verá en el directorio `lists_verif` que los hablantes se dividen en tres grupos. Un grupo de 50. usuarios del sistema `users`, un grupo de 50 impostores `impostors` y el resto de usuarios `others`. Entienda la función de los ficheros creados.

   b) Añada un comando para estimar el modelo general. Para ello puede utilizar los datos de entrenamiento que considere conveniente, entre `users.train`, `others.train` o `users_and_others.train`.

   c) 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 (correcta) o los impostores (falsa y aleatoria).

El resultado de esta ejecución guardelo en un fichero (ej.: `' ... > result_verif.log'`, y ejecute `spk_verif_score.pl` pasándole este fichero. En esta fase de validación, se elige el mejor umbral (que debería guardarse para utilizarlo en la fase operativa o del test final).

Ampliación

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.