UNIVERSIDAD POLITECNICA DE VALENCIA

ESCUELA POLITECNICA SUPERIOR DE GANDIA

Máster en Ingeniería Acústica

Desarrollo y análisis de clasificadores de señales de audio

TESIS DE MASTER

Autor:
Fabián Aguirre Martín

Director/es:
D. Miguel Ferrer Contreras
D. María de Diego Antón

GANDIA, Julio de 2017

Desarrollo y análisis de clasificadores de señales de audio

Autor: Fabián Aguirre Martín

Director: Miguel Ferrer Contreras

Codirectora: María de Diego Antón

Resumen — Actualmente, la tecnología tiende cada vez más a unirse al ámbito de la Inteligencia Artificial, donde los propios sistemas toman decisiones. En las últimas décadas se han desarrollado diversos sistemas fruto de una combinación entre el procesado de audio y los algoritmos de Aprendizaje Automático, principalmente aplicaciones orientadas a voz y música. Sin embargo, el reconocimiento y clasificación de sonidos desde un punto de vista genérico es actualmente un campo objeto de investigación de particular interés. En este trabajo, se aporta una visión general de todo el proceso llevado a cabo para realizar clasificación automática de audio, utilizando el procesado digital de audio denominado Extracción de Características, y los algoritmos de clasificación del Aprendizaje Automático Supervisado basados en Reconocimiento de Patrones.

Palabras clave: Extracción de Características, Aprendizaje Automático, Reconocimiento de Patrones

Abstract — Nowadays, technology tends to join the Artificial Inteligence field, where systems take decisions. In the last decades, there has been developed diverse systems as a combination of audio processing and Machine Learning algorithms, mainly aplications focused on voice and music. However, sound recognition and classification from a general point of view is an investigación field of particular interest. This work, gives a general vision of the whole process carried out to make Automatic Audio Classification, using digital audio processing as Feature Extraction, and the Supervised Machine Learning classification algorithms based on Pattern Recognition.

Keywords: Feature Extraction, Machine Learning, Pattern Recognition

Autor: Fabián Aguirre Martín, email: faagmar@epsg.upv.es
Fecha de Entrega: 11 de septiembre de 2017

ÍNDICE

I. INTRODUCCIÓN

La clasificación y reconocimiento de señales de audio es un actualmente un campo de gran interés. Durante los últimos años se han desarrollado diversas aplicaciones resultantes de una combinación entre procesado de audio y Aprendizaje Automático (*Machine Learning*). Entre las más desarrolladas se encuentran aquellas enfocadas al ámbito de la voz y la música, como por ejemplo el reconocimiento de voz o del hablante (*Automatic Speaker Recognition*), el reconocimiento y transcripción del habla (*Automatic Speech Recognition*), o incluso la recuperación de información musical basada en Reconocimiento de Patrones (*Music Information Retrieval*) [1].

Un campo menos desarrollado dentro de la clasificación automática de señales de audio (*Automatic Audio Classification* o *Audio Signal Classification*), pero actualmente en continua investigación, es el reconocimiento de ruidos ambientales o del entorno (*Environmental Sound Recognition*) [2], el cual tiene diversas aplicaciones, como por ejemplo la detección de eventos sonoros en sistemas de seguridad, recuperación de datos multimedia basada en el audio a través del etiquetado en forma de metadatos, ecualización automática para audífonos [3], etc.

En este trabajo se pretende dar una visión general de todo el proceso de clasificación, desde la extracción de características (*features*) del audio, hasta los algoritmos de *Machine Learning* utilizados para clasificación. El tipo de aprendizaje automático llevado a cabo a lo largo del trabajo será supervisado, lo cual implica que los datos o audios utilizados en la fase de aprendizaje o entrenamiento estén etiquetados en clases [4]. También se realizará un sistema de clasificación y etiquetado de audio en tiempo real, como ejemplo de aplicación del proceso, incluyendo la clasificación de forma jerárquica.

En el capítulo 2 se establecen los objetivos del trabajo, seguido del estado del arte de esta tecnología en el capítulo 3. El marco teórico se desarrolla en el capítulo 4, describiendo cada paso del proceso de clasificación: recopilación de bases de datos, extracción de características del audio (*Feature Extraction*) y funcionamiento de distintos algoritmos de clasificación. En el capítulo 5 se explica la realización del software desarrollado como herramienta de clasificación de audio, así como el desarrollo del clasificador en tiempo real. Dentro de este capítulo, también se incluyen los resultados obtenidos de las distintas clasificaciones realizadas. Por último, en los capítulos 5 y 6, se establecen las posibles líneas futuras del trabajo y las conclusiones, respectivamente.

Finalmente, se incluyen algunos de los códigos generados con *Matlab* 2017a en el Anexo II, así como el manual de usuario del software desarrollado (Anexo I).

II. OBJETIVOS

El objetivo principal de este trabajo es realizar una herramienta de clasificación que permita diseñar clasificadores utilizando distintos tipos de características (*features*), así como distintos tipos de algoritmos de aprendizaje automático (*Machine Learning*). La base de datos principal del proyecto será *UrbanSound*8K [5], orientada a ruido urbano. Sin embargo, el sistema debe ser versátil y permitir introducir nuevas bases de datos de audio para poder ampliar la clasificación de audio desde un punto de vista genérico, para así, tener la posibilidad de entrenar distintos clasificadores y organizarlos posteriormente en una estructura jerárquica.

Esta herramienta contará con una interfaz gráfica que permita al usuario procesar las bases de datos, entrenar clasificadores y evaluar los resultados mediante audios de test, pudiendo indicar el porcentaje de datos que se destinarán a entrenamiento y a evaluación al incorporar una nueva base de datos.

El algoritmo base de aprendizaje automático que el sistema deberá incorporar es *Random Forest* [6], el cual es un algoritmo basado en árboles de decisión (*decisión trees*) [7]. Además, se debe incorporar otros clasificadores para poder realizar comparativas y escoger la mejor opción ante el diseño o entrenamiento de un nuevo clasificador.

También, se pretende realizar un sistema que funcione en tiempo real, clasificando el audio a través de una estructura taxonómica de clasificadores que permita identificar muestras de sonido cada pocos segundos. El sistema deberá ser capaz de procesar el audio a partir de un archivo de audio o de una entrada de micrófono, indicando mediante una interfaz gráfica la clasificación realizada sobre la forma de onda.

La implementación será llevada a cabo utilizando *Matlab* 2017a. Para la aplicación de tiempo real se empleará la herramienta *playrec* [8], la cual es gratuita y contiene un conjunto de funciones implementadas en *Matlab* que permiten el acceso a tarjetas de audio utilizando la librería *PortAudio* para llevar a cabo la entrada y salida del mismo.

III. ESTADO DEL ARTE

La clasificación y el reconocimiento de señales de audio es un campo tradicionalmente enfocado a voz y música. En el ámbito de la voz, existen diversos sistemas basados en reconocimiento y transcripción del habla (*Automatic Speech Recognition*), reconocimiento o verificación del hablante (*Automatic Speaker Recognition*) e identificación del hablante (*Automatic Speaker Identification*), o incluso otros tipos de reconocimiento, ya sea idioma, emociones, género, etc. En cuanto al ámbito musical, existen programas que permiten la recuperación de información musical (*Music Information Retrieval*), como por ejemplo la famosa aplicación *Shazam*, capaces de devolver al usuario información de una canción a partir de una grabación de audio de pocos segundos. El reconocimiento de audio también encuentra su lugar en la clasificación automática de instrumentos musicales o incluso en la transcripción musical (*Automatic Music Transcription*), la cual transcribe el audio en notación musical [9].

Sin embargo, la clasificación automática de audio (*Automatic Audio Classification*) toma un papel más amplio dentro del campo de análisis de escena auditiva (*Auditory Scene Analysis*) [9], el cual engloba cualquier sonido que una persona pueda escuchar. También se contemplan los sonidos ambientales o del entorno, ya sea ruido o cualquier sonido distinto a música y/o voz. A este campo se le denomina *Environmental Sound Recognition* (ESR) [2], el cual encuentra aplicación en sistemas de etiquetado de audio para una posterior recuperación basada en búsqueda escrita por teclado, sistemas de seguridad (junto con imagen y video), sistemas de robótica con reconocimiento auditivo del entorno, sistemas incorporados en audífonos para mejorar la inteligibilidad del habla mediante ecualización automática, etc. Si bien, cualquier sistema de este tipo suele estar enfocado a una aplicación específica, ya que, la clasificación de audio desde un punto de vista genérico, o utilizando grandes bases de datos, tiende a ser poco efectiva [2]. Esto se debe principalmente a las características no-estacionarias, ya que, la música y la voz, tienen un sentido melódico y rítmico, lo cual no tiene por qué ocurrir con cualquier otro tipo de sonido [2].

Es importante mencionar que en aplicaciones de clasificación de audio basadas en Aprendizaje Automático Supervisado [4] no siempre conllevan una relación directa entre aplicación-algoritmo, ya que distintos algoritmos de *Machine Learning* pueden ser útiles para la misma aplicación de clasificación. No ocurre lo mismo con las *features* utilizadas en el pre-procesado del audio, ya que la gran mayoría de aplicaciones utilizan los MFCC (*Mel-Frequency Cepstral Coefficients*) [13] como *features* principales.

La clasificación de ruido urbano es un campo actualmente en crecimiento. La base de datos gratuita más extensa de audio etiquetado de ruido urbano es *UrbanSound* [5], recopilada por los grupos de investigación *Music and Audio Research Laboratory* y *Center for Urban Science and Progress* de la universidad de Nueva York. Proponen una taxonomía bastante completa a la hora de clasificar sonidos del entorno [10]. Sin embargo, la base de datos contiene únicamente 10 clases.

En cuanto al ámbito de la robótica, también se investiga para enseñar a las máquinas a escuchar, a lo que denominan *Machine Hearing* [11]. En el instituto MARCS (*Western Sydney University*) se ha diseñado un sistema en tiempo real para enseñar a un robot humanoide a escuchar [12]. Este sistema permite al robot detectar sonidos, clasificarlos y reconocerlos en un futuro. El proceso de aprendizaje se lleva a cabo mediante interacción humana, de tal manera que, si el robot no es capaz de clasificar un nuevo sonido, pueda preguntar a su supervisor de qué sonido se trata para así poder etiquetarlo. Utiliza un algoritmo de Aprendizaje Automático No Supervisado basado en los MFCC, con un entrenamiento básico previo de algunas clases predefinidas. Sin embargo, este sistema no establece ningún tipo de taxonomía entre clases, por lo que utiliza un método de clases libre, en el cual, si no se categoriza un nuevo sonido, se crea una nueva clase. En la figura 1 se puede observar al robot humanoide Nao.

Fig. 1. Nao Robot [12]

IV. APRENDIZAJE AUTOMÁTICO SUPERVISADO

El Aprendizaje Automático de las máquinas (*Machine Learning - ML)* tiene como objetivo desarrollar algoritmos que permitan aprender sobre un conjunto de observaciones, de tal manera que sea posible establecer hipótesis generales o predicciones sobre nuevas observaciones, y de esta forma, tomar decisiones automáticamente, dotando así a un sistema de Inteligencia Artificial.

El Aprendizaje Automático es utilizado en una gran variedad de ámbitos, desde el reconocimiento de voz, hasta la exploración o minería de datos, existiendo una gran variedad de algoritmos. Este tipo de algoritmos tienen dos fases fundamentales: la fase de entrenamiento y la fase de evaluación.

La **fase de entrenamiento** consiste principalmente en proporcionar observaciones o ejemplos al sistema, de los cuales éste debe aprender. En este punto surgen dos alternativas:

- Aprendizaje Automático Supervisado (*Supervised Machine Learning*)
- Aprendizaje Automático No Supervisado (*Unsupervised Machine Learning*)

El primero consiste en proporcionar al sistema un conjunto de datos o ejemplos de aprendizaje etiquetados, es decir, contienen la respuesta correcta. Sin embargo, en el método no supervisado, los datos de entrenamiento no están etiquetados, dando lugar a algoritmos de agrupamiento o *clustering* que permitan descubrir nuevos patrones de clasificación de un cierto conjunto de observaciones. En este trabajo se utilizará únicamente el aprendizaje supervisado, proporcionando siempre al sistema la etiqueta correcta de cada observación durante la fase de entrenamiento.

La **fase de evaluación** es llevada a cabo una vez el sistema ha sido entrenado. Esta fase consiste en proporcionar al sistema un subconjunto de datos o ejemplos, que el propio sistema debe etiquetar. Conociendo las verdaderas etiquetas de los datos proporcionados sería posible verificar los resultados de la evaluación, caracterizando así al sistema en función de sus aciertos y fallos. En este tipo de algoritmos existen dos tipos de **errores**:

- Falsa Alarma (*False Alarm Rate)*: probabilidad que tiene una determinada clase de aceptar erróneamente muestras pertenecientes a otra clase.
- Falso Rechazo (*False Rejection Rate*): probabilidad que tiene una determinada clase de rechazar erróneamente muestras que pertenecen a dicha clase.

Por ejemplo, supongamos un sistema de biometría basado en reconocimiento de voz. En un primer lugar, sería necesario que el sistema aprenda cómo es la voz de una determinada persona. Para ello, sería necesario realizar un procesado del audio que contiene la voz extrayendo ciertas características de la misma. Posteriormente, estas características serían introducidas al sistema

de aprendizaje indicándole de qué persona se trata. Finalmente, para evaluar el sistema se deberían introducir datos de la persona en cuestión, así como datos de otros sujetos. El sistema habrá acertado en muchos casos afirmando que el sujeto es quien dice ser. Sin embargo, habrá un determinado porcentaje de casos en los que el sistema haya rechazado al sujeto siendo éste la persona en cuestión (falso rechazo), mientras que otro determinado porcentaje haya sido aceptado cuando realmente no se trataba de la persona correcta (falsa alarma).

Es importante mencionar que todo algoritmo de Aprendizaje Automático basado en Reconocimiento de Patrones requiere un procesado previo de la información con la cual se trabajará. Este procesado consiste en extraer las características o *features* relevantes de la información a tratar (***Feature Extraction***) [14], las cuales serán los datos a introducir al sistema, tanto en la fase de entrenamiento como en la fase de evaluación. Es decir, si se trata de una aplicación de audio, será necesario extraer características clave del audio (por ejemplo, el centroide espectral), al igual que si se trata de una aplicación de imagen, será necesario procesar las imágenes para obtener las características más relevantes.

Generalmente, los algoritmos de *ML* son utilizados en dos tipos de aplicaciones, **clasificación** y regresión. Sin embargo, en este trabajo únicamente se tratarán los algoritmos con el objetivo de clasificar, concretamente, señales de audio.

En la figura 2 se muestra el esquema general del proceso de desarrollo de un clasificador [4]. Como se puede apreciar, el primer paso tras determinar cuál es el problema a resolver es recopilar una base de datos (***database***). La elección de la base de datos a utilizar es un aspecto fundamental del clasificador, ya que se trata de los ejemplos de los cuales éste debe aprender. Una vez definida la base de datos, es necesario procesar cada uno de los ejemplos para extraer las características más significativas (***feature extraction***). Todas las *features* seleccionadas conforman, para todas las observaciones de la base de datos, el conjunto de datos de entrenamiento (***training dataset)***, entrenando así el algoritmo elegido. Es conveniente elegir adecuadamente el algoritmo de clasificación a utilizar, para de esta forma obtener los resultados óptimos. También se debe reservar una parte de la base de datos para definir un conjunto de datos de test o evaluación (***test dataset)*** para caracterizar el clasificador, aunque existen otras técnicas que también permiten caracterizarlo. Finalmente, se aceptará o no el clasificador resultante en función de los resultados obtenidos. En caso de no ser aceptado, se podrá recurrir a la adquisición de nuevos datos de entrenamiento, la búsqueda de nuevas *features* que permitan obtener información relevante, la elección de otro algoritmo de clasificación o la configuración de los parámetros que intervienen en el algoritmo utilizado.

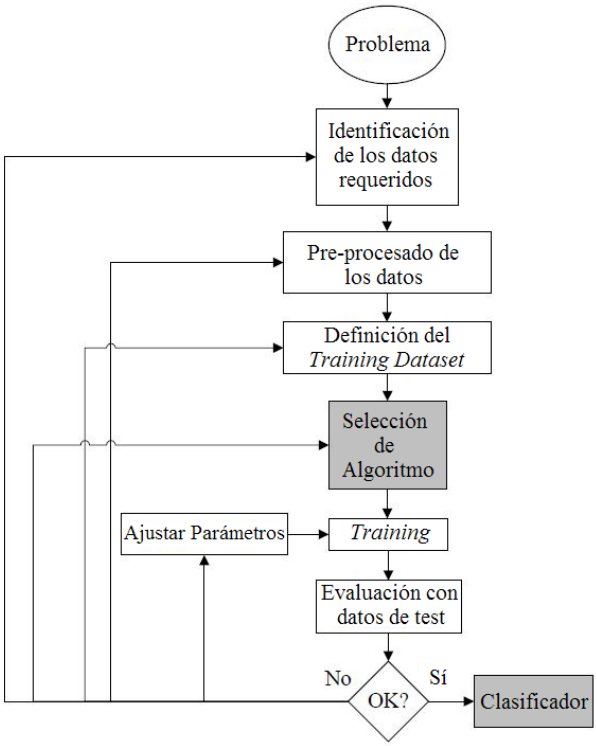

Fig. 2. Proceso de desarrollo de un clasificador supervisado [4]

IV.1. BASES DE DATOS

Siguiendo el esquema de la figura 2, el primer paso sería tener una idea clara de qué tipo de información se desea clasificar. El objetivo de este trabajo consiste en clasificar señales de audio en un conjunto de clases, por lo que la información básica vendrá dada en forma de audio.

La principal base de datos utilizada en este trabajo es la **UrbanSound8K Database** [5], recopilada a partir de audios extraídos de la página web www.freesound.org (siendo así gratuita). Se trata de una base de datos orientada a ruido urbano, la cual consta de 8732 audios (27 horas), de duración menor o igual a 4 segundos, etiquetados en 10 clases distintas:

- *Air Conditioner* (Aire acondicionado)
- *Car Horn* (Bocina de coche)
- *Children Playing* (Niños jugando)
- *Dog Bark* (Ladrido de perro)
- *Drilling* (Taladradora)
- *Engine Idling* (Motor en ralentí)
- *Gun Shot* (Disparo)
- *Jackhammer* (Martillo neumático)
- *Siren* (Sirena)
- *Street Music* (Música de calle)

Se han utilizado otras bases de datos con el objetivo de ampliar la naturaleza de las señales de audio empleadas en las clasificaciones, incluyendo así música y voz, y no únicamente ruido urbano.

Entre estas bases de datos se encuentra la base de datos **IRMAS** (*Instrument Recognition in Musical Audio Signals*), desarrollada por Juan J. Bosch, Ferdinand Fuhrmann y Perfecto Herrera en el *Music Technology Group* de la Universidad Pompeu Fabra (Barcelona) [15], en la cual incluyen 6705 audios de entrenamiento y 2874 audios de test, categorizados en 11 clases distintas de instrumentos musicales. La calidad de los audios viene determinada por una profundidad de bit de 16 bits, y una frecuencia de muestreo de 44,1 kHz. A continuación, se muestran las clases de la base de datos:

- Violonchelo
- Clarinete
- Flauta
- Guitarra Acústica
- Guitarra Eléctrica
- Órgano
- Piano
- Saxo
- Trompeta
- Violín
- Voz cantando

Entre las bases de datos utilizadas para audio de tipo voz se encuentran la **CMU PDA Database** [16], creada y grabada por Yasunari Obuchi en la Carnegie Mellon University (CMU), en el *Robust Speech Recognition Group*, y la **CMU SIN Database** (*Speech In Noise*), por lo que los audios contienen ruido. Concretamente se han utilizado las particiones PDAs y PDAm de 16 kHz. Todos los audios contienen habla inglesa de Estados Unidos. Para entrenar los clasificadores se ha utilizado la partición PDAm y la partición PDAs (hablantes 1-6), mientras que para evaluación se ha utilizado la partición PDAs (hablantes 7-11) y la base de datos CMU SIN. Entre los datos de evaluación relativos a los audios de tipo voz también se han utilizado

También se ha utilizado la base de datos **GTZAN *Music/Speech Database*** [17], de voz y música, la cual contiene 120 pistas de audio muestreadas a 22050 Hz y 16 bits de profundidad. Cada clase (Voz – Música) contiene 60 audios. Los audios de voz únicamente han sido utilizados para evaluación, mientras que los de música han sido utilizados tanto para entrenamiento como para evaluación. Para la clasificación de música y géneros musicales se ha utilizado la base de datos **GTZAN *Genre Collection*** [17], la cual incluye 100 audios de cada una de las siguientes clases de género musical:

- Blues
- Clásica
- Country
- Disco
- Hip-Hop
- Jazz
- Metal
- Pop
- Reggae
- Rock

IV.2. PRE-PROCESADO: FEATURE EXTRACTION

Tal y como se ha comentado anteriormente, es necesario depurar la información a clasificar para crear el denominado *training dataset* o conjunto de datos de entrenamiento. Para ello, es necesario realizar un pre-procesado de cada audio con el objetivo de extraer la información más relevante que permita resolver el problema en cuestión, en este caso, clasificar un conjunto de señales de audio en un determinado número de clases. Por ello, la extracción de características es una parte fundamental dentro cualquier sistema de Reconocimiento de Patrones.

El proceso de obtención de características del audio que describan al mismo en diferentes aspectos es denominado *Feature Extraction*. De tal forma que cada descriptor o *feature* aporte algún tipo de información sobre el audio. Al realizar el pre-procesado del audio, cada *feature* tomará un valor (*feature value*), y cada audio será descrito mediante un conjunto de *features*, es decir, cada audio será descrito con un conjunto de valores, formando así un ejemplo del *training dataset*, que a su vez contiene todos los audios o ejemplos de entrenamiento. Es importante destacar que a la hora de definir un *training dataset*, todos los ejemplos que forman el mismo deben estar descritos por exactamente las mismas *features*.

Existen diferentes tipos de *features* [18], y cada una de ellas aporta en teoría una información distinta. Si bien, es normal que entre distintas *features* haya cierta correlación, lo cual no es deseable, ya que, si dos *features* tienen alta correlación entre sí, estarán aportando la misma información, pudiendo ser redundante una ellas. Por el contrario, cuanta menos correlación haya entre *features*, más información distinta se tendrá para describir el audio. Es importante evitar la redundancia entre *features* para que el funcionamiento del clasificador sea más eficiente.

El proceso de selección de *features* (*feature selection*) normalmente puede ir asociado a un conocimiento previo del problema a resolver, utilizando así *features* cuyo significado permita discriminar entre clases distintas. Sin embargo, es muy común realizar el proceso extrayendo todas las *features* posibles para posteriormente probar cuáles de ellas en conjunto proporcionan el mejor resultado posible. También existen algoritmos que tratan de eliminar la redundancia, proporcionando así una reducción de dimensionalidad, como por ejemplo el algoritmo PCA (*Principal Component Analysis*).

Para visualizar a simple vista la correlación entre dos *features*, es habitual representar las mismas en forma de *scattering*. Esta representación consiste básicamente en enfrentar el valor de cada *feature* sobre un eje bidimensional para un gran conjunto de audios o ejemplos, conformando así una nube de puntos en el denominado espacio de características (*feature space*). Puesto que cada *feature*, representa una dimensión, el *scattering* siempre viene dado en forma bidimensional, es decir, para dos *features* en concreto.

En la figura 3 se muestra un ejemplo de *scattering* entre las features MFCC1 (*Mel-Frequency Cepstral Coefficient*) y el centroide espectral (*Centroid*).

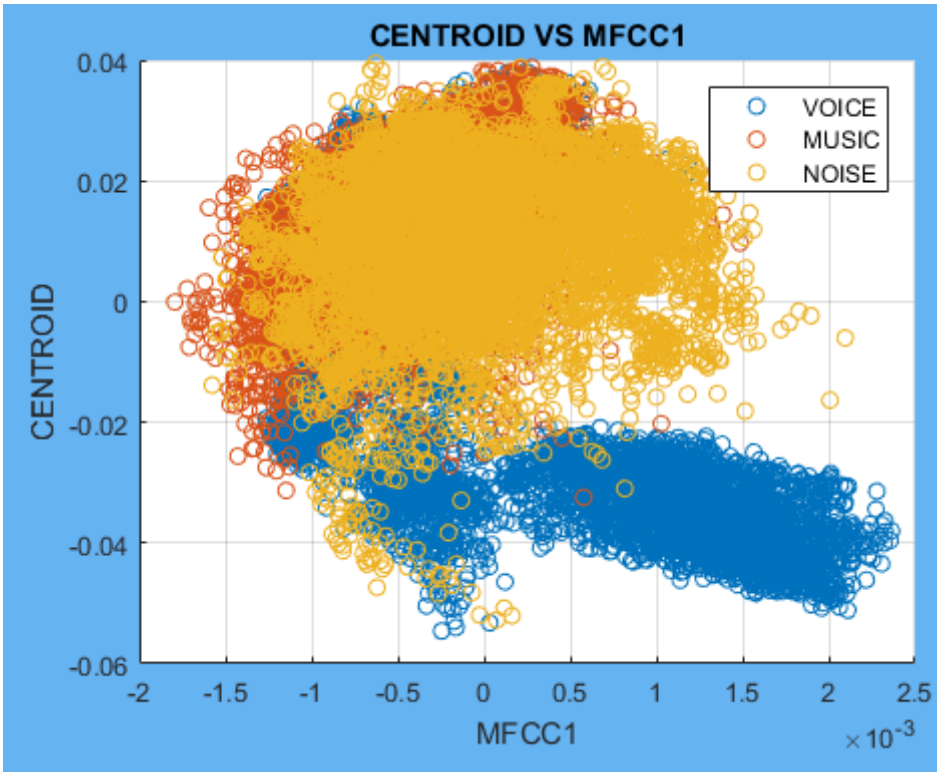

Fig. 3 Scattering MFCC1 VS Centroid

Como se puede apreciar en la figura 3, se trata de una clasificación entre voz, música y ruido, cada una de las clases representada con un color. Para cada una de ellas se representa el valor de las dos *features* en cuestión, formado cada una de ellas un eje (eje bidimensional). Como se puede apreciar, estas *features* no presentan mucha correlación entre sí, ya que, de alguna manera, juntas permiten discriminar la voz respecto de la música y el ruido. En cuanto a la música y el ruido, se puede observar que existe cierto solapamiento entre ambas clases, lo cual, a priori no permite discriminar entre estas dos clases. Sin embargo, no hay que perder de vista que se trata de una representación bidimensional, es decir, sería posible que incluyendo una nueva *feature*, y, por tanto, una tercera dimensión, estas clases quedasen separadas entre sí en el plano tridimensional.

En caso de realizar un *scattering* con la misma *feature* en cada eje, la correlación sería del 100 %, por lo que el resultado sería una línea recta, tal y como se puede apreciar en la figura 4.

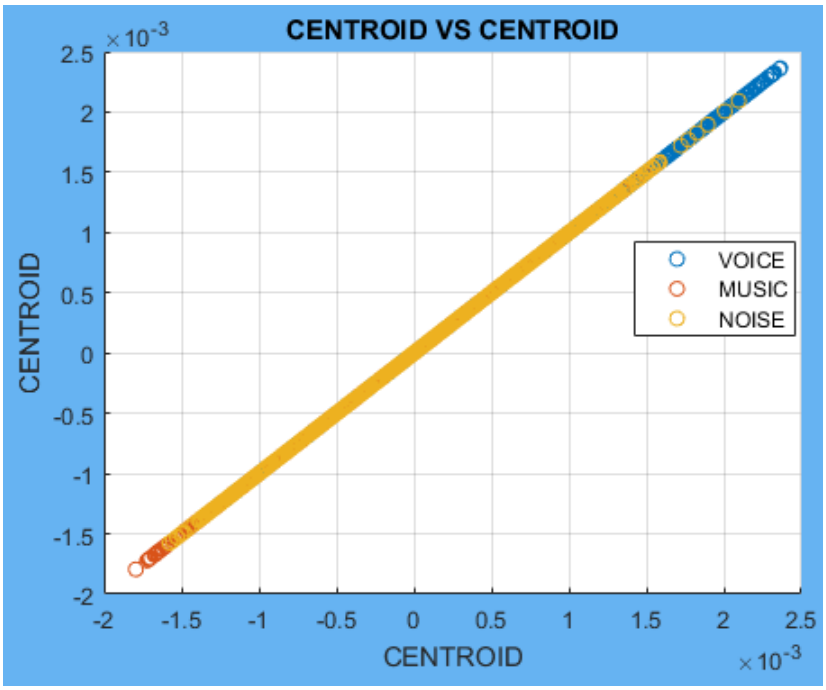

Fig. 4 Scattering de correlación total

Visualizando la representación de la figura 4, es posible determinar cuándo dos *features* tienen correlación entre sí. Por ejemplo, en la figura 5, se muestra el *scattering* entre dos *features* (*roll-off* y *centroid*) que presentan cierta correlación entre ellas, por lo que la representación tiende en cierto modo a una recta.

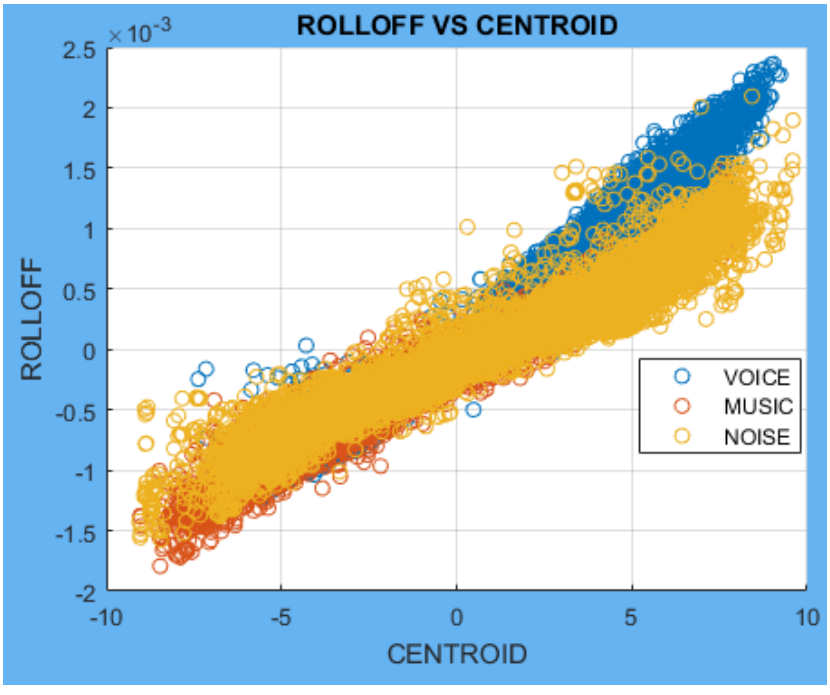

Fig. 5 Scattering Roll-Off VS Centroid

En el caso de *features* orientadas al audio, existen distintos tipos de taxonomías, como por ejemplo la mostrada en la figura 6 [18].

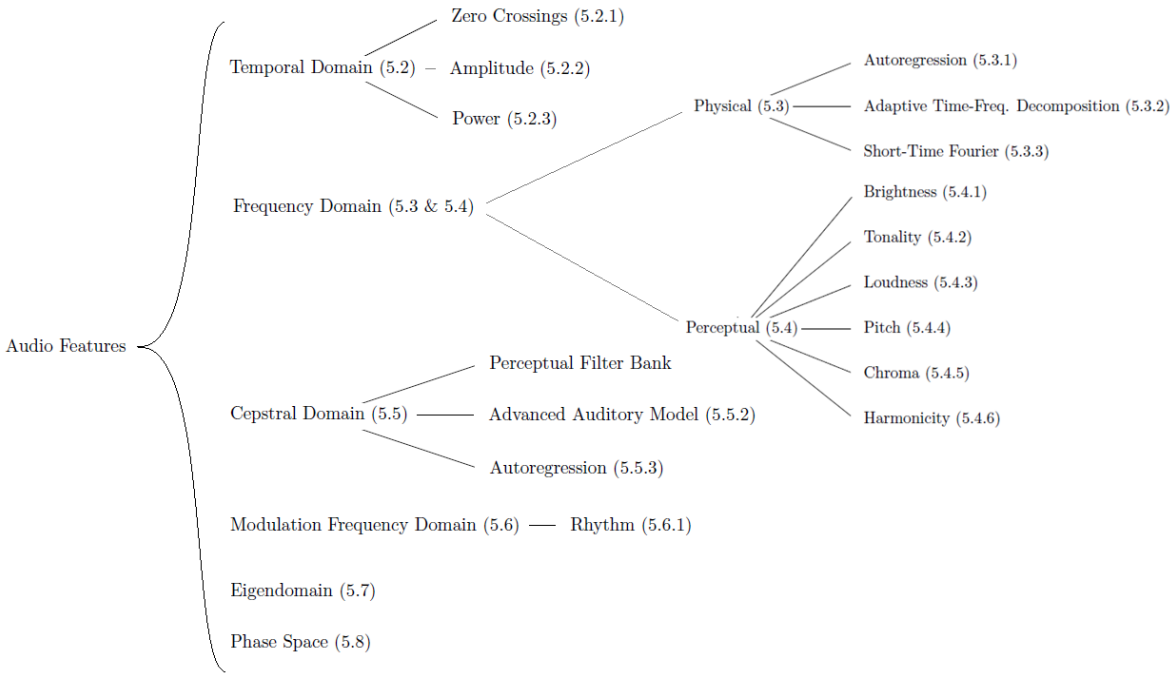

Fig. 6. Taxonomía de *features* [4]

Otro tipo de taxonomía [14] clasifica los distintas *features* en: temporales, de energía, espectrales, armónicas y perceptuales. Sin embargo, en este trabajo se han simplificado la clasificación a tres categorías:

- **Dominio Cepstral**: aquellas obtenidas en el domino cepstral.
  - *Mel-Frequency Cepstral Coefficients* (MFCC) + Δ-MFCC + ΔΔ-MFCC
- **Dominio Espectral**: aquellas obtenidas en el dominio de la frecuencia que definen la forma del espectro.
  - *Centroid*
  - *Spread*
  - *Skewness*
  - *Kurtosis*
  - *Slope*
  - *Decrease*
  - *Roll-off*
  - *Flux (media)*
  - *Flux (varianza)*
  - *MPEG7 Centroid (ASC)*
  - *MPEG7 Spread (ASS)*
  - *MPEG7 Flatness (ASF)*
- **Dominio Temporal**: aquellas obtenidas en el domino del tiempo cuyas características están basadas en la forma de onda, envolvente y en la autocorrelación.
  - *Pitch*
  - *Zero-Crossing Rate* (ZCR)
  - *Log-Attack Time*
  - *AM Index (Amplitude Modulation)*

Como paso previo a la extracción de dichas *features*, es habitual realizar algún tipo de mejora en el audio a tratar, por ejemplo, eliminar ruido y componente continua, o incluso separación de fuentes. En este trabajo no se ha realizado ningún algoritmo de supresión de ruido ni separación de fuentes, pero sí se ha contemplado la eliminación de componente continua o *DC Offset* (*Direct Current*). Para ello, se utiliza un filtro IIR paso alto (*Infinite Pulse Response*) utilizado para aplicaciones de tiempo real, el cual viene determinado por la expresión (1), en este caso $\alpha = 0.95$.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1 - z^{-1}}{1 - \alpha z^{-1}} \tag{1}$$

IV.2.1. Mel-Frequency Cepstral Coefficients (MFCC)

Los MFCC [13,19] son *features* obtenidas en el dominio cepstral, el cual se obtiene como la Transformada Inversa de Fourier del logaritmo natural de la magnitud del espectro (*Spectrum*), dando lugar al dominio *Cepstrum*. Estas *features* han demostrado ser muy útiles en el ámbito del reconocimiento de voz (ASR – *Automatic Speaker Recognition*). Habitualmente se utilizan junto con su derivada (Δ-MFCC) y su segunda derivada (ΔΔ-MFCC), las cuales aportan información temporal (prosodia, acento…), ya que, describen la variación de cada MFCC.

El algoritmo de extracción conlleva un conjunto de operaciones, tal y como se muestra en la figura 7.

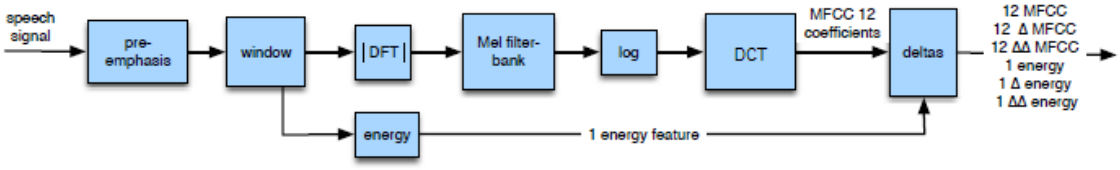

Fig. 7 Proceso de extracción de los MFCC [13]

En realidad, los MFCC son *features* en forma de vector [13], es decir, se trata de más de un valor, pudiendo obtenerse tantos MFCC como coeficientes de la Transformada Discreta del Coseno (DCT), además de sus derivadas.

En primer lugar, se realiza un **preénfasis** (*pre-emphasis*) de la señal, lo que equivale a un filtrado tipo FIR (*Finite Impulse Response*) para realzar la alta frecuencia, el cual se muestra en la expresión (2). Este realce proporciona una mejor relación señal a ruido (SNR – *Signal to Noise Ratio*) [19], además de introducir aproximadamente +6 dB/octava para compensar la atenuación de alta frecuencia en el proceso de generación del habla.

$$H(z) = 1 - \alpha z^{-1} \qquad 0{,}9 \leq \alpha \leq 1 \tag{2}$$

Posteriormente, se realiza un **enventanado** de la señal, dividiendo la señal en bloques o *frames* (***framing***) de menor duración con cierto solapamiento entre sí, aplicando una ventana (Hamming, Hanning…) a cada *frame*. En la figura 8 se puede observar este proceso, dividiendo la señal en bloques de tamaño *L* (*frame length*), con cierto solapamiento entre sí (*overlapping*). A cada bloque se le multiplica por una ventana muestra a muestra (en este caso Hamming, ya que introduce menos distorsión que otras ventanas), de tal forma que las muestras de los extremos de cada bloque tengan menor peso que las muestras centrales, ya que, de lo contrario, se producirían efectos indeseados en alta frecuencia. Además, puesto que existe solapamiento, no se tiene pérdida de información por el efecto de atenuación de los extremos de cada bloque.

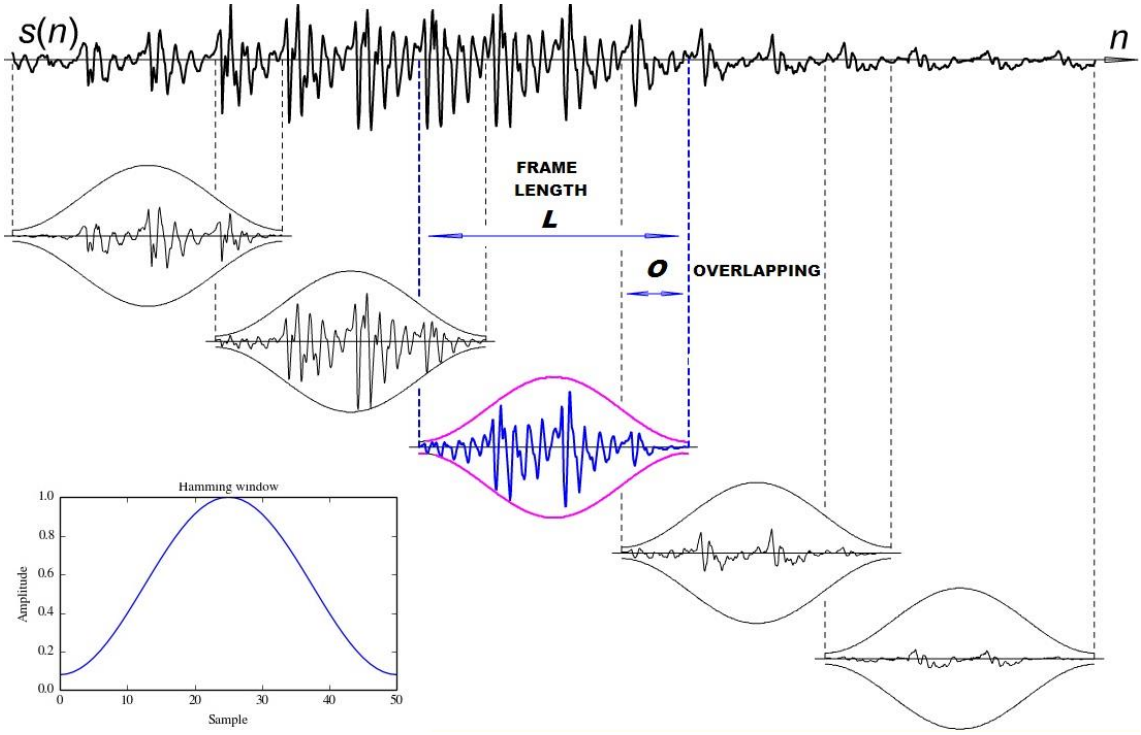

Fig. 8. Enventanado

De cada *frame* se obtiene la **Transformada Discreta de Fourier** (DFT – *Discrete Fourier Transform*), la cual se obtiene de la expresión (3), obteniendo así la información de cada bloque en el dominio espectral. En la implementación práctica, este proceso es realizado mediante al algoritmo **FFT** (*Fast Fourier Transform*) o **Transformada Rápida de Fourier**.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N} \qquad 0 \leq k \leq N-1 \tag{3}$$

Donde *k* es un punto determinado de la DFT de los *N* posibles, siendo *N* mayor o igual a *L* (longitud de cada bloque de muestras).

Posteriormente, se aplica un **banco de filtros Mel** sobre cada *frame* para reducir el espectro en potencia (al cuadrado) a un conjunto de bandas espectrales. Este filtrado está basado en la percepción humana, teniendo así mayor resolución en baja frecuencia y menor en alta frecuencia. A diferencia de la escala lineal de frecuencias utilizada en el cómputo de la FFT, la escala Mel es una escala perceptual de tonalidades equidistantes, es decir, es proporcional al logaritmo de las frecuencias lineales, asemejándose así a la percepción humana [19]. En la expresión (4) se presenta la ecuación que permite pasar de frecuencias a frecuencias Mel.

$$Mel(f) = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{4}$$

El banco de filtros Mel es por tanto un conjunto de filtros triangulares equiespaciados entre sí en la escala Mel, da tal forma que, al volver a la escala lineal de frecuencias, las frecuencias centrales de dichos filtros se distribuyen de forma logarítmica en el espectro, dejando de ser filtros equiespaciados. Para devolver estas frecuencias Mel a frecuencias lineales, basta con aplicar la función inversa de la expresión (4), la cual se muestra en la expresión (5).

$$f = 700 \cdot \left(10^{\frac{Mel(f)}{2595}} - 1\right) \tag{5}$$

En la figura 9 se puede observar cómo quedarían estos filtros distribuidos en el espectro. Se trata de un ejemplo en cual se ha utilizado una frecuencia de muestreo de 16000 Hz, con un tamaño de FFT de 1024 puntos y 34 filtros Mel.

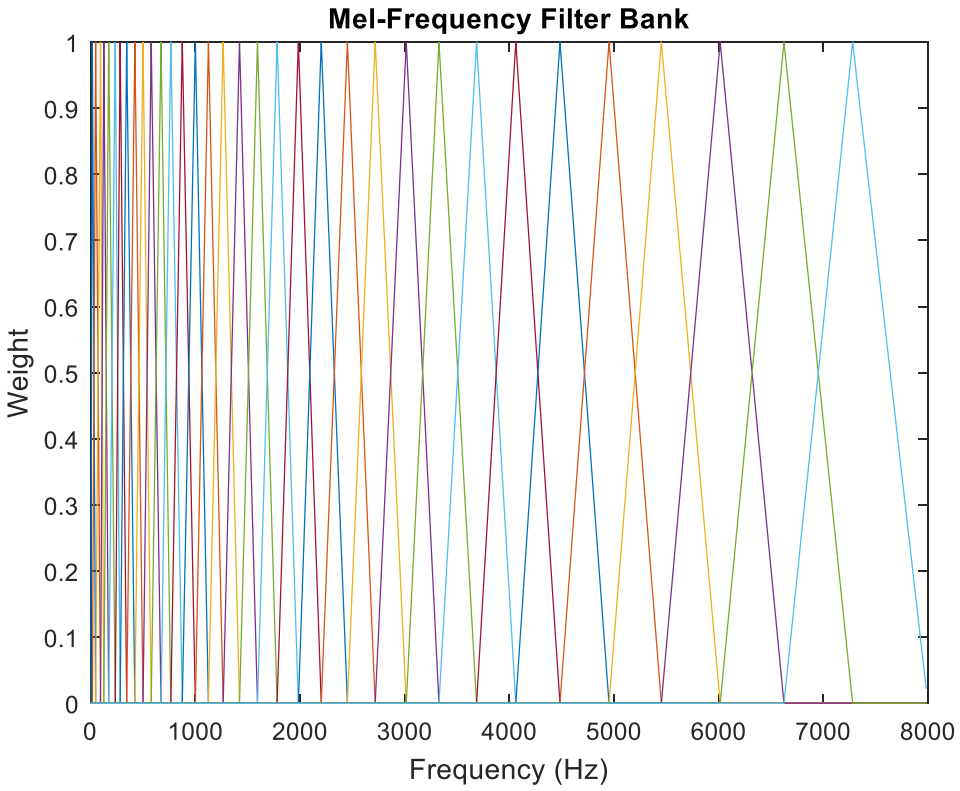

Fig. 9. *Mel-Frequency Filter Bank*

Este conjunto de filtros viene dado según la expresión (6) [19], donde *k* es un punto ó *bin* de la FFT, *m* es el número del filtro, siendo *f(m)* el *bin* correspondiente a la frecuencia central del filtro. Por lo tanto, para cada filtro, se tiene un vector de ponderaciones de cada punto de la FFT, ponderando con gran peso aquellos puntos que se encuentren alrededor de la frecuencia central, y con valor nulo para el resto del espectro. En la figura 10 se muestra el resultado de cada filtro para cada punto de la FFT. Nótese que los filtros se encuentran perfectamente solapados, da tal manera que el peso total de cada *bin* distribuido en dos filtros es igual a la unidad, satisfaciendo la expresión (7) (no necesariamente para el último filtro).

$$Hm(k) = \begin{cases} 0\ , & k < f(m-1) \\ \dfrac{k - f(m-1)}{f(m) - f(m-1)}\ , & f(m-1) \le k \le f(m) \\ \dfrac{f(m+1) - k}{f(m+1) - f(m)}\ , & f(m) \le k \le f(m+1) \\ 0\ , & k > f(m+1) \end{cases} \tag{6}$$

$$\sum_{m=0}^{M-1} H_m(k) = 1 \tag{7}$$

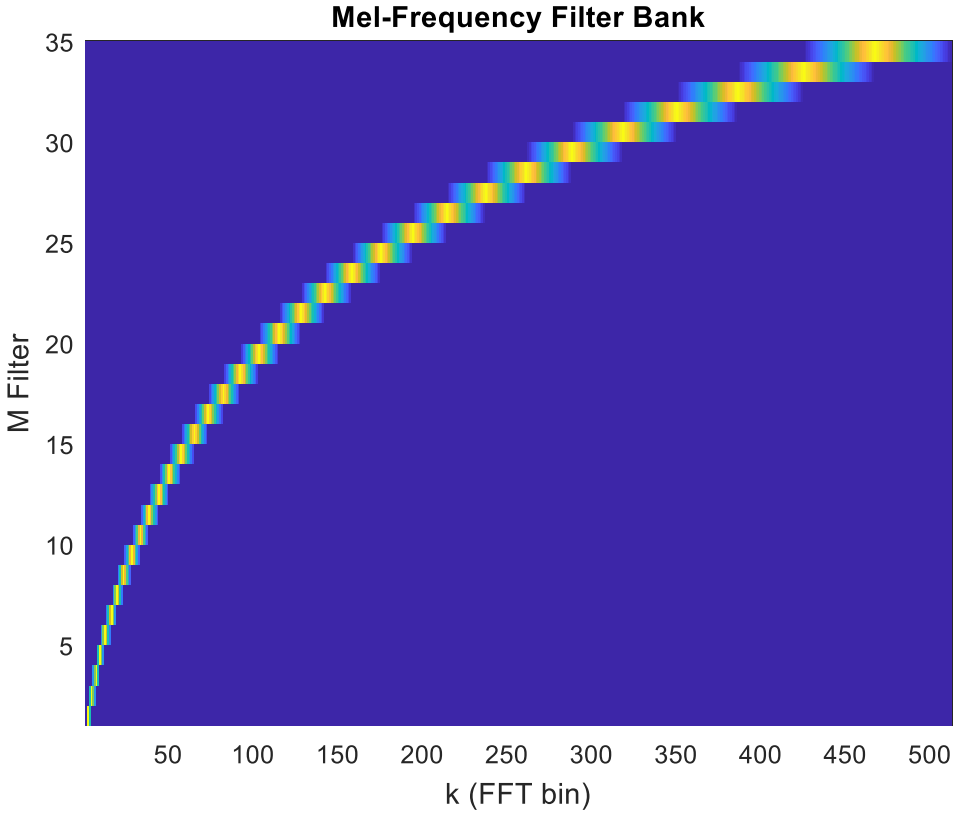

Fig. 10. Mel-Frequency Filter Bank

Este filtrado se aplica a cada *frame*, obteniendo M valores del espectro, correspondientes a cada una de las bandas espectrales definidas por los filtros triangulares. Una vez se tiene la magnitud de cada una de las bandas, se aplica el **logaritmo natural**, ya que, la percepción humana de la sonoridad también es procesada de una forma logarítmica. En la expresión (8) se muestra la ecuación que describe este proceso [4].

$$S(m) = log\left[\sum_{k=0}^{N-1} |X(k)^2| \cdot H_m(k)\right] \qquad 0 \leq m \leq M \tag{8}$$

Finalmente, se aplica la **Transformada Discreta del Coseno** o **DCT** (*Discrete Cosine Transform*) para obtener el ***cepstrum***. El término cepstrum proviene de las cuatro primeras letras de ***spectrum*** al revés, ya que, se trata del espectro del logaritmo del espectro de la señal, es decir, trata el logaritmo del espectro de la señal como una nueva señal en sí misma. En realidad, el dominio cepstral se obtiene mediante la **Transformada Discreta de Fourier Inversa** (IDFT – *Inverse Discrete Fourier Transform*). Sin embargo, puesto que en el proceso de cálculo de los MFCC no se contempla la parte imaginaría, es habitual realizar este proceso mediante la DCT, la cual se aplica directamente sobre la magnitud logarítmica, según la expresión (9).

$$C(k) = \sum_{m=0}^{M-1} S(m)\cos(\pi k(m + 1/2)/M) \quad 0 \leq k < K \tag{9}$$

Donde *K* es el número máximo de coeficientes de la DCT, determinando *k* el número del MFCC a extraer. *M* determina el número máximo de puntos sobre los que se aplicará la DCT, que coincide con el número de filtros Mel empleados, siendo *m* el indicador de cada filtro.

En la figura 11 se muestra de forma representativa cada componente de la DCT aplicada sobre un conjunto de 34 filtros. Como se puede apreciar, la primera componente de la DCT (*k=0*) equivale a la energía de la señal, ya que, la ponderación para cada filtro es la unidad. La segunda componente es una relación entre baja y alta frecuencia. A medida que aumenta el orden de las componentes de la DCT aumenta el número de ciclos del coseno. Puesto que se trata de un dominio logarítmico, las sumas y restas equivalen a multiplicaciones y divisiones, respectivamente, por lo que conceptualmente la DCT realiza una serie de ratios entre bandas espectrales. Esto hace que la información más relevante se concentre en los primeros coeficientes cepstrales (MFCC), decorrelando así la información y eliminando redundancia.

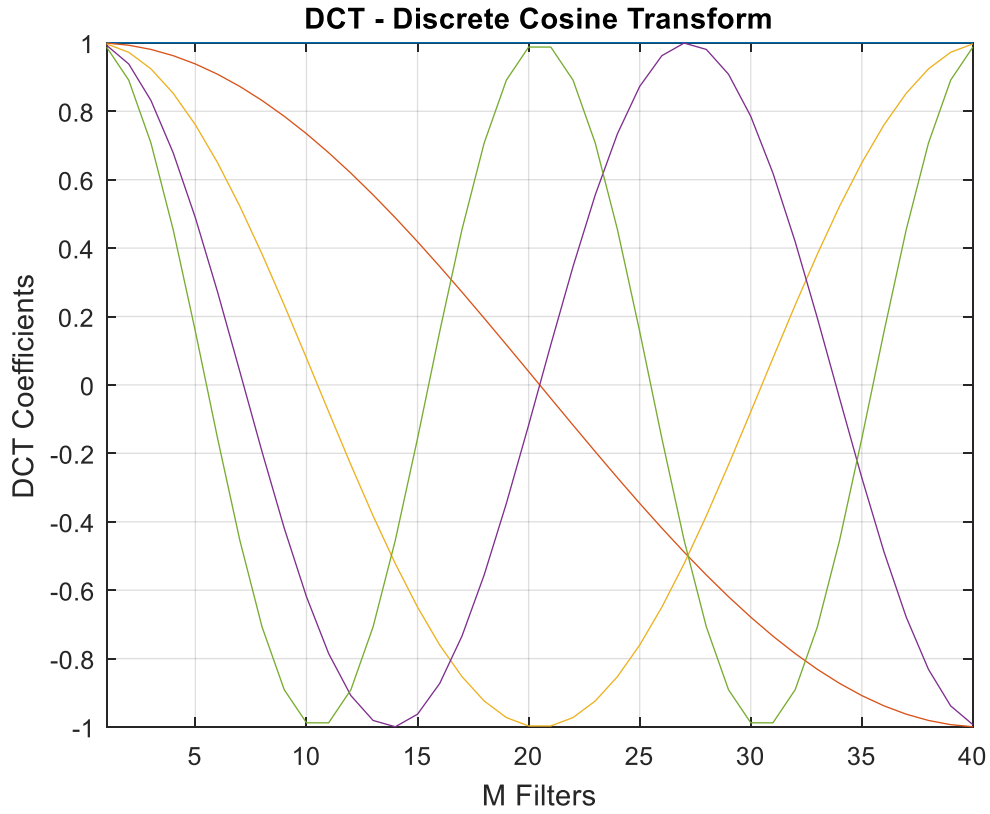

Fig. 11. DCT - *Discrete Cosine Transform*

Por último, se calculan la primera (Δ-MFCC) y segunda derivada (ΔΔ-MFCC) para cada MFCC. Estas se obtienen mediante la diferencia entre MFCC de distintos *frames* consecutivos.

IV.2.2. Centroide Espectral (Spectral Centroid)

El centroide espectral es un tipo de *feature* estadístico que aporta información acerca de la forma del espectro, concretamente define el centro de gravedad del espectro, es decir, considera el espectro como una distribución de probabilidad, la cual viene determinada por la amplitud del espectro para cada frecuencia, obteniendo así la frecuencia **media** ponderada en amplitud [14]. El centroide se calcula según la expresión 10, donde *x* es la frecuencia y *p(x)* la amplitud (o probabilidad) del espectro normalizado, en la frecuencia *x*.

$$\begin{cases} \mu = \int x\, p(x)\, \delta x \\ p(x) = \dfrac{amplitud(x)}{\sum_x amplitud(x)} \end{cases} \tag{10}$$

Existen otros tipos de centroide, como el centroide armónico, el cual puede ser muy útil a la hora de discriminar entre instrumentos musicales. Sin embargo, éste implica la extracción previa de la nota fundamental, así como sus armónicos.

En la figura 12 se muestra un ejemplo de dos centroides armónicos obtenidos para un cuenco tibetano excitado por fricción y por golpe. Como se puede apreciar, el centroide está muy correlado con el *pitch* de la señal, ya que la tonalidad percibida tendrá mucha parte de la energía del espectro. Sin embargo, tal y como se puede observar en el ejemplo de excitación por golpe, la excitación de armónicos produce un desplazamiento del centroide hacía frecuencias mayores.

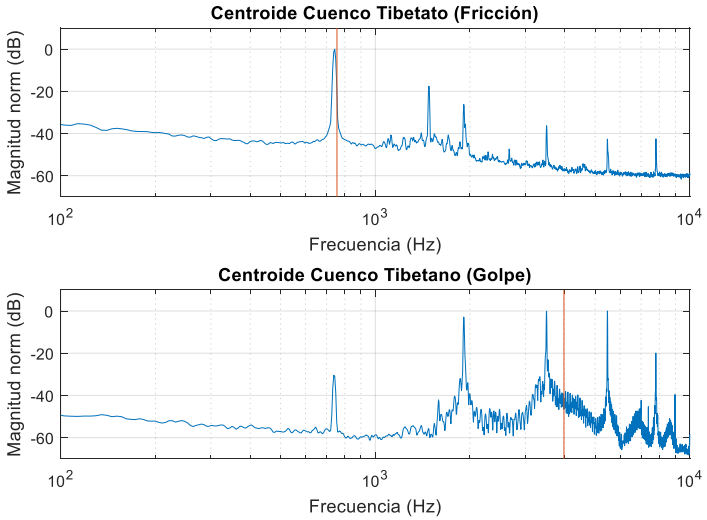

Fig. 12. Ejemplo Centroide

IV.2.3. Envergadura Espectral (Spectral Spread)

El *spread* o envergadura espectral define la concentración de energía alrededor de la media [14], es decir, del centroide. En términos estadísticos se correspondería con la varianza, que se obtiene según la expresión (11), siendo su raíz la desviación típica σ, la cual se utiliza para el cálculo de parámetros de orden superior (como la asimetría o la curtosis espectral).

$$\sigma^2 = \int (x-\mu)^2 \cdot p(x)\, \delta x \tag{11}$$

IV.2.4. Asimetría Espectral (Spectral Skewness)

Al igual que el *spread*, el *skewness* o asimetría [14] depende también del centroide, aportando información de la asimetría energética del espectro respecto a su centroide (media). Esta asimetría se calcula según la expresión (12).

$$\begin{cases} m_3 = \int (x-\mu)^3 \cdot p(x)\, \delta x \\ \gamma_1 = \dfrac{m_3}{\sigma^3} \qquad (skewness) \end{cases} \tag{12}$$

En función del valor del *skewness*, el espectro será simétrico o asimétrico con mayor energía hacia la izquierda (menor frecuencia) o hacia la derecha (mayor frecuencia) del centroide:

$$\begin{cases} \gamma_1 = 0 \rightarrow \text{Distribución simétrica} \\ \gamma_1 < 0 \rightarrow \text{Distribución asimétrica hacia la derecha} \\ \gamma_1 > 0 \rightarrow \text{Distribución asimétrica hacia la izquierda} \end{cases}$$

IV.2.5. Curtosis Espectral (Spectral Kurtosis)

La curtosis espectral describe el grado de planicie o "picudez" del espectro alrededor del centroide. La curtosis es calculada según la expresión (13) [14].

$$\begin{cases} m_4 = \int (x-\mu)^4 \cdot p(x)\, \delta x \\ \gamma_2 = \dfrac{m_4}{\sigma^4} \qquad (kurtosis) \end{cases} \tag{13}$$

Al igual que el *skewness*, la curtosis indica lo plano o picudo que es el espectro en función de los valores obtenidos, respecto a una distribución normal ($\gamma_2 = 3$):

$$\begin{cases} \gamma_2 = 3 \rightarrow \text{Distribución normal} \\ \gamma_2 < 3 \rightarrow \text{Distribución más plana} \\ \gamma_2 > 3 \rightarrow \text{Distribución más picuda} \end{cases}$$

IV.2.6. Spectral Roll-off

El *roll-off* también considera el espectro como una distribución de probabilidad, siendo éste el valor para el cual se supera una determinada probabilidad en la distribución de probabilidad acumulada. En este trabajo, se ha utilizado el *roll-off* al 0.85 de probabilidad, aunque también suele utilizarse como 0.95 [14]. En términos espectrales, el *roll-off* sería aquella frecuencia para la cual se tiene el 85 % de la energía por debajo de la misma (distribución de energía acumulada).

En la figura 13 se muestra un ejemplo para un roll-off obtenido al 85 %. Se representa según la expresión (14), donde *fc* es la frecuencia de *roll-off* y *fs* la frecuencia de muestreo.

$$\sum_{0}^{fc} a^2(f) = 0.85 \sum_{0}^{fs/2} a^2(f) \tag{14}$$

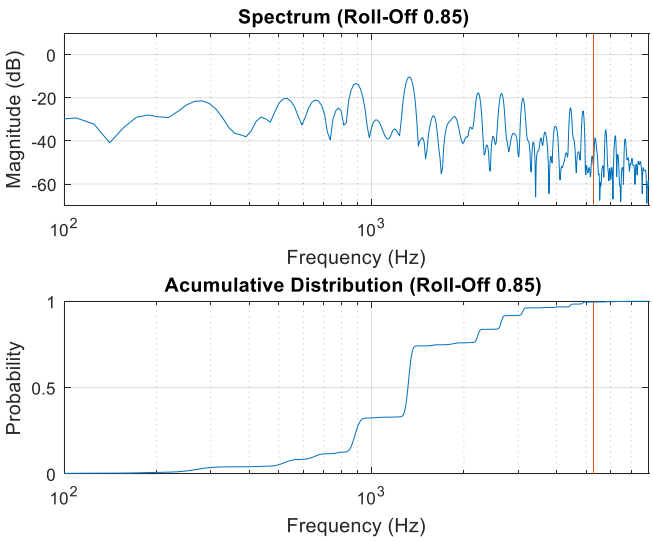

Fig. 13. Ejemplo de *Roll-Off* 85 %

IV.2.7. Pendiente Espectral (Spectral Slope)

El *slope* o pendiente espectral describe el grado de decaimiento de la amplitud espectral. Se obtiene directamente de realizar una regresión lineal, obteniendo así la pendiente de la recta [14]. El cálculo se obtiene directamente de la expresión (15), donde *a(k)* es la amplitud correspondiente al *bin* frecuencial *k*, y *f(k)* la frecuencia correspondiente a dicho *bin* frecuencial.

$$\begin{cases} \hat{a}(f) = slope \cdot f + constante \\ slope = \dfrac{1}{\sum_k a(k)} \dfrac{N\sum_k f(k)\cdot a(k) - \sum_k f(k)\cdot \sum_k a(k)}{N\sum_k f^2(k) - (\sum_k f(k))^2} \end{cases} \quad (15)$$

En la figura 14 se muestra la pendiente obtenida para un espectro dado, se representa la ecuación de la recta de la expresión (15), sin tener en cuenta la constante. Además, el primer término del cálculo del *slope* normaliza la pendiente en energía, por lo que la representación utiliza el *slope* desnormalizado.

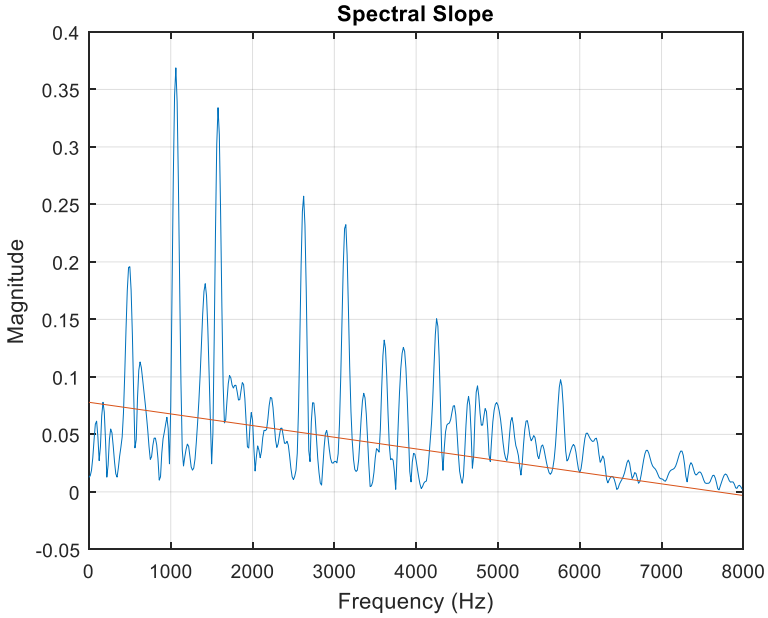

Fig. 14. Ejemplo de pendiente espectral (*spectral slope*)

IV.2.8. Decaimiento Espectral (Spectral Decrease)

Al igual que el *slope*, el decaimiento o *decrease* describe el grado de decaimiento de la amplitud espectral. Sin embargo, el cómputo de esta *feature* tiene mayor similitud con la percepción humana [14]. El cálculo del *spectral decrease* viene dado por la expresión (16).

$$decrease = \frac{1}{\sum_{k=2}^{K} a(k)} \sum_{k=2}^{K} \frac{a(k) - a(1)}{k-1} \quad (16)$$

IV.2.9. Variación Espectral (Spectral Flux)

La variación espectral, también denominada *spectral flux*, aporta información acerca de la variación espectral que existe entre *frames* consecutivos. Esta *feature* es útil para diferenciar entre música y sonidos del entorno [20]. Se calcula según la expresión (17) [14], obteniendo el *flux* correspondiente a una trama *t*, que depende de la amplitud del espectro de la trama anterior *a(t-1, k)*. En ocasiones, también es calculado como la diferencia cuadrática entre espectros [20].

$$flux(t) = 1 - \frac{\sum_k a(t-1,k)a(t,k)}{\sqrt{\sum_k a(t-1,k)^2} \cdot \sqrt{\sum_k a(t,k)^2}} \tag{17}$$

Este parámetro toma valores entre 0 y 1, siendo casi nulo cuando existe gran similitud entre espectros, y viceversa. En la figura 15 se muestra un ejemplo de la variación espectral existente en un audio en el que únicamente se escucha lluvia, otro en el que se escucha un clarinete y otro en el que se escucha una voz. Tal y como se puede apreciar, la variación que sufre el sonido producido por la lluvia a lo largo del tiempo apenas varía, podría decirse incluso que la variación que existe entre *frames* es casi constante. Sin embargo, tanto en el caso del clarinete como en la voz, las variaciones son muy abruptas.

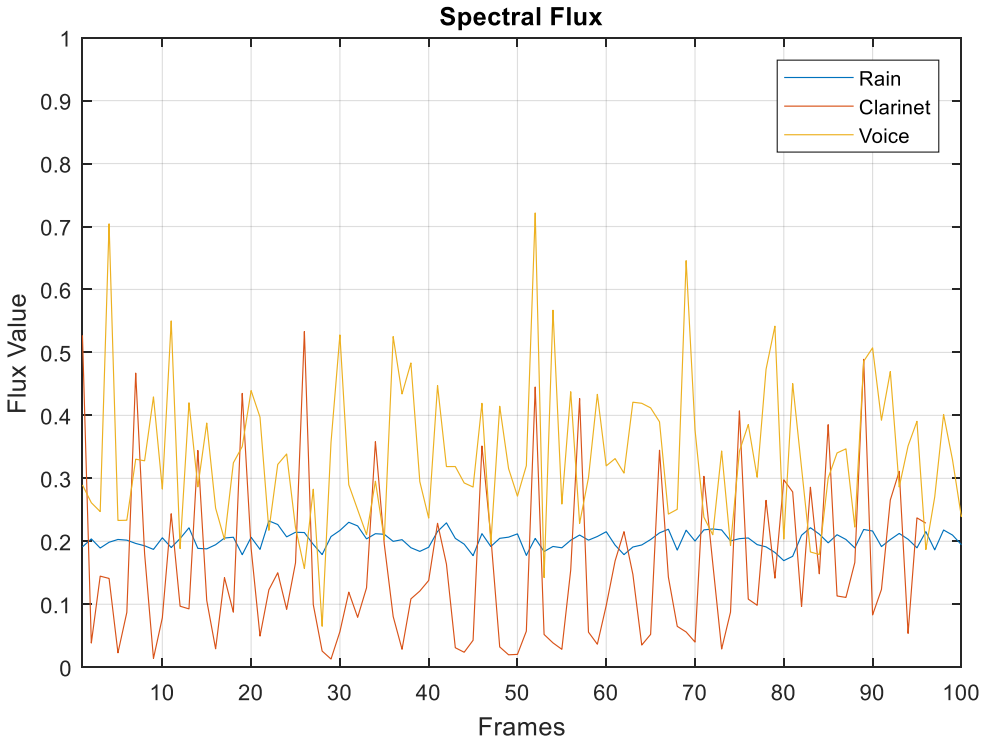

Fig. 15. Ejemplo de variación espectral (*Spectral Flux*)

Para finalmente obtener la *feature*, se realiza un promediado de la variación, pudiendo obtener también su varianza.

IV.2.10. Audio Spectrum Centroid (MPEG-7)

MPEG-7 es un estándar de la Organización Internacional de Normalización y la Comisión Electrotécnica Internacional (ISO/IEC) desarrollado por el grupo MPEG (*Moving Picture Experts Group*) que tiene como finalidad la descripción de contenidos de información audiovisual para una mejor gestión de los mismos gracias a los metadatos. En concreto "MPEG-7 Audio" aporta un conjunto de *features* de bajo nivel, utilizando en este trabajo únicamente tres de ellas, obtenidas en el dominio espectral: *Audio Spectrum Centroid*, *AS-Spread* y *AS-Flatness*.

El *Audio Spectrum Centroid* (ASC) de MPEG-7 describe el centroide espectral en una escala de frecuencia logarítmica del espectro, definiendo así el centro de gravedad del mismo de una forma más perceptual [20]. Antes de proceder al cálculo del mismo, se realiza una suma de aquellas frecuencias por debajo de 62.5 Hz, estableciendo como frecuencia central de dicha banda 31.25 Hz. De esta forma, se evita que el cálculo del ASC proporcione demasiado peso a la componente continua y las bandas de muy baja frecuencia. El cálculo se obtiene según la expresión (18) [1] tras haber obtenido el espectro en potencia mediante la FFT.

$$ASC = \frac{\sum_{k\prime=0}^{\frac{N_{FFT}}{2}-K_{low}} \log_2\left(\frac{f'(k')}{1000}\right) P'(k')}{\sum_{k\prime=0}^{\frac{N_{FFT}}{2}-K_{low}} P'(k')} \tag{18}$$

Donde $K_{low} = \lfloor 62.5\ Hz/\Delta F \rfloor$, siendo ΔF la resolución espectral ($\Delta F = fs/N_{FFT}$) y *f'(k') y P'(k')*, frecuencias y potencias para los nuevos valores de los índices de las frecuencias discretas (*k'*) una vez unificada la baja frecuencia (es decir, *f(k'=0)=31,25 Hz*).

IV.2.11. Audio Spectrum Spread (MPEG7)

Al igual que el ASC, el *Audio Spectrum Spread* aporta la misma información que el *spectral spread*, salvo que utiliza la escala de frecuencias logarítmica, indicando la distribución del espectro alrededor del ASC [1]. El cálculo se obtiene mediante la expresión (19).

$$ASS = \sqrt{\frac{\sum_{k\prime=0}^{\frac{N_{FFT}}{2}-K_{low}} \left[\log_2\left(\frac{f'(k')}{1000}\right) - ASC\right]^2 P'(k')}{\sum_{k\prime=0}^{\frac{N_{FFT}}{2}-K_{low}} P'(k')}} \tag{19}$$

IV.2.12. Audio Spectrum Flatness (MPEG7)

El *Audio Spectrum Flatness* (ASF) es otra *feature* de bajo nivel de MPEG-7, la cual realiza una comparación del espectro de la señal con un espectro totalmente plano. Esta comparación se realiza por bandas de frecuencia, para posteriormente realizar una media de todas las bandas, obteniendo un único escalar que describe la planicie total. La planicie espectral de una banda viene dada como el ratio entre la media geométrica y la media aritmética de los coeficientes del

espectro en potencia de dicha banda [20]. Valores bajos de ASF implicarán presencia de componentes armónicas, mientras que valores altos implicarán ruido o señales impulsivas.

IV.2.13. Zero Crossing Rate (ZCR)

El *Zero-Crossing Rate* es un tipo de *feature* obtenida en el dominio temporal, mide el número de cruces por cero de una señal. Los sonidos que presentan periodicidad suelen tener menor valor de ZCR que aquellos con gran presencia de ruido [14]. El cálculo del ZCR puede obtenerse según la expresión (20) [1], donde *N* es el número de muestras de la señal *s(n)* y *fs* su frecuencia de muestreo.

$$ZCR = \frac{1}{2}\left(\sum_{n=1}^{N-1}\left|signo\big(s(n)\big) - signo(s(n-1))\right|\right)\frac{fs}{N} \tag{20}$$

El ZCR resulta útil para determinar si una señal contiene o no voz [21]. También puede utilizarse como una estimación rápida del *pitch*, sin embargo, solo aporta resultados aceptables cuando la señal contiene una gran periodicidad.

IV.2.14. Pitch

El pitch es una de las percepciones básicas de un sonido, describe la frecuencia percibida en una escala de altura, grave-agudo. Si bien, también puede entenderse como la frecuencia fundamental de un sonido. Esta *feature* puede venir dada en forma de histograma (*Pitch Histogram*), donde el análisis del *pitch* viene determinado generalmente por notas musicales.

En este caso, el *pitch* ha sido orientado hacia la búsqueda de la frecuencia fundamental, para ello, existen distintos métodos, como la autocorrelación, el dominio espectral y el dominio cepstral, o una combinación de estos [18]. Por lo general, la obtención de la frecuencia fundamental comienza por el cálculo de la autocorrelación, precisando posteriormente con alguna otra técnica de las mencionadas. El *pitch* utilizado en este trabajo viene dado únicamente por la autocorrelación.

La autocorrelación permite obtener patrones repetitivos dentro de una misma señal. Ésta consiste en una multiplicación de la señal original por ella misma desplazada un intervalo de tiempo, por ello, cuando este intervalo coincide con un periodo de la señal, la autocorrelación toma un valor alto, ya que, si la señal es pseudo-periódica, dos periodos consecutivos serán muy parecidos entre sí. En el caso de una señal de ruido blanco, la autocorrelación únicamente tendrá un valor alto con desplazamiento nulo, ya que la señal es completamente aleatoria y no existirá ningún proceso repetitivo. La autocorrelación se define según la expresión (21).

$$R_x(m) = E\{\, x[n]\, x[n-m]\,\} \tag{21}$$

En la figura 16 se muestra un ejemplo de un determinado *frame* de una señal instrumental de una flauta. Como se pude apreciar, la señal presenta gran periodicidad, por lo que, cuando el desplazamiento coincide con un periodo (en este caso, 2.8 ms), entonces la autocorrelación coincide con un máximo. Por lo tanto, el *pitch* vendrá determinado por el desplazamiento del primer máximo significativo de la autocorrelación, en este caso: *pitch = 1/2.8 ms ≈ 347 Hz*.

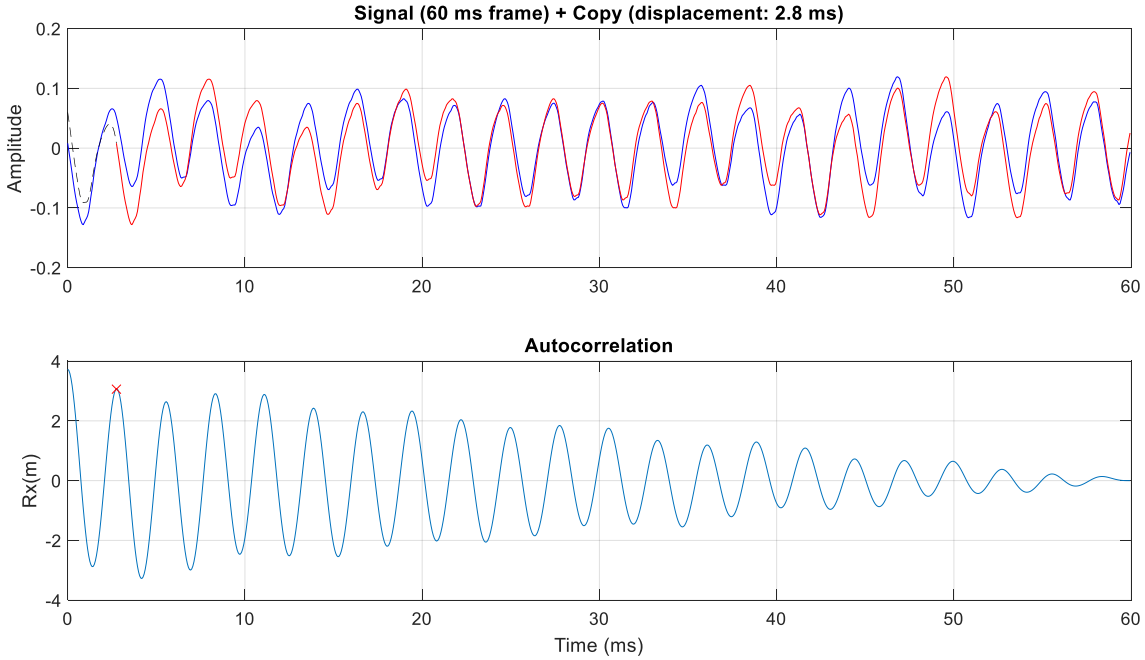

Fig. 16. Ejemplo de obtención del *pitch* mediante autocorrelación

En la figura 17 de muestra el espectro del *frame* analizado. Como se puede observar, el valor de la nota fundamental coincide con el valor obtenido a través de la autocorrelación.

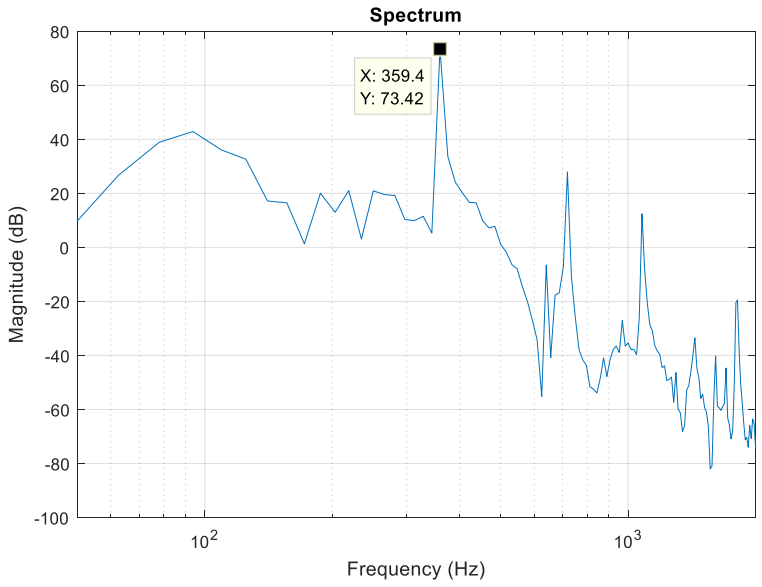

Fig. 17. Espectro del *frame* analizado

En el caso de la FFT, la precisión del cálculo viene determinada por la resolución física *(Δf=1/L(segundos)=1/60ms=16,6 Hz* para una ventana rectangular, *L=longitud de la ventana*) y la resolución computacional *(Δf=fs/N_{FFT}=16000 Hz/1024=15,6 Hz*), en este caso 16,6 Hz y 15,6 Hz, respectivamente. Sin embargo, la precisión de la autocorrelación depende del desplazamiento mínimo, es decir, una muestra, dependiendo así de la frecuencia de muestreo, y de la longitud del *frame*, el cual determinará el mínimo *pitch* posible en la detección.

En las figuras 16 y 17 se trata de un caso bastante sencillo, ya que se trata de un tono bien definido, por lo que bastaría con analizar el espectro para determinar el tono. Sin embargo, para señales más ruidosas, la obtención del *pitch* puede ser errónea, por ello, se recurre a combinar el método de la autocorrelación con alguno de los otros métodos (espectral y cepstral) para precisar resultados. En las figuras 18 y 19, se muestra un ejemplo de una guitarra eléctrica distorsionada tocando una nota La (440 Hz) con otros instrumentos de fondo, obteniendo un espectro con gran contenido armónico bastante confuso (figura 19). Sin embargo, el pitch detectado en la autocorrelación determinado por el primer máximo es de 444 Hz (2.25 ms), tal y como se puede apreciar en la figura 20, resultado del algoritmo de detección de *pitch* desarrollado.

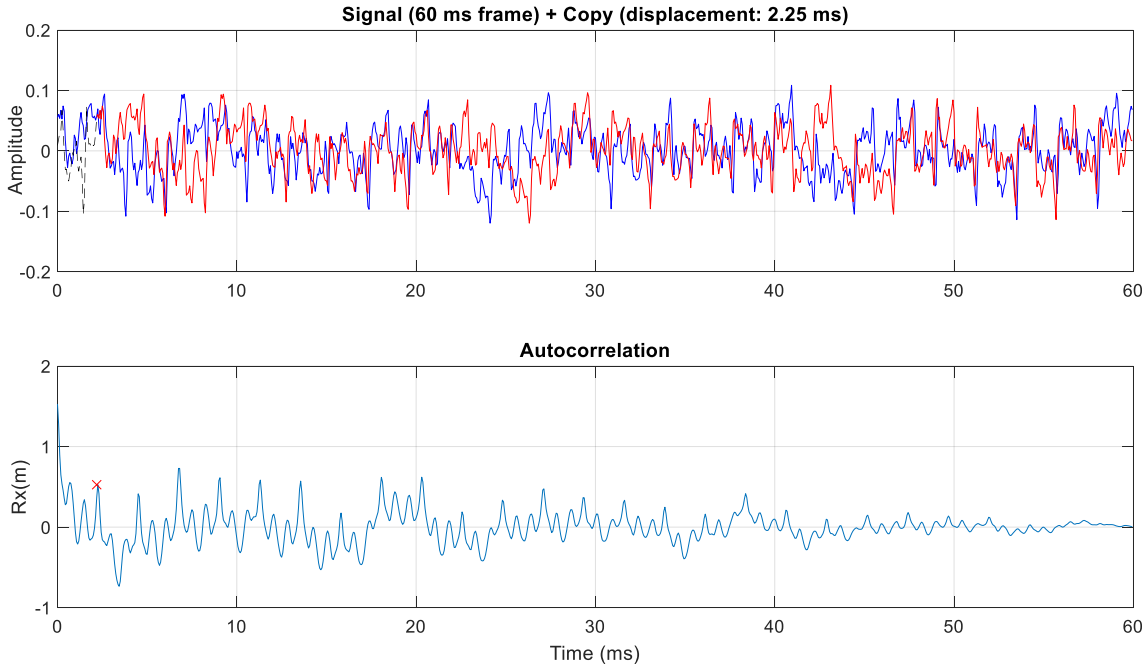

Fig. 18. Ejemplo de autocorrelación en una señal con guitarra eléctrica

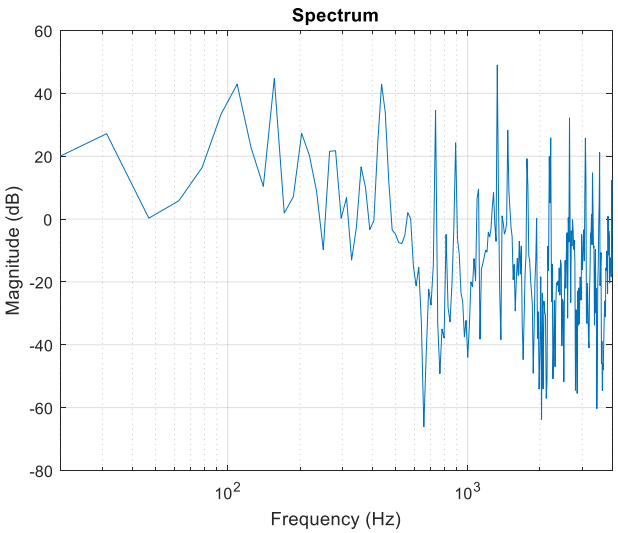

Fig. 19. Espectro de la señal

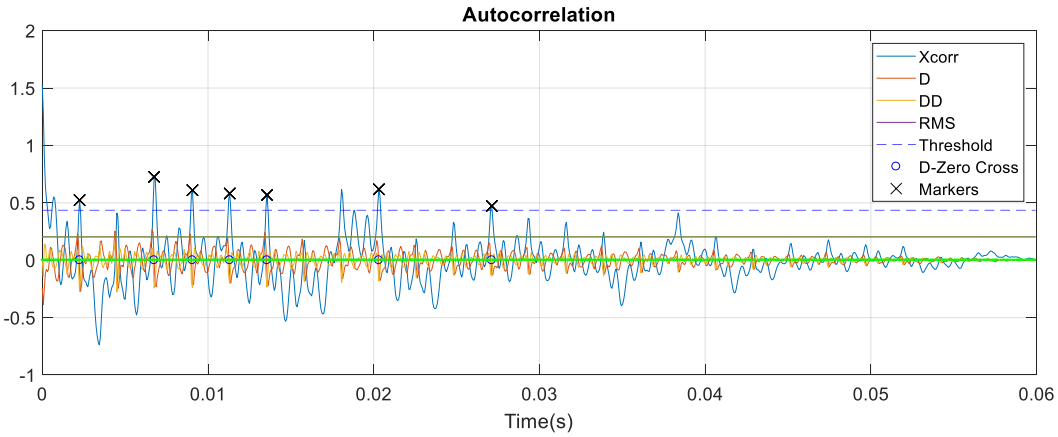

Fig. 20. Ejemplo de detección de *pitch*

En la figura 21 se puede apreciar el pitch detectado a lo largo del tiempo. Como se puede apreciar, hay casos en los que no es posible detectar el *pitch*, estableciendo en dichos casos un *pitch* nulo.

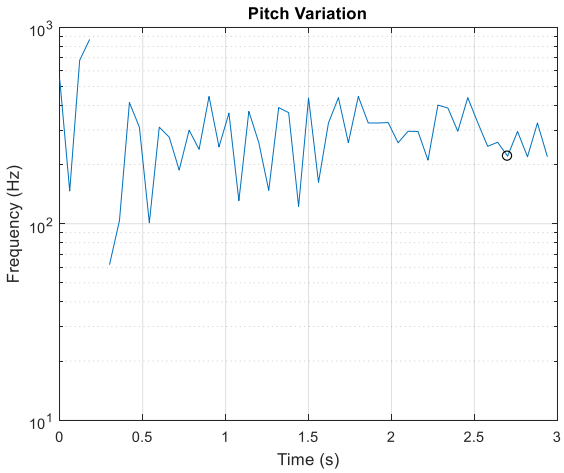

Fig. 21. Variación del *pitch*

IV.2.15. Índice de modulación AM (AM Index)

El índice de modulación se ha venido utilizando en acústica como índice de degradación de la señal acústica a la hora de evaluar la inteligibilidad del habla en un recinto. Para calcular este índice, se obtiene la envolvente de la señal, extrayendo su valor máximo y su valor medio por cada segundo. El cálculo se realiza para cada segundo según la expresión (22), promediando los resultados obtenidos.

$$m(\%) = 100 \cdot \frac{I_{max} - I_0}{I_{max}} \tag{22}$$

En la figura 22 se muestra un ejemplo de índice de modulación obtenido sobre una señal de voz. Como se pude apreciar, la envolvente de la señal de voz contiene cambios abruptos y grandes diferencias entre valores máximos y valores mínimos, por lo que el índice de modulación toma un valor alto. Sin embargo, en la figura 23 se muestra un ejemplo de ruido de lluvia, donde el sonido es más constante, obteniendo así un índice de modulación mucho menor.

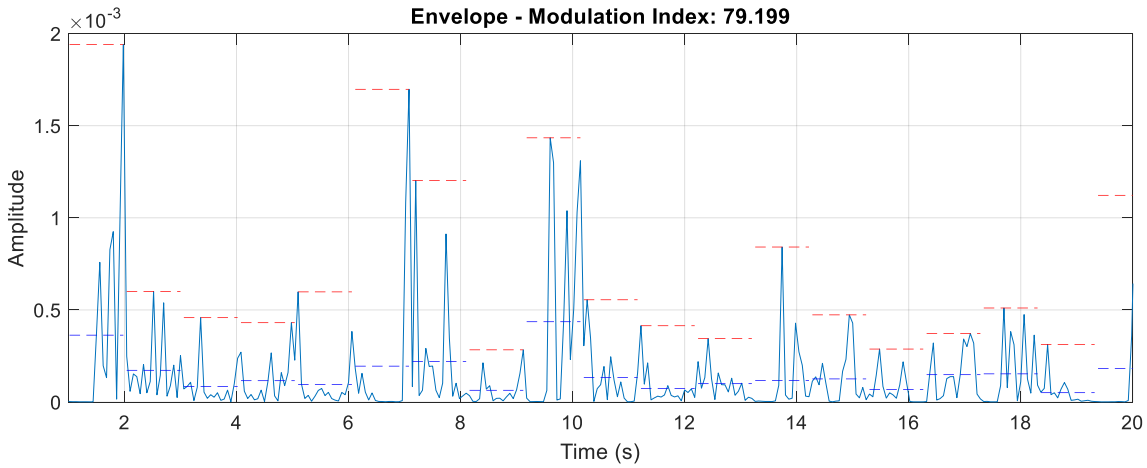

Fig. 22. Ejemplo Índice de Modulación de una señal de voz

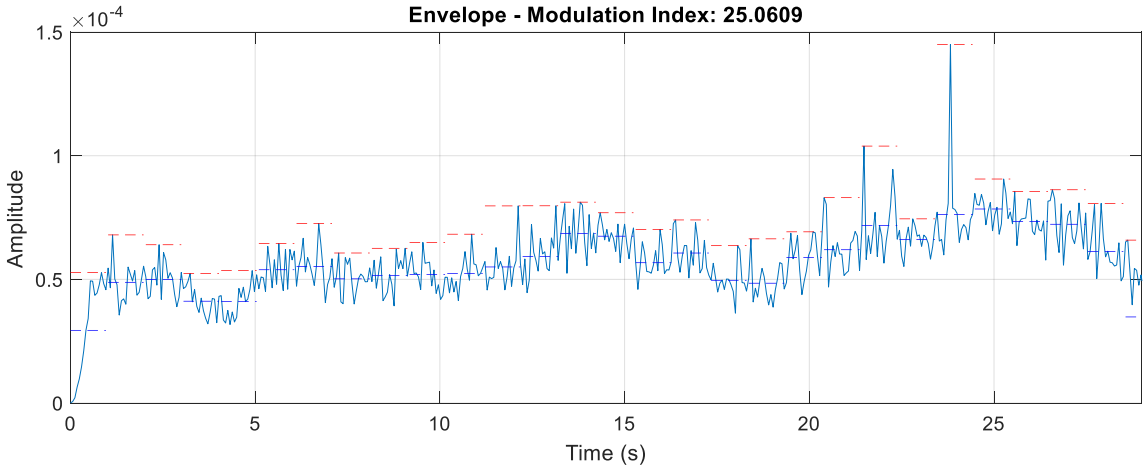

Fig. 23. Ejemplo de Índice de Modulación en una señal de ruido de lluvia

IV.2.16. Tiempo de ataque logarítmico (Log-Attack Time)

El *log-attack* es un tipo de *feature* obtenida en el dominio temporal. Describe el tiempo de ataque de una señal (aplicando después un logaritmo). En los sintetizadores se utiliza el llamado ADSR (*Attack - Decay - Sustain - Release*), mostrado en la figura 24, para describir la envolvente de una nota, definiendo así la evolución temporal de la misma. Sin embargo, esta información no es tan sencilla de extraer ya que, no todas las señales presentan *sustain* o *decay*.

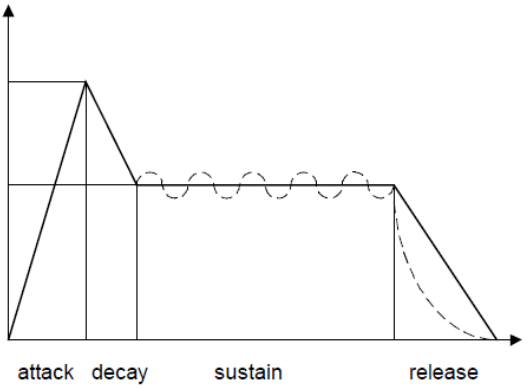

Fig. 24. ADSR (*Attack - Decay - Sustain - Release*)

En este trabajo, únicamente se ha realizado la extracción del ataque, ya que es un descriptor perceptual importante [14]. Para ello, se determinan los puntos de comienzo y fin del incremento de la energía durante el ataque, estableciendo como *threshold* de comienzo el 25 % de su valor RMS (*Root Mean Square)* de la señal, aplicando posteriormente la expresión (23).

$$LogAttack\ Time = \log_{10}(stop_{attack} - start_{attack}) \tag{23}$$

En la figura 25, se muestra un resultado obtenido tras extraer el tiempo de ataque con el algoritmo desarrollado de una señal musical de una flauta. Esta *feature* es más utilizada en el ámbito musical, ya que puede ser importante a la hora de caracterizar instrumentos.

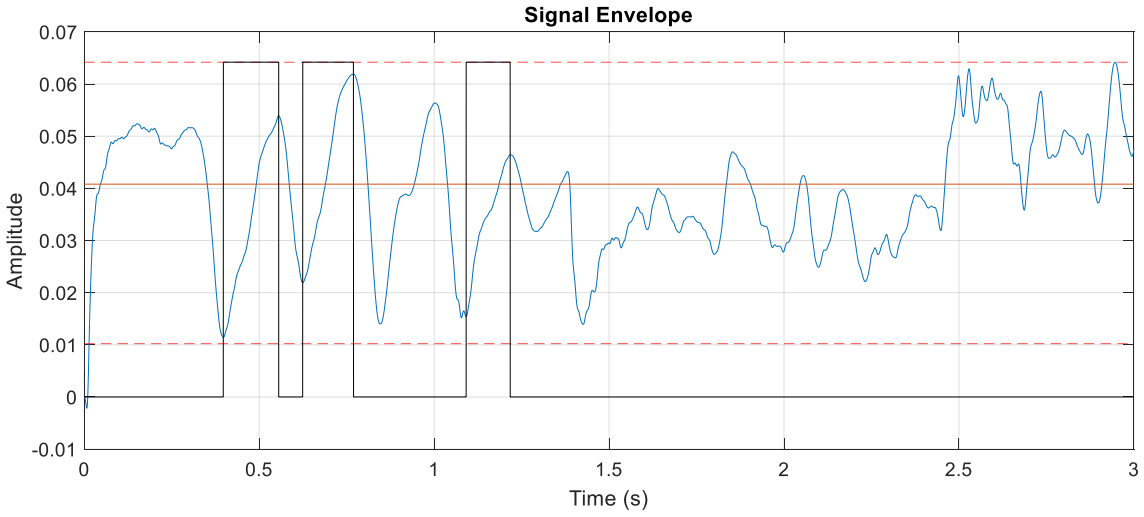

Fig. 25. Detección del tiempo de ataque

IV.3. CLASIFICACIÓN

Una vez extraídas todas las *features*, el *training dataset* será una matriz o una tabla que contenga todas las *features* para cada uno de los ejemplos de entrenamiento, incluyendo la clase a la que pertenece cada uno de los ejemplos (aprendizaje supervisado), siendo posible realizar un entrenamiento del clasificador, utilizando algún algoritmo de clasificación de Aprendizaje Automático. Un ejemplo de formato de *training dataset* podría ser el mostrado en la tabla 1 [4].

Tabla 1. Ejemplo de *Training Dataset*

Audio	***Feature* 1**	***Feature* 2**	**…**	***Feature* n**	**Clase**
1	F1	F2	…	Fn	Voz
2	F1	F2	…	Fn	Voz
…	F1	F2	…	Fn	Voz
1000	F1	F2	…	Fn	Música
…	F1	F2	…	Fn	Música
2000	F1	F2	…	Fn	Ruido
…	F1	F2	…	Fn	Ruido

Es habitual realizar la media y varianza de cada una de las columnas (*features*) y realizar una normalización para obtener una variable aleatoria normal estandarizada. Es decir, $X \sim N(\mu, \sigma^2)$ (variable aleatoria normal) tiene su propia media y varianza y, para normalizar esta variable a una distribución normal estándar de media cero y varianza unidad, es necesario definir la nueva variable aleatoria normal estandarizada $Z \sim N(0,1)$, la cual se muestra en la expresión (24). Esta normalización será aplicada de igual forma a cada nueva instancia que se presente al clasificador, utilizando la media y varianza obtenidas en el *training dataset*.

$$Z = \frac{X - \mu}{\sigma^2} \tag{24}$$

El formato del *test dataset* sería igual que el *dataset* de entrenamiento salvo que no incluiría la clase a la que pertenece cada ejemplo. Este *test dataset* determinará si el clasificador cumple con los requisitos necesarios. En caso de obtener resultados insatisfactorios, es posible realizar una nueva selección de *features* (*feature selection*) y volver a entrenar el clasificador.

Para evaluar los resultados de la clasificación existen diferentes técnicas. La más común de todas ellas es dividir la base de datos para tener un gran porcentaje para entrenamiento (*training dataset*), y el resto para test (***test dataset***).

Otra técnica muy común es la validación cruzada (***cross-validation***). Ésta consiste en dividir la base de datos en K subconjuntos o particiones (*k-folds*), típicamente 5 o 10. Posteriormente se realizan K iteraciones o entrenamientos con su correspondiente evaluación, entrenando cada uno de ellos con K-1 particiones y reservando una para test, de tal forma que, en cada uno de los entrenamientos, se reserve una partición de test distinta. Por último, se realiza una media aritmética de los resultados obtenidos, definiendo así la precisión del clasificador [4]. Esta técnica, muy utilizada en sistemas de inteligencia artificial, es muy útil para tener una idea de cómo se comportaría el sistema utilizando toda la base de datos para entrenar el clasificador, ya que independiza el resultado de las particiones utilizadas en el entrenamiento. Sin embargo, el coste computacional es mayor que reservando un *test dataset* concreto.

Una tercera técnica es la denominada *leave-one-out*, la cual sigue la misma filosofía que la validación cruzada salvo que, en este caso, cada partición de test es un único ejemplo de la base de datos, por lo que conlleva un gran coste computacional, aunque buena precisión.

Los resultados de la clasificación vienen generalmente dados en forma de **matriz de confusión**. Consiste en una matriz cuyas filas y columnas se corresponden con las clases, concretamente las filas serían las clases reales, y las columnas las clases predichas. De esta manera, el resultado óptimo sería que cada ejemplo de test sea predicho como la clase que realmente es, lo cual situaría al ejemplo en la diagonal de la matriz (misma fila, misma columna). Cuando este ejemplo se predice de forma errónea ocurren dos tipos de error: falso rechazo, para la clase real, ya que uno de sus ejemplos ha sido rechazado; y, por otro lado, la falsa alarma, ya que ese ejemplo será predicho como otra clase incorrecta, por lo tanto, esa clase tendrá un error de falsa alarma, aceptando un ejemplo que no le corresponde. En la figura 26 se muestra un ejemplo de matriz de confusión para una clasificación de la base de datos UrbanSound8K.

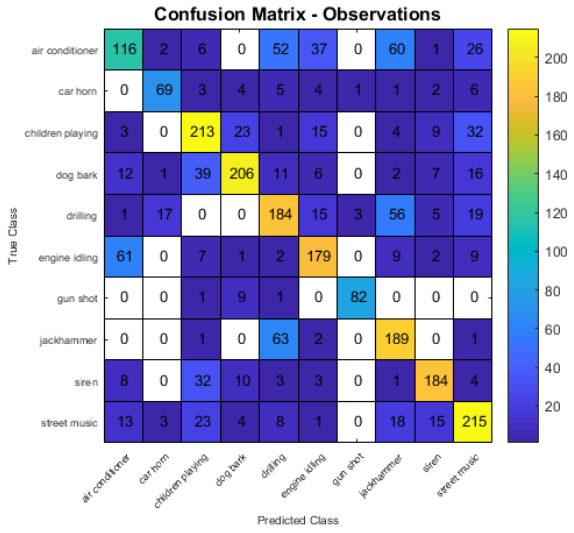

Fig. 26. Matriz de confusión. Observaciones

La matriz de confusión puede venir dada en forma de observaciones (número de ejemplos – ver figura 26) o en forma de porcentajes, ya sean porcentajes por filas (aciertos y falsos rechazos – ver figura 27.a), o por columnas (predicciones acertadas y falsas alarmas – ver figura 27.b).

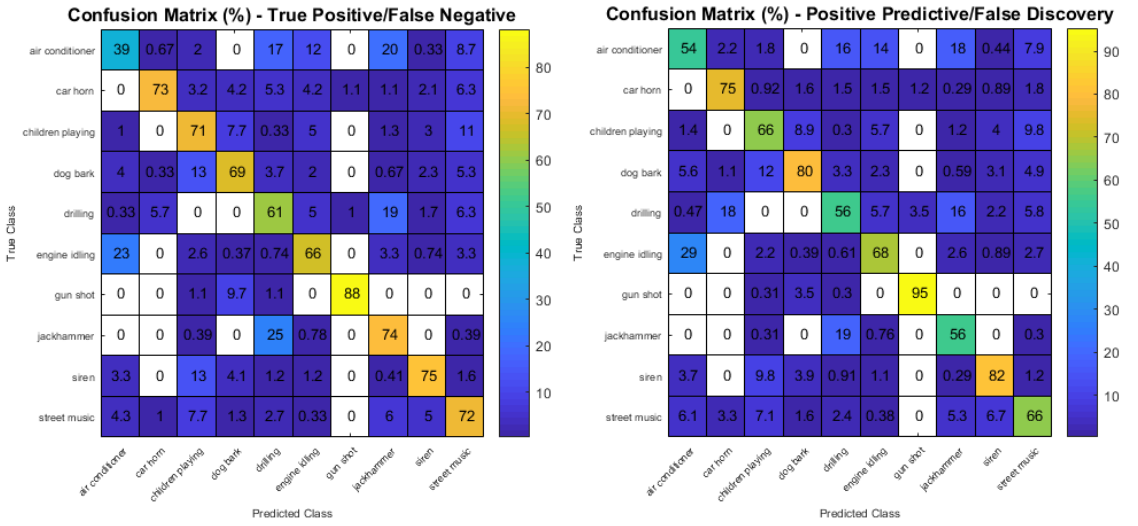

Fig. 27. a) Porcentaje de aciertos y falsos rechazos b) Porcentaje de predicciones acertadas y falsas alarmas

La forma más común es la de porcentaje de aciertos y falsos rechazos, ya que aporta una idea de la precisión del clasificador para cada clase, y de la precisión global, media de las precisiones de cada clase. Para tener la evaluación más completa, también es necesario obtener los porcentajes de error, los cuales se muestran en la figura 28. El falso rechazo (*false rejection*) se obtiene como la suma de porcentajes fuera de la diagonal para cada fila, es decir, de todos los audios introducidos de una clase, cuántos han sido rechazados; mientras que la falsa alarma se obtiene como la suma de porcentajes fuera de la diagonal para cada columna, es decir, de todas las predicciones de una misma clase, cuántas son erróneas.

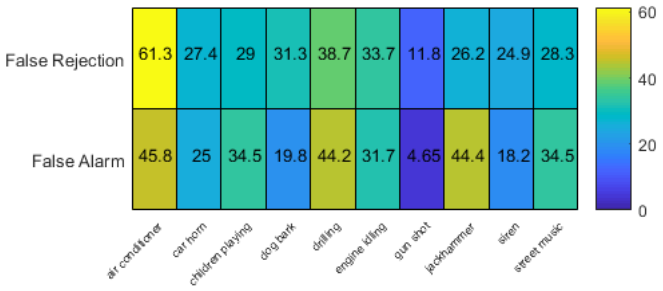

Fig. 28. Falsa alarma y falso rechazo

Existe una gran variedad de algoritmos de clasificación, como por ejemplo los basados en lógica y árboles de decisión (*Decision Trees*), aprendizaje estadístico (*Bayesian Networks*), criterios de vecindad o distancia (*k-Nearest Neighbors*), redes neuronales (*Neural Networks*), *clustering* (*k-means*), máquinas de vectores de soporte (*Support Vector Machines*), etc. Sin embargo, en este trabajo únicamente se utilizarán los algoritmos: *Random Forest*, SVM y k-NN.

IV.3.1. Random Forest

Random Forest es un algoritmo de ensamblado basado en árboles de decisión. El concepto de ensamblado se produce al utilizar, en este caso, más de un árbol de decisión para una misma clasificación. Sin embargo, para entender este algoritmo es necesario analizar previamente un árbol de decisión [22].

El árbol de decisión es un tipo de algoritmo muy utilizado en Inteligencia Artificial. Su estructura está basada en nodos (*nodes*), ramas (*branches*) y hojas (*leaves*). En la figura 29 se muestra la estructura del algoritmo. El primer nodo es denominado nodo raíz (*root node*), correspondiéndose cada nodo a una única *feature*. Cada rama representa un rango de valores de dicha *feature* para dividir (*split*) los datos, indicando el camino entre dos nodos. Los nodos finales se denominan hojas, que son nodos a partir de los cuales no se continúa subdividiendo los datos, obteniendo así un histograma de los datos existentes en cada hoja, el cual será utilizado como resultado para una predicción futura, siendo la clase ganadora aquella con mayor presencia en la hoja.

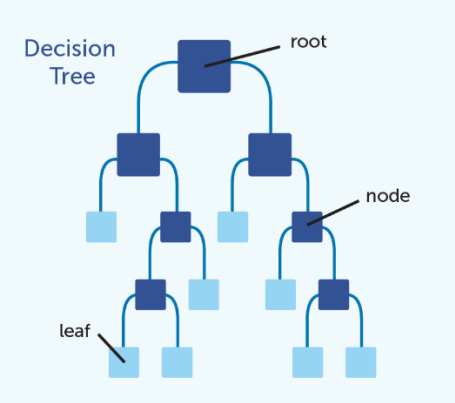

Fig. 29. Árbol de Decisión

En la figura 30 se muestran los procesos de entrenamiento y test del clasificador. En la primera fase, se determinan los umbrales de cada nodo y se obtienen los histogramas de probabilidad de cada hoja. En la fase de test, se realiza el camino raíz-hoja obteniendo así el resultado de la clasificación.

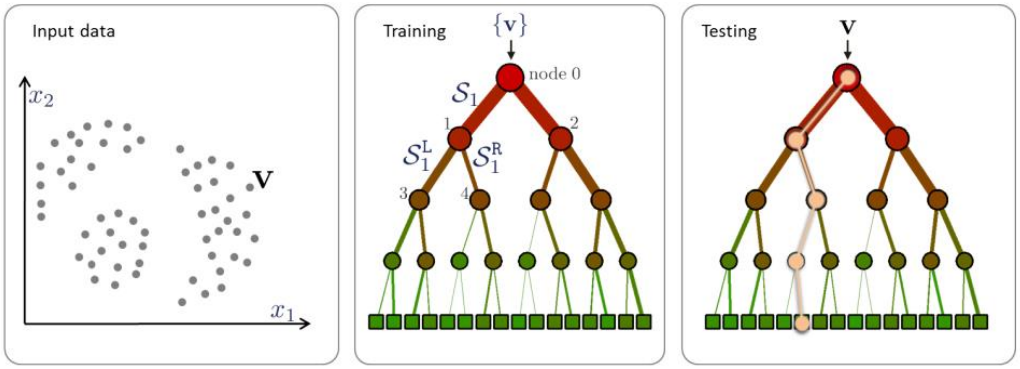

Fig. 30. Ejemplo de entrenamiento y test con árboles de decisión [7]

En la **fase de entrenamiento**, tal y como se muestra en la figura 31, se tiene el *training dataset* (ver figura 31.a) a la entrada del nodo raíz, teniendo todo el conjunto de datos de *N* dimensiones (*features*) con los que será entrenado el árbol (ver figura 31.b), donde cada punto en el espacio de características es un vector de valores para cada *feature*: $\boldsymbol{v_n} = (x_1, x_2, \ldots, x_n)$. Puesto que cada nodo se corresponde con una *feature*, y cada *feature* con una dimensión de las *N* iniciales, el proceso consiste en ir subdividiendo los datos en cada dimensión, de tal manera que cada nodo presente un histograma de clases de los datos que han tomado dicho camino.

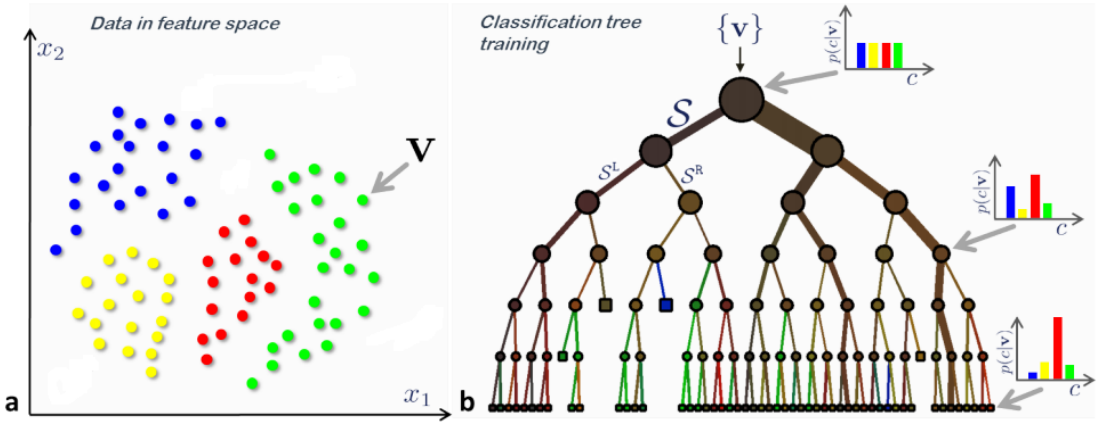

Fig. 31. a) *Training Dataset* b) Fase de entrenamiento del árbol de decisión [7]

En primer lugar, se divide la información en el nodo raíz, separando los datos en función de un umbral específico que mejor separe la información de la *feature* asociada a dicho nodo (existen distintos criterios para determinar este umbral: *Gini index, variance reduction, information gain*… [23]). Una vez dividida la información, se separa en distintos nodos en función del umbral. En cada uno de estos nodos se produce otra subdivisión de los datos de la misma forma que en el nodo anterior. Finalmente, aquellos nodos que no pueden subdividir más la información quedan como hojas de una única clase, y, aquellos nodos que superen la profundidad del árbol o *tree depth* (parámetro configurable, representa el máximo número de subdivisiones), quedarán como hojas representando un histograma con el número de datos que contienen de cada clase.

En la **fase de test**, mostrada en la figura 32, el clasificador utiliza todos los umbrales determinados en la fase de entrenamiento, de tal manera que el vector de test en cuestión seguirá un camino desde el nodo raíz hasta una hoja en función de los valores de las *features*. La hoja resultante representará un histograma de probabilidad para cada una de las clases, es decir, la probabilidad que tiene el ejemplo o vector $\boldsymbol{v}$ de pertenecer a cada una de las clases, tal y como muestra la expresión 25 [7], siendo la clase ganadora aquella con mayor probabilidad (es decir, mayor número de ejemplos de una clase durante la fase de entrenamiento en dicha hoja).

$$p_t(c|\boldsymbol{v}) \tag{25}$$

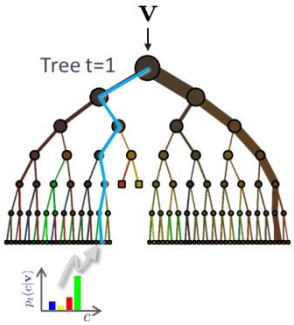

Fig. 32. Fase de test de un árbol de decisión [7]

El algoritmo *Random Forest* es un método de ensamblado (*ensemble method*) basado en árboles de decisión, es decir, en lugar de entrenar un único árbol de decisión, se entrenan un gran número de estos, de tal manera que entre todos ellos se obtenga una solución más precisa. A cada uno de estos árboles de decisión dentro del algoritmo de ensamblado se le denomina *weak learner*, y cada uno de ellos es entrenado con un subconjunto de *features* de las *N* posibles. La selección de este subconjunto de *features* entre las *N* que describen cada ejemplo se realiza de forma aleatoria, entrenando cada árbol con el mismo número de *features*, pero siendo estas diferentes. A este proceso se le denomina *Bagging* o *Bootstrap Aggregation*.

El proceso de entrenamiento se realiza para *T* árboles (parámetro configurable), y, de la misma forma, el test se realiza sobre *T* árboles, aportando cada uno de ellos un resultado distinto, tal y como se muestra en la figura 33 [7].

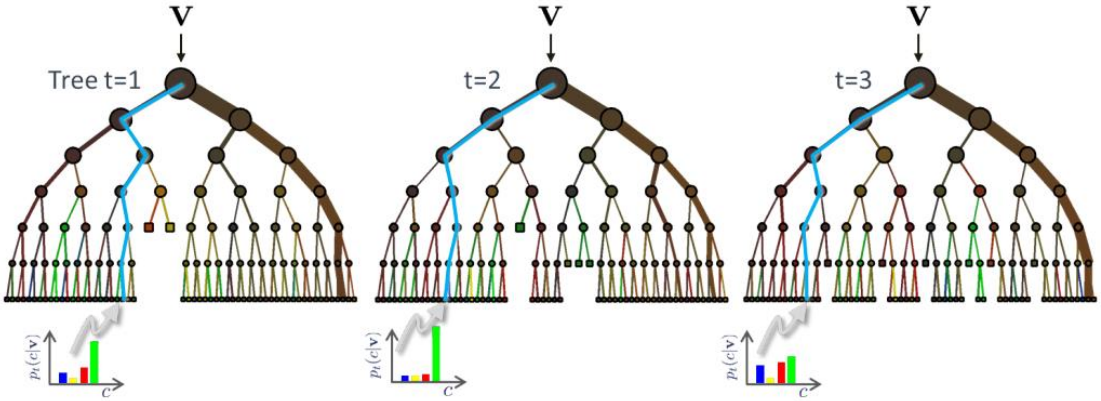

Fig. 33. *Random Forest*: fase de test [7]

El resultado de la clasificación vendrá dado por el resultado conjunto de todos los árboles, el cual puede realizarse mediante un promediado, o bien mediante multiplicación de los resultados, según se muestra en la expresión 26 [7].

$$\begin{cases} p(c|\boldsymbol{v}) = \dfrac{1}{T}\displaystyle\sum_{t=1}^{T} p_t(c|\boldsymbol{v}) \\ p(c|\boldsymbol{v}) = \dfrac{1}{T}\displaystyle\prod_{t=1}^{T} p_t(c|\boldsymbol{v}) \end{cases} \tag{26}$$

Este proceso de ensamblado tiene la ventaja de seleccionar los resultados con mayor confianza, proporcionando así robustez frente a árboles "ruidosos" [7]. La tasa de error disminuye de forma exponencial a medida que aumenta el número de árboles, hasta obtener un valor constante aproximadamente, tal y como se muestra en el ejemplo de la figura 34.

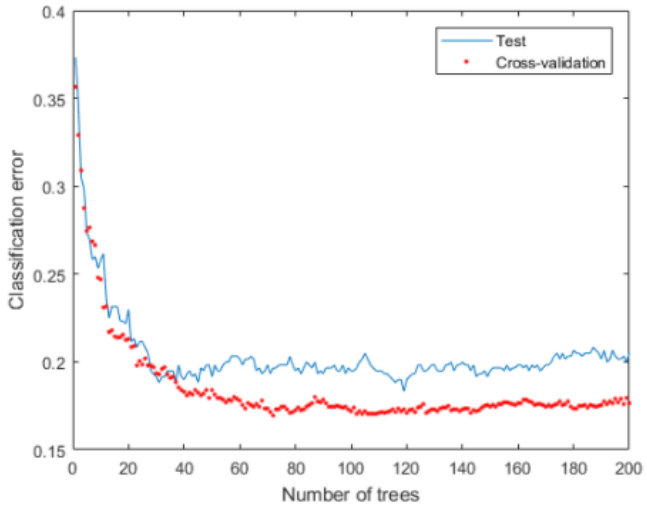

Fig. 34. El error de la clasificación disminuye con el número de árboles

Sin embargo, no ocurre lo mismo con la profundidad de árbol, ya que, a medida que la información se subdivide, el modelo se ajusta cada vez más a los datos, lo que se denomina *overfitting*, produciendo que el clasificador no sea capaz de **generalizar** una nueva entrada. Si en la etapa de entrenamiento se subdivide la información hasta que todas las hojas sean puras, es decir, que únicamente contengan una clase, el clasificador estará sobreentrenado. Para solucionar esto, se realiza una poda o *pruning* de las hojas del árbol, limitando la profundidad del árbol, obteniendo así mayor grado de generalización.

Una particularidad interesante de este algoritmo es que permite clasificar un ejemplo para cada uno de los árboles por separado, es decir, en paralelo o de forma distribuida, uniendo posteriormente los resultados, lo cual puede resultar muy útil para aplicaciones de tiempo real.

IV.3.2. k-Nearest Neighbors

El algoritmo k-NN es un tipo de algoritmo estadístico basado en los ejemplos de entrenamiento (*instance-based learning*) [4]. Al contrario que otros algoritmos, éste requiere menos tiempo de cómputo durante la fase de entrenamiento, pero requiere más tiempo en el proceso de clasificar una muestra. Está basado en el principio de cercanía de los ejemplos con características similares, siendo el algoritmo más básico el vecino más cercano (*nearest neighbor*). En este caso, el algoritmo se basa en los *k* vecinos más cercanos para predecir la clase de la muestra en cuestión.

El algoritmo consiste en determinar un espacio de características *n*-dimensional (*n features*) en el cual se encuentran todas las muestras de entrenamiento, todas ellas etiquetadas con su clase correspondiente. Durante el proceso de clasificación de una nueva muestra, ésta se proyecta en dicho espacio *n*-dimensional y se obtiene la distancia relativa entre la muestra $\boldsymbol{y}= (y_1, \ldots, y_n)$ y cada uno de los ejemplos o muestras de entrenamiento: $\boldsymbol{x}= (x_1, \ldots, x_n)$. Esta distancia se puede medir con diferentes métricas, siendo la más habitual la distancia euclídea, la cual se muestra en la expresión (27). Una vez se identifican los *k* ejemplos con menor distancia relativa a la nueva muestra, se selecciona la clase con mayor probabilidad, es decir, aquella con mayor número de ejemplos entre los *k* vecinos [4].

$$Euclidean: \qquad D(x,y) = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2} \tag{27}$$

Sin embargo, existen otros métodos para medir dicha distancia que tratan de minimizar la distancia entre muestras pertenecientes a la misma clase, y maximizar aquellas con distinta clase. Algunas de estas métricas se muestran en las expresiones mostradas en (28) [4]. En la distancia de Minkowsky, *r* representa el orden del método (distancia euclídea generalizada).

$$\begin{cases} Minkowsky: & D(x,y) = \left(\sum_{i=1}^{n} |x_i - y_i|^r\right)^{1/r} \\ Manhattan: & D(x,y) = \sum_{i=1}^{n} |x_i - y_i| \\ Chebychev: & D(x,y) = \max_i |x_i - y_i| \\ Camberra: & D(x,y) = \sum_{i=1}^{n} \frac{|x_i - y_i|}{|x_i + y_i|} \end{cases} \tag{28}$$

El valor de *k* influye en gran medida en el resultado de la clasificación. Sin embargo, no existe un método que permita elegir un *k* adecuado, salvo la realización de pruebas con distintos valores de *k*.

IV.3.3. Support Vector Machine

El algoritmo SVM es un poco más complejo que los anteriores, por lo que únicamente se hará una breve introducción al mismo. Este algoritmo trata de dividir las regiones definidas por las clases en el espacio de características utilizando para ello un hiperplano (*hyperplane*) n-dimensional [4]. El caso más sencillo para interpretar este algoritmo sería un espacio de características 2D, con únicamente dos clases, donde el hiperplano equivaldría a una línea, tal y como se muestra en la figura 35.

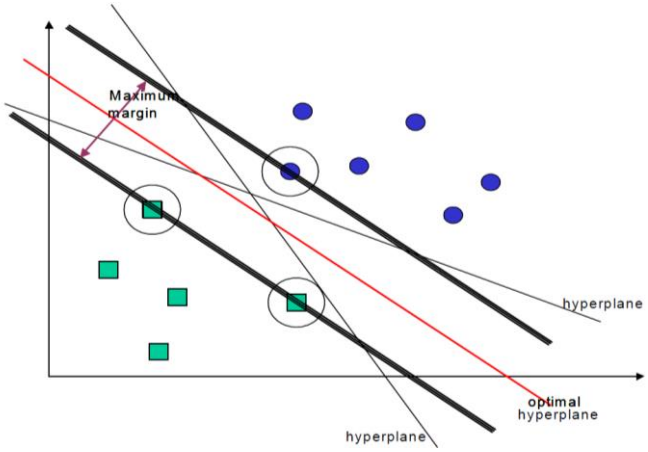

Fig. 35. SVM: hiperplano óptimo (máximo margen) [4]

Tal y como se puede apreciar en la figura 35, el algoritmo trata de buscar el mejor hiperplano posible, que es aquel con mayor margen entre muestras de distintas clases, suponiendo que las clases sean linealmente separables. Maximizando dicho margen, se reduce el error de generalización [4], siendo así una división justa para ambas clases. Aquellos puntos (ejemplos) que definan el margen son denominados *support vector points*, obteniendo la solución como la combinación lineal de todos ellos.

El *training dataset* se compone de ejemplos que contienen dos variables ($\boldsymbol{x_i}$, y_i) donde el primer término $\boldsymbol{x_i}$ es la entrada (*input*) y el segundo término y_i la salida (*output*). Puesto que se trata de una clasificación binaria, la salida y_i únicamente puede tomar dos valores, +1 para patrones positivos (una clase) y -1 para patrones negativos (otra clase) [24]. Por lo tanto, el hiperplano óptimo debe ser una función que permita decidir entre ± 1 para un nuevo vector de entrada $\boldsymbol{x_i}$. Si los datos de entrenamiento son linealmente separables, entonces debe existir ($\mathbf{w}$,b) tal que:

$$\begin{cases} \boldsymbol{w^T x_i} + b \geq 1 & para\ todo\ x_i \in clase\ positiva \\ \boldsymbol{w^T x_i} + b \leq -1 & para\ todo\ x_i \in clase\ negativa \end{cases}$$

Donde $\boldsymbol{w}$ es un vector de pesos, $\boldsymbol{x}$ la variable de entrada y b una constante, obteniendo la regla de decisión según la expresión (29).

$$f(\boldsymbol{x}) = sgn(\boldsymbol{w} \cdot \boldsymbol{x} + b) \tag{29}$$

Para resolver el problema de optimización, se introducen multiplicadores de Lagrange ($\alpha_i \geq 0$) y una función de Lagrange [24], la cual se muestra en la expresión (30), donde N es el número total de ejemplos del *training dataset*.

$$L(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{N} \alpha_i\big((\boldsymbol{x_i} \cdot \boldsymbol{w}) + b\big) - 1 \qquad \boldsymbol{\alpha} = (\alpha_i, \dots, \alpha_N) \tag{30}$$

Aplicando el teorema de Kuhn Tucker [24], el cual determina que las derivadas parciales de L con respecto a $\boldsymbol{w}$ y b deben ser igual a 0, se obtienen las condiciones mostradas en la expresión (31).

$$\begin{cases} \sum_{i=1}^{N} \alpha_i y_i = 0 \\ \boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i x_i \end{cases} \tag{31}$$

Este teorema implica que α_i debe satisfacer las condiciones de Karush-Kuhn Tucker y por tanto aquellas muestras de entrenamiento que no caigan en el margen óptimo tendrán un α_i nulo, siendo el resto vectores de soporte. Finalmente, la regla de decisión podría ser según la expresión (32) [24], siendo $\boldsymbol{x_i}$ cada uno de los vectores de entrenamiento y $\boldsymbol{x_j}$ una nueva muestra a clasificar, donde $\alpha_i \in SV$. En un caso real, en el que los datos no sean linealmente separables, estas expresiones se componen de más variables, las cuales no se explicarán en este trabajo. Además, pueden utilizarse polinomios de distinto orden (por ejemplo, cúbico).

$$f(\boldsymbol{x}) = sgn\left(\sum_{i=i}^{N} \alpha_i y_i (\boldsymbol{x}_i \cdot \boldsymbol{x}_j) + b\right) \tag{32}$$

El algoritmo SVM es un algoritmo binario, discriminando únicamente entre dos clases. Sin embargo, es posible utilizar distintos métodos cuando se trata de una clasificación **multi-clase**. Uno de estos métodos es el "uno contra el resto" (*one-versus-rest method*) [24], el cual consiste en entrenar un clasificador por cada clase (*K* clases), enfrentando dicha clase contra todas las demás, agrupadas como una sola clase, obteniendo así *K* clasificadores binarios. Por lo tanto, el resultado de una clasificación vendrá dado por la clase ganadora. Otro de los métodos más utilizados es el denominado "uno contra uno" (*one-versus-one method*) [24], el cual sigue la estrategia del más votado. En este método, por cada clase, se entrena un clasificador contra cada una de las demás clases. Por lo tanto, el número de clasificadores a entrenar viene dado según la expresión (33). El resultado de una clasificación vendrá dado por un vector de puntuaciones $\boldsymbol{w} = (w_1, \dots, w_i, \dots, w_K)$, el cual se obtiene como el número de veces que cada una de las clases ha sido ganadora en los enfrentamientos, siendo la clase ganadora aquella con mayor puntuación.

$$Número\ de\ clasificadores = \frac{K(K-1)}{2} \tag{33}$$

V. DESARROLLO E IMPLEMENTACIÓN

En este trabajo se han realizado dos programas orientados a la clasificación de señales de audio. En primer lugar, se ha diseñado una herramienta que permita entrenar clasificadores y analizar los resultados, utilizando distintos tipos de *features* y distintos algoritmos de clasificación. Posteriormente, con los clasificadores entrenados, se ha diseñado un sistema que analiza y clasifica el audio en tiempo real, para señales recogidas por micrófono o desde un archivo.

V.1. HERRAMIENTA DE CLASIFICACIÓN

En la figura 36 se muestra la interfaz gráfica del programa desarrollado en *Matlab* 2017a, que permite llevar a cabo todo el proceso de clasificación mostrado en la figura 2. Este programa se compone de distintos módulos, entre los cuales se encuentran:

- El módulo general, que permite seleccionar las clases a clasificar, así como incorporar nuevas bases de datos para crear nuevas clases. También es el módulo de configuración.
- El módulo de selección de *features*, el cual permite seleccionar las *features* con las que se desea realizar un clasificador. También dispone de un módulo que permite realizar un *scattering* entre dos *features*, visualizando el mismo en la ventana de representación.
- El módulo de clasificación, en el cual se debe indicar qué clasificador utilizar, así como los parámetros correspondientes a cada uno de los algoritmos. En este módulo se puede seleccionar el modo de representación de la matriz de confusión, así como realizar una prueba de test de validación cruzada. También permite guardar el clasificador una vez entrenado correctamente.

Tras la ejecución, se representa la matriz de confusión, con sus respectivos errores de falso rechazo y falsa alarma, así como un histograma de probabilidad de una de las muestras de test.

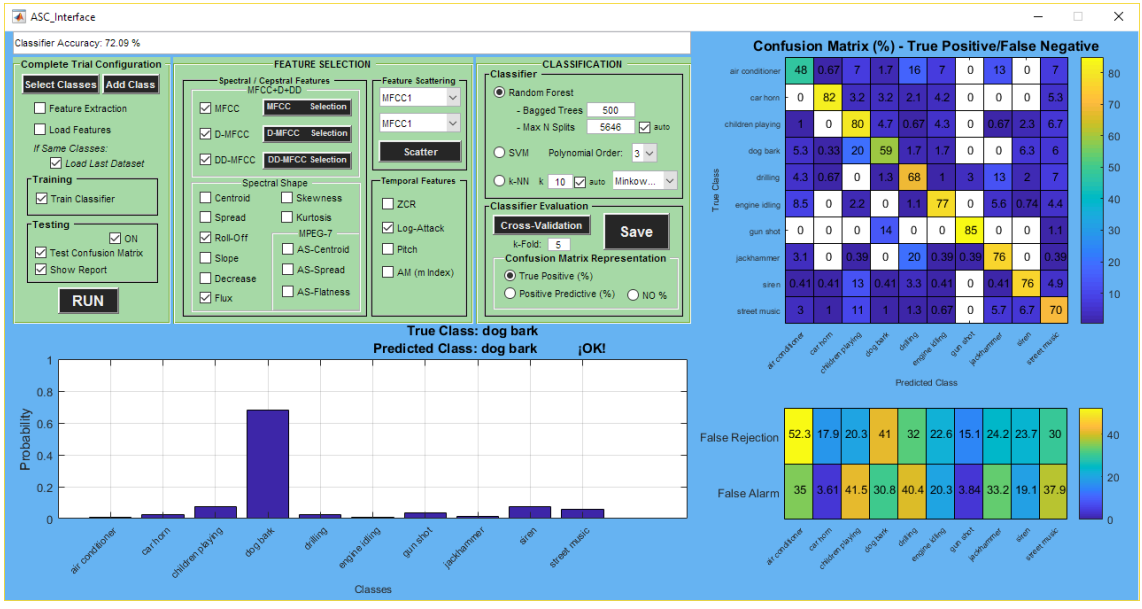

Fig. 36. Interfaz de la herramienta de clasificación

V.1.1. Módulo de Selección de Features (Feature Selection)

En la figura 37 se muestra en módulo de selección de *features*. Como se puede apreciar este se divide en subsecciones. Por un lado, se encuentran las *features* cepstrales y espectrales, que comprenden los MFCC y la *features* espectrales que definen la forma del espectro, y, por otro lado, las *features* temporales. También se tiene una sección que permite realizar un *scattering* de las dos *features* seleccionadas, el cual se puede observar en la figura 37.b.

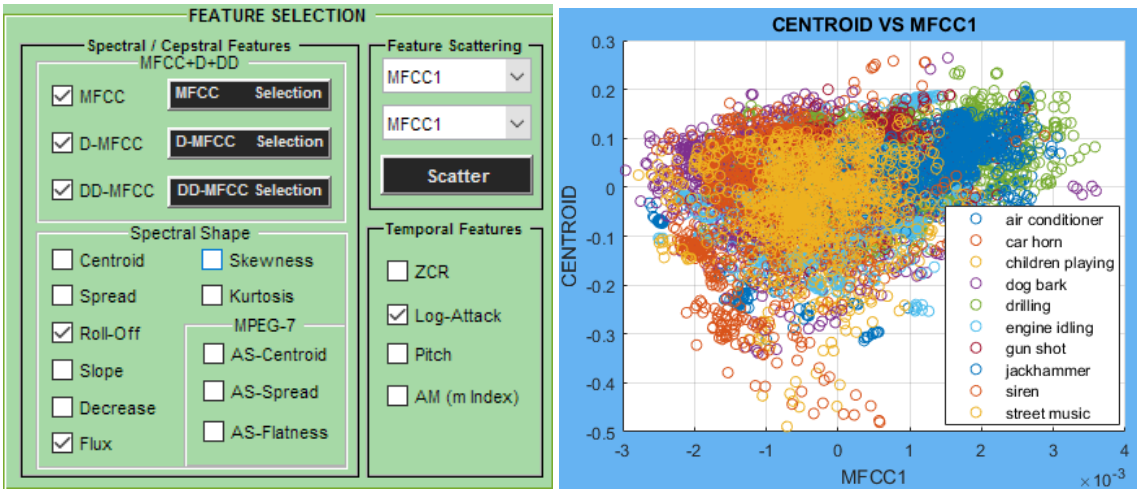

Fig. 37. a) Módulo *Feature Selection*. b) *Scattering*

Puesto que los MFCC son *features* en forma de vector, es posible seleccionar los MFCC que se desean utilizar, al igual que sus derivadas. Para ello, se tienen las pestañas "*MFCC Selection*", "*D-MFCC Selection*" y "*DD-MFCC Selection*", las cuales muestran una ventana independiente donde poder seleccionar dichas *features*. Esta ventana se muestra en la figura 38.

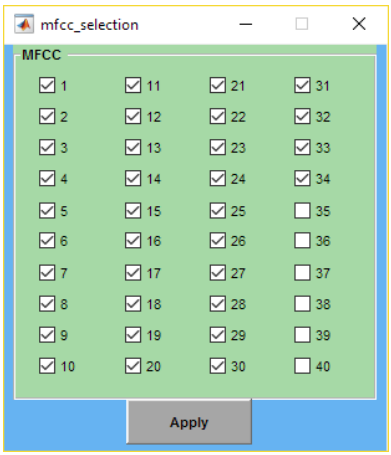

Fig. 38. Selección de MFCC

V.1.2. Módulo de Clasificación

En cuanto al módulo de clasificación, el cual se muestra en la figura 39, se puede observar que permite utilizar tres algoritmos: *Random Forest*, SVM y k-NN. Para el algoritmo *Random Forest* es posible determinar el número de árboles, así como la profundidad de árbol (*tree depth*). En el caso de SVM, es posible cambiar el orden polinomial de la función núcleo (*kernel*) del algoritmo. Finalmente, para el algoritmo k-NN, se puede determinar el tipo de distancia relativa que se desea utilizar (Euclídea, Minkowsky…).

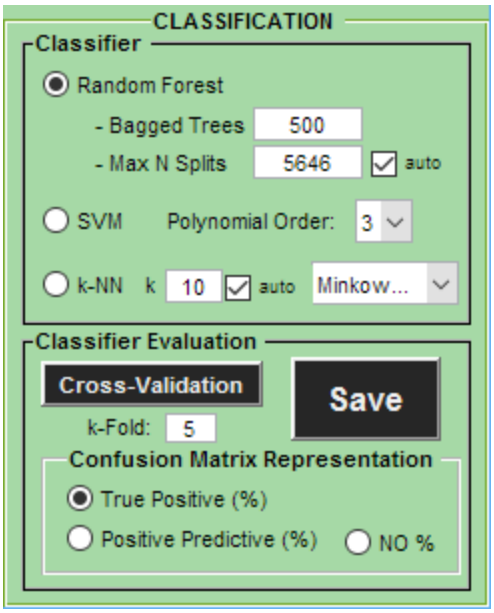

Fig. 39. Módulo de Clasificación

En la sección de evaluación, este módulo también permite cambiar el modo de visualización de la matriz de confusión, ya sea en modo observaciones (sin porcentajes), en modo porcentajes de aciertos y falsos rechazos, o en modo porcentajes de predicciones acertadas y falsas alarmas.

También permite realizar una evaluación mediante *cross-validation* pudiendo elegir el número de particiones. Cabe destacar que para esta prueba se utilizan todos los datos de cada clase, tanto de entrenamiento como de test, para que la propia evaluación realice las particiones necesarias. Si se utiliza una base de datos en la que distintos audios provengan de una misma grabación, es preferible no utilizar dicha evaluación, ya que los resultados serán más favorables. Además, se debe tener en cuenta que los datos de test nunca pueden ser datos de entrenamiento.

Tras haber realizado el entrenamiento y la evaluación de un clasificador, en este módulo también se permite guardar el clasificador. Es posible guardar el clasificador directamente o volver a entrenar el mismo incluyendo, finalmente, los datos de test como parte del entrenamiento, lo que en teoría aumentaría la precisión del clasificador.

V.1.3. Módulo general: configuración

El módulo general, el cual se muestra en la figura 40, incluye dos ventanas auxiliares con distinta funcionalidad: *Select Classes* y *Add Class*. También contiene la configuración correspondiente a la ejecución.

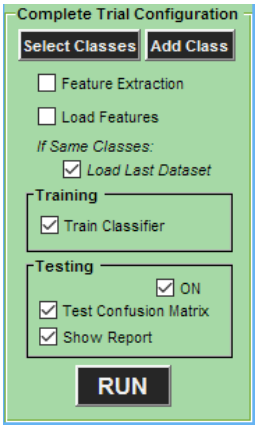

Fig. 40. Módulo general

La ventana *Select Classes* permite seleccionar las clases que se desean clasificar, la cual se muestra en la figura 40. Esta ventana también permite realizar una mezcla de clases, es decir, unir un conjunto de clases para formar una única clase. En la figura 41 se puede observar que existen un conjunto de clases añadidas, entre las cuales se encuentran un conjunto de instrumentos, y, a su vez, existe una clase denominada "*instruments*" que es la unión de todos los instrumentos independientes. En cuanto a las clases predefinidas (*UrbanSound8K*), es posible determinar el porcentaje de base de datos que se desea destinar a entrenamiento y test.

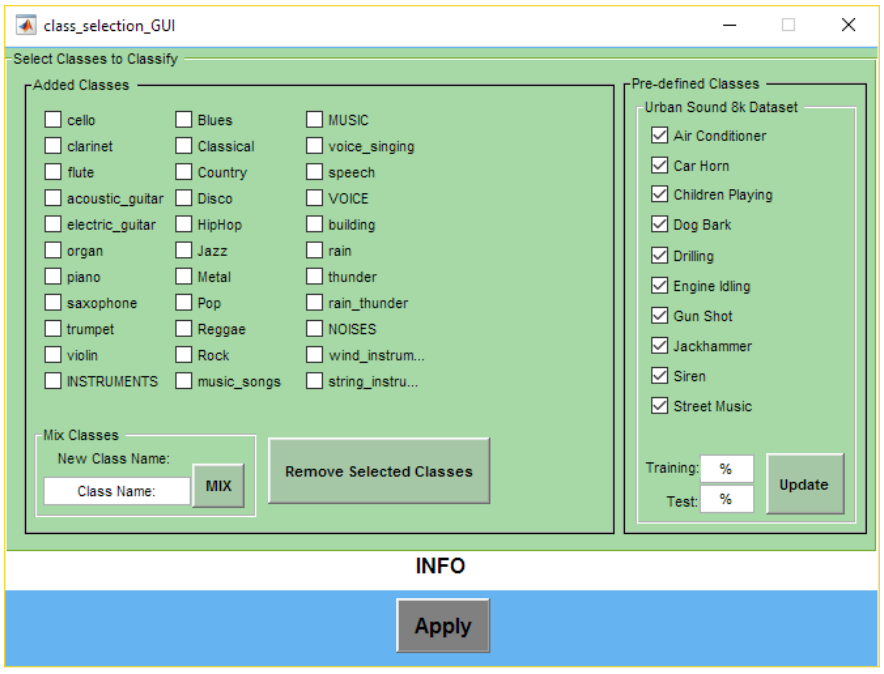

Fig. 41. Ventana de selección de clases

En cuanto a la ventana *Add Class*, mostrada en la figura 42, ésta permite incorporar nuevas clases al sistema partiendo de una base de datos. Su interfaz es sencilla, simplemente se necesita el directorio donde se ubican todos los audios (puede contener subdirectorios), el porcentaje de entrenamiento y test que se desea utilizar, y el nombre de la nueva clase. Una vez se tienen los datos, se procede a crear las **listas** de entrenamiento y test de dicha clase, para lo cual se debe pulsar "*create benchmarks*". Si no ha habido ningún problema durante el proceso, se mostrará un mensaje en el panel de información indicando que las listas se han creado correctamente. Estas listas son archivos de texto que pueden ser manipulados manualmente para incluir o excluir determinados ejemplos si fuera necesario.

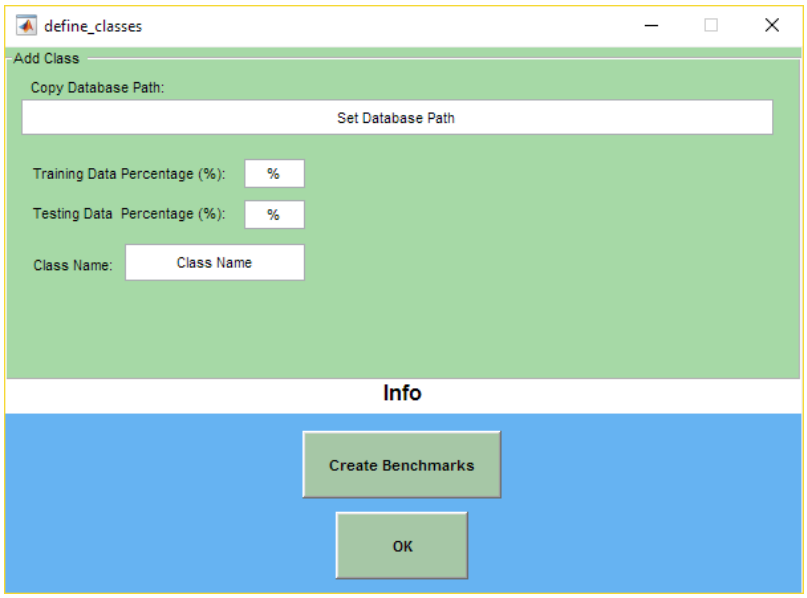

Fig. 42. Ventana para añadir clases

Por último, antes de realizar la ejecución del programa, se debe seleccionar una de las tres casillas correspondientes a la extracción de *features*:

- *Feature Extraction*: debe ser seleccionada cuando no se haya realizado ninguna extracción previa sobre las clases a utilizar.
- *Load Features*: debe ser seleccionada cuando se desee realizar una clasificación con clases cuyas *features* hayan sido previamente extraídas. En caso de que algunas clases ya hayan pasado por el proceso de extracción y otras no, aquellas cuyas *features* no hayan sido guardadas y, por tanto, no sea posible su carga, pasarán por la extracción de *features*.
- *Load Last dataset*: esta última opción se debe utilizar cuando se vaya a realizar una ejecución con las mismas clases que en la ejecución anterior. Esto permite cambiar la selección de *features* para un nuevo entrenamiento, ahorrando el tiempo de carga.

También se debe indicar si se desea entrenar el clasificador, si se desea evaluar el mismo, y si se desea generar un informe automático.

V.2. EJECUCIÓN Y FUNCIONAMIENTO

Una vez definida la interfaz gráfica con todas sus opciones, se procede a explicar el funcionamiento del software. En primer lugar, se define la estructura del programa, la cual está basada en un *front-end* y un *back-end*.

- El *front-end* se encarga de procesar cada uno de los audios para realizar la extracción de *features*, de tal forma que el *back-end* reciba el *training dataset* o el *test dataset*.
- El *back-end* comprende la parte de Aprendizaje Automático, tanto en entrenamiento como en evaluación, utilizando para ello los datos recibidos desde el *front-end*.

Cada una de las clases a clasificar, se corresponde con dos listas o *benchmarks*, una de entrenamiento y otra de test, las cuales contienen los directorios donde se encuentran cada uno de los audios. Estas listas son creadas cada vez que se incorpora una nueva clase al sistema, perteneciendo ambas a la misma base de datos, pero cada una de ellas con un porcentaje de la misma.

V.2.1. Front-end: Feature Extraction

En la figura 43 se muestra el diagrama de flujo del procesado llevado a cabo en el *front-end* tanto en la fase de entrenamiento como en la fase de test.

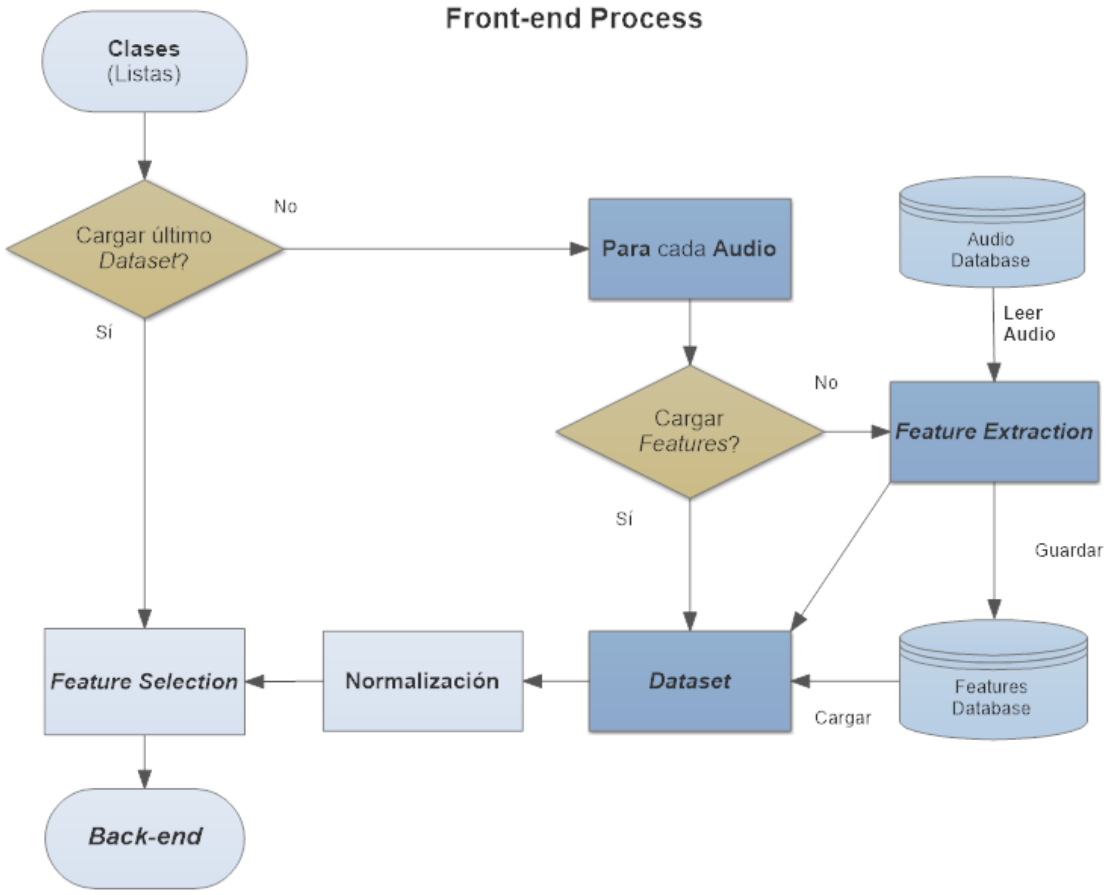

Fig. 43. Diagrama de Flujo del *front-end*

En primer lugar, se reciben las listas de aquellas clases que se desea clasificar. Si anteriormente se ha realizado una ejecución con las mismas clases, pero con distintas *features*, es posible utilizar el mismo *dataset*, ya que éste contiene todas las *features* posibles para posteriormente poder realizar la selección. De esta manera es posible evitar el tiempo de cómputo de extracción de *features*.

En caso de tratarse de clases distintas a las clases utilizadas en la ejecución anterior, es necesario cargar dichas clases. Si la base de datos de las clases a utilizar ha pasado previamente por el proceso de extracción, las *features* de cada audio estarán almacenadas en la base de datos de *features*. Esta base de datos tiene la misma estructura que la base de datos de audio, de tal manera que cada audio tiene un archivo con extensión ".wav" en la base de datos de audio, y su correspondiente archivo con mismo nombre y extensión ".mat" en la base de datos de *features*, conteniendo todas las *features* implementadas en el programa. Si el sistema detecta que no es posible cargar un archivo de *features* de un audio porque no se ha realizado la extracción previamente, entonces se realiza el proceso de extracción para dicho audio. En la fase de entrenamiento, una vez recopiladas todas las *features* de todos los audios de cada clase, se conforma un *training dataset*, del cual se extrae la media y varianza de cada *feature* (ver tabla 2) y se normaliza conforme a una distribución gaussiana; y, en la fase de evaluación, el *test dataset* será normalizado con la media y varianza obtenidas del *training dataset* correspondiente.

Tabla 2. *Dataset* + Media y Varianza

Audio	***Feature* 1**	***Feature* 2**	**…**	***Feature* n**	**Clase**
1	f_1	f_2	…	f_n	1
2	f_1	f_2	…	f_n	1
…	f_1	f_2	…	f_n	1
1000	f_1	f_2	…	f_n	2
…	f_1	f_2	…	f_n	2
2000	f_1	f_2	…	f_n	3
…	f_1	f_2	…	f_n	3
Media	μ_1	μ_2		μ_n	
Varianza	σ^2_1	σ^2_2		σ^2_n	

En el caso de incorporar nuevas clases al sistema, estas necesitan pasar por el proceso de extracción de características. Por lo tanto, para cada audio de cada clase, se realiza el proceso de extracción, guardando el resultado en la base de datos de *features* con el mismo nombre del audio original, pero con distinta extensión. Puesto que de cada audio se extrae un vector de *features*, éste se une a la tabla o *dataset* que contiene todos los vectores de *features* de todos los audios, incluyendo junto al vector la clase a la que pertenece.

El proceso de ***Feature Extraction*** es realizado según el diagrama de flujo de la figura 44. Como se puede apreciar, el audio de entrada es diezmado a una frecuencia de muestreo de 16000 Hz (si es necesario). Esto es debido a que no todos los audios de las bases de datos tienen la misma frecuencia de muestreo. Por ello, se ha elegido como frecuencia de muestreo la mínima posible para poder utilizar el máximo número de audios de las bases de datos. Después, cada audio recibe un procesado común para eliminar la componente continua de la señal. A continuación, se presentan algunas de las características básicas de los procesos de extracción, tanto en el dominio temporal como en los dominios espectral y cepstral:

- Frecuencia de muestreo: ***fs = 16000 Hz***
- Features Espectrales / Cepstrales:
  - *Framing*: 60 ms (960 muestras)
  - Solapamiento: 50 %
  - Ventana: Hamming
  - NFFT: 1024 puntos
  - Cepstral (MFCC):
    - Filtros Mel: M=35
    - Coeficiente DCT: C=40
- Features Temporales:
  - *Framing*: 25 ms (400 muestras)
  - Solapamiento: 0 %
  - Ventana: Rectangular
  - Envolvente: Exponencial

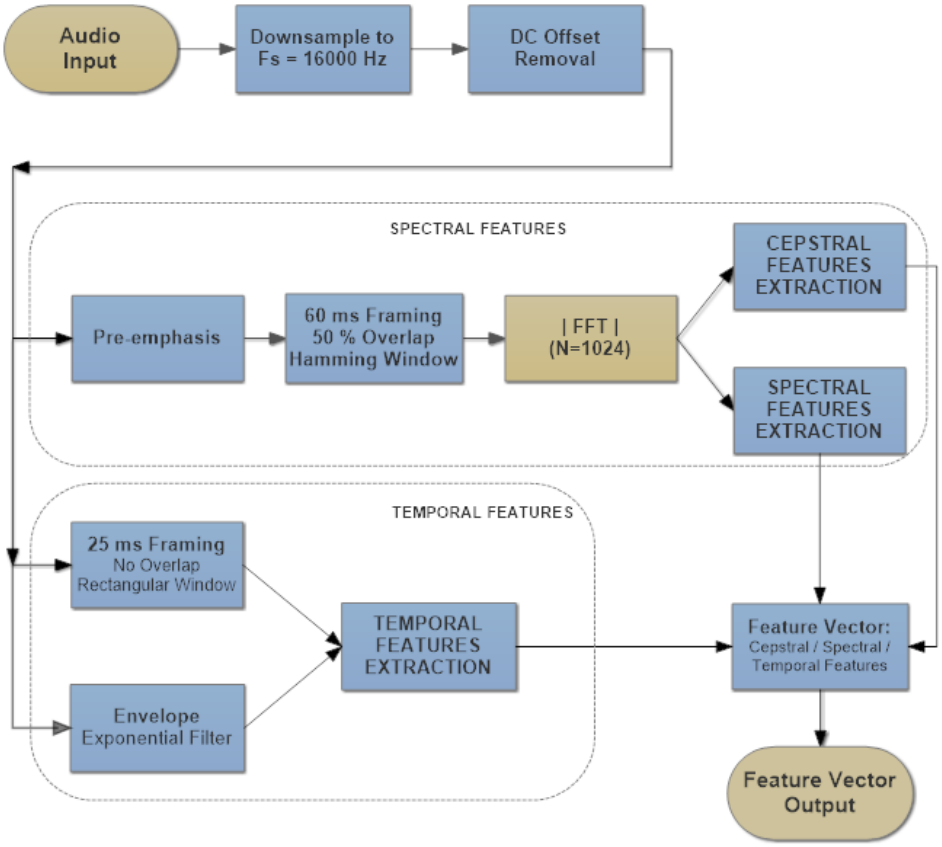

Fig. 44. Diagrama de Flujo de *Feature Extraction*

En el procesado de features espectrales y cepstrales, se realiza un preénfasis de la señal seguido de un enventanado Hamming por bloques de 60 milisegundos (960 muestras) con 50 % de solapamiento entre sí (480 muestras de solapamiento). Posteriormente se obtiene la DFT de cada uno de los bloques mediante la Transformada Rápida de Fourier (FFT), utilizando 1024 puntos. Posteriormente se obtiene el módulo de la primera mitad del espectro (512 puntos), utilizando los espectros obtenidos de cada uno de los bloques para realizar el procesado de cada una de las *features* implementadas, algunas de ellas utilizando el espectro en potencia y otras en amplitud.

En el procesado de las features temporales, se realiza un enventanado (rectangular) de 25 milisegundos sin solapamiento, el cual es utilizado para la extracción de *features* como el *pitch* y el ZCR. También se obtiene la envolvente de la señal utilizando para ello un filtrado exponencial, la cual se utiliza en *features* como el *log-attack*. El índice de modulación utiliza otra envolvente obtenida a partir de la energía de cada *frame*. El filtrado exponencial se realiza muestra a muestra según la expresión (34), donde *x(n)* es la señal, *e(n)* la envolvente y α el factor de olvido, de tal manera que, si α es cercana a la unidad, la envolvente obtendrá las variaciones rápidas de la señal, siendo muy próxima a ésta, y, por el contrario, si α es cercana a la cero, la envolvente variará más lentamente, dando mayor peso a las muestras pasadas. Este método de obtención de la envolvente es rápido y útil en aplicaciones de tiempo real.

$$e(n) = \alpha \cdot x(n) + (1 - \alpha) \cdot e(n-1) \qquad 0 \leq \alpha \leq 1 \qquad (34)$$

El proceso de extracción de cada *feature* se describe en el apartado IV.2. Para más detalle sobre la implementación de cada una de las funciones de extracción, consular el Anexo II.

Una vez extraídas las *features*, se realiza un promediado de cada una de ellas para obtener el valor medio de cada *feature* en cada audio, formando el vector de *features* final que describe a cada audio, conteniendo las *features* ceptrales, espectrales y temporales. Finalmente, todos los vectores de todos los audios se unen en una misma tabla o matriz de forma organizada para poder formar el *dataset*.

Por último, se realiza el proceso de normalización correspondiente a dicho *dataset*. El *dataset* normalizado es guardado para una posible ejecución posterior, previamente a la selección de features. Finalmente, el proceso de selección de features es implementado como un vector lógico que indica con un valor 1 aquellas features seleccionadas y un valor 0 aquellas descartadas. Con dicho vector se seleccionan las columnas a utilizar del *dataset*, correspondientes a cada una de las features.

V.2.2. Back-end

El *back-end* tiene dos fases fundamentales, el entrenamiento y la evaluación. Ambas fases se pueden observar en la figura 45, donde se muestra el proceso general de la herramienta y el funcionamiento del *back-end*.

En la fase de entrenamiento, se entrena el clasificador con el *training dataset* obtenido en el *front-end*, el cual, como se ha visto previamente, puede ser cargado o bien creado cargando *features* o extrayendo las mismas. En esta fase se debe escoger el algoritmo a utilizar, pudiendo escoger entre los algoritmos: *Random Forest*, SVM y k-NN. El clasificador entrenado será guardado junto con el vector lógico (unos y ceros) de selección de *features* y la normalización empleada en el *training dataset*.

En la fase de evaluación, se realizará la normalización y selección de features para el *test dataset* de la misma forma que para el training *dataset*, y se entregará al clasificador para obtener los resultados de la evaluación. Es importante destacar que el test *dataset* entregado al clasificador (no el *test dataset* guardado) contiene una columna menos que el *training dataset*, ya que no se incluye la clase a la que pertenece cada observación, la cual es extraída antes de introducir los datos al clasificador.

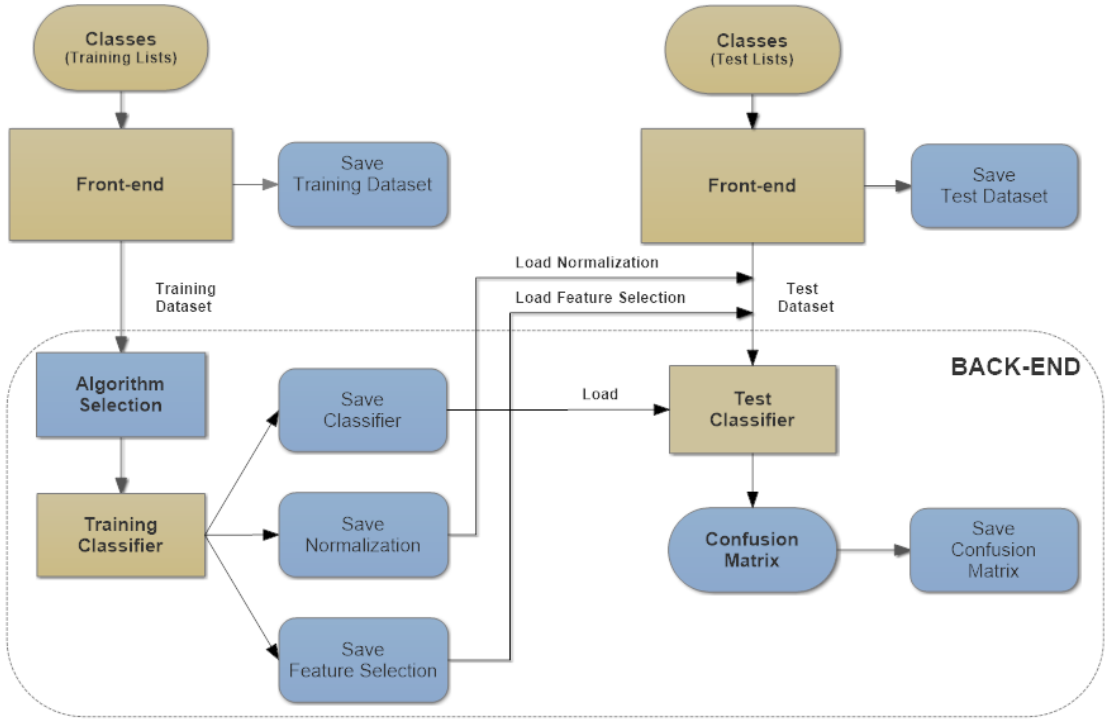

Fig. 45. Diagrama de bloques: *Back-end*

El *test dataset* guardado también contiene la clase a la que pertenece cada observación debido a que cuando se realiza la evaluación *cross-validation*, se unen los *datasets* de entrenamiento y test para formar un único *dataset* que el sistema divide en particiones para realizar dicha evaluación.

Para realizar cada uno de los **algoritmos de clasificación**, se ha utilizado la *toolbox* de estadística y *Machine Learning* de *Matlab* 2017a. Se han utilizado las funciones *fitcensemble* para el algoritmo *Random Forest*, *fitcecoc* para SVM y *fitcknn* para k-NN.

Para realizar el ensamblado *Random Forest* con *fitcensemble*, primero se realiza un árbol plantilla utilizando la función *templateTree*, donde es posible indicar la profundidad de árbol (*tree depth*). Después, se incluye dicha plantilla como parámetro de la función de ensamblado, indicando también el número de ciclos, es decir, el número de árboles, y el método de ensamblado, que en el caso de *Random Forest* es "*bag*" (*Bagging – Bootstrap Aggregation*).

En cuanto al algoritmo SVM, puesto que se trata de un algoritmo binario, es necesario indicar qué tipo de método debe utilizar para una clasificación multi-clase. Para ello, primero se crea una plantilla SVM con la función *templateSVM*, donde es posible indicar el orden del polinomio de la función *kernel* (núcleo). Después, se incluye dicha plantilla como parámetro en la función *fitcecoc*, en la cual se indica el método multi-clase en el campo "*coding*", utilizando en este caso el método *one-versus-one*, el cual realiza *K(K-1)/2* clasificadores para *K* clases.

Por último, para el algoritmo k-NN, se utiliza la función *fitcknn*, en la cual se puede indicar el tipo de distancia relativa (Euclídea, Minkowski, …), el exponente en caso de que tenga (por ejemplo, Minkowski) y el número de vecinos *k* que determinará el resultado de una clasificación futura.

Para familiarizarse con el uso de estas funciones, es muy útil la aplicación de *Matlab Classification Learner App*, en la cual se puede realizar cualquier tipo de clasificación. La interfaz de esta aplicación se muestra en la figura 46. Para realizar una clasificación con esta aplicación tan solo se debe importar un *dataset* que contenga tanto las *features* como sus etiquetas (*labels*) de cada observación. Una vez realizada cualquier tipo de clasificación, el programa genera el código necesario para la misma utilizando las funciones de la *toolbox* de *Machine Learning*, por lo que puede ser muy didáctico a la hora de utilizar dichas funciones.

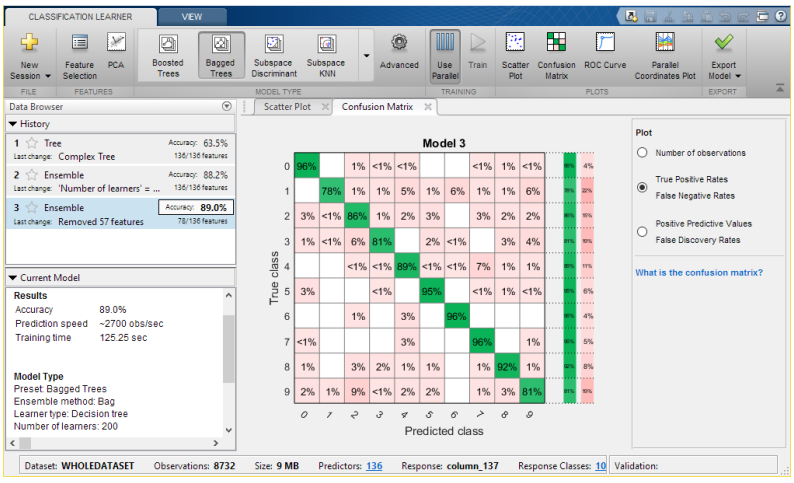

Fig. 46. *Classification Learner App*

V.3. RESULTADOS

Uno de los problemas destacados en los estudios realizados por los autores de *UrbanSound* es que los sonidos de ruido urbano habitualmente contienen ruido de fondo, lo que dificulta la clasificación. Cabe decir que se utilizaron distintos algoritmos de clasificación sin optimizar los parámetros de los mismos (parámetros por defecto) para evaluar la utilidad de la base de datos, introduciendo un vector de *features* de 225 dimensiones, entre las cuales se encuentran los 25 primeros MFCC.

Como se puede apreciar en la figura 47, se obtiene una precisión aproximada de 70% para algoritmos como SVM y *Random Forest* (ver figura 47.a). En un experimento realizado con sujetos, utilizando secuencias de 4 segundos de duración, se identificaron los sonidos con un 82% de precisión [10] como media de las 10 clases. También, señalan un decrecimiento en la precisión de la clasificación cuando se tratan muestras de audio de duración menor a 4 segundos aproximadamente [10], tal y como se puede apreciar en la figura 47.b, lo que resulta importante a la hora de realizar una clasificación en tiempo real.

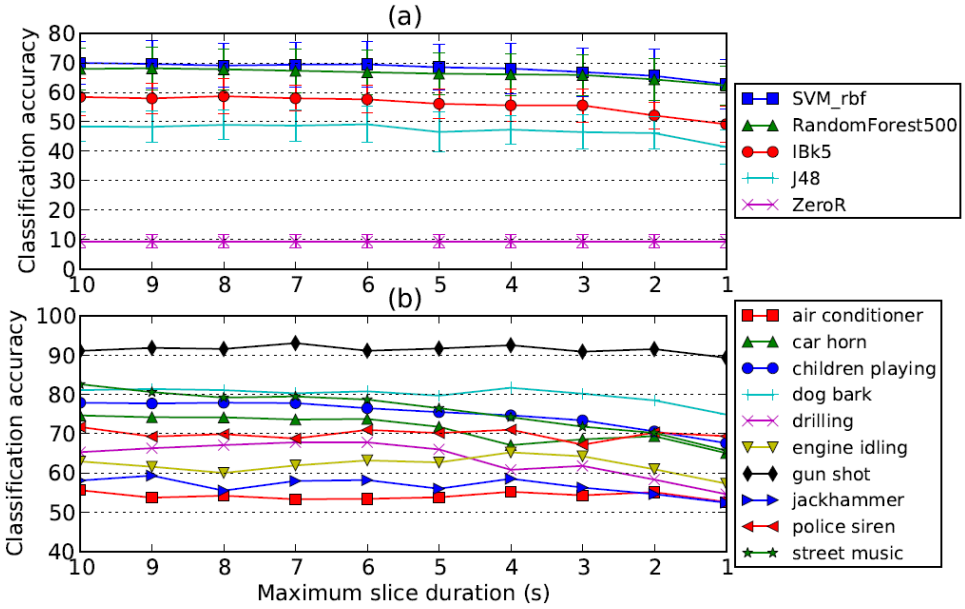

Fig. 47. Experimentos *UrbanSound* [10]

Puesto que cada clasificación depende de las clases utilizadas, en las siguientes secciones se muestran los resultados obtenidos para cada uno de los clasificadores entrenados. En particular, para la base de datos *UrbanSound*, se realiza una comparativa entre los distintos algoritmos de clasificación, así como las distintas *features* seleccionadas a la hora de clasificar.

V.3.1. Clasificador UrbanSound8K

La base de datos *UrbandSound8K* contiene sonidos extraídos de una misma grabación de audio. Sin embargo, la base de datos está dividida en 10 particiones, haciendo que los audios de cada partición sean únicos. Esto hace que a la hora de definir el *training dataset* y el *test dataset*, cada uno debe estar compuesto por particiones distintas, en este caso se han utilizado 7 particiones para entrenamiento y 3 para evaluación. Una validación cruzada del dataset conjunto (*training* + test) resultaría en una evaluación optimista, ya que la validación cruzada automática de *Matlab* divide las particiones de forma aleatoria, por lo que este tipo de evaluación para esta base de datos únicamente debería realizarse de forma manual, entrenando 10 clasificadores con 9 particiones cada uno para entrenamiento y 1 para test, sin repetir las particiones de test en cada uno de los clasificadores.

En la tabla 3 se muestran los resultados obtenidos utilizando cada uno de los algoritmos. En todos los casos se ha utilizado el mismo conjunto de *features*, concretamente los 25 primeros MFCC más el resto de *features* implementadas. Para determinar el valor de precisión de la clasificación con cada algoritmo se han realizado 5 evaluaciones con cada uno de ellos y se ha realizado una media.

Tabla 3. Resultados entre distintos algoritmos de clasificación

Algoritmo / Evaluación	***Random Forest (500 árboles)***	***SVM (Orden 3)***	***k-NN***
1	69,23 %	66,23 %	55,85 % (*k=10*)
2	69,36 %	66,19 %	55,20 % *(k=20)*
3	68,09 %	66,07 %	54,85 % *(k=30)*
4	69,09 %	66,35 %	53,80 % *(k=40)*
5	68,43 %	66,30 %	57,71 % (*k=50*)
Media	**68,84 %**	**66,23 %**	**55,48 %**

Como se puede observar el clasificador con mejores resultados es *Random Forest*. Cabe decir que la precisión mencionada se obtiene con la media de precisiones de cada clase, sin tener en cuenta la falsa alarma, por lo que se podría realizar una comparativa más exhaustiva. Esto implica que una clase pueda tener más porcentaje de acierto utilizando un algoritmo que tiene una precisión global menor. Sin embargo, la precisión global permite tener una idea general del comportamiento del clasificador.

En las figuras 48, 49 y 50, se muestran los informes automáticos de los mejores resultados obtenidos de cada algoritmo, *Random Forest*, SVM, y k-NN, respectivamente, los cuales han sido generados por la herramienta de clasificación desarrollada.

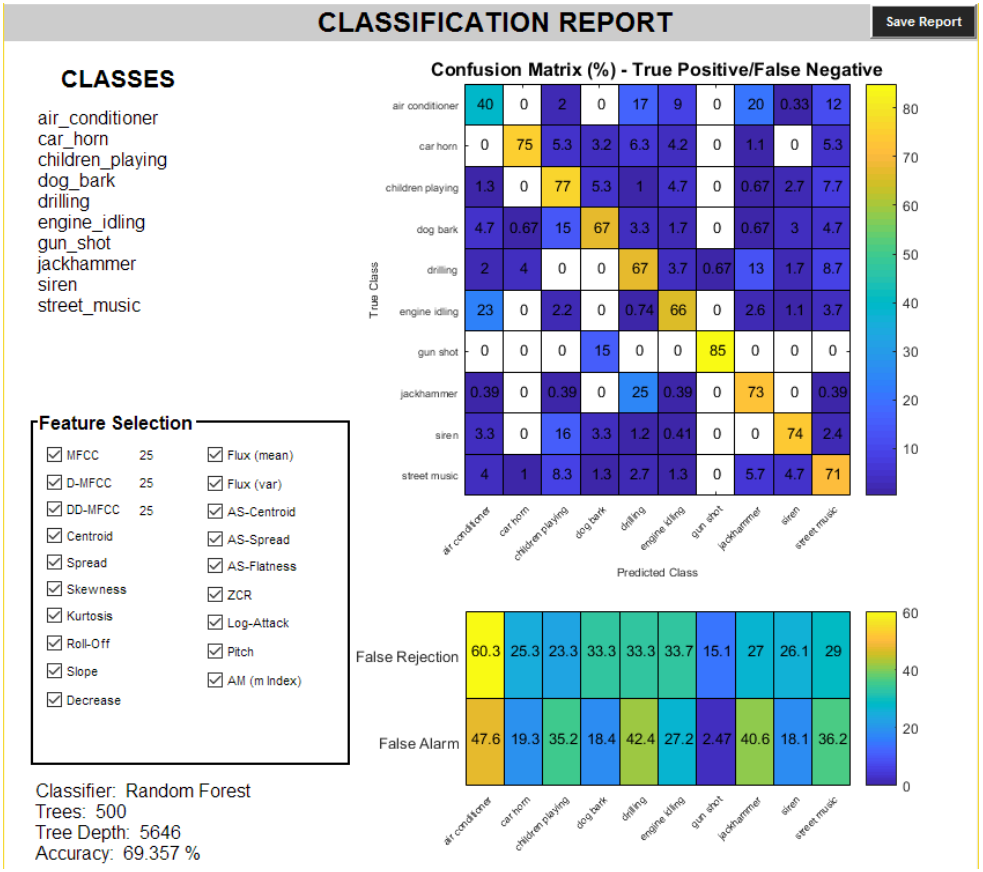

Fig. 48. Informe Clasificación *UrbanSound8K* con *Random Forest*

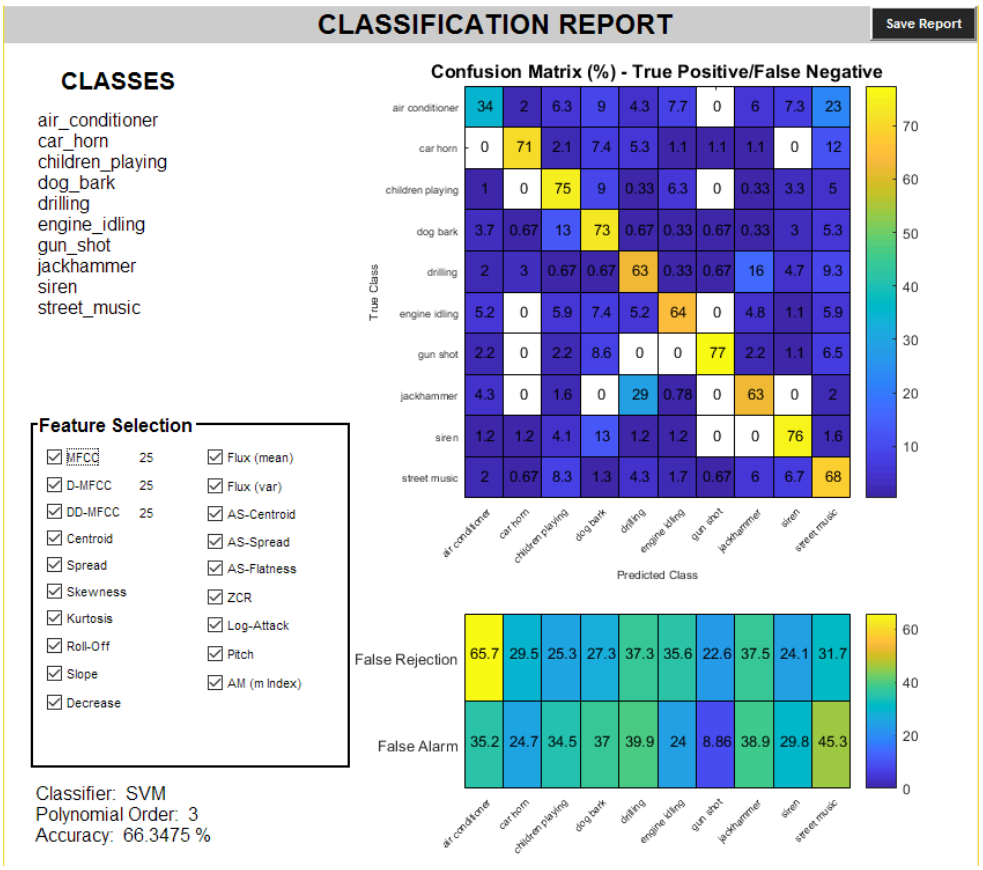

Fig. 49. Informe Clasificación *UrbanSound8K* con SVM

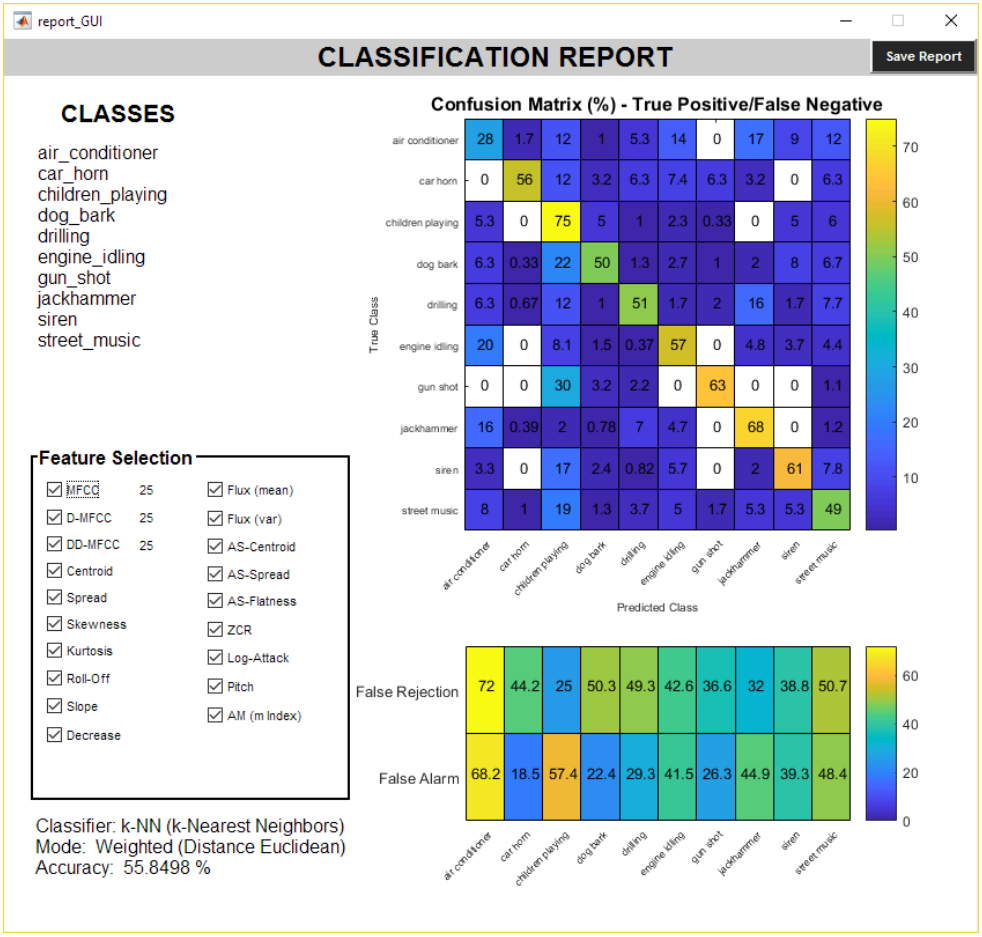

Fig. 50. Informe Clasificación *UrbanSound8K* con k-NN

Los MFCC son las *features* principales en muchos sistemas de reconocimiento de señales de audio. Por ello, se ha realizado una comparativa de resultados utilizando distintos MFCC con el objetivo de determinar cuántos MFCC son útiles en una clasificación de tipos de ruido, comenzando por 2 MFCC, junto con sus derivadas, hasta 40 MFCC en pasos de 2. Los resultados se muestran en la tabla 4. El algoritmo utilizado para realizar esta comparativa ha sido *Random Forest*.

Tabla 4. Resultados en función del número de MFCCs utilizados

n MFCC + **n** D-MFCC + **n** DD-MFCC

n	***Resultados (%)***	***n***	***Resultados (%)***
2	36,11	22	60,16
4	46,51	24	60,84
6	53,47	26	60,44
8	57,22	28	61,95
10	56,94	30	61,02
12	59,20	32	62,23
14	60,46	34	64,97
16	60,05	36	64,39
18	60,13	38	64,60
20	60,30	40	64,30

En la figura 51 se muestran los resultados de la tabla 4, donde se puede apreciar que el número de MFCCs utilizados en una clasificación, junto con sus derivadas, resultan en un rápido crecimiento hasta aproximadamente n=14, donde se produce cierta estabilidad. Sin embargo, a partir de n=26 aproximadamente, comienza una pequeña etapa de incremento en la precisión, alcanzando una precisión de 64 %, casi la precisión obtenida utilizando todas las *features* con los 25 primeros MFCC, lo cual demuestra la gran eficiencia de los MFCC.

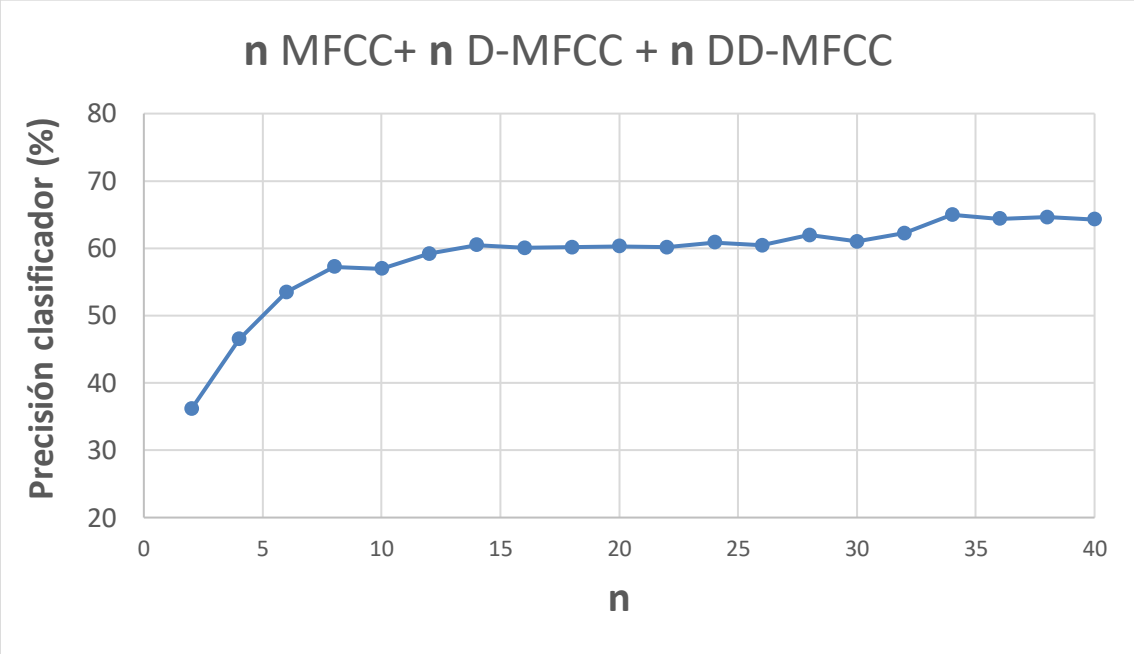

Fig. 51. Gráfica representativa de la precisión de clasificación frente al número de MFCC utilizados

A continuación, en la figura 52 se muestra el resultado obtenido de la clasificación utilizando los primeros 34 MFCC, donde se obtiene el máximo de precisión en la gráfica de la figura 51, un 65 %.

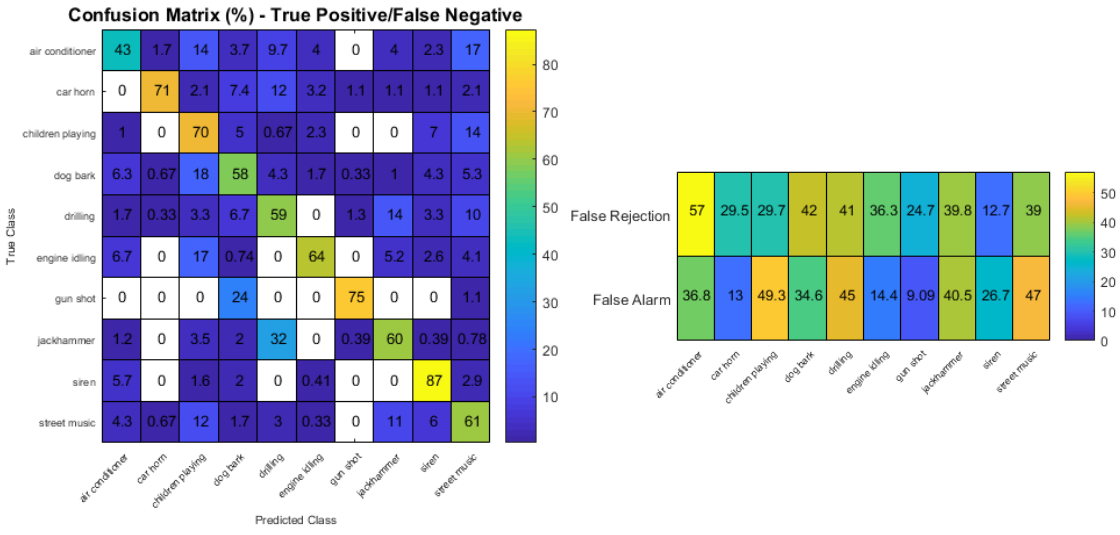

Fig. 52. Resultados utilizando los 34 primeros MFCC

Para comprobar la utilidad de las demás *features*, en la tabla 4 se muestran los resultados obtenidos utilizando los 34 primeros MFCC junto con una *feature* añadida, así como el resultado de utilizar dicha *feature* junto con únicamente el primer MFCC, es decir, la energía. En amarillo se muestran las tres *features* que mayor precisión aportan, tanto en conjunto con los MFCC como en solitario con únicamente el MFCC1. Como se puede observar, las *features* que aportan más información en conjunto con los MFCC son el *Flux*, el *Log-Attack* y el *Roll-off*. En solitario siguen siendo el *Flux* y el *Roll-off* las *features* que más información aportan, sin embargo, el *Log-Attack* en solitario no ofrece buenos resultados, siendo la tercera mejor el índice de modulación (*m Index*). En el Anexo III se muestran los informes de los resultados de cada *feature* individual junto con el MFCC1, esto permite averiguar para qué clases es útil cada *feature*. Por ejemplo, el *Log-Attack* resulta muy útil para predecir la clase disparo (*gun shot*).

Tabla 5. Resultados utilizando distintas *features*

FEATURES	*MFCC (1-34) + D (1-34) + DD (1-34)*	*MFCC1*
Centroid	65,58 %	25,15 %
Spread	64,74 %	18,19 %
Skewness	66,27 %	23,70 %
Kurtosis	65,29 %	21,36 %
Roll-Off	66,77 %	28,03 %
Slope	64,63 %	25,61 %
Decrease	63,36 %	22,75 %
Flux (2)	69,88 %	31,08 %
AS-Centroid	65,07 %	25,55 %
AS-Spread	62,89 %	21,68 %
AS-Flatness	63,89 %	23,24 %
ZCR	64,10 %	22,18 %
Log-Attack	68,06 %	23,92 %
Pitch	65,82 %	24,76 %
m Index (AM)	65,48 %	26,11 %

Conociendo qué *features* aportan información extra a los MFCC, es posible realizar una mejor selección de las mismas para obtener mejores resultados. Para ello, en la tabla 6 se muestran resultados para distintas combinaciones de *features* en conjunto con los MFCC, con el objetivo de hallas el mejor resultado posible.

Tabla 6. Resultados para distintas combinaciones de *features*

FEATURES	*MFCC (1-34) + D (1-34) + DD (1-34)*
Flux + Roll-Off	70,44 %
Flux + Log-Attack	71,26 %
Flux + Roll-Off + Log-Attack	72,59 %
Flux + Log-Attack + m Index	72,15 %
Flux + Roll-Off + Log-Attack + m Index	71,70 %
Flux + Roll-Off + Log-Attack + Skewness	70,92 %

Finalmente, en la figura 53 se muestra el mejor resultado obtenido para la base de datos *UrbanSound8K*, utilizando los MFCC, DMFCC y DDMFCC (1-34) en conjunto con el *flux* o variación espectral, el *roll-off* espectral al 85 % y el tiempo de ataque *log-attack*. Como se puede observar, la clase "aire acondicionado" (*air conditioner*) es la que mayor falso rechazo tiene y, la clase *gun shot*, la clase mejor detectada, con el menor falso rechazo y la menor falsa alarma. *Children playing* (niños jugando) es la clase con mayor falsa alarma, principalmente procedente de las clases *dog bark* (ladrido), *siren* (sirena) y *street music* (música de calle).

Un dato interesante es que, las clases *jackhammer* (martillo neumático) y *drilling* (taladradora) tienen gran falsa alarma. Sin embargo, la mayor parte de dicha falsa alarma se produce entre ellas mismas, por lo que, se podría crear una clase conjunta, denominada por ejemplo "construcción", en la cual se incluyesen ambas. De esta forma, habría más porcentaje de acierto clasificando ambas como herramientas de construcción que especificando qué tipo de máquina, para lo cual podría diseñarse un nuevo clasificador más específico.

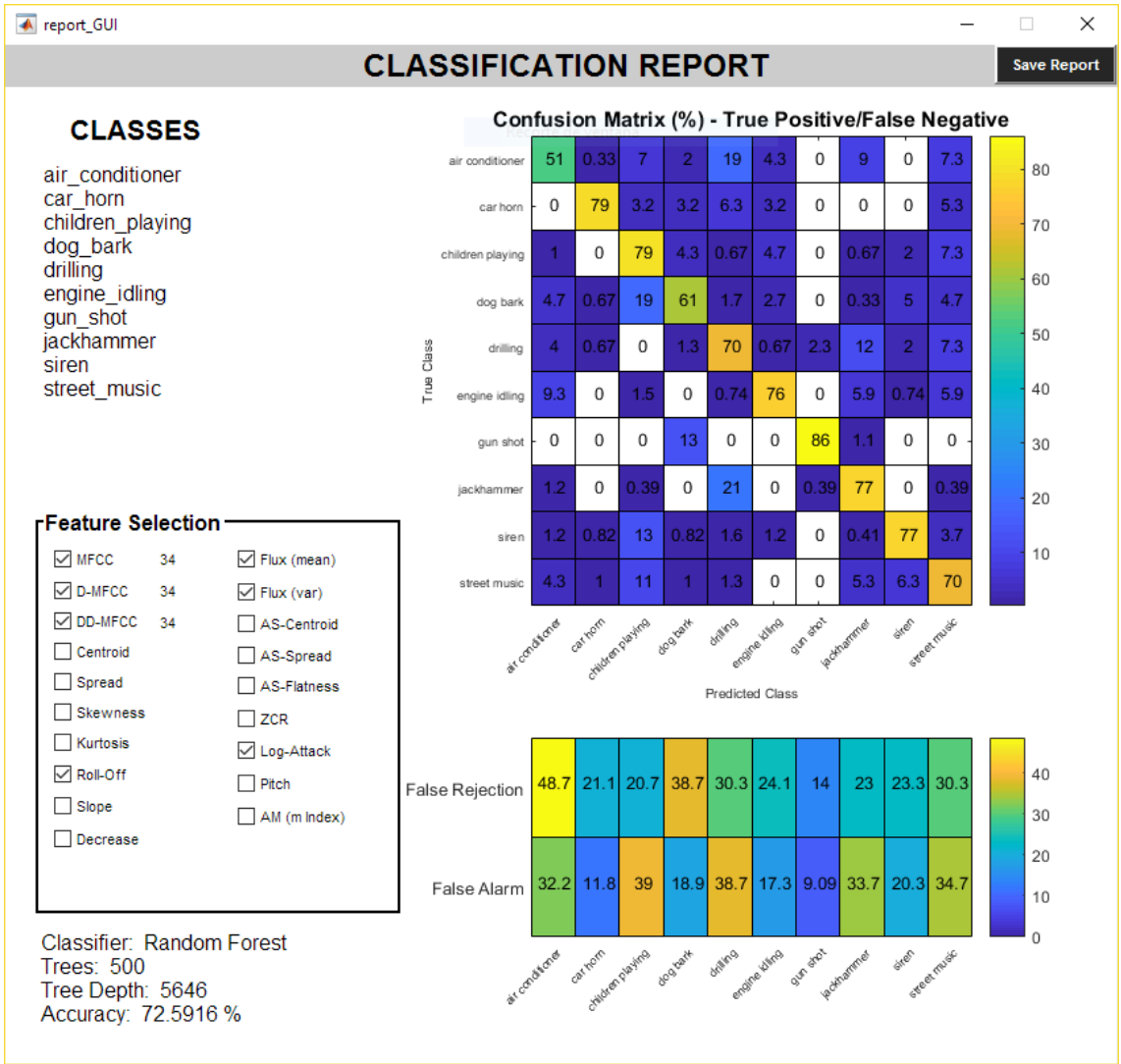

Fig. 53. Mejor resultado: MFCC (1-34) + DMFCC (1-34) + DDMFCC (1-34) + *Flux* + *Roll-Off* + *Log-Attack*

V.3.2. Clasificador Ruido

Para la aplicación de clasificación en tiempo real, se han realizado distintos clasificadores con el objetivo de organizarlos jerárquicamente. Entre ellos, se tiene un clasificador de ruido basado en la base de datos de *UrbanSound*. Sin embargo, en este clasificador se ha excluido la clase *air conditioner* y se ha incluido una nueva clase de lluvia, denominada *rain*. Además, las clases *drilling* y *jackhammer* se han unificado en una nueva clase denominada *building*, para reducir el error. En la figura 54 se muestran el informe de los resultados de dicho clasificador.

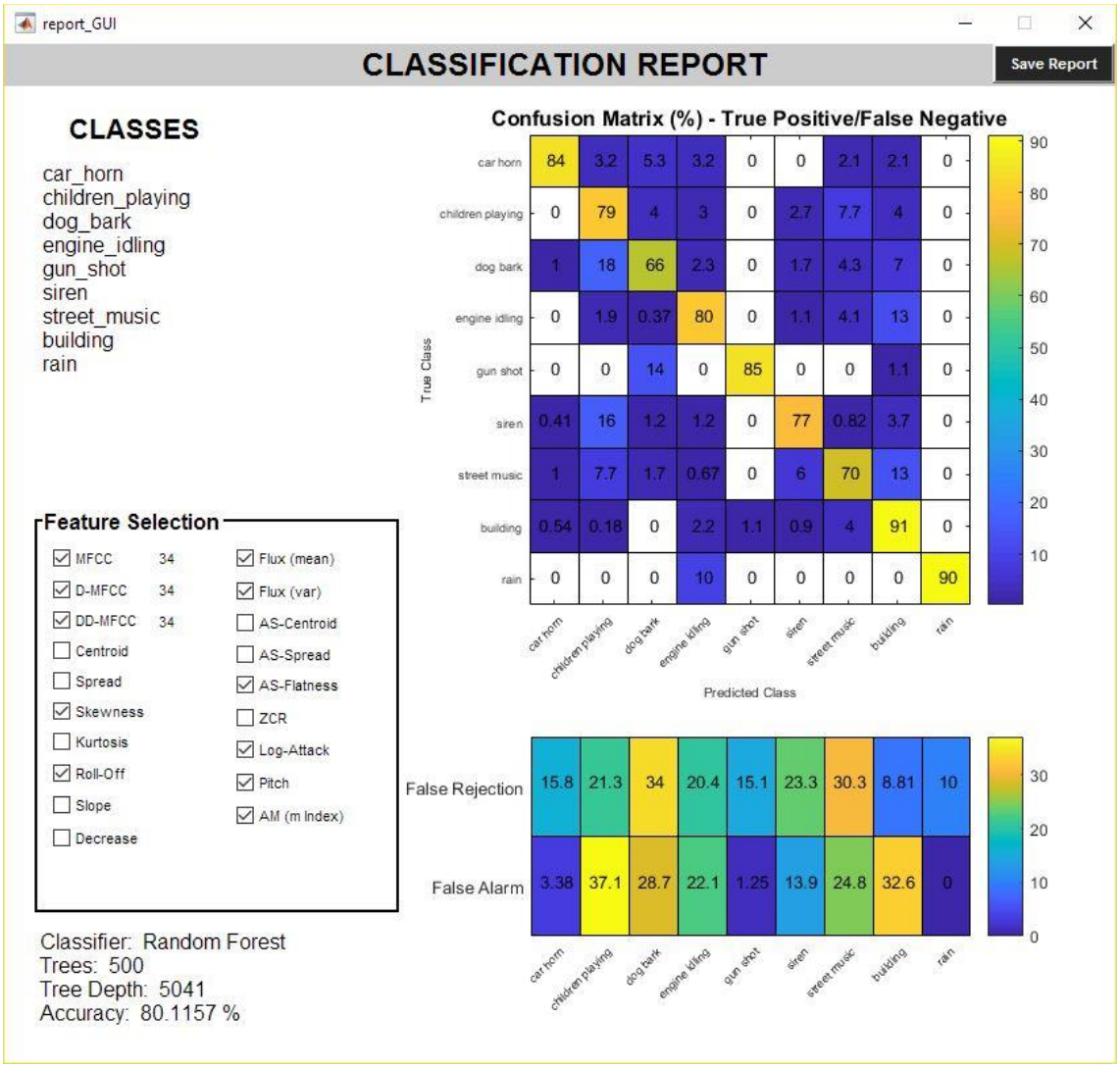

Fig. 54. Informe del clasificador de ruido

V.3.3. Clasificador Música

El clasificador de música creado para la aplicación de tiempo real ha sido entrenado con distintos géneros musicales de la base de datos GTZAN *Genre Collection*.

En la figura 55 se muestra el informe generado, en el cual se puede observar que los resultados no son muy buenos, excepto para las clases de música clásica, pop y metal. También se puede apreciar que los resultados para las clases Blues, Reggae y Rock son especialmente malos.

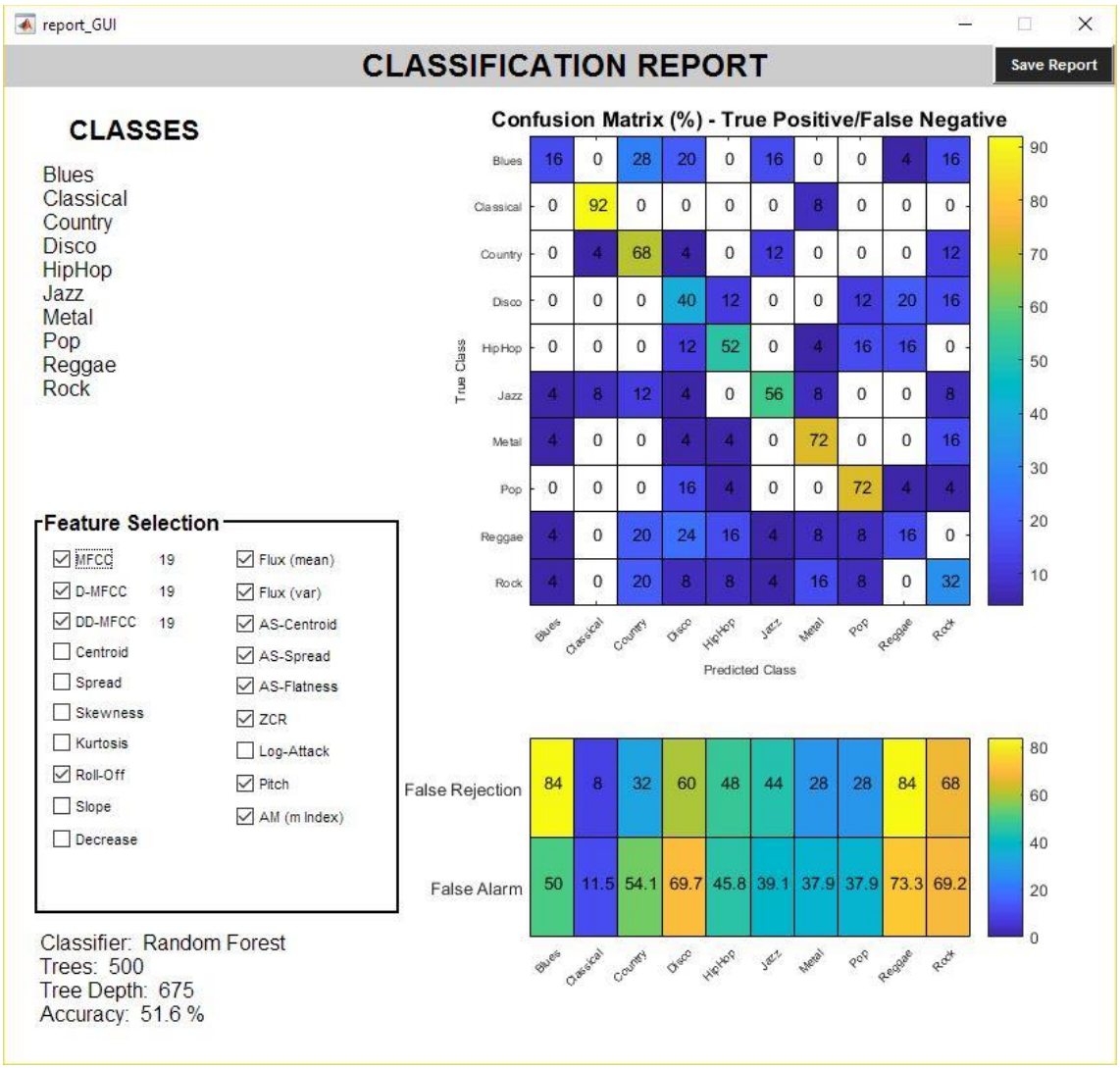

Fig. 55. Informe del clasificador de Música

V.3.4. Clasificador Voz

En la figura 56 se muestra el resultado de la clasificación de voz. Como se puede observar la clasificación únicamente se realiza para discriminar entre una voz cantando o hablando. No se necesitan demasiadas *features* para obtener estos resultados, tan solo los MFCC (1-16) y el *flux* o variación espectral. Además, solamente se ha realizado el clasificador con 15 árboles, lo cual resulta en un entrenamiento y una predicción muy rápida.

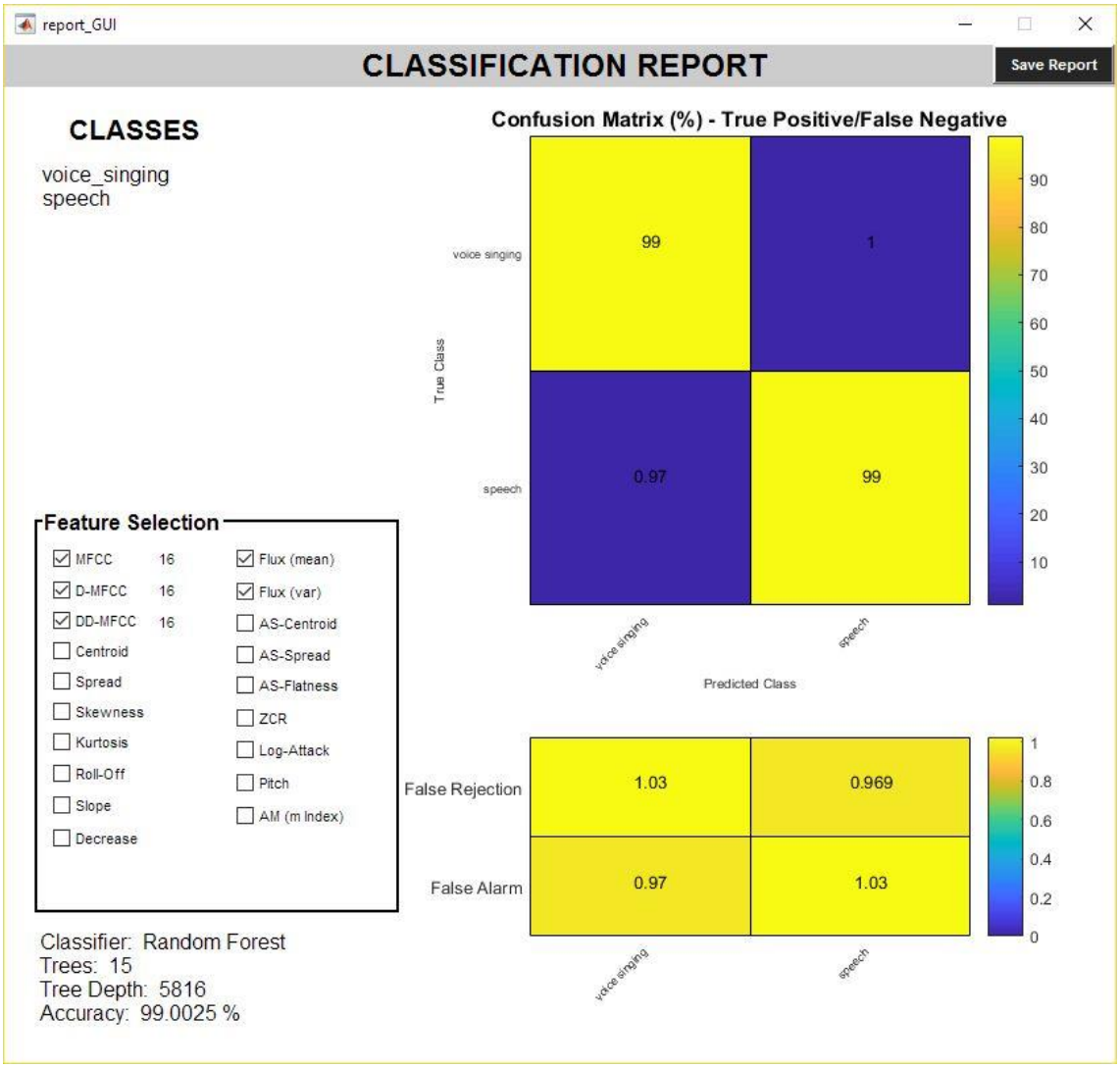

Fig. 56. Informe del clasificador de Voz

V.3.5. Clasificador Voz – Música – Ruido

En la figura 57 se muestra el clasificador de primer orden utilizado en la aplicación de tiempo real. Este clasificador determina si la señal de audio se trata de voz, música o ruido. Posteriormente, el audio se predice con los clasificadores de segundo orden. Como se puede apreciar, la precisión es alta para discriminar entre estas clases. Se puede observar que la clase música tiene alta falsa alarma, principalmente procedente de la voz, lo cual puede ser debido a que la voz ha sido entrenada para predecir si se trata de una voz cantando, por lo que también tendrá características musicales.

Para el entrenamiento de la clase música, se han utilizado todos los instrumentos de la base de datos IRMAS y la base de datos GTZAN de géneros musicales. Las clases voz y ruido han sido entrenadas con las clases de sus clasificadores correspondientes.

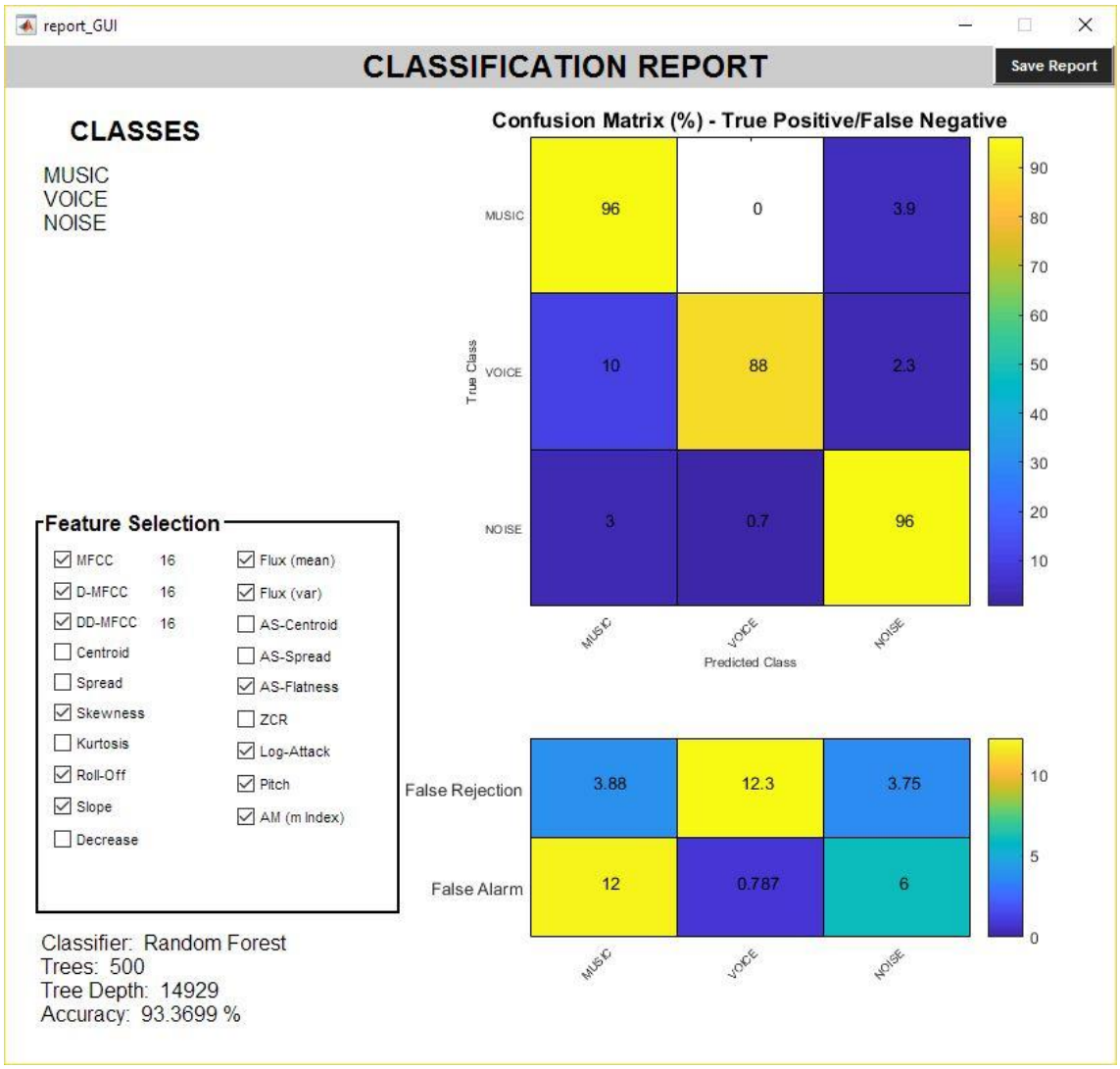

Fig. 57. Informe del clasificador principal: Voz - Música - Ruido

V.4. CLASIFICADOR EN TIEMPO REAL

La aplicación de clasificación en tiempo real desarrollada está basada en una jerarquía de clasificadores. El clasificador de primer orden tiene como finalidad determinar si la señal de audio se trata de una señal de voz, música o ruido. En función de la clase determinada, otro clasificador de segundo orden entrará en juego, determinando qué tipo de ruido, qué tipo de música, o qué tipo de voz. En la figura 58 se muestra la taxonomía de este clasificador, indicando las clases que contiene cada uno de los clasificadores de segundo orden.

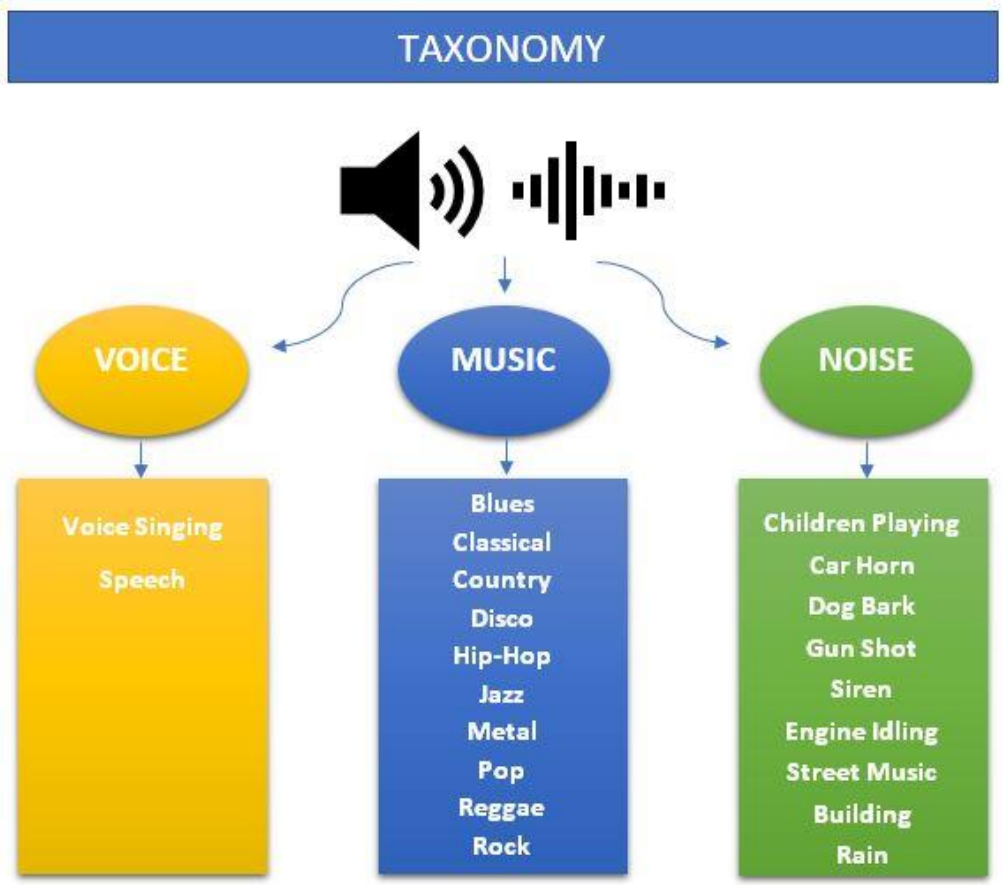

Fig. 58. Taxonomía del clasificador en tiempo real

Para el desarrollo de la aplicación se ha utilizado *PlayRec*, una herramienta gratuita basada en la librería de código abierto *PortAudio*, la cual permite el acceso a tarjetas de sonido para obtener entrada y salida de audio en tiempo real desde *Matlab*. En este caso, el audio puede ser introducido mediante la tarjeta de audio del ordenador, una tarjeta externa, o bien mediante un archivo de audio. La aplicación utiliza las mismas características que la herramienta de clasificación, ya que las *features* deben ser extraídas de la misma forma para una correcta clasificación. Cabe destacar que la predicción se realiza por cada 4 segundos de señal, ya que según los estudios de *UrbanSound* [10] los resultados de tipo ruido con menor tiempo de análisis consiguen menos precisión.

En la figura 59 de muestra la interfaz gráfica de esta aplicación. Como se puede observar, la aplicación se encarga de etiquetar el audio, con la clase y subclase a la que pertenece, cada 4 segundos. En la línea de tiempo se muestran las etiquetas, mientras que en la parte superior se muestran los histogramas de probabilidad tanto del clasificador de primer orden como del clasificador de segundo orden que actúe en cada momento.

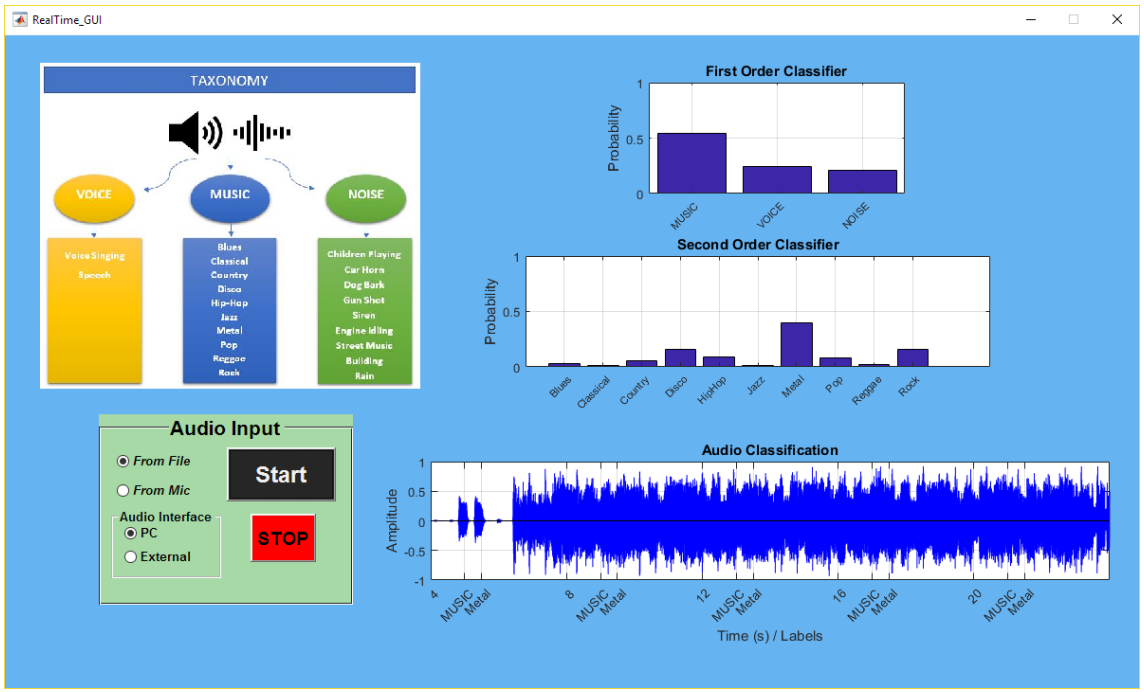

Fig. 59. Aplicación de clasificación en tiempo real

VI. LÍNEAS FUTURAS

El reconocimiento automático de señales de audio es un campo relativamente joven, por lo que se trata de un campo actualmente en investigación. Como se ha podido apreciar en los resultados del trabajo, uno de los factores más importantes a la hora de realizar una correcta clasificación es el uso de determinadas *features*. Por ello, una de las líneas futuras más importantes para obtener mejores resultados sería la incorporación de otras *features* ya existentes, así como el desarrollo e investigación de nuevas *features* que permitan discriminar mejor entre distintas clases.

Una aportación que podría añadirse fácilmente al trabajo sería la incorporación de un algoritmo de reducción de dimensionalidad, como por ejemplo PCA (*Principal Component Analysis*), ya que, estos algoritmos, conservan la información importante, creando un nuevo conjunto de *features* ortogonales entre sí a partir de las *features* originales de una selección, reduciendo así el número de estas, y, por tanto, la dimensionalidad. El hecho de reducir la dimensionalidad hace que el algoritmo de clasificación sea más eficiente, con menor coste computacional y mejores resultados, ya que, en definitiva, se reduce la redundancia y correlación entre *features*.

En el campo del *Machine Learning* existen una gran variedad de algoritmos que permiten realizar clasificación. Entre ellos, uno de los más utilizados actualmente para sistemas de audio son las redes neuronales (*Neural Networks*). Este tipo de algoritmo imita de alguna manera las redes neurales del cerebro humano, por lo que, si una persona es capaz de discriminar entre distintos tipos de señales de audio utilizando estas redes, cabe pensar que una máquina con una algoritmia similar podría realizar este cometido. Si bien, no hay que olvidar que se trataría únicamente del algoritmo de clasificación, el cual depende directamente de las *features* aportadas al mismo, es decir, de la extracción de información del audio.

Siguiendo en la línea del aprendizaje automático, en este trabajo únicamente se ha utilizado el aprendizaje supervisado. Sin embargo, se podría crear un sistema que utilizase aprendizaje no supervisado, incluyendo algoritmos de *clustering* que permitiesen crear nuevas clases, sin estar restringido a un determinado número de estas. También se podría crear un ensamblado de algoritmos, cuyo resultado fuese aportado por un consenso entre distintos algoritmos de distinta naturaleza, como por ejemplo *Random Forest* y SVM, entre otros.

En cuanto al pre-procesado del audio, se podría incorporar un algoritmo de separación de fuentes (*source separation*), ya que, en los sonidos de tipo ruido, puede existir más de una fuente al mismo tiempo. La supresión de ruido para determinados ruidos que afecten negativamente al sistema también podría ser interesante.

Por último, mencionar que, para aplicaciones de tiempo real, se podría realizar una red distribuida de micrófonos y procesadores, pudiendo elegir en cada momento la señal de micrófono más útil o el resultado consensuado de una clasificación de todas ellas. Además, el hecho de distribuir los procesadores permitiría mayor velocidad de cómputo para aquellos algoritmos que puedan paralelizar sus procesos, como por ejemplo el algoritmo *Random Forest*.

VII. CONCLUSIONES

En este trabajo se ha podido experimentar todo el proceso de clasificación de señales de audio, desde la extracción de características (*Feature Extraction*) hasta los algoritmos de clasificación de Aprendizaje Automático Supervisado (*Supervised Machine Learning*).

Las bases de datos juegan un papel muy importante dentro de la clasificación, ya que son éstas las que poseen los ejemplos con los que el sistema de debe aprender, por lo que una buena base de datos es crucial para realizar un buen entrenamiento. Esto es aún más importante cuando se trata de grabaciones de distintos tipos de ruido, ya que es precisamente el ruido el que dificulta en muchas ocasiones una correcta clasificación. Otro aspecto importante referente a las bases de datos es la determinación de ejemplos destinados a entrenamiento o evaluación, ya que una evaluación que contenga datos de entrenamiento resultará en unos resultados favorables, pero no reales.

Se ha podido observar la influencia que tiene una buena selección de *features* (*Feature Selection*) para obtener los resultados óptimos. Los MFCC han demostrado su gran capacidad para aportar información útil, siendo estos las *features* principales en cada clasificación realizada. Por otro lado, *features* como la variación o *flux* espectral, el *roll-off* o el tiempo de ataque han demostrado ser útiles para clasificaciones de ruido, concretamente para la base de datos *UrbanSound*. En la clasificación de géneros musicales, los resultados han sido bastante pobres. Sin embargo, esto no quiere decir que no sea posible obtener mejores resultados, ya que, se podrían implementar *features* más adecuadas al ámbito musical como detección de tempo y seguimiento de pitch (*pitch tracking*), o incluso transcripción de la misma para otro tipo de clasificación no necesariamente basada en el audio.

En cuanto a los algoritmos de *Machine Learning* utilizados, destaca el algoritmo *Random Forest*, cuyos resultados han sido mejores al resto de clasificadores en cada una de las evaluaciones.

Respecto a la realización de una clasificación jerárquica, no se ha realizado una prueba de evaluación general, sino únicamente pruebas de cada uno de los clasificadores por separado. Sin embargo, conceptualmente cabe decir que la ventaja de una clasificación jerárquica es que los errores de falsa alarma y falso rechazo disminuyen en cuanto a importancia se refiere, ya que, no es tan grave un error producido entre dos subclases pertenecientes a una misma clase de orden superior, que un error producido entre clases totalmente distintas. En este sentido, se podría establecer un umbral a la hora de determinar una subclase y, en caso de no superarse, utilizar únicamente la clase de orden superior. Los umbrales también pueden ser utilizados para disminuir la falsa alarma en contra de aumentar un falso rechazo, y viceversa, lo cual viene en muchos casos determinado por la seguridad que requiera la aplicación en concreto.

Por último, cabe decir que los objetivos se han alcanzado con éxito, realizando una herramienta de clasificación versátil y de fácil manejo. Permite incorporar bases de datos, extraer y seleccionar distintas *features* del audio, y entrenar y evaluar clasificadores utilizando distintos algoritmos de Aprendizaje Automático. También se ha realizado la aplicación de clasificación en tiempo real, utilizando una estructura jerárquica de clasificadores.

BIBLIOGRAFÍA

[1] M. Al-Maathidi M and Francis F. L. (2015) "Audio Content Feature Selection and Classification" en *IEEE International Conference on Progress in Informatics and Computing (PIC)*. Nanjing, China. December 2015

[2] S. Chachada and C.-C. J. Kuo, "Environmental sound recognition: A survey," en *Proc. Asia Pac. Signal Inf. Process. Assoc. Annu*. Summit Conf., Kaohsiung, Taiwan, 2013, pp. 1–9.

[3] Gil-Pita, R., Ayllón, D., Ranilla, J., Llerena-Aguilar, C., & Díaz, I. (2015). "A computationally efficient sound environment classifier for hearing aids". *IEEE Transactions on Biomedical Engineering*. Vol. 62, n. 10, October 2015 pp. 2358-2368.

[4] S. B. Kotsiantis (2007), "Supervised Machine Learning: A Review of Classification Techniques", en *Proceedings of the 2007 conference on Emerging Artificial Intelligence Applications in Computer Engineering*: *Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, p.3-24, June 10, 2007

[5] Center for Urban Science and Progress (NYU). *Urban Sound Datasets*. Disponible: <https://serv.cusp.nyu.edu/projects/urbansounddataset/> [Consulta: 8 de septiembre de 2017].

[6] L. Breiman, (2001) "Random forests" en *Machine Learning*, vol. 45, no. 1, pp. 5–32.

[7] A. Criminisi, J. Shotton and E. Konukoglu (2011), "Decision Forests for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning" Microsoft Research technical report TR-2011-114.

[8] Playrec, Multi-Channel Matlab Audio. *Playrec*. Disponible: <http://www.playrec.co.uk>/ [Consulta: 8 de septiembre de 2017].

[9] D. Gerhard, (2003) "Audio Signal Classification: History and Current Techniques," en *Technical Report, Department of Computer Science*, University of Regina, TR-CS 2003-07, pp. 1–38.

[10] J. Salamon, C. Jacoby, and J. P. Bello (2014) "A dataset and taxonomy for urban sound research," en *22nd ACM International Conference on Multimedia* (ACM-MM'14), Orlando, FL, USA, Nov. 2014.

[11] R. Lyon. (2010) "Machine hearing: An emerging field" en *Signal Processing Magazine, IEEE*, vol. 27, no. 5, pp. 131-139, Sept 2010.

[12] D'Arcy, T., Stanton, C., Bogdanovych, A. (2013) "Teaching a robot to hear: a real-time on-board sound classification system for a humanoid robot". en *Proceedings of Australasian Conference on Robotics and Automation*. Sydney, Australia (2013).

[13] Dan Jurafsky and James H. Martin (2009) “Feature Extraction: MFCC vectors” (Sección 9.3) en *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*, Pearson Prentice Hall, Upper Saddle River, N.J., Second edition,

[14] G. Peeters, “A large set of audio features for sound description (similarity and classification) in the CUIDADO project”, Technical report, IRCAM, 2004.

[15] Music Technology Group (Universidad Pompeu Fabra). *IRMAS Dataset*. Disponible: <https://www.upf.edu/web/mtg/irmas> [Consulta: 8 de septiembre de 2017].

[16] Carnegie Mellon University. *The CMU Audio Databases* Disponible: <http://www.speech.cs.cmu.edu/databases/pda/README.html> [Consulta: 8 de septiembre de 2017].

[17] Music Analysis, Retrieval and Synthesis for Audio Signals (MARSYAS). *GTZAN Genre and Music/Speech Collection.* Disponible: <http://marsyasweb.ap.pspot.com/download/data_sets/> [Consulta: 8 de septiembre de 2017].

[18] Dalibor Mitrovic, Matthias Zeppelzauer, and Christian Breiteneder (2010) “Features for content based audio retrieval”. *Advances in Computers*, vol. 78, pp. 71–150, 2010.

[19] S. C. Joshi, and A. N. Cheeran (2014) “MATLAB Based Feature Extraction Using Mel Frequency Cepstrum Coefficients for Automatic Speech Recognition”, *International Journal of Science, Engineering and Technology Research* (IJSETR), vol. 3, Issue 6, June 2014, pp. 1820-1823.

[20] Hariharan Subramanian (2004) “Audio Signal Classification”, *M.Tech. Credit Seminar Report, Electronic Systems Group*, EE. Dept, IIT Bombay, November 2004.

[21] R. Cachu, S. Kopparthi, B. Adapa, and B. Barkana, (2008) “Separation of voiced and unvoiced using zero crossing rate and energy of the speech signal,” en *ASEE*, Dec. 2008.

[22] L. Breiman J. Friedman C. J. Stone R. A. Olshen (1984) "*Classification and Regression Trees*" Londres: Taylor & Francis 1984.

[23] Liu, X., Song, M., Tao, D., Liu, Z., Zhang, L., Chen, C., & Bu, J. (2013). “Semi-supervised node splitting for random forest construction” en *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* 2013, pp. 492-499.

[24] Camastra F, Vinciarelli A. (2007) *Machine Learning for Audio, Image and Video Analysis*. London: Springer-Verlag.

ANEXO I: MANUAL DE USUARIO

HERRAMIENTA DE CLASIFICACIÓN DE AUDIO

En este manual se pretende dar una idea del funcionamiento general del clasificador, tanto para entender su estructura, como para realizar clasificaciones de una forma práctica. Esta herramienta se muestra en la figura 60.

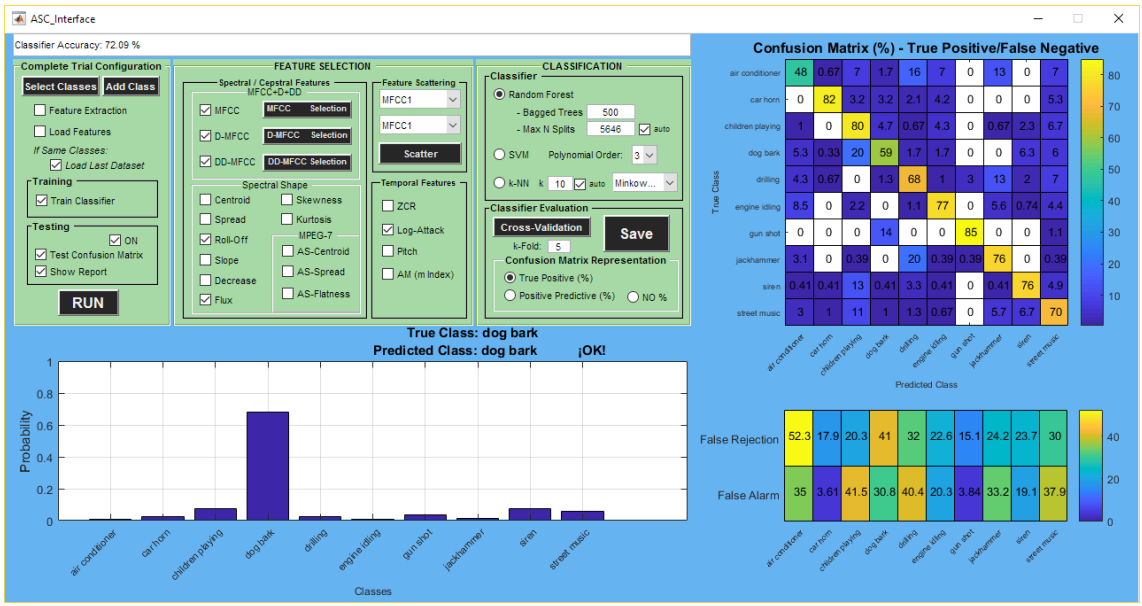

Fig. 60. Herramienta de clasificación de desarrollada

En primer lugar, en la figura 61 se muestra la estructura de carpetas sobre las cuales está programado el software.

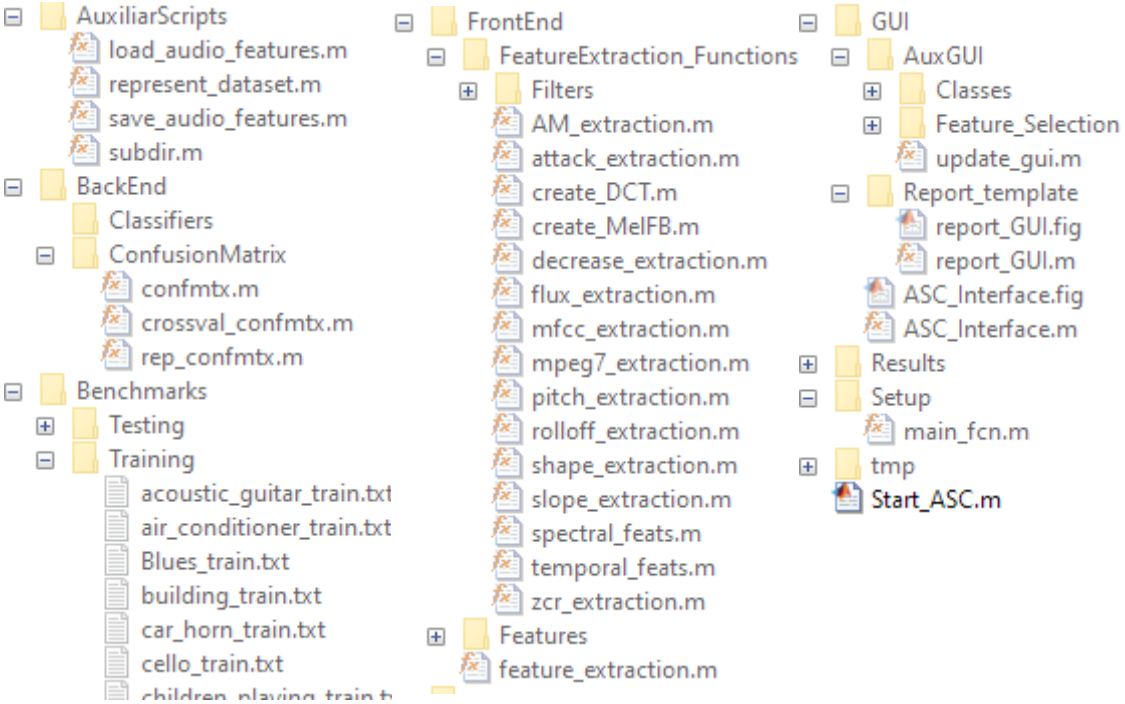

Fig. 61. Estructura de carpetas del programa

Las carpetas que contiene las funciones más relevantes, además de las interfaces (carpeta GUI), son las carpetas *Setup*, *FrontEnd* y *BackEnd*. En *Setup* se encuentra la función *main*, la cual realiza una ejecución completa: extracción de características, entrenamiento y evaluación. Las carpetas *FrontEnd* y *BackEnd* contienen las funciones necesarias para llevarse a cabo, concretamente la carpeta *FrontEnd* contiene todas las funciones de extracción (las cuales se adjuntan en el Anexo II) y el *BackEnd* las funciones de evaluación de la matriz de confusión. Además, la función FrontEnd también contiene la base de datos de *features*, extraídas de todos los audios.

En la carpeta Benchmarks se guardan todas las listas de entrenamiento y test que contienen los *paths* de todos los audios. Estas listas son simples archivos de texto que pueden ser editados de forma manual si se necesita incluir o excluir algún audio.

La carpeta GUI contiene todas las ventanas de tipo interfaz que permiten utilizar el programa de forma interactiva. En la carpeta temporal "tmp" se guardan algunos datos que el programa carga y guarda en diferentes ejecuciones, como, por ejemplo, el último clasificador realizado junto con sus clases y selección de *features*. Por último, en la carpeta *results* se guardan los resultados generados por la interfaz que genera un informe automático de cada clasificación.

Una vez entendida esta estructura, sería posible modificar alguna función o algún archivo si fuera necesario. Para evitar navegar por las carpetas y mantener siempre el mismo espacio de trabajo, la función "*start_ASC.m*" se encarga de establecer el *workspace* e inicializar la interfaz principal de la herramienta.

Para realizar una clasificación en primer lugar se deben añadir bases de datos para crear clases en caso de que no las haya. Para ello, bastaría con introducir el directorio donde se encuentran los audios, especificar un porcentaje de entrenamiento y test, y determinar el nombre de la clase, en la pestaña "*Add Class*", la cual se muestra en la figura 62.

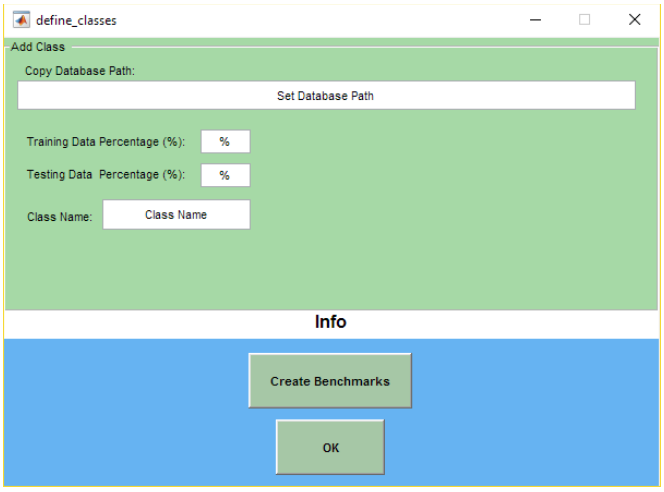

Fig. 62. Interfaz *Add Class*

Si ya existen clases incorporadas en el programa, simplemente habría que seleccionar las clases que se desean clasificar en la pestaña "*Select classes*", la cual se muestra en la figura 63.

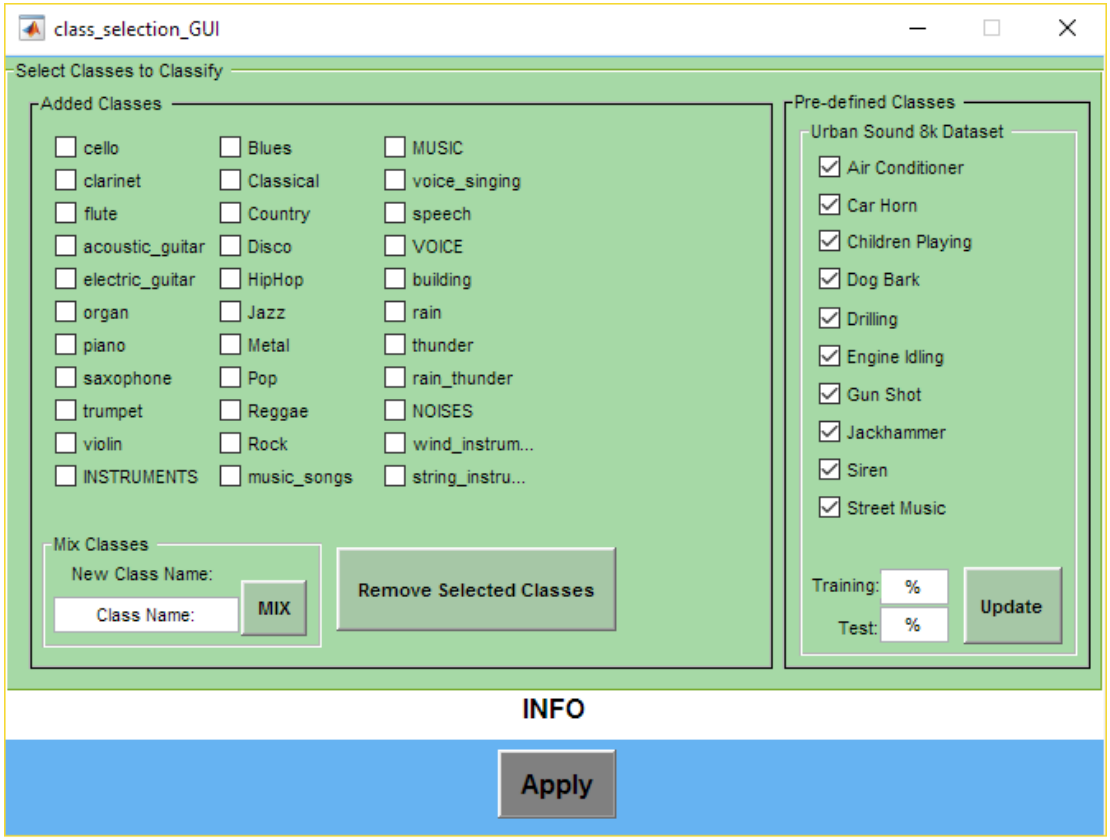

Fig. 63. Interfaz *Select Classes*

Una vez seleccionadas las clases, se realiza una selección de features en el módulo "*Feature Selection*", el cual se muestra en la figura 64, donde también se puede realizar un *scattering*. Además, se puede realizar una selección específica de los MFCC, D-MFCC y DD-MFCC.

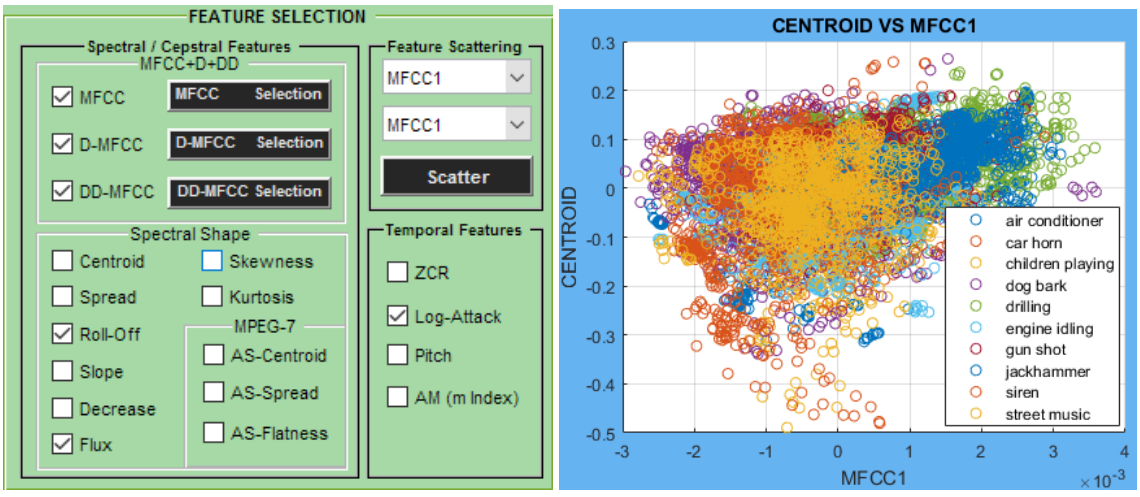

Fig. 64. *Feature Selection*

El clasificador a utilizar, con sus respectivos parámetros, se debe especificar en el módulo de clasificación, el cual se muestra en la figura 65.

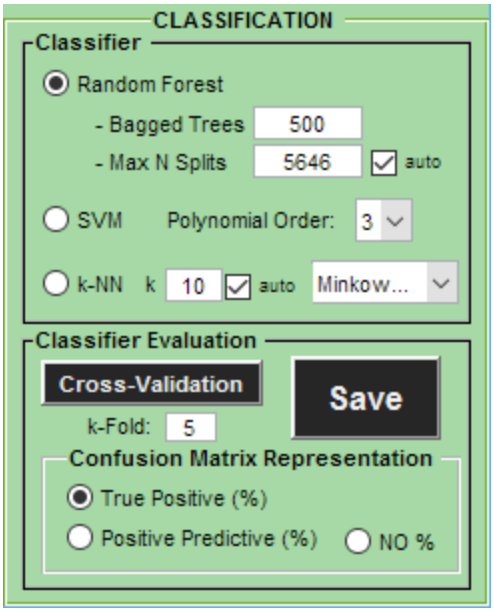

Fig. 65. Módulo de clasificación

Por último, se realiza la ejecución desde el módulo general de configuración, donde se especifica si se desea extraer las *features*, cargar las mismas si han sido extraídas previamente, o incluso cargar el último *dataset*. En la figura 66 se muestra dicho módulo.

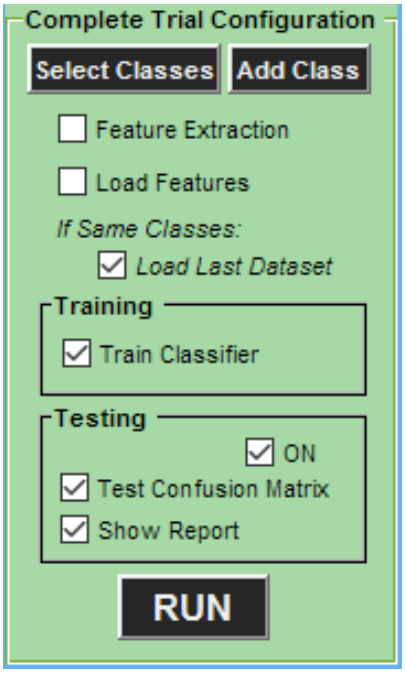

Fig. 66. Módulo general de configuración. Ejecución.

Por último, tras la ejecución, se mostrarán los resultados en la propia interfaz y se generará un informe automático de los resultados de la clasificación, el cual se muestra en la figura 67, siempre y cuando se haya seleccionado la pestaña "*show report*" en el módulo general.

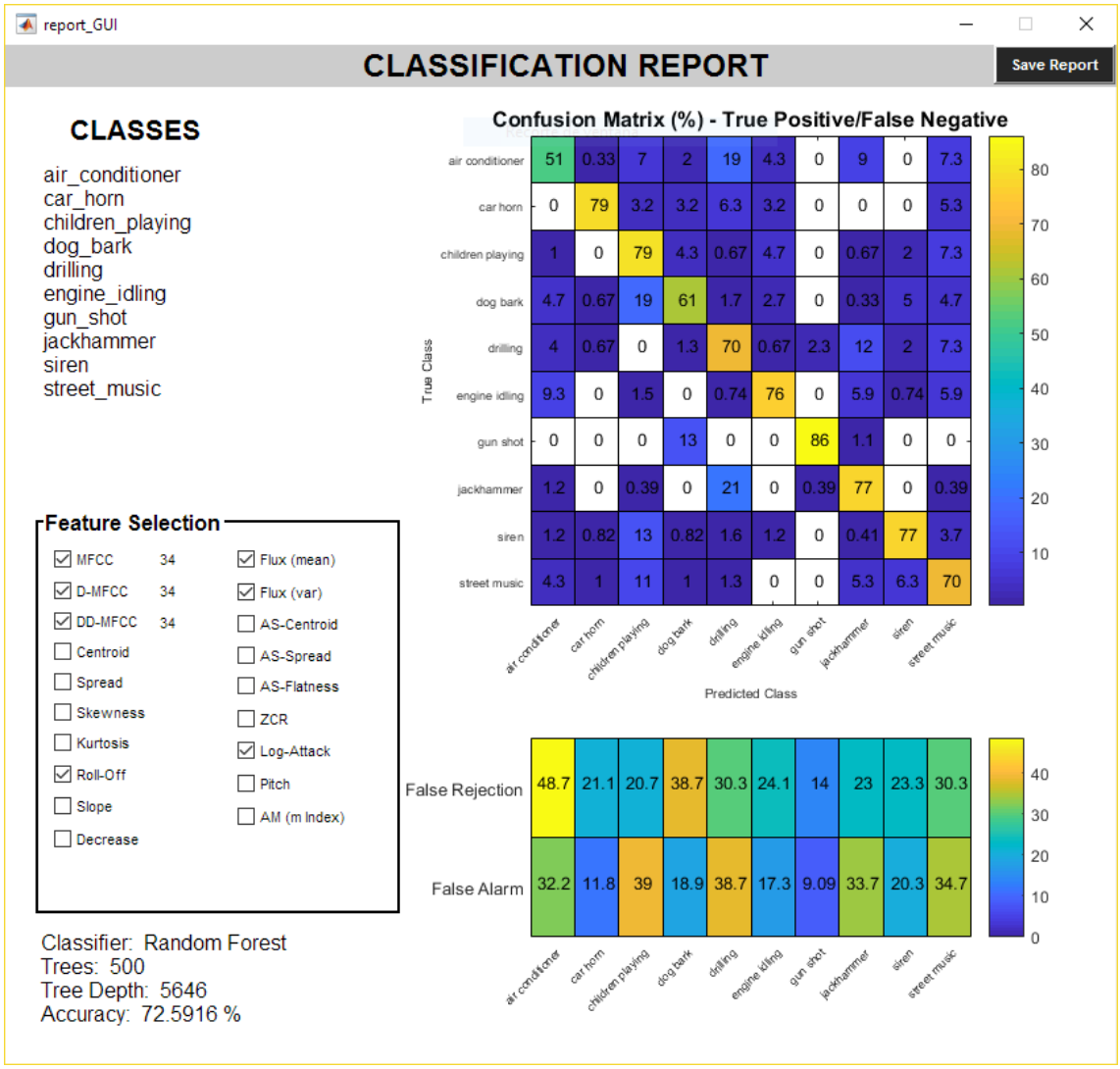

Fig. 67. Informe automático de la clasificación

CLASIFICADOR EN TIEMPO REAL

La aplicación de clasificación en tiempo real, la cual se muestra en la figura 68, utiliza las mismas funciones de extracción que la herramienta de clasificación.

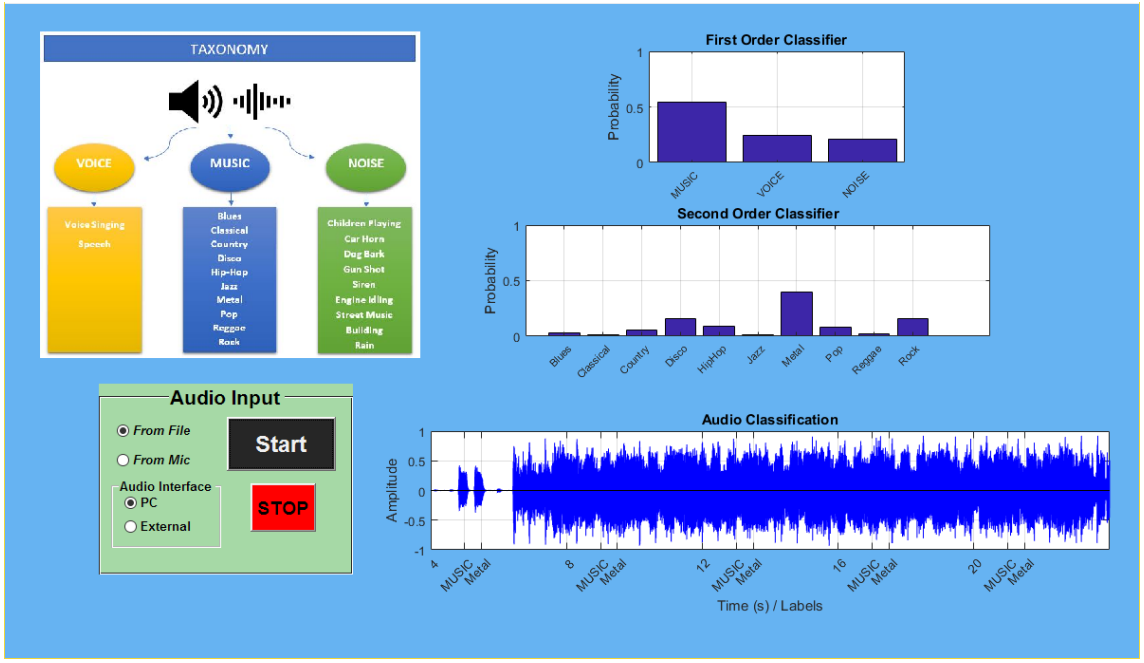

Fig. 68. Clasificación en tiempo real

La estructura de carpeta del programa se muestra en la figura 69. En este caso, en la carpeta *BackEnd* se incluyen los clasificadores de primer y segundo orden de la clasificación jerárquica.

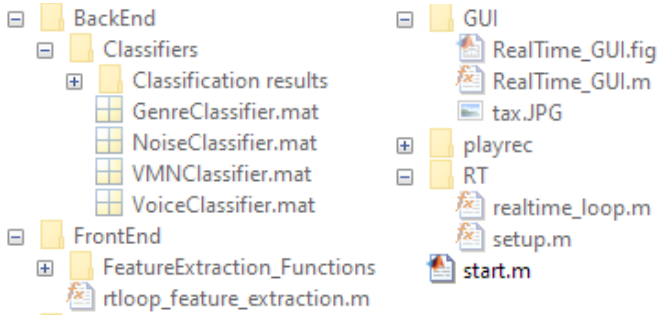

Fig. 69. Estructura de carpeta del clasificador en tiempo real

En la carpeta *playrec* se incluyen todas las funciones de dicha herramienta que permiten las entrada y salida de audio en tiempo real. Al igual que en la herramienta de clasificación, también se tiene un script *start* que establece el espacio de trabajo e inicia la interfaz gráfica. Es importante mencionar que en la función *setup* se pueden especificar los identificadores de las tarjetas de audio para evitar tener que introducir los mismos en cada ejecución. Por último, destacar que la función principal que realiza la clasificación en tiempo real es "*realtime_loop.m"*.

ANEXO II: CÓDIGO IMPLEMENTADO EN MATLAB

En este anexo se presentan los códigos más importantes de la herramienta de clasificación desarrollada. Las funciones de tipo interfaz y auxiliares, así como el resto de archivos necesarios, se pueden encontrar en los archivos adjuntados electrónicamente. Se incluye:

- Función principal
- Funciones de *Feature Extraction*

HERRAMIENTA DE CLASIFICACIÓN

Función Principal

```
function main_fcn(config,handles)
%% MAIN FUNCTION ---------------------------------------------------------- %
% Función que realiza una prueba completa de clasificación, extrayendo    %
% características de los audios pertenecientes a las clases que se        %
% deseen clasificar o cargando los datos si ya han sido extraidos         %
% previamente. Estos datos son clasificados mediante distintos tipos de   %
% clasificadores, realizando una prueba de test para evaluar el           %
% clasificador y obtener resultados.                                      %
%                                                                         %
% Procedimiento:                                                          %
%                                                                         %
%       - Training                                                        %
%           * Feature Extraction                                          %
%           * Train Classifier                                            %
%                                                                         %
%       - Testing                                                         %
%           * Feature Extraction                                          %
%           * Classifier Testing --> Confusion Matrix                     %
%                                                                         %
%           * Audio Sample Prediction (example)                           %
%                                                                         %
%.........................................................................%
%       - Input:                                                          %
%           * config(struct):                                             %
%               'feature_selection'                                       %
%               'classes'                                                 %
%               - Configuración general:                                  %
%                     'do_FE'                                             %
%                     'load_data'                                         %
%                     'do_trainClassifier'                                %
%                     'do_test'                                           %
%                     'do_testCmtx'                                       %
%                     'report'                                            %
%               - Configuración Clasificador:                             %
%                     'RF_classifier'                                     %
%                          'n_trees'                                      %
%                          'autosplit'                                    %
%                          'MaxNumSplits'                                 %
%                     'SVM_classifier'                                    %
%                          'SVMorder'                                     %
%                     'kNN_classifier'                                    %
%                          'kNN_mode'                                     %
%                          'auto_k'                                       %
%                          'k_neighbors'                                  %
%                                                                         %
%           * handles (representation)                                    %
%.........................................................................%
%                                                                         %
%                                                 Fabián Aguirre Martín   %
%                                                               2017      %
```

```
%_____________________________________________________________________%
debug=0;
%% WORKSPACE_____________________________________________________________%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              WORKSPACE                                %
%.....................................................................%
% --------------------------- Directories ------------------------------ %
workpath = '.'; % workpath = pwd;
temp_path=[workpath '/tmp/'];
% ..................................................... Front End ...... %
frontend_path=[workpath '/FrontEnd'];
featpath_train=[frontend_path '/Features/'];
featpath_test=[frontend_path '/Features/'];
if ~exist(featpath_train,'dir'); mkdir(featpath_train); end
if ~exist(featpath_test,'dir');  mkdir(featpath_test);  end
% ..................................................... Back end ....... %
backend_path=[workpath '/Backend/'];
models_path=[backend_path 'Classifiers/'];
if ~exist(models_path, 'dir'); mkdir(workpath,models_path); end
% ..................................................... Benchmarks ..... %
benchmark_path=strcat(workpath,'/Benchmarks/');
training_path=strcat(benchmark_path, 'Training/');
testing_path=strcat(benchmark_path, 'Testing/');
% ..................................................... Results ........ %
results_path=[workpath '/Results'];
if ~exist(results_path,'dir'); mkdir(results_path); end
%_____________________________________________________________________%
%% CONFIGURATION_________________________________________________________%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              CONFIGURATION                            %
%.....................................................................%
% --> Feature Extraction ------------------------------------------------ %
do_FE=config.do_FE;
%       - =1 ON  -> Feature Extraction (all features will be extracted)   %
%       - =0 OFF -> Load features                                         %
%                                                                         %
load_dataset=config.load_data;
%       - =1 ON  -> Load last dataset (allows new feature selection)      %
%       - =0 OFF -> Load or extract features of each audio                %
%                                                                         %
% --> Classifier Training ------------------------------------------------ %
training_classifier=config.do_trainClassifier;
%       - =1 ON  -> Train a new classifier                                %
%       - =0 OFF -> Load the classifier if exists                         %
%                                                                         %
% --> Classifier Testing ------------------------------------------------- %
testing=config.do_test;
    test_sample=1;                          % Evaluate an audio sample    %
    test_confusionmtx=config.do_testCmtx;   % Calculate Confusion Matrix  %
%_____________________________________________________________________%
%% CLASSES_______________________________________________________________%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                 CLASSES                               %
%.....................................................................%
% Example:                                                              %
% classes={'air_conditioner';                                           %
%          'car_horn';                                                  %
%          'children_playing';                                          %
%          'dog_bark';                                                  %
%          'drilling';                                                  %
%          'engine_idling';                                             %
%          'gun_shot';                                                  %
%          'jackhammer';                                                %
%          'siren' ;                                                    %
%          'street_music'};                                             %
% ..................................................................... %
classes=config.classes;     % Classes Selected      (cell)
n_class=length(classes);    % Number of Classes
```

```
%                                                                       %
% ............................. BENCHMARKS ............................. %
train=cell(n_class, 1);
test=cell(n_class, 1);
for i=1:n_class
%       - Training Benchmarks
    train{i,1}=[training_path char(classes{i}) '_train.txt'];
%       - Testing Benchmarks
    test{i,1}=[testing_path char(classes{i}) '_test.txt'];
end
%_______________________________________________________________________%
% Initialization
message='';
% feat_dim=sum(config.feature_selection.on_off);

%% TRAINING_____________________________________________________________%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                              TRAINING                                 %
%.....................................................................%
%   --> Front End                                                       %
%           - Feature Extraction (or Load features)                     %
%           - Normalization                                             %
%           - Feature Selection                                         %
%                                                                       %
%   --> Back End                                                        %
%           - Train Classifier                                          %
%_____________________________________________________________________%
TRAINDATA=[];
if training_classifier || do_FE
    if ~load_dataset
    %% ____________________________ FRONT-END ____________________________ %
    for clss=1:n_class
        % ..................... Audio Input ............................. %
        % Lectura de Listas de entrenamiento        (Training Benchmarks)
        fid=fopen(train{clss},'r');
        train_data=textscan(fid,'%s'); fclose(fid);
        %% ---------------------------------------- FEATURE EXTRACTION -- %
        [m,~]=size(train_data{1});
        features=zeros(length(config.feature_selection.on_off),m);
        for i=1:m  %----> Audio_i (class_n)
            audio_path=char(train_data{1}(i));
            if do_FE
                features(:,i)=feature_extraction(audio_path,config);
                 % Save features
                message=save_audio_features(features(:,i),audio_path,...
                                            featpath_train);
            else % Load features
                [features(:,i), message]=load_audio_features(audio_path,...
                                            featpath_train, config);
            end
            %___________________________________________________________
            % features =                                                %
            %          Audio1  Audio2  Audio3    Audioi                %
            %         ---------------------------------                 %
            %        [ MFCC1   MFCC1   MFCC1 ... MFCC1 ]                %
            %        [ MFCC2   MFCC2   MFCC2 ... MFCC2 ]                %
            %        [   .       .       .         .   ]                %
            %        [   .       .       .         .   ]                %
            %        [   .       .       .         .   ]                %
            %        [ MFCCn   MFCCn   MFCCn ... MFCCn ]                %
            %        [   .       .       .         .   ]                %
            %        [   .       .       .         .   ]                %
            %        [  ZCR      ZCR     ZCR       ZCR  ]               %
            %        [   .       .       .         .   ]                %
            %        [   .       .       .         .   ]                %
            %        [   .       .       .         .   ]                %
            %        [ FeatX   FeatX   FeatX     FeatX ]                %
            %         ---------------------------------                 %
```

```
            % --> Information                                                 %
            str = sprintf([' TRAINING (Feature Extraction) \t\t ' ...       %
                'Class:  %s  \t\t %s'],char(classes{clss}),message);        %
            set(handles.info,'string',str);drawnow; % display(str);         %
            %_____________________________________________________________%
        end
        % ----------> class i
        % DataStruct
        % eval(['train_features.' classes{clss} '= feats_train;']);
        % DataMatrix
        feats_train=features';
        [m_audios , n_feats] = size(feats_train);
        feats_train(:,n_feats+1)=(clss-1)*ones(m_audios,1);% Include labels
        TRAINDATA=[TRAINDATA; feats_train];
        %_____________________________________________________________%

    end
    %% ------------------------------------------------ NORMALIZATION ---- %
    feats_mean=mean(TRAINDATA(:,1:end-1),1);
    feats_var=var(TRAINDATA(:,1:end-1),0,1);

    norm.feats_mean=feats_mean;
    norm.feats_var=feats_var;
    save('./tmp/normalization.mat','norm');

    TRAINDATA(:,1:end-1)=bsxfun(@minus,TRAINDATA(:,1:end-1),feats_mean);
    TRAINDATA(:,1:end-1)=bsxfun(@rdivide,TRAINDATA(:,1:end-1),feats_var);
    % Save DATASET
    save([temp_path 'TRAINDATA.mat'], 'TRAINDATA');
    else
        % Load DATASET
        if exist([temp_path 'TRAINDATA.mat'],'file')
           load([temp_path 'TRAINDATA.mat']);
           load('./tmp/normalization.mat');
        else
            erstr='Error:    There is no training data to Load. ';
            disp(erstr);set(handles.info,'string',erstr);drawnow;
            training_classifier=0;
        end
    end
    %_____________________________________________________________%

    %% ------------------------------------------- FEATURE SELECTION -- %
    if ~isempty(TRAINDATA)
        [large,M]=size(TRAINDATA);
        feats=find(config.feature_selection.on_off==1);
        if debug;represent_dataset(TRAINDATA,'TRAIN DATASET');end
        if max(feats)<M && isempty(find(isnan(TRAINDATA(:,feats)),1))
            data=TRAINDATA(:,feats);    %- Data to Train
            labels=TRAINDATA(:,M)'+1;   %- Class Labels
        else
            erstr='Error: Features Selected are not in the Training Data.';
            disp(erstr);set(handles.info,'string',erstr);drawnow;
            training_classifier=0;
        end
    end % (FrontEnd.)
    %_____________________________________________________________%

%% ______________________________ BACK-END ______________________________ %
if training_classifier
    %% ------------------------------------------------ CLASSIFIER ---- %
    if config.RF_classifier
        % ------------------------------------------- Random Forest --- %
        name='RandomForest_Classifier';
        % ---> Information
        str = sprintf([' Training Classifier: \t\t ' ...
            '%s  \t\t Wait.............................'],name);
        set(handles.info,'string',str);drawnow; % display(str);
```

```
        %.....................................................................%
        if config.autosplit
            config.MaxNumSplits=round(0.9*large);
            set(handles.n_splits,'String',num2str(config.MaxNumSplits));
            templ = templateTree('MaxNumSplits',config.MaxNumSplits);
        else
            templ = templateTree('MaxNumSplits',config.MaxNumSplits);
        end

        classifier=fitcensemble(data, ...
                                labels, ...
                               'Method', 'Bag', ...
                               'NumLearningCycles', config.n_trees, ...
                               'Learners', templ);
        %_____________________________________________________________________%
    elseif config.SVM_classifier
        % ------------------------------------------------- SVM ------- %
        name='SVM_Classifier';
        % ---> Information
        str = sprintf([' Training Classifier: \t\t ' ...
            '%s  \t\t Wait.............................'],name);
        set(handles.info,'string',str);drawnow; % display(str);

        %.....................................................................%
        polyOrder=config.SVMorder;
        template = templateSVM(...
            'KernelFunction', 'polynomial', ...
            'PolynomialOrder', polyOrder, ...
            'KernelScale', 'auto', ...
            'BoxConstraint', 1, ...
            'Standardize', true);
        classifier = fitcecoc(...
            data, ...
            labels, ...
            'Learners', template, ...
            'Coding', 'onevsone'); % 1vs1: K(K-1)/2 BinarySVM Classifiers

        % classifier=fitcsvm(data,labels); % For Binary Classification
        %_____________________________________________________________________%
    elseif config.kNN_classifier
        % ------------------------------------------------- k-NN ------ %
        name='kNN_Classifier';
        % ---> Information
        str = sprintf([' Training Classifier: \t\t ' ...
            '%s  \t\t Wait.............................'],name);
        set(handles.info,'string',str);drawnow; % display(str);
        if config.auto_k
            k_n=n_class;
            config.k_n=k_n;
        else
            k_n=config.k_neighbors;
            config.k_n=k_n;
        end
        %.....................................................................%
        switch config.kNN_mode
            case 1
            classifier = fitcknn(data, ...
                                 labels, ...
                                'Distance', 'Minkowski', ...
                                'Exponent', 3, ...
                                'NumNeighbors', k_n, ...
                                'DistanceWeight', 'Equal', ...
                                'Standardize', true);
            case 2
                distance='Euclidean';
            case 3
                distance='Chebychev';
        end
```

```
        if config.kNN_mode~=1
                        classifier = fitcknn(data, ...
                                 labels, ...
                                'Distance', distance, ...
                                'Exponent', [], ...
                                'NumNeighbors', k_n, ...
                                'DistanceWeight', 'SquaredInverse', ...
                                'Standardize', true);
        end
        %_____________________________________________________________________%
    end
    % Save Classifier
    % save([models_path name '_F' num2str(feat_dim) '.mat'], 'classifier');
    % Save complete classifier
    trained_classifier.classifier=classifier;
    trained_classifier.feature_selection=config.feature_selection;
    trained_classifier.norm=norm;
    trained_classifier.classes=classes;
    save([temp_path 'trained_classifier.mat'], 'trained_classifier');
end % (BackEnd.)
end %.....................................................................%
%_______________________________END-TRAINING_________________________________%

%% TESTING_____________________________________________________________________%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                               TESTING                                       %
%.....................................................................%
%   --> Front End                                                             %
%           - Feature Extraction (or Load features)                           %
%           - Normalization                                                   %
%           - Feature Selection                                               %
%                                                                             %
%   --> Back End                                                              %
%           - Test Classifier --> Confusion Matrix                            %
%           - Audio Sample Prediction (example)                               %
%_____________________________________________________________________________%
TESTDATA=[];
if testing || do_FE
    if ~load_dataset
    %% ____________________________ FRONT-END ____________________________ %
    disp('Testing');
    for clss=1:n_class
        % ..................... Audio Input ............................. %
        % Lectura de Listas de test        (Testing Benchmarks)
        fid=fopen(test{clss},'r');
        test_data=textscan(fid,'%s'); fclose(fid);
        %% ---------------------------------------- FEATURE EXTRACTION -- %
        [m,~]=size(test_data{1});
        features=zeros(length(config.feature_selection.on_off),m);
        for i=1:m  %----> Audio_i (class_n)
            audio_path=char(test_data{1}(i));
            if do_FE
                features(:,i)=feature_extraction(audio_path,config);
                 % Save features
                message=save_audio_features(features(:,i),audio_path,...
                                            featpath_test);
            else % Load features
                [features(:,i), message]=load_audio_features(audio_path,...
                                            featpath_test, config);
            end
            %_____________________________________________________________
            % features =                                                    %
            %          Audio1  Audio2  Audio3    Audioi                   %
            %         ---------------------------------                   %
            %        [ MFCC1   MFCC1   MFCC1 ... MFCC1 ]                  %
            %        [ MFCC2   MFCC2   MFCC2 ... MFCC2 ]                  %
            %        [   .       .       .         .   ]                  %
            %        [   .       .       .         .   ]                  %
```

```
            %       [   .       .       .         .   ]                   %
            %       [ MFCCn   MFCCn   MFCCn ... MFCCn ]                   %
            %       [   .       .       .         .   ]                   %
            %       [   .       .       .         .   ]                   %
            %       [  ZCR     ZCR     ZCR       ZCR  ]                   %
            %       [   .       .       .         .   ]                   %
            %       [   .       .       .         .   ]                   %
            %       [   .       .       .         .   ]                   %
            %       [ FeatX   FeatX   FeatX     FeatX ]                   %
            %        ---------------------------------                    %
            % --> Information                                              %
            str = sprintf([' TESTING (Feature Extraction) \t\t ' ...       %
                'Class:  %s  \t\t %s'],char(classes{clss}),message);       %
            set(handles.info,'string',str);drawnow; % display(str);        %
            %_____________________________________________________________%
        end
        % ----------> class i
        % DataStruct
        % eval(['test_features.' classes{clss} '= feats_test;']);
        % DataMatrix
        feats_test=features';
        [m_audios , n_feats] = size(feats_test);
        feats_test(:,n_feats+1)=(clss-1)*ones(m_audios,1);
        TESTDATA=[TESTDATA; feats_test];
        %_____________________________________________________________%
    end
    %% ------------------------------------------------ NORMALIZATION ---- %
    load('./tmp/normalization.mat');
    TESTDATA(:,1:end-1)=bsxfun(@minus,TESTDATA(:,1:end-1),norm.feats_mean);
    TESTDATA(:,1:end-1)=bsxfun(@rdivide,TESTDATA(:,1:end-1),norm.feats_var);
    % Save DATASET
    save([temp_path 'TESTDATA.mat'], 'TESTDATA');
    else
        % Load DATASET
        if exist([temp_path 'TESTDATA.mat'],'file')
            load([temp_path 'TESTDATA.mat']);
        else
            erstr='Error:    There is no Testing Data to Load. ';
            disp(erstr);set(handles.info,'string',erstr);drawnow;
            testing=0;
        end
    end
    %_____________________________________________________________%
    %% ------------------------------------------- FEATURE SELECTION -- %
    if ~isempty(TESTDATA)
        [N,M]=size(TESTDATA);

        feats=find(config.feature_selection.on_off==1);
        if debug;represent_dataset(TESTDATA,'TEST DATASET');end
        if max(feats)<M && isempty(find(isnan(TESTDATA(:,feats)),1))
            data=TESTDATA(:,feats);             %- Data to Test
            test_label_data=TESTDATA(:,M)'+1;   %- Class Labels
        else
            erstr='Error:  Features Selected are not in the Testing Data.';
            disp(erstr);set(handles.info,'string',erstr);drawnow;
            testing=0;
        end
    end % (FrontEnd.)
    %_____________________________________________________________%
    %% _____________________________ BACK-END _____________________________ %
    if testing
        if config.RF_classifier
            name='RandomForest_Classifier';
        elseif config.SVM_classifier
            name='SVM_Classifier';
        elseif config.kNN_classifier
            name='kNN_Classifier';
        end
```

```
    if exist([temp_path 'trained_classifier.mat'],'file')
        % Load Classifier
        load([temp_path 'trained_classifier.mat']);
        classifier=trained_classifier.classifier;
        %% ------------------------------------ CONFUSION MATRIX ----- %
        % Test Data Should be untrained data
        if test_confusionmtx
            str='Calculating Confusion Matrix.....................';
            set(handles.info,'string',str);drawnow; % disp(str);
            set(handles.cnfmtx_mode1,'Value',1);
            [confusion_matrix, message]=confmtx(classifier,data,...
                                    test_label_data,classes,handles);
            set(handles.info,'string',message);drawnow;

            % Save Confusion Matrix_____________________________________
            last_cnfmtx.confusion_matrix=confusion_matrix;                %
            last_cnfmtx.classes=classes;                                  %
            save('./tmp/confusion_matrix.mat','last_cnfmtx');             %
            %_______________________________________________________________%
            % Generate Report___________________________________________
            filename='/results';
            outpath=[results_path filename];
            % Show GUI: 1-YES, 0-NO
            showGUI=config.report;
            report_GUI(config, confusion_matrix, outpath, showGUI);
        end
        %% ---------------------------------- AUDIO SAMPLE PREDICTION -- %
        if test_sample
            sample=round(N*rand(1,1));
            tic % Time of Prediction
            % Prediction
            [Yfit,scores] = predict(classifier,data(sample,:));
            if config.SVM_classifier
                scores=(1+scores)/sum(1+scores);
            end
            toc
            classes=strrep(classes,'_',' ');
            if Yfit==test_label_data(sample)
                result='¡OK!';
            else
                result='¡INCORRECT!';
            end
            Yfit_class=classes(Yfit);
            correct_anw=classes(test_label_data(sample));
            str = {['         True Class: ' char(correct_anw)]; ...
            ['Predicted Class: ' char(Yfit_class) '           ' result ]};
            % Representation ..............................................%
            axes(handles.axes2);
            bar(scores);grid on
            xlabel(handles.axes2,'Classes');
            ylabel(handles.axes2,'Probability');
            title(handles.axes2,str,'HorizontalAlignment','left');
            handles.axes2.XAxis.FontSize=8;
            set(handles.axes2,'XtickLabel',classes,...
                              'XtickLabelRotation',45);
            ylim([0 1]);drawnow
        end
        %_______________________________________________________________%

    else
        erstr='Error:    The Classifier Does Not Exist. ';
        disp(erstr);set(handles.info,'string',erstr);drawnow;
    end
    %___________________________________________________________________%
    end % (BackEnd.)
end %.....................................................................%
%_____________________________END-TESTING_____________________________%
end
```

Feature Extraction

```
function [ features ] = feature_extraction( audio_path ,config)
%% FEATURE EXTRACTION ------------------------------------------------- %
% Función que realiza el cálculo de todas las features.                 %
% Su función es leer el audio de su path correspondiente, realizar un   %
% control de canal seleccionando aquel con mayor energía y posteriormente %
% un cambio en la frecuencia de muestreo si es necesario (a 16000 Hz)   %
% para que toda la extracción se procese a la misma frecuencia de       %
% muestreo. También se elimina la componente continua de la señal.      %
%                                                                       %
%                                                                       %
%   Input:                                                              %
%       - audio_path:    audio signal path                             %
%       - config:        configuration struct                          %
%   Output:                                                             %
%       - features (vector)                                            %
%               * spectral_features                                     %
%               * temporal_features                                     %
%                                                                       %
%.......................................................................%
%                                                                       %
%                                                  Fabián Aguirre Martín  %
%                                                                 2017  %
%_______________________________________________________________________%

feature_selection=config.feature_selection;
% Lectura del audio
% disp(audio_path);
[x,fs]=audioread(audio_path); % Read audio

% Selección de canal con mayor energia
[~,n]=size(x);
if n > 1
    Energy=sum(x.^2,1);
    % MaxCh=find(Energy==max(Energy));
    x=x(:,Energy==max(Energy));
    x=x(:,1);
end

if fs ~= 16000
        Tx=(0:length(x)-1)./fs;
        fs=16000;   % New fs
        x=resample(x,Tx,fs);
end
%  % Sound Check
%  sound(x,fs);
%  pause;

%% --------------------------- DC OFFSET ------------------------------- %
% *Optional
% - DC removal (Elimina la comoponente continua de la señal)
b=[1 -1];
a=[1 -0.9];
x=filter(b,a,x);
%_______________________________________________________________________%

%% ----------------------- FEATURE EXTRACTION --------------------------- %
features=zeros(length(feature_selection.on_off),1);
%% ---- SPECTRAL FEATURES ------------------------------------------------ %
[spectral_features]=spectral_feats(x,fs);
```

```
                % -> spectral_features.mfcc:
                %
                %   Frame1  Frame2  Frame3    Framei
                % [ MFCC1   MFCC1   MFCC1 ... MFCC1 ]
                % [ MFCC2   MFCC2   MFCC2 ... MFCC2 ]
                % [   .       .       .         .   ]
                % [   .       .       .         .   ]
                % [   .       .       .         .   ]
                % [ MFCCn   MFCCn   MFCCn ... MFCCn ]
                % .
                % .
                % .
                % -> spectral_features.centroid:
                %
                %    Frame1     Frame2     Frame3       Framei
                % [ CENTROID   CENTROID   CENTROID ... CENTROID ]
                % .
                % .
                % .
                % -> spectral_features.rolloff
                %
                %    Frame1     Frame2     Frame3       Framei
                % [ ROLL-OFF   ROLL-OFF   ROLL-OFF ... ROLL-OFF ]
% Mean Values
features(1:40)=mean(spectral_features.mfcc,2);
features(41:80)=mean(spectral_features.dmfcc,2);
features(81:120)=mean(spectral_features.ddmfcc,2);
features(121)=mean(spectral_features.centroid,2);
features(122)=mean(spectral_features.spread,2);
features(123)=mean(spectral_features.skewness,2);
features(124)=mean(spectral_features.kurtosis,2);
features(125)=mean(spectral_features.asc,2);
features(126)=mean(spectral_features.ass,2);
features(127)=mean(spectral_features.asf,2);
features(128)=mean(spectral_features.rolloff,2);
features(129)=mean(spectral_features.slope,2);
features(130)=mean(spectral_features.decrease,2);
features(131)=mean(spectral_features.flux,2);
features(132)=var(spectral_features.flux);

%% ---- TEMPORAL FEATURES ------------------------------------------------ %
temporal_features=temporal_feats(x,fs);
features(133)=temporal_features.zcr;
features(134)=temporal_features.logattack;
features(135)=temporal_features.pitch_vector;
features(136)=temporal_features.AM;

                %   Audio
                % [ MFCC1  ]
                % [ MFCC2  ]
                % [   .    ]
                % [   .    ]
                % [   .    ]
                % [ MFCCn  ]
                % [   .    ]
                % [   .    ]
                % [   .    ]
                % [ROLL-OFF]
                % [   .    ]
                % [FLUX var]
                % [  ZCR   ]
                % [   .    ]
                % [   .    ]
                % [   AM   ]
end
```

Features Espectrales y Cepstrales

```
function [ spectral_features ] = spectral_feats(x,fs)
%% Spectral Features Extraction ------------------------------------------ %
% Función que realiza la extracción de características                     %
% (Feature Extraction) de todas aquellas características o "features"      %
% cuyo procedimiento de extracción se realice a través de la FFT,          %
% es decir, en el dominio de la frecuencia.                                %
%                                                                          %
%   Input:                                                                 %
%       - x:    audio signal                                               %
%       - fs:   sample rate                                                %
%   Output:                                                                %
%       - spectral features                                                %
%               * MFCC + D-MFCC + DD-MFCC                                  %
%               * CENTROID + SPREAD + SKEWNESS + KURTOSIS                  %
%               * ROLL-OFF                                                 %
%               * SLOPE                                                    %
%               * DECREASE                                                 %
%               * AUDIO SPECTRUM CENTROID + SPREAD + FLATNESS              %
%.........................................................................%
%                                                                          %
% - Process:                                        Variables              %
%       - DC Offset Removal                                                %
%       - Pre-emphasis                                                     %
%       - Frame Blocking                                                   %
%               * Window size:                      W                      %
%               * Overlap:                          percent                %
%       - Windowing                                                        %
%               * Window (hamming, hanning...):     win                    %
%       - DFT                                                              %
%               * FFT size:                         nfft                   %
%                                                                          %
%       %% MFCC Extraction                                                 %
%       - Mel frequency Filter-Bank                                        %
%               * Mel Filters:                      M                      %
%       - DCT                                                              %
%               * DCT Coefficients:                 C                      %
%       - D + DD                                                           %
%                                                                          %
%       %% CENTROID/SPREAD/SKEWNESS/KURTOSIS                               %
%       %% ROLL-OFF                                                        %
%       %% SLOPE                                                           %
%       %% DECREASE                                                        %
%       %% AUDIO SPECTRUM CENTROID/SPREAD/FLATNESS                         %
%       %% SPECTRAL FLUX                                                   %
%                                                                          %
%                                                  Fabián Aguirre Martín   %
%_________________________________________________________________________%
%% -------------------- Configuration Parameters ------------------------ %
L=length(x);        % - Audio length     (Longitud Audio)
W=round(fs*60e-3);  % - Window size      (Tamaño de ventana)
win=hamming(W);     % - Window           (Ventana)
percent=50;         % - Overlapping      (Solapamiento)
nfft=1024;          % - FFT size         (Tamaño FFT)
dB_blwmax=15;       % - dB Energy Control(dB to MAX)   default: 10dB
% - MFCC Parameters
M=35;               % - Mel Filters      (Número de Filtros MEL)
C=40;               % - DCT Coefficients (Número de Coeficientes de la DCT)
%_________________________________________________________________________%
if length(x)<W; x(W)=0; end
if length(x)<2*W; x=[x; x; x]; L=3*W; end
%% ------------------------- PRE-EMPHASIS ------------------------------- %
% *Optional
% Emphasize the high-frequency (Realza la alta frecuencia)
h=[1 -0.9]; % FIR filer
x=filter(h,1,x);
%_________________________________________________________________________%
```

```
%% ------------------------ FRAME BLOCKING ------------------------------ %
ovlap=percent*W/100;     % - Overlapping samples         (Muestras de Solape)
m=1:ovlap:L-W;           % - Displacement                (Desplazamiento)
N=length(m);             % - Number of windows (Frames) (Número de ventanas)
% - Block division vector   (Vector de división en bloques)
ind_vector=(0:W-1)'*ones(1,N) + ones(W,1)*m;
% - Frame blocked signal    (Señal dividida en bloques)
x=x(ind_vector);
%_____________________________________________________________________________%

%% ---------------------- Frame Energy Control ------------------------- %
energy=sum(abs(x),1); % Envelope
framesW = (energy >= max(energy)./(10.^(dB_blwmax/10)) & energy~=0);
x=x(:,framesW);  % x [W,N]
N=sum(framesW);  % N logic frames: 1
%_____________________________________________________________________________%

%% -------------------------- WINDOWING ---------------------------------- %
% Windowing (Enventanado)
xwin=bsxfun(@times,x,win);
%_____________________________________________________________________________%

%% --------------------- Spectral Magnitude DFT ------------------------- %
X=fft(xwin,nfft);        % FFT
X=abs(X(1:nfft/2,:));    % Keep only absolute half FFT
X2=X.^2;                 % Energy
%_____________________________________________________________________________%

%% CEPSTRAL: MFCCs + D-MFCCs + DD-MFCCs -------------------------------- %
[mfcc ,dmfcc, ddmfcc] = mfcc_extraction( X2, nfft, fs , M, C , N );
%_____________________________________________________________________________%
%% CENTROID / SPREAD / SKEWNESS / KURTOSIS ----------------------------- %
[centroid, spread, skewness, kurtosis] = shape_extraction(X, nfft, fs, N);
%_____________________________________________________________________________%
%% ROLL-OFF  ----------------------------------------------------------- %
rolloff = rolloff_extraction(X2, nfft, fs, N);
%_____________________________________________________________________________%
%% SPECTRAL SLOPE  ----------------------------------------------------- %
slope = slope_extraction(X, nfft, fs);
%_____________________________________________________________________________%
%% SPECTRAL DECREASE  -------------------------------------------------- %
decrease = decrease_extraction(X, nfft,N);
%_____________________________________________________________________________%
%% AUDIO SPECTRUM CENTROID/SPREAD/FLATNESS ----------------------------- %
[asc , ass, asf] = mpeg7_extraction(X2, nfft, fs, N);
%_____________________________________________________________________________%
%% SPECTRAL FLUX ------------------------------------------------------- %
flux = flux_extraction(X, nfft, fs);
%_____________________________________________________________________________%
% MFCC + D + DD
spectral_features.mfcc=mfcc;
spectral_features.dmfcc=dmfcc;
spectral_features.ddmfcc=ddmfcc;
% SHAPE Descriptors
spectral_features.centroid=centroid;
spectral_features.spread=spread;
spectral_features.skewness=skewness;
spectral_features.kurtosis=kurtosis;
spectral_features.rolloff=rolloff;
spectral_features.slope=slope;
spectral_features.decrease=decrease;
% MPEG-7
spectral_features.asc=asc;
spectral_features.ass=ass;
spectral_features.asf=asf;
% FLUX
spectral_features.flux=flux;
end
```

Features Temporales

```
function [ temporal_features ] = temporal_feats(s,fs)
%% Temporal Features Extraction  ---------------------------------------- %
% Función que realiza la extracción de características                    %
% (Feature Extraction) de todas aquellas características o "features"     %
% cuyo procedimiento de extracción se realice en el dominio del tiempo.   %
%                                                                         %
%   Input:                                                                %
%       - x:    audio signal                                              %
%       - fs:   sample rate                                               %
%   Output:                                                               %
%       - temporal_features                                               %
%               * ZCR                                                     %
%               * LOG-ATTACK                                              %
%               * PITCH                                                   %
%               * m Index (AM)                                            %
%.........................................................................%
%                                                                         %
%                                                 Fabián Aguirre Martín   %
%_________________________________________________________________________%
%% -------------------- Configuration Parameters ----------------------- %
L=length(s);                % - Audio length        (Longitud Audio)
frame_time=25e-3;           % - Default window size time 20 ms
W=round(fs*frame_time);     % - Window size         (Tamaño de ventana)
% m Index:
mIndex_time=1;              % - Window size time Envelope Analysis (1 s)
%_________________________________________________________________________%
%% ------------------------ FRAME BLOCKING ------------------------------ %
% Non-Overlapping Framing
N=ceil(L/W);        % N frames
s(N*W)=0;
L=length(s);
x=reshape(s,W,N);   % Framed Signal
%_________________________________________________________________________%
%% --------------------------- ENVELOPE ----------------------------------- %
% Envelope 1: Frame Energies
x_shape=sum(x.^2,1);                        % Frame Energy
% x_shape=sum(x.^2.*(frame_time/W),1);      % Frame Energy
% t_shape=0:frame_time:(N-1)*frame_time;    % Time vector
% x_rms=rms(x_shape);                       % RMS value
%.........................................................................
% Envelope 2: Exponential Filter
t=1/fs:1/fs:length(s)/fs;
a=0.01;
s_env=zeros(1,L-1);s_env(1)=s(1);
for n=2:L
    s_env(n)=a*abs(s(n))+(1-a)*s_env(n-1);
end
% Filter envelope ripple
load('Filter.mat');
s_env=filter(Hd,s_env);
s_rms=rms(s_env);
%_________________________________________________________________________%
%--------------------------- EXTRACTION ------------------------------- %
%% ------------------------------------------------- ZCR  ------------ %
temporal_features.zcr=zcr_extraction(x,fs,W);
%_________________________________________________________________________%
%% ------------------------------------------- Log-Attack --------- %
temporal_features.logattack = attack_extraction(s_env, s_rms, t);
%_________________________________________________________________________%
%% ------------------------------------------------ Pitch ------------ %
temporal_features.pitch_vector = pitch_extraction( x, fs, W ,N);
%_________________________________________________________________________%
%% ------------------------------------------- m AM_Index --------- %
temporal_features.AM=AM_extraction(x_shape,fs,W,mIndex_time);
%_________________________________________________________________________%
end
```

Feature: Mel Frequency Cepstral Coefficients

```
function [ mfcc, dmfcc, ddmfcc] = mfcc_extraction( X2, nfft, fs , M, C, N  )
%% Mel Frequency Cepstral Coefficients extraction ----------------------- %
%                                                                          %
%   Input:                                                                 %
%       - X2: Spectrum X.^2                                                %
%       - nfft: FFT size                                                   %
%       - fs: sample rate                                                  %
%       - M: Mel Filters                                                   %
%       - C: DCT Coefficients                                              %
%       - N: number of frames                                              %
%                                                                          %
%   Output:                                                                %
%       - MFCC                                                             %
%       - D-MFCC                                                           %
%       - DD-MFCC                                                          %
%                                                                          %
% - Process:                                                               %
% %       - Pre-emphasis                (done)                             %
% %       - DC Offset Removal           (done)                             %
% %       - Frame Blocking              (done)                             %
% %       - Windowing                   (done)                             %
% %       - DFT                         (done)                             %
%       - Mel frequency Filter-Bank                                        %
%       - DCT                                                              %
%                                                                          %
%..........................................................................%
%                                                                          %
%                                              Fabián Aguirre Martín       %
%__________________________________________________________________________%

% ------------------------ Energy -> MFCC0 ------------------------------ %
%
% - MFCC0 is almost the energy of the signal
energy=10*log10(sum(X2,1));
%__________________________________________________________________________%

%% -------------------- MEL FREQUENCY FILTER-BANK ------------------------%
%
% Take or create the Mel Filter-Bank for a specific number of filters
if ~exist(['./FrontEnd/FeatureExtraction_Functions/Filters/MelFB_' ...
        num2str(M) 'x' num2str(nfft/2) '.mat'],'file');
    mel=create_MelFB(fs, nfft, M);
    save(['./FrontEnd/FeatureExtraction_Functions/Filters/MelFB_' ...
        num2str(M) 'x' num2str(nfft/2)], 'mel');
else
    load(['MelFB_' num2str(M) 'x' num2str(nfft/2)]) % loads variable: mel
end

% Mel Filtering
X2=log(mel*X2);
%..........................................................................%
%    MEL FREQUENCY FILTER-BANK          FFT
% [ 1_1 1_2 1_3 1_4 1_5 ... 1_Nfft]    [ 1   ] = Mx128 * 128x1 = Mx1
% [ 2_1 2_2 2_3 2_4 2_5 ... 2_Nfft]    [ 2   ]      _______________
% [ 3_1 3_2 3_3 3_4 3_5 ... 3_Nfft]    [ 3   ]     |               |
% [             .                 ] * [ 4   ]     |  40xFRAMES    |
% [             .                 ]   [ 5   ]     |_______________|
% [             .                 ]   [ .   ]
% [ M_1 M_2 M_3 M_4 M_5 ... M_Nfft]   [ .   ]
%                                     [ .   ]
%                                     [Nfft]
%__________________________________________________________________________%
```

```
%% ------------------------- DCT -> MFCCs ------------------------------ %
%
% Take or create the DCT matrix for a specific number of DCT coefficients
% and Mel frequency filters
%
if ~exist(['./FrontEnd/FeatureExtraction_Functions/Filters/DCT'  ...
        num2str(C) 'x' num2str(M) '.mat'],'file');
    dct=create_DCT(C,M);
    save(['./FrontEnd/FeatureExtraction_Functions/Filters/DCT' ...
        num2str(C) 'x' num2str(M)], 'dct');
else
    load(['DCT' num2str(C) 'x' num2str(M)]) % loads variable: DCT (matrix)
end
%
% - - - - - - - - - -> MFCC extraction final step <- - - - - - - - - - - -
%         (CEPSTRAL COEFFICIENTS) -> HOMOMORPHIC DOMAIN
mfcc=dct*X2;
mfcc(1,:)=energy;
%
%.........................................................................%
%   DCT1 DCT2 DCT3             DCT_N   MEL_FFT
% [ 1_1  1_2  1_3  1_4  1_5 ... 1_M]   [ 1  ] = CxM * Mx1 = Mx1
% [ 2_1  2_2  2_3  2_4  2_5 ... 2_M]   [ 2  ]      _____________
% [ 3_1  3_2  3_3  3_4  3_5 ... 3_M]   [ 3  ]     |             |
% [               .               ] * [ 4  ]     | CxFRAMES(N) |
% [               .               ]   [ 5  ]     |_____________|
% [               .               ]   [ .  ]
% [ C_1  C_2  C_3  C_4  C_5 ... C_M]   [ .  ]
%                                      [ .  ]
%                                      [Nfft]
%_________________________________________________________________________%


%% ------------------------- Delta MFCC ------------------------------- %
t1=2:N;
t2=1:N-1;
dmfcc=mfcc(:,t1)-mfcc(:,t2);
%_____________________________________________________________________%

%% ------------------------- 2Delta MFCC ------------------------------ %
ddmfcc=dmfcc(:,t1(1:end-1))-dmfcc(:,t2(1:end-1));
%_____________________________________________________________________%
end


function mel_matrix  = create_MelFB(fs, nfft, M )
%% MEL Filter-Bank ------------------------------------------------------%
% Función que crea el banco de filtros MEL para determinados valores de:  %
%                                                                         %
%   Input:                                                                %
%       - fs: sample rate                                                 %
%       - Nfft: FFT size                                                  %
%       - M: Number of Mel filters                                        %
%                                                                         %
%   Output:                                                               %
%       - mel_matrix: filter matrix                                       %
%                                                                         %
%       MEL FREQUENCY FILTER-BANK           Triangular Filters            %
% [ 1_1  1_2  1_3  1_4  1_5  ...  1_Nfft]       Filter 1                  %
% [ 2_1  2_2  2_3  2_4  2_5  ...  2_Nfft]       Filter 2                  %
% [ 3_1  3_2  3_3  3_4  3_5  ...  3_Nfft]       Filter 3                  %
% [             .                       ]         .                     %
% [             .                       ]         .                     %
% [             .                       ]         .                     %
% [ 40_1 40_2 40_3 40_4 40_5 ... 40_Nfft]       Filter 40                 %
%                                                                         %
% Example:                                                                %
%                                                                         %
```

```
%       MEL FREQUENCY FILTER-BANK            FFT                          %
% [ 1_1  1_2  1_3  1_4  1_5  ...  1_Nfft]   [ 1  ] = 40x128 * 128x1 =40x1 %
% [ 2_1  2_2  2_3  2_4  2_5  ...  2_Nfft]   [ 2  ]      _________          %
% [ 3_1  3_2  3_3  3_4  3_5  ...  3_Nfft]   [ 3  ]     |         |         %
% [             .                       ] * [ 4  ]     |  40x1   |         %
% [             .                       ]   [ 5  ]     |_________|         %
% [             .                       ]   [ .  ]                        %
% [ 40_1 40_2 40_3 40_4 40_5 ... 40_Nfft]   [ .  ]                        %
%                                           [ .  ]                        %
%                                           [Nfft]                        %
%                                                                         %
%.........................................................................%
%                                                                         %
%                                              Fabián Aguirre Martín      %
%_________________________________________________________________________%
% fs=16000;       % Sample rate
% Nfft=1024;      % FFT size
% M=34;           % Number of Filters
debug=1;

minMel=102.2;   % Minimum Mel Central Frequency
% Mel frequency vector
mel_f=minMel:minMel:M*minMel;
mel_f=[ minMel/2 mel_f (M+1)*minMel];

% flow=0;
% mlow=2595*log10(1+flow/700);
% fhigh=fs/2;
% mhigh=2595*log10(1+fhigh/700);
% mel_f=linspace(mlow, mhigh, M+2);
% Mel to Hz
f=700.*(10.^(mel_f./2595)-1);
% Frequency bins (k)
f_bin=floor(f.*((nfft/2+1)/fs));
% Intialization of the Mel Filter Matrix
mel_matrix=zeros(M,nfft/2);
for m=2:M+1
    for k=1:nfft/2   % k: bin
        % Initialization -> if k<mel_f(m-1); mel_matrix(m-1,k) =0; end
        if k < f_bin(m-1)
            mel_matrix(m-1,k)=0;
        elseif (k >= f_bin(m-1) ) && ( k <= f_bin(m))
            mel_matrix(m-1,k)=(k-f_bin(m-1))/(f_bin(m) - f_bin(m-1));
        elseif (f_bin(m) <= k ) && ( k <=f_bin(m+1))
            mel_matrix(m-1,k)=(f_bin(m+1)-k)/(f_bin(m+1)-f_bin(m));
        elseif k > f_bin(m+1)
            mel_matrix(m-1,k)=0;
        end

        % Initialization -> if k > mel_f(m+1); mel_matrix(m-1,k) =0; end
    end
end
mel_matrix((isnan(mel_matrix)))=0;
if debug
    figure;
    f_ax=(0:1/nfft:0.5-1/nfft)*fs;
    plot(f_ax,mel_matrix);
    title('Mel-Frequency Filter Bank');
    xlabel('Frequency (Hz)');ylabel('Weight');
    mel_matrix=[mel_matrix; zeros(1,nfft/2)];
    mel_matrix=[mel_matrix, zeros(M+1,1)];
    figure;pcolor(mel_matrix);shading flat;
    title('Mel-Frequency Filter Bank');
    xlabel('k (FFT bin)'); ylabel('M Filter');
end

% save(['MelFB' num2str(nfft/2)], 'mel_matrix');
end
```

```
function DCT_matrix = create_DCT(C,M)
%% DCT ------------------------------------------------------------------------%
% Function that creates a DCT matrix with C DCT coefficientes and M            %
% columns.                                                                     %
%                                                                              %
%                     DCT1 DCT2 DCT3              DCT_N                        %
%                   [ 1_1  1_2  1_3  1_4  1_5 ... 1_M]  -> MEL_FFT           %
%                   [ 2_1  2_2  2_3  2_4  2_5 ... 2_M]                       %
%                   [ 3_1  3_2  3_3  3_4  3_5 ... 3_M]                       %
%                   [                .                ]                      %
%                   [                .                ]                      %
%                   [                .                ]                      %
%                   [ C_1  C_2  C_3  C_4  C_5 ... C_M]                       %
%                                                                            %
%.............................................................................%
%    DCT1 DCT2 DCT3              DCT_N   MEL_FFT                             %
% [ 1_1  1_2  1_3  1_4  1_5 ... 1_M]    [ 1  ] = CxM * Mx1 = Mx1              %
% [ 2_1  2_2  2_3  2_4  2_5 ... 2_M]    [ 2  ]       _____________            %
% [ 3_1  3_2  3_3  3_4  3_5 ... 3_M]    [ 3  ]      |             |           %
% [                .                ] * [ 4  ]      | CxFRAMES(N) |           %
% [                .                ]    [ 5  ]      |_____________|           %
% [                .                ]    [ .  ]                             %
% [ C_1  C_2  C_3  C_4  C_5 ... C_M]    [ .  ]                             %
%                                       [ .  ]                             %
%                                       [Nfft]                             %
%                                                                            %
%.............................................................................%
%                                                                            %
%                                                  Fabián Aguirre Martín     %
%____________________________________________________________________________%
debug=1;

% C=13; % DCT Coefficients
% M=20; % Mel Filters

m=0:M-1;
for k=0:C-1
    DCT_matrix(k+1,:)=cos((pi*(k*(m+1/2)/M)));
end

if debug
    figure;
    plot(1:M,DCT_matrix);
    xlim([1 Inf]);
    grid on;
    title('DCT - Discrete Cosine Transform');
    xlabel('M Filters');
    ylabel('DCT Coefficients');
end


end
```

Features: Spectral Centroid, Spread, Skewness, Kurtosis

```
function [centroid, spread, skewness, kurtosis]=shape_extraction(X,nfft, fs,N)
%% SHAPE DESCRIPTORS EXTRACTION ----------------------------------------- %
% Función que realiza la extracción de las principales features de forma  %
% espectral:                                                              %
%       - Centroid  (Centro de gravedad del espectro)                     %
%       - Spread    (Expansión de la energía respecto al centroide)       %
%       - Skewness  (Asimetría)                                           %
%       - Kurtosis  (Espectro plano o picudo)                             %
%                                                                         %
%   Input:                                                                %
%       - X:    framed signal                                             %
%       - nfft: FFT size                                                  %
%       - fs:   sample rate                                               %
%       - N:    N frames                                                  %
%                                                                         %
%   Output:                                                               %
%       - Centroid                                                        %
%       - Spread                                                          %
%       - Skewness                                                        %
%       - Kurtosis                                                        %
%.........................................................................%
%                                                                         %
%                                                  Fabián Aguirre Martín  %
%_________________________________________________________________________%
if length(X(:,1))>nfft/2; X=X(1:nfft/2,:); end
% Frequency vector
f=(0:nfft/2-1).*fs/nfft;
%% --------------------------- CENTROID ---------------------------------- %
num=f*X;
%                           Frame1  Frame2     FrameN
% f*X =[f0 f1 f2 ... f_n]*[X_k0    X_k0  ...  X_k0] =
%                          [X_k1    X_k1  ...  X_k1]
%                          [ .       .          .  ]
%                          [X_kn    X_kn  ...  X_kn]
%
%             Frame1  Frame2  Frame3  Frame4       FrameN
%      f*X = [ fX1     fX2     fX3     fX4    ...    fXN ]
%
denom=sum(X,1);
%             Frame1  Frame2  Frame3  Frame4       FrameN
%         X = [X_k0    X_k0    X_k0    X_k0  ...    X_k0 ]
%             [X_k1    X_k1    X_k1    X_k1  ...    X_k1 ]
%             [ .       .       .       .            .   ]
%             [X_kn    X_kn    X_kn    X_kn         X_kn ]
%
%             Frame1  Frame2  Frame3  Frame4       FrameN
%     sum_X = [X_1     X_2     X_3     X_4    ...    X_N  ]
%
centroid=num./denom;            % Mean
%_________________________________________________________________________%
f_nor=bsxfun(@minus,f'*ones(1,N),centroid);
%% --------------------------- SPREAD ------------------------------------ %
num=sum((f_nor.^2).*X,1);
spread=num./denom;              % Variance
%_________________________________________________________________________%
%% --------------------------- SKEWNESS ---------------------------------- %
num=sum((f_nor.^3).*X,1);
m3=num./denom;
skewness=m3./(spread.^(3/2));   % Asymmetry
%_________________________________________________________________________%
%% --------------------------- KURTOSIS ---------------------------------- %
num=sum((f_nor.^4).*X,1);
m4=num./denom;
kurtosis=m4./(spread.^2);       % Peakedness/Flatness
%_________________________________________________________________________%
end
```

Feature: Spectral Roll-Off

```
function rolloff = rolloff_extraction(X, nfft, fs, N)
%% ROLL-OFF Extraction ------------------------------------------------ %
% Función que determina el roll-off espectral de cada frame al 85%, es  %
% decir, la frecuencia a la cual se obtiene el 85% de la energía en la  %
% función de distribución acumulada.                                    %
%                                                                       %
%   Input:                                                              %
%       - X:    framed signal                                           %
%       - nfft: FFT size                                                %
%       - fs:   sample rate                                             %
%       - N:    N frames                                                %
%                                                                       %
%   Output:                                                             %
%       - rolloff                                                       %
%                                                                       %
%.......................................................................%
%                                                                       %
%                                                 Fabián Aguirre Martín %
%_______________________________________________________________________%
debug=0;

p=0.85;

if length(X(:,1))>nfft/2; X=X(1:nfft/2,:); end

Px_tot=sum(X,1);
for i=1:N
    Px=0;
    k=0;
    while Px < p*Px_tot(1,i)
        k=k+1;
        Px=Px+X(k,i);
    end
    rolloff(1,i)=k./(nfft/2);
end

if debug
    frame=1;
    fdp=X(:,frame)/sum(X(:,frame));
    fda(1)=0;
    for i=1:nfft/2
        fda(i+1)=fda(i)+fdp(i);
    end

    figure;
    f=(0:nfft/2-1).*(fs/nfft);
    subplot(2,1,1)
    semilogx(f,10.*log10(fdp));
    hold on;
    semilogx([rolloff(1,frame)*fs rolloff(1,frame)*fs],[10 -70]);
    axis([100 8000 -70 10])
    grid on;
    title('Spectrum (Roll-Off 0.85) ');
    xlabel('Frequency (Hz)');
    ylabel('Magnitude (dB)')
    subplot(2,1,2)
    semilogx(f,fda(2:end));
    hold on;
    semilogx([rolloff(1,frame)*fs rolloff(1,frame)*fs],[1 0]);
    axis([100 8000 0 1])
    grid on;
    title('Acumulative Distribution (Roll-Off 0.85) ');
    xlabel('Frequency (Hz)');
    ylabel('Probability')
end
end
```

Feature: Spectral Slope

```
function slope = slope_extraction(X, nfft, fs)
%% Slope Extraction ------------------------------------------------------ %
% Función que determina la pendiente (slope) del espectro.                 %
%                                                                          %
%   Input:                                                                 %
%       - X:    framed signal                                              %
%       - nfft: FFT size                                                   %
%       - fs:   sample rate                                                %
%                                                                          %
%   Output:                                                                %
%       - slope                                                            %
%                                                                          %
%..........................................................................%
%                                                                          %
%                                                 Fabián Aguirre Martín    %
%__________________________________________________________________________%
debug=0;
if length(X(:,1))>nfft/2; X=X(1:nfft/2,:); end

f=(0:nfft/2-1).*fs/nfft; % f = [f0 f1 f2 f3 f4 ................. f_nfft/2]
denom=0.5*nfft.*sum(f.^2)-(sum(f)).^2; % scalar
sum_X=sum(X,1);
%
%              Frame1  Frame2  Frame3  Frame4      FrameN
%         X = [a_k0    a_k0    a_k0    a_k0  ...   a_k0 ]
%             [a_k1    a_k1    a_k1    a_k1  ...   a_k1 ]
%             [a_k2    a_k2    a_k2    a_k2  ...   a_k2 ]
%             [ .       .       .       .           .   ]
%             [ .       .       .       .           .   ]
%             [ .       .       .       .           .   ]
%             [a_kn    a_kn    a_kn    a_kn        a_kn ]
%
%              Frame1  Frame2  Frame3  Frame4      FrameN
%     sum_X = [a_k     a_k     a_k     a_k   ...   a_k  ]
%__________________________________________________________________________%
num=0.5*nfft.*(f*X)-sum(f).*sum_X;
%
%                       Frame1  Frame2     FrameN
% f*X =[f0 f1 ... f_n]*[a_k0    a_k0  ...  a_k0] = [ fak   fak ... fak]
%                      [a_k1    a_k1  ...  a_k1]
%                      [a_k2    a_k2  ...  a_k2]
%                      [ .       .          .  ]
%                      [ .       .          .  ]
%                      [ .       .          .  ]
%                      [a_kn    a_kn  ...  a_kn]
%
%              Frame1  Frame2  Frame3  Frame4      FrameN
%      f*X = [ fak     fak     fak     fak   ...   fak ]
%__________________________________________________________________________%
% True Slope:       slope=num/denom;
% Normalized Slope
slope=(1./sum_X).*num/denom;

if debug
    slope=num/denom; % True slope
    [~, N]=size(X);
    frame=round(N/2);
    figure;
    plot(f,X(:,frame));grid on
    hold on
    a_f=slope(frame).*f+rms(X(:,frame));
    plot(f,a_f);
    xlabel('Frequency (Hz)');ylabel('Magnitude');title('Spectral Slope');
end
end
```

Feature: Spectral Decrease

```
function decrease = decrease_extraction(X, nfft, N)
%% Decrease Extraction ------------------------------------------------ %
% Función que determina la pendiente de caída del espectro.             %
%                                                                       %
%   Input:                                                              %
%       - X:    framed signal                                           %
%       - nfft: FFT size                                                %
%       - N:    N frames                                                %
%                                                                       %
%   Output:                                                             %
%       - decrease                                                      %
%                                                                       %
%.......................................................................%
%                                                                       %
%                                              Fabián Aguirre Martín    %
%_______________________________________________________________________%

if length(X(:,1))>nfft/2
    X=X(1:nfft/2,:);
end

sum_X=sum(X(2:end,:),1);
%
%              Frame1  Frame2  Frame3  Frame4       FrameN
%         X = [a_k0    a_k0    a_k0    a_k0  ...    a_k0 ]
%          -> [a_k1    a_k1    a_k1    a_k1  ...    a_k1 ]
%           . [a_k2    a_k2    a_k2    a_k2  ...    a_k2 ]
%           . [ .       .       .       .            .   ]
%           . [ .       .       .       .            .   ]
%           . [ .       .       .       .            .   ]
%          -> [a_kn    a_kn    a_kn    a_kn         a_kn ]
%
%              Frame1  Frame2  Frame3  Frame4       FrameN
%     sum_X = [a_k     a_k     a_k     a_k    ...    a_k  ]
%_______________________________________________________________________%

acum=zeros(1,N);
for k=2:nfft/2
    acum=acum+(X(k,:)-X(1,:))./(k-1);
    %
    %  Frame1  Frame2         FrameN    Frame1  Frame2         FrameN
    %  [a_kn    a_kn    ...    a_kn ] - [a_k0    a_k0    ...    a_k0 ]
    %  ____________________________________________________________
    %                        (k - 1)
    %
end

% Spectral Decrease
decrease=(1./sum_X).*acum;

end
```

Features: MPEG-7 (AS-Centroid, AS-Spread. AS-Flatness)

```
function [ASC, ASS, ASF] = mpeg7_extraction(X, nfft, fs, N)
%% MPEG-7 Features Extraction ---------------------------------------- %
% Función que obtiene las features MPEG-7:                            %
%       - Centroid                                                    %
%       - Flatness                                                    %
%       - Spread                                                      %
%                                                                     %
%   Input:                                                            %
%       - X:    framed signal  (Power Spectrum)                       %
%       - nfft: FFT size                                              %
%       - fs:   sample rate                                           %
%       - N:    N frames                                              %
%                                                                     %
%   Output:                                                           %
%       - ASC:  Audio Spectrum Centroid                               %
%       - ASS:  Audio Spectrum Spread                                 %
%       - ASF:  Audio Spectrum Flatness                               %
%                                                                     %
%.....................................................................%
%                                                                     %
%                                                 Fabián Aguirre Martín   %
%_____________________________________________________________________%

if length(X(:,1))>nfft/2
    X=X(1:nfft/2,:);
end

f=(0:nfft/2-1).*fs/nfft;
df=fs/nfft;
Klow=floor(62.5/df);
k=1:Klow;
X=[sum(X(k,:),1); X(Klow+1:end,:)];
f=[31.25, f(Klow+1:end)];

% X=bsxfun(@rdivide,X,max(X,1));
maximum=max(X,[],1);
acum=zeros(1,N);
for i=1:nfft/2-Klow+1
    %% Audio Spectrum Flatness
    flat_spectr=maximum-X(i,:);
    %% Audio Spectrum Centroid
    weights=log(f(i)/1000).*X(i,:);
    acum=acum+weights;
end
denom=sum(X,1);
ASC=acum./denom;

ASF=2*sum(flat_spectr,1)./nfft;
%% Audio Spectrum Spread
f=f'*ones(1,N);
acum=zeros(1,N);
for i=1:nfft/2-Klow+1
    weights=((log(f(i,:)/1000)-ASC).^2).*X(i,:);
    acum=acum+weights;
end
denom=sum(X,1);
ASS=acum./denom;

end
```

Feature: Spectral Flux

```
function flux = flux_extraction(X, nfft, fs)
%% Slope Extraction ------------------------------------------------------ %
% Función que determina la variación (flux) temporal del espectro.         %
%                                                                          %
%   Input:                                                                 %
%       - X:    framed signal                                              %
%       - nfft: FFT size                                                   %
%       - fs:   sample rate                                                %
%                                                                          %
%   Output:                                                                %
%       - flux                                                            %
%                                                                          %
%.........................................................................%
%                                                                          %
%                                                 Fabián Aguirre Martín    %
%_________________________________________________________________________%
debug=0;

[M,N]=size(X);
if M>nfft/2; X=X(1:nfft/2,:); end

%-------------------------------------------------------------------------%
t=2:N;
t_1=1:N-1;
num = sum(X(:,t_1).*X(:,t),1);
%              Frame1  Frame2  Frame3  Frame4      FrameN-1
%    X(t-1) = [a_k0    a_k0    a_k0    a_k0  ...   a_k0 ]
%             [a_k1    a_k1    a_k1    a_k1  ...   a_k1 ]
%             [a_k2    a_k2    a_k2    a_k2  ...   a_k2 ]
%             [ .       .       .       .           .   ]
%             [ .       .       .       .           .   ]
%             [ .       .       .       .           .   ]
%             [a_kn    a_kn    a_kn    a_kn        a_kn ]
%
%                                x(.*)
%
%              Frame2  Frame3  Frame4  Frame5      FrameN
%      X(t) = [a_k0    a_k0    a_k0    a_k0  ...   a_k0 ]
%             [a_k1    a_k1    a_k1    a_k1  ...   a_k1 ]
%             [a_k2    a_k2    a_k2    a_k2  ...   a_k2 ]
%             [ .       .       .       .           .   ]
%             [ .       .       .       .           .   ]
%             [ .       .       .       .           .   ]
%             [a_kn    a_kn    a_kn    a_kn        a_kn ]
%.............________________________________________.................%
%                           + (sum(,1))
%
%            Frame_1-2 Frame_2-3 Frame_3-4 Frame_4-5   Frame_N-(N-1)
%      num = [a_k       a_k       a_k       a_k   ...   a_k  ]
%.........................................................................%
% Normalization (0-1)
denom = sqrt(sum(X(:,t_1).^2,1).*sum(X(:,t).^2,1));
% Spectral Flux (Succsesive Spectrums)
% ~0 -> Similar
% ~1 -> Dissimilar
flux = 1 - num./denom;

if debug
    figure;
    plot(2:N,flux);grid on
    xlabel('Frames');
    title('Spectral Flux');
    ylabel('Flux Value');
    ylim([0 1]);
end
end
```

Feature: Zero Crossing Rate

```
function ZCR=zcr_extraction(x,fs,W)
%% ZERO CROSSING RATE extraction ---------------------------------------- %
% Función que determina el índice de cruces de por cero.                  %
%                                                                         %
%   - Input:                                                              %
%       - x:  signal                                                      %
%       - fs: sample rate                                                 %
%       - W:  window size                                                 %
%                                                                         %
%   - Output:                                                             %
%       - ZCR: Zero-Crossing Rate                                         %
%.........................................................................%
%                                                                         %
%                                                  Fabián Aguirre Martín  %
%_________________________________________________________________________%
z=sign(x(2:end,:).*x(1:end-1,:));
ZCR=mean(sum(z==-1,1))/(W/fs);
end
```

Feature: m Index (AM)

```
function m_index=AM_extraction(x_envelope,fs,W,env_time)
%% AM_extraction -------------------------------------------------------------%
% Función que determina el índice de modulación de amplitud de la           %
% envolvente de la señal.                                                   %
%                                                                           %
%   - Input:                                                                %
%       - x_envelope: signal envelope                                       %
%       - fs:         sample rate                                           %
%       - W:          envelope window size                                  %
%       - env_time:   envelope window time                                  %
%                                                                           %
%   - Output:                                                               %
%       - m_index: Modulation Index (AM)                                    %
%                                                                           %
%...........................................................................%
%                                                                           %
%                                                  Fabián Aguirre Martín    %
%___________________________________________________________________________%
debug=0;
%% Envelope Framing (Default Analysis Time: 1 s)
new_W=round((fs/W)*env_time);
N=ceil(length(x_envelope)/new_W);           % N frames (def. N seconds)
x_envelope(N*new_W)=0;
x=reshape(x_envelope,new_W,N);              % re-Framed Signal
%% Modulation Index
maximum=max(x,[],1);
meanvalue=mean(x,1);
m_index=mean(100*(maximum-meanvalue)/maximum);

%% Representation (DEBUG)
if debug
    t=0:W/fs:length(x_envelope)*W/fs-W/fs;
    t2=t;  t2(N*new_W)=t(end); t2=reshape(t2,new_W,N);
    figure;
    plot(t,x_envelope);grid on
    for i=1:N
    hold on;
    plot(t2(:,i),maximum(i)*ones(1,new_W),'r--',...
         t2(:,i),meanvalue(i)*ones(1,new_W),'b--');
    end
    title(['Envelope - Modulation Index: ' num2str(m_index)]);
    xlabel('Time (s)');
    ylabel('Amplitude');
end
end
```

Feature: Log-attack time

```
function logattack = attack_extraction(x_shape, x_rms, t)
%% Log-Attack Extraction ------------------------------------------------ %
% Función que determina el tiempo de ataque medio de la señal.            %
%                                                                         %
%   Input:                                                                %
%       - x_shape:    framed signal (envelope)                            %
%       - x_rms:      RMS value                                           %
%       - t:          time vector                                         %
%                                                                         %
%   Output:                                                               %
%       - logattack                                                       %
%                                                                         %
%.........................................................................%
%                                                                         %
%                                                 Fabián Aguirre Martín   %
%_________________________________________________________________________%
debug=0;
N=length(x_shape);
start_attack=25;
% Initialization Thresholds
th_start=start_attack/100*x_rms;
th_end=max(x_shape);

logic_frames=x_shape>=x_rms ;
t1=2:N;
t2=1:N-1;
d_shape=[0 , (x_shape(t1) - x_shape(t2))];
logic_d=[0 , (logic_frames(t1) - logic_frames(t2))];
starts=find(logic_d==1);

% Initialization
attack=[];n=0;atck_pos=zeros(length(t),1);
for i=1:length(starts)
    j=starts(i);
    z=j;
    while j<N && d_shape(j)>=0 ; j=j+1 ; end    % Buscar final del ataque
    while z>1 && d_shape(z)>=0 ; z=z-1 ; end    % Buscar inicio del ataque

    % Controlar que cada "nota" o pico tenga al menos 6 dB de diferencia
    if x_shape(j-1)/x_shape(z+1) > 2  % 20log10(2) = 6 dB
        n=n+1;
        % k=j-z-2; attack(i)=t(k+1);
        attack(n)=t(j-z-1);
        atck_pos(z+1:j-1)=1;
    end
end
if isempty(attack)
    logattack=0;
else
    logattack=log(mean(attack));    % Media de los ataques encontrados
end

% Debug Mode ----------------------------------------------------------- %
if debug
    figure;
    plot(t,x_shape);grid on
    hold on
    plot(t,x_rms*ones(length(t),1),t,th_start*ones(length(t),1),'--r' , t,
th_end*ones(length(t),1),'--r' );
    hold on
    plot(t,max(x_shape)*logic_frames,'y',t,max(x_shape).*atck_pos,'black');
    hold off
    xlabel('Time (s)');ylabel('Amplitude');title('Signal Envelope');
end
%-------------------------------------------------------------------------%
end
```

Feature: Pitch

```
function [ pitches ] = pitch_extraction( x, fs, W ,N)
%% PITCH extraction ------------------------------------------------------%
% Función que determina el pitch (si existe) de una señal                 %
%                                                                         %
%   - Input:                                                              %
%       - x:  signal                                                      %
%       - fs: sample rate                                                 %
%       - W:  envelope window size                                        %
%       - N:  N frames                                                    %
%                                                                         %
%   - Output:                                                             %
%       - pitch: mean pitch                                               %
%                                                                         %
%.........................................................................%
%                                                                         %
%                                                 Fabián Aguirre Martín   %
%_________________________________________________________________________%

%% Autocorrelación ------------------------------------------------------ %
% - Signal: x blocks 60 ms
s_cor=zeros(2*W-1,N);
for i=1:N
    s_cor(:,i)=xcorr(x(:,i));
end
s_cor=s_cor(ceil((2*W-1)/2:end),:);
s_cor=bsxfun(@minus, s_cor, mean(s_cor,1));
rmscor=rms(s_cor);
t_cor=0:1/fs:W/fs-1/fs;
% Derivada
t1=2:W;
t2=1:W-1;
d_cor=[s_cor(t1,:)-s_cor(t2,:); zeros(1,N)];
dd_cor=[ zeros(1,N); d_cor(t1(1:end-1),:)-d_cor(t2(1:end-1),:);zeros(1,N)];
% Detectar derivada con primer paso por cero con segunda derivada negativa
zero_x= [ s_cor(t2,:).*s_cor(t1,:);zeros(1,N)];     % Zero Crossings
zero_dcor=[ zeros(1,N);d_cor(t2,:).*d_cor(t1,:) ];  % Maximos (D ZeroX)

for frame_i=1:N
    if debug && frame_i==frame
        disp('');
    end
    ZX=find(zero_x(:,frame_i)<0);       % Posiciones de los cruces por 0
    % Detectar si es una trama sonora ó una trama sorda
    if length(ZX)>8 && any(s_cor(ZX(1):end,frame_i) > 0.25*s_cor(1,frame_i))
        %% ---------------------- TRAMA SONORA -------------------------- %
        % Posiciones de los máximos (D=0)
        dzero_cross=zero_dcor(:,frame_i)<0;
        % Posición del pico máximo
        offset=ZX(1);
        maximum_pos=offset-
1+find(s_cor(offset:end,frame_i)==max(s_cor(offset:end,frame_i)));
        max_positions=[]; p=0.6;
        while length(max_positions)<3
            m=1;
            threshold=p*s_cor(maximum_pos(1),frame_i);
            for i=1:length(ZX)-1
                getpeak=ZX(i)-1+find(dzero_cross(ZX(i):ZX(i+1))==1 ...
                    & dd_cor(ZX(i):ZX(i+1),frame_i) < 0 ...
                    & s_cor(ZX(i):ZX(i+1),frame_i) > threshold);
                if ~isempty(getpeak)
                    [~, reindx]=sort(s_cor(getpeak,frame_i),'descend');
                    max_positions(m)=getpeak(reindx(1));m=m+1;
                end
            end
            p=p-0.05;
        end
```

```
        % Determinar el PITCH a través de los máximos ------------------- %
        % Diferencia Primeros máximos
        pitch_dif1=1/abs(t_cor(max_positions(1))-t_cor(max_positions(2)));

        % Maximas posiciones ordenadas por amplitud
        [~, reindx]=sort(s_cor(max_positions),'descend');
        max_positions=max_positions(reindx);

        % Diferencia Mayores máximos
        pitch_dif2=1/abs(t_cor(max_positions(1))-t_cor(max_positions(2)));

        pitches=1./t_cor(max_positions);
        confianza=abs(pitch_dif2-pitches)/pitch_dif2;
        pitch_def=pitches(confianza==min(confianza));
        if isempty(pitch_def)
            pitch_vector(frame_i,1)=pitch_dif1;
            pitch_vector(frame_i,2)=pitch_dif2;
        else
            pitch_vector(frame_i)=mean(pitch_def);
        end
        pitch_vector(frame_i,1)=pitch_dif1;
        pitch_vector(frame_i,2)=pitch_dif2;


    else    % TRAMA SORDA ------------------------------------------------ %
        pitch_vector(frame_i,1)=0;
        pitch_vector(frame_i,2)=0;
        if debug && frame_i==frame
            okmax_rep=1;
            ZC_rep=1;
            lim=0;
            frame=round((N-frame_i-1)*rand(1,1))+1+frame_i;
        end
    end
end
pitch_vector=mean(pitch_vector,2);
pitch_vector(pitch_vector==Inf)=0;
pitches=mean(pitch_vector(pitch_vector~=0));
if isempty(pitches) || isnan(pitches)
    pitches=0;
end
%_____________________________________________________________________________%

end
```

ANEXO III: RESULTADOS ADICIONALES

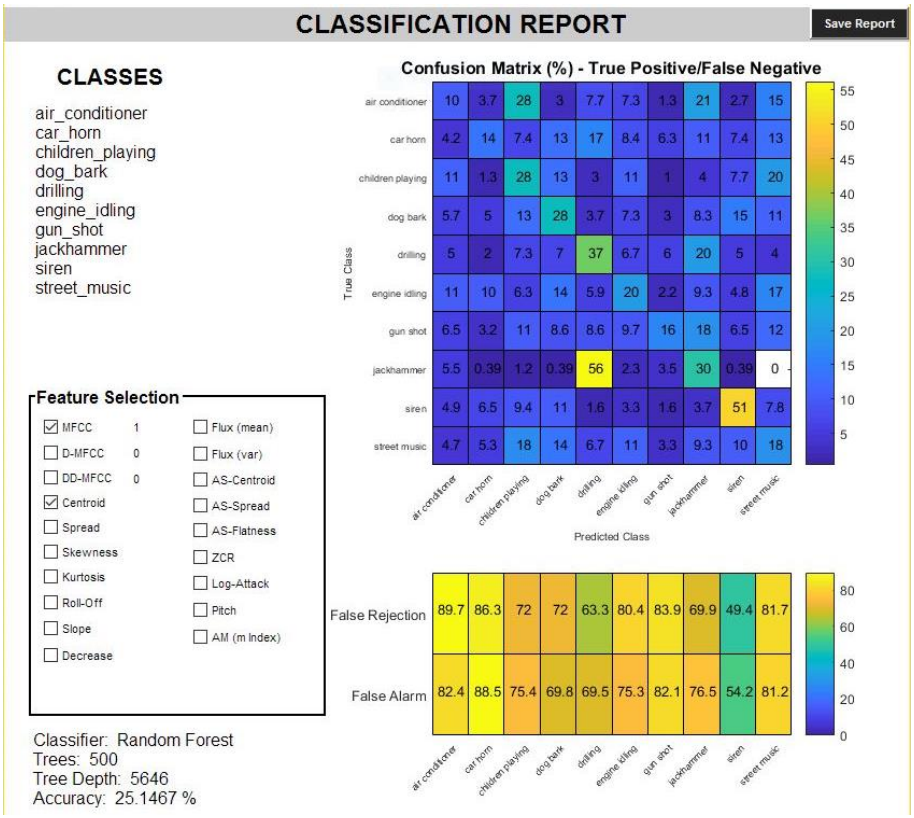

Fig. 70. *Centroid* y MFCC1

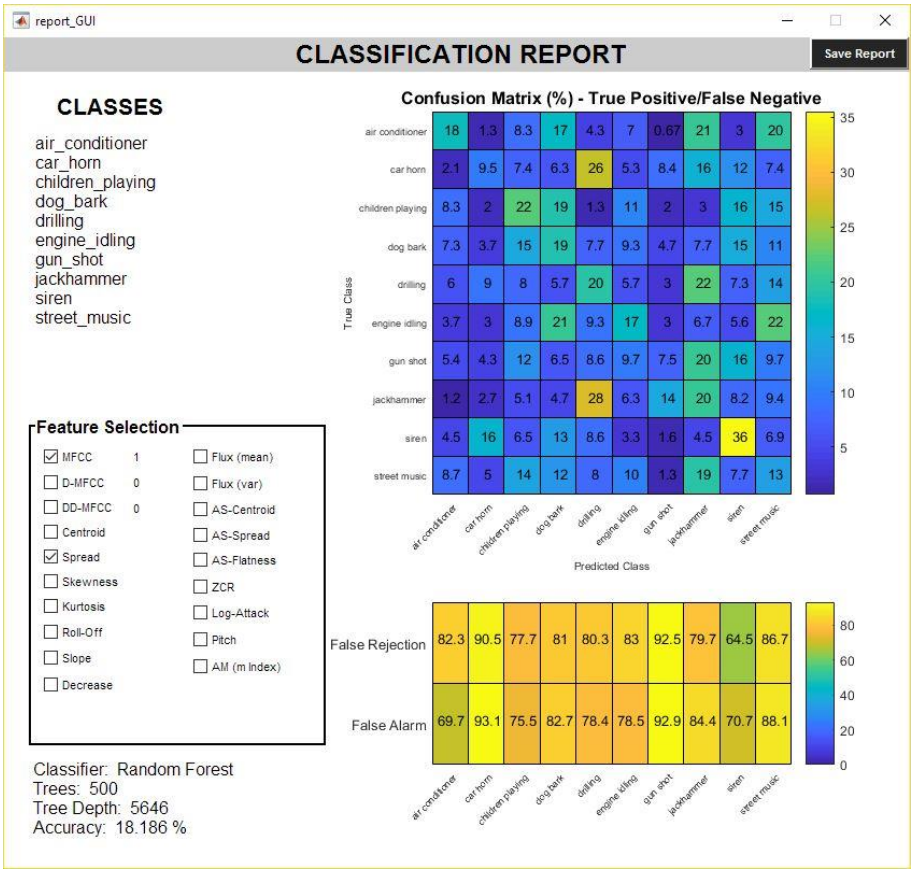

Fig. 71. *Spread* y MFCC1

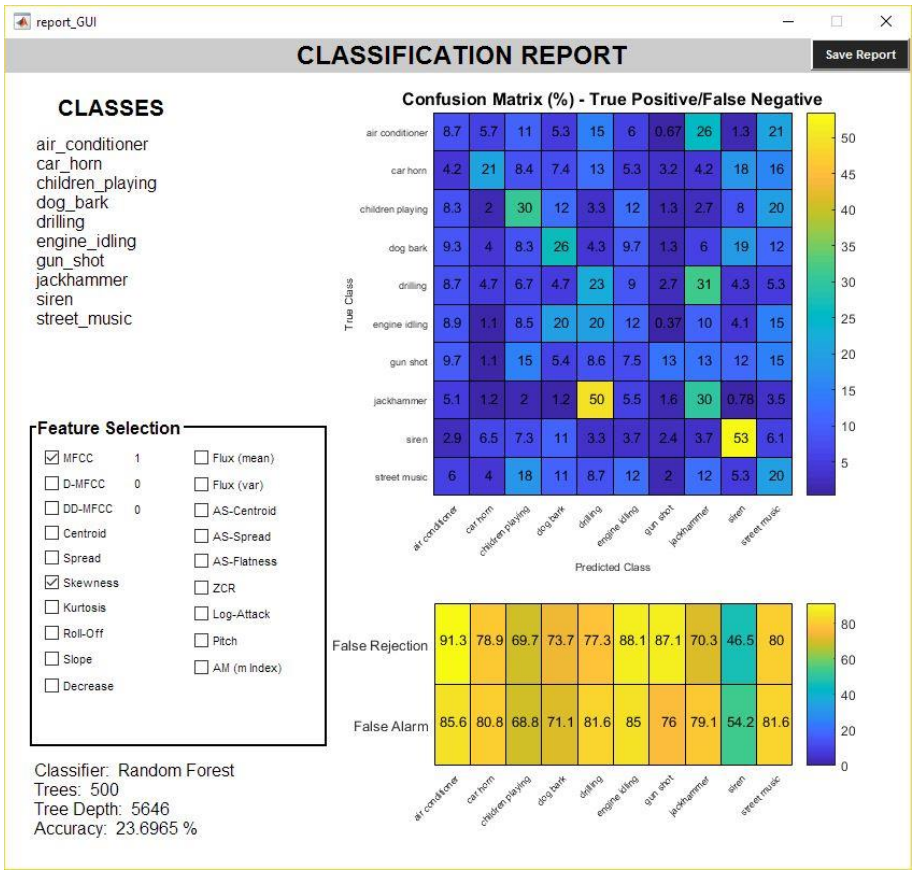

Fig. 72. *Skewness* y MFCC1

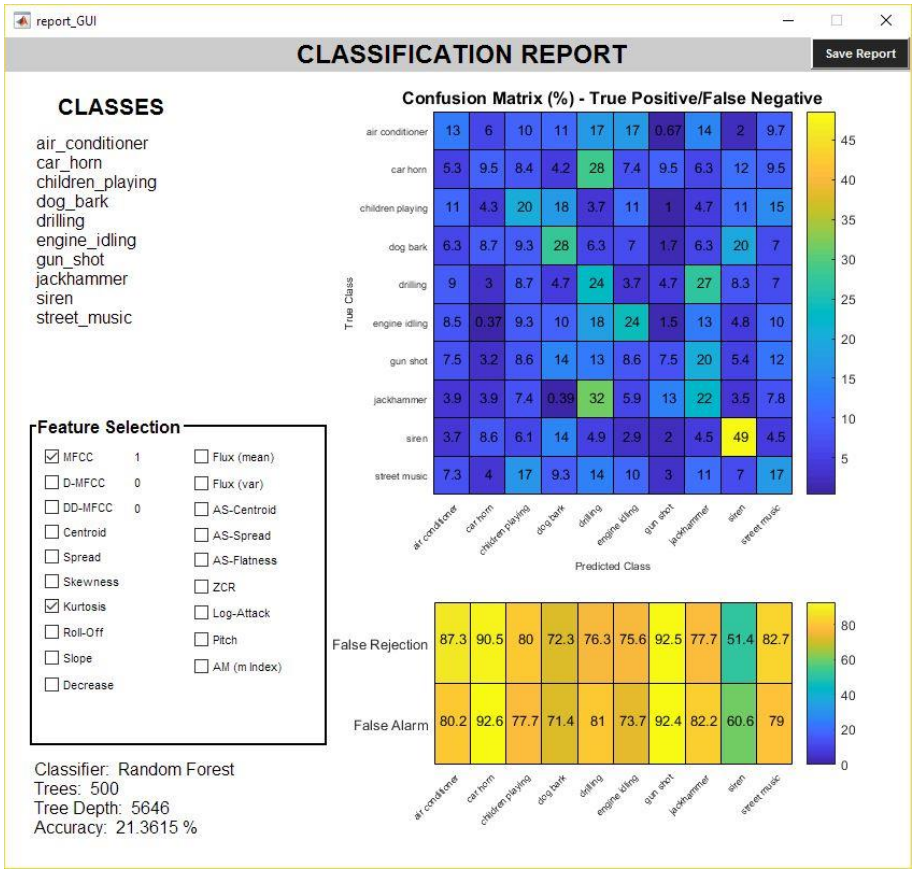

Fig. 73. *Kurtosis* y MFCC1

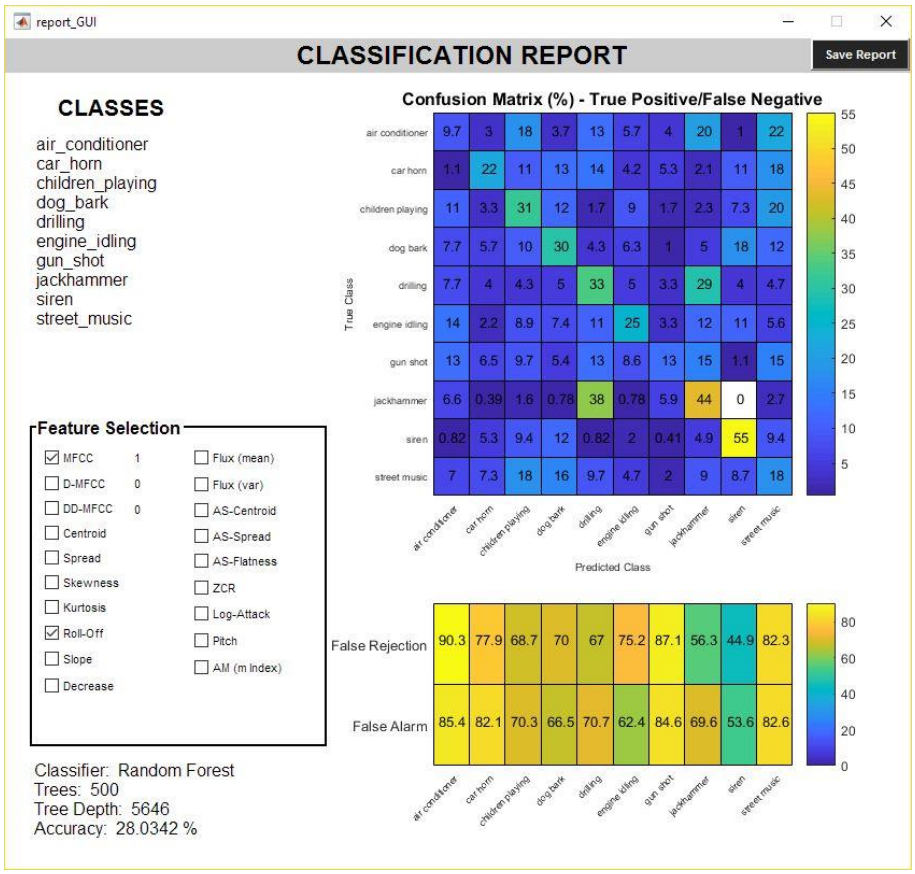

Fig. 74. *Roll-off* y MFCC1

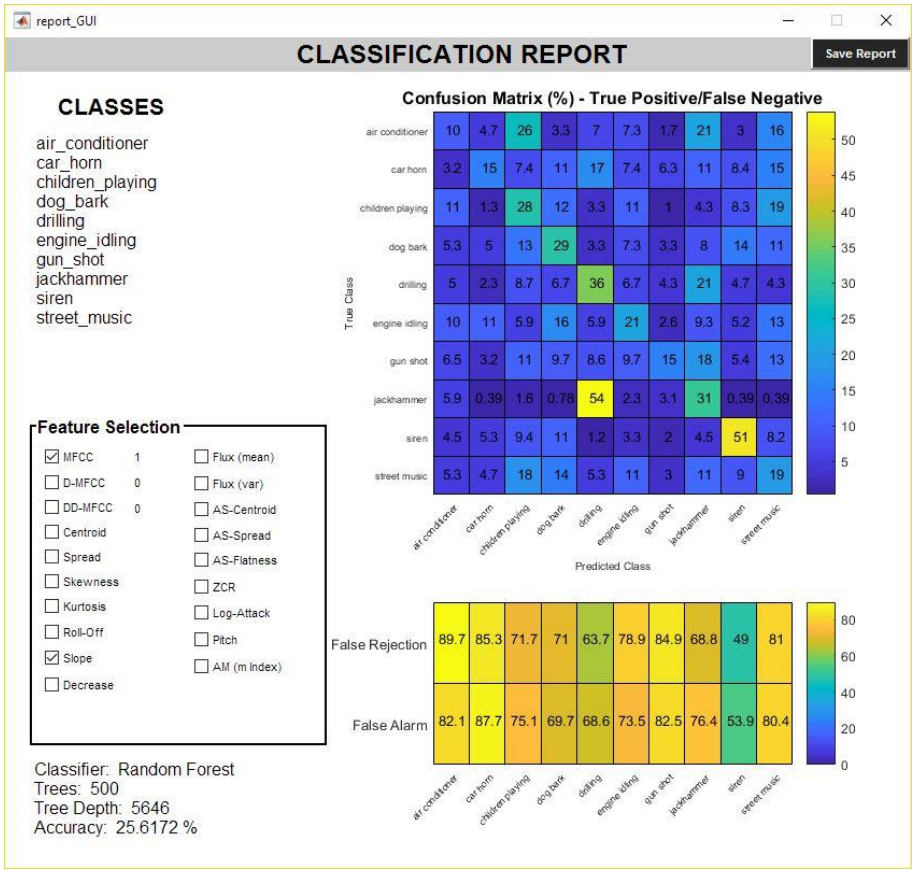

Fig. 75. *Slope* y MFCC1

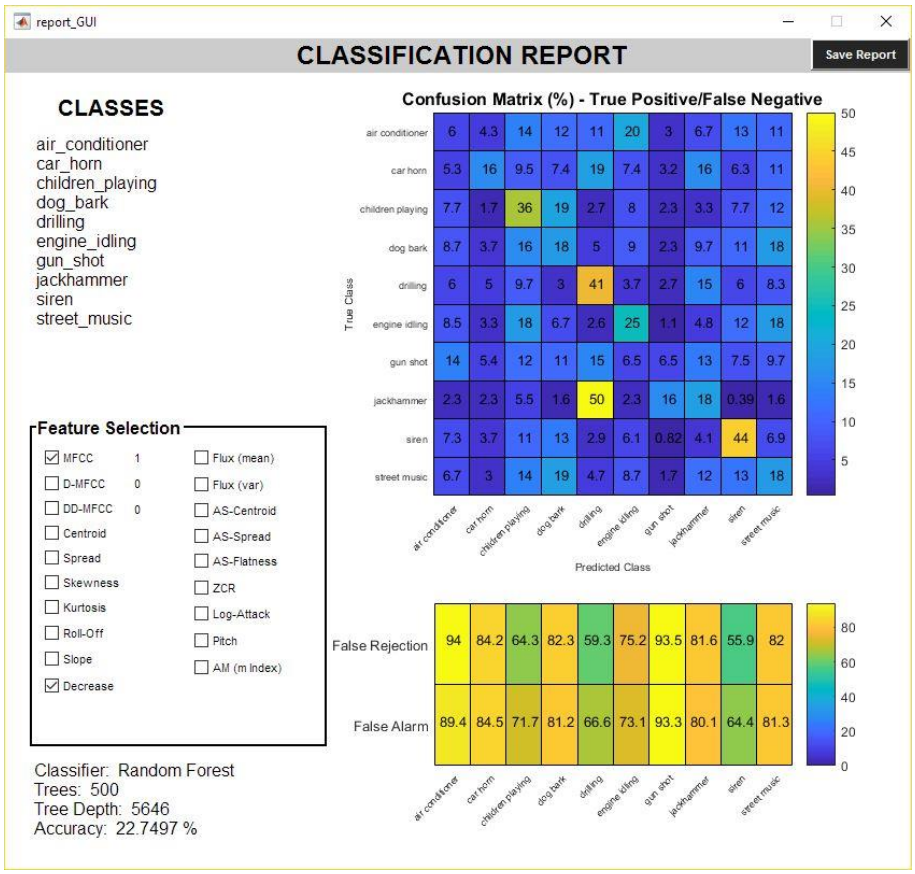

Fig. 76. *Decrease* y MFCC1

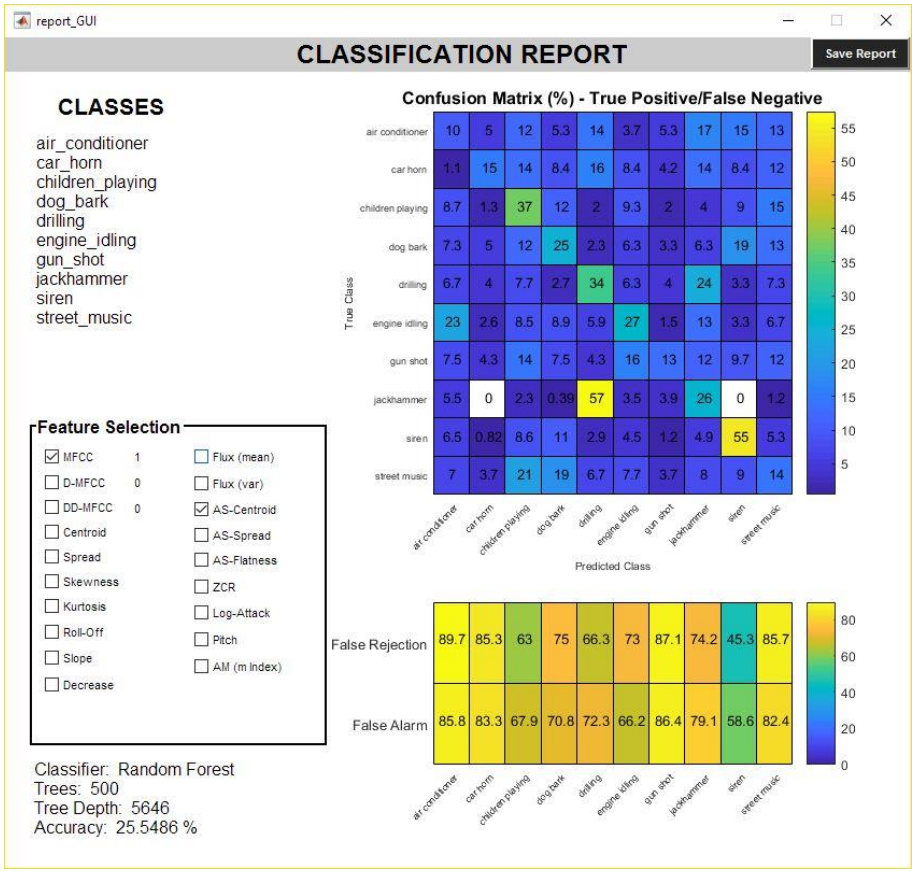

Fig. 77. *AS-Centroid* (MPEG7) y MFCC1

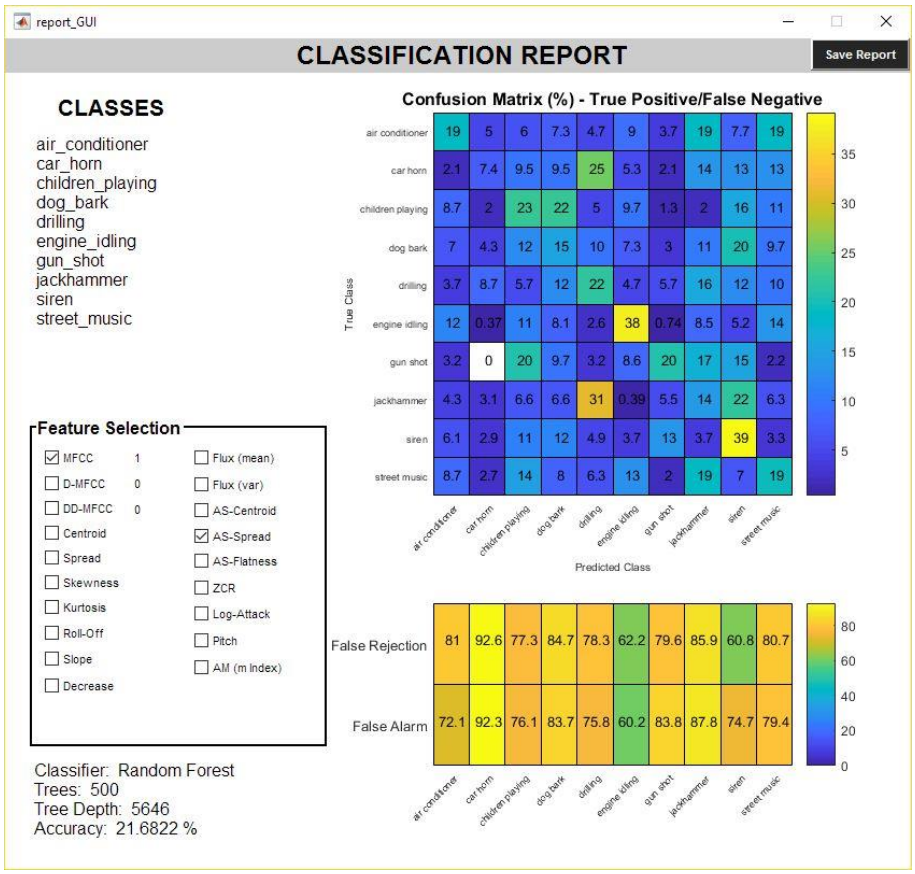

Fig. 78. *AS-Spread* (MPEG7) y MFCC1

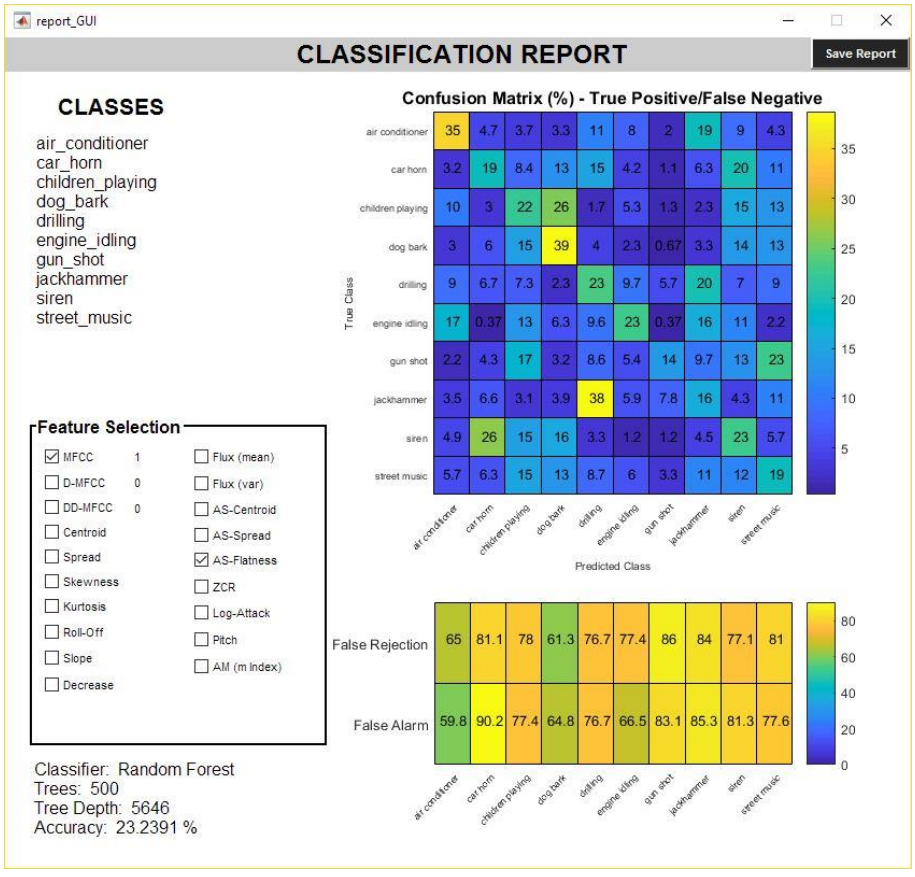

Fig. 79. *AS-Flatness* (MPEG7) y MFCC1

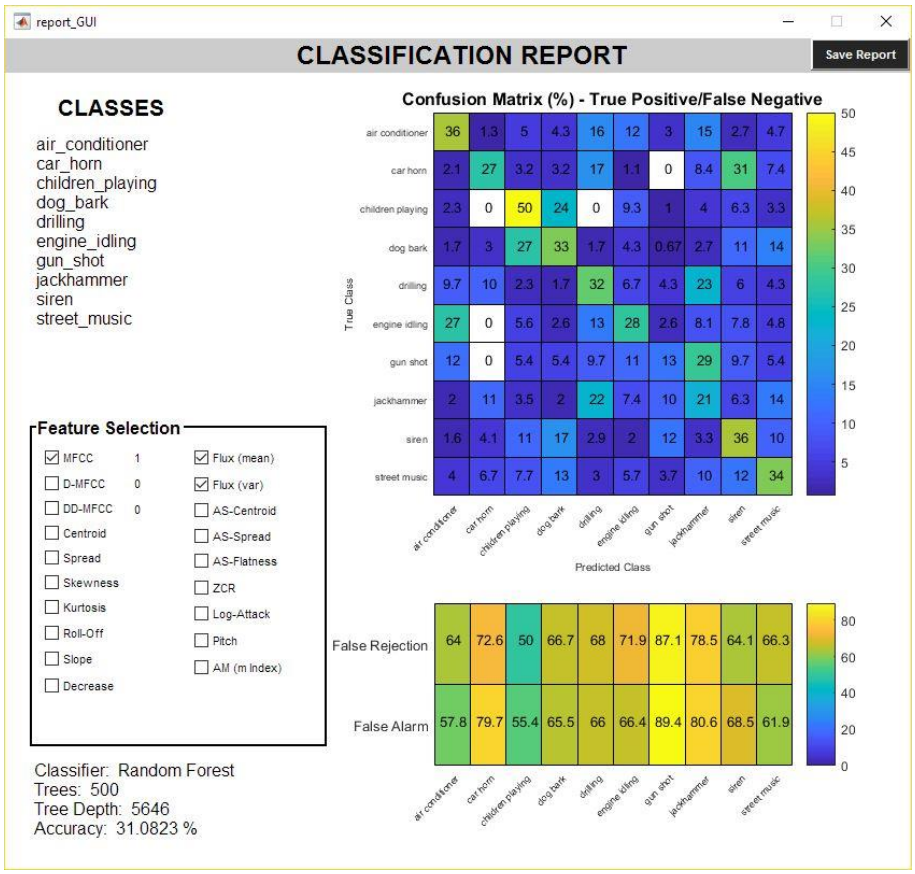

Fig. 80. *Spectral Flux* y MFCC1

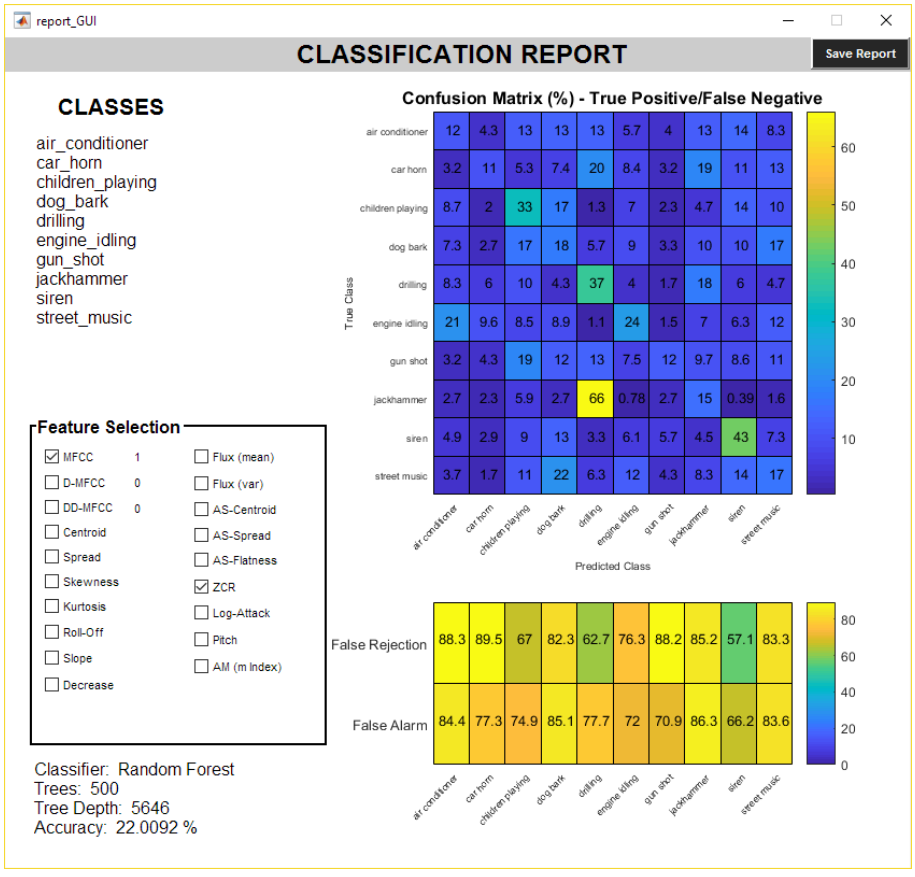

Fig. 81. ZCR y MFCC1

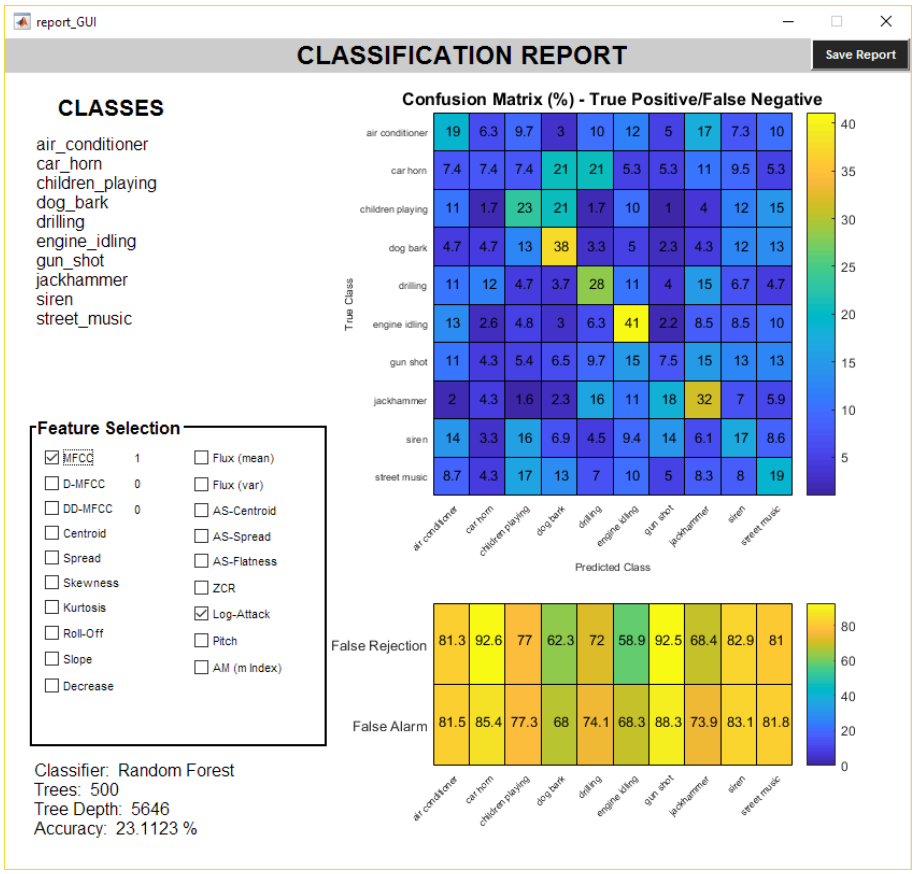

Fig. 82. *Log-Attack time* y MFCC1

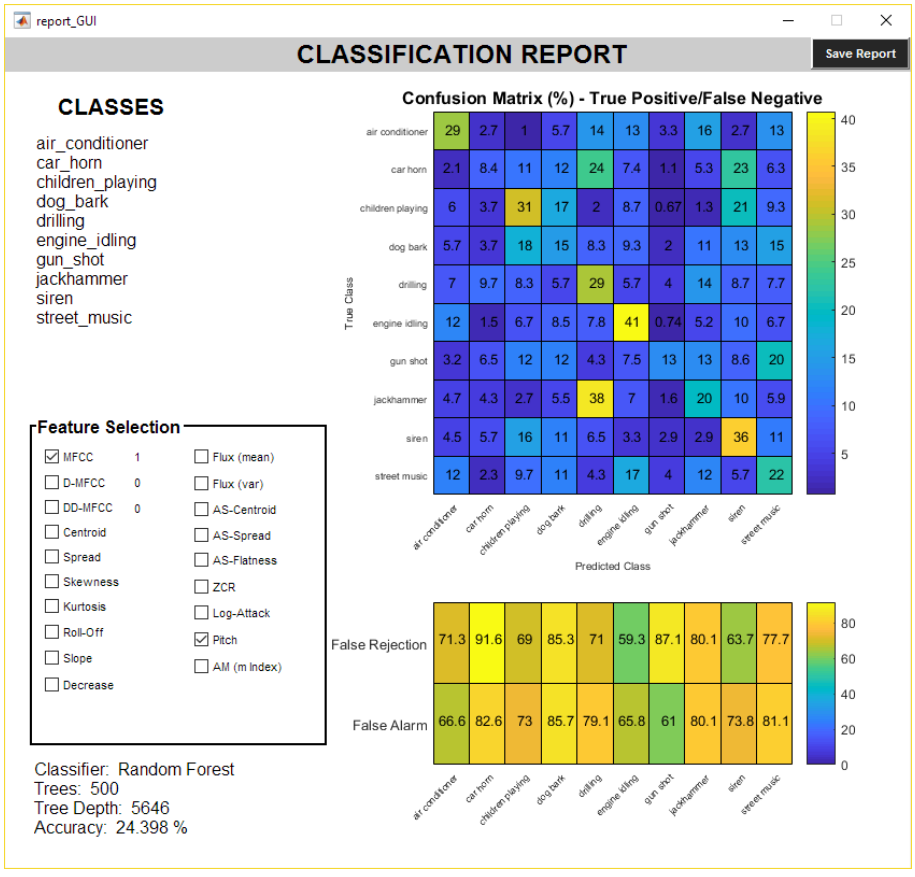

Fig. 83. *Pitch* y MFCC1

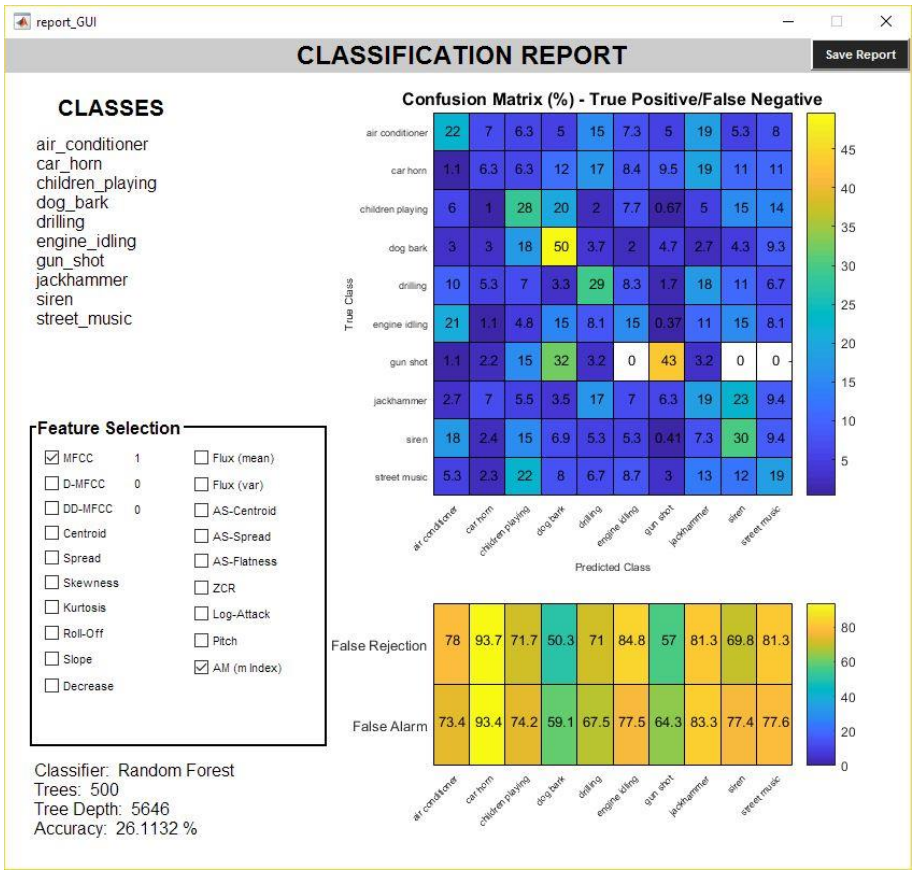

Fig. 84. *m Index* (AM) y MFCC1