UNIVERSIDAD INTERNACIONAL DE LA RIOJA

Universidad Internacional de La Rioja

Escuela Superior de Ingeniería y Tecnología

Máster Universitario en Análisis y Visualización de Datos Masivos

Desarrollo de un algoritmo de detección de voz en castellano con TinyML

Trabajo Fin de Máster

Tipo de trabajo: Desarrollo de software

Presentado por: Tejedor Ferrero, Roberto

Director/a: Agüero Iglesia, Carlos

Resumen

Los sistemas de detección de voz (VAD, “voice activity detection”) persiguen detectar la presencia de voz y diferenciar las secciones de voz de las de silencio o ausencia de voz. Son importantes para las aplicaciones basadas en la voz, tales como la codificación y el reconocimiento de voz. La literatura recoge diversas metodologías para alcanzar este objetivo y, en este caso, se presenta un VAD para lengua castellana basado en la detección de fonemas vocálicos.

Por otro lado, Tiny Machine Learning (TinyML) es un campo reciente que supone un punto de encuentro entre los métodos de aprendizaje automático y los sistemas embebidos, con la premisa de llevar las técnicas de inteligencia artificial hasta dispositivos de bajo coste, baja capacidad de procesamiento y limitada capacidad de almacenamiento, como pueden ser los microcontroladores.

Este TFM se plantea con el objetivo de recrear todas las etapas que permiten la integración en un microcontrolador de un sistema de inteligencia artificial, concretamente, de un sistema VAD implementado sobre la plataforma Arduino. Esto incluye la creación de la base de datos necesaria, la programación y entrenamiento del algoritmo de detección de voz, su posterior adaptación e implementación en el microcontrolador (TinyML) y, por último, el análisis y validación de los resultados obtenidos.

Palabras Clave: TinyML, detección de voz, Arduino

Abstract

Voice detection systems (VAD, “voice activity detection”) seek to detect the presence of voice and differentiate the voice sections from those of silence or absence of voice. They are important for speech-based applications such as coding and speech recognition. The literature includes various methodologies to achieve this objective and, in this case, a VAD for the Spanish language based on the detection of vowel phonemes is presented.

On the other hand, Tiny Machine Learning (TinyML) is a recent field that represents a meeting point between machine learning methods and embedded systems, with the premise of bringing artificial intelligence techniques to low-cost devices with low processing capacity and limited storage capacity, such as microcontrollers.

This TFM is proposed with the objective of recreating all the stages that allow the integration into a microcontroller of an artificial intelligence system, specifically, a VAD system implemented on the Arduino platform. This includes the creation of the necessary database, the programming and training of the voice detection algorithm, its subsequent adaptation and implementation in the microcontroller (TinyML) and, finally, the analysis and validation of the obtained results.

Keywords: TinyML, voice detection, Arduino

Índice de contenidos

Índice de tablas

Índice de figuras

1. Introducción

Los procedimientos de detección de voz (VAD) son básicos para las aplicaciones de procesamiento y constituyen un importante paso previo a las mismas. Su función es distinguir el silencio/ruido, las secciones con ausencia de voz y aquellas en la que hay presencia de voz, ya que la existencia de secciones sin voz degrada considerablemente el rendimiento de las aplicaciones de procesamiento.

Las técnicas VAD se han basado tradicionalmente en dos orientaciones:

- Basadas en el dominio del tiempo: short-time average energy (STAE), short-time average magnitude (STAM), zero-crossing rate (ZCR) …
- Basadas en el dominio de la frecuencia: espectro, cepstro ...

Típicamente, se extraen las características (en uno o ambos dominios) de la señal de audio y se comparan con umbrales de referencia, de manera que si se superan dichos umbrales se infiere la presencia de voz en la señal. Más recientemente se han empezado a emplear técnicas de aprendizaje profundo sobre las características de la señal de audio dando origen a toda una nueva gama de algoritmos VAD.

Tres factores son determinantes para medir la utilidad de un sistema VAD:

- Latencia: el tiempo que se tarda en determinar la presencia/ausencia de voz desde que se recibe la señal de audio
- Recursos computacionales necesarios
- Robustez frente al ruido

Los dos primeros afectarán al entorno de aplicación. Así, por ejemplo, las aplicaciones en tiempo real requerirán una baja latencia y aquellas que se ejecuten en entornos móviles deberán exigir poca capacidad computacional. La robustez frente al ruido, por el contrario, será crítica en cualquier tarea, tratándose de hecho de uno de los problemas más complejos a los que se enfrenta cualquier sistema VAD.

En este TFM, siguiendo la idea del desarrollo de un algoritmo VAD aplicable a entornos de uso móvil en tiempo real, se plantea un sistema VAD específico para lengua castellana, basado en la detección de fonemas vocálicos por medio de una red neuronal y su implementación en la plataforma Arduino, concretamente en Arduino Nano 33 BLE Sense. Al tratarse de un microcontrolador de 32-bit tipo ARM® Cortex®-M4 CPU a 64 MHz, se hace necesario el desarrollo de un algoritmo ligero y adaptado a los pocos recursos disponibles de manera que no haya una penalización debido a la latencia.

Es en el campo de la adaptación de los procedimientos de aprendizaje automático a equipos de bajos recursos donde surge el ámbito de desarrollo conocido como TinyML, cuyas herramientas son utilizadas a fin de llevar a cabo la implementación final del algoritmo VAD desarrollado.

El resto del TFM está organizado de la siguiente manera. En el capítulo 2 se recoge información sobre los sistemas VAD, el campo de TinyML y el análisis fonético de la lengua castellana. Se analiza primeramente la evolución de los sistemas VAD, presentando las distintas soluciones que se han ido desarrollando a lo largo del tiempo. Seguidamente se presentan los conceptos relativos al TinyML, sus motivaciones y usos, así como una breve presentación de la plataforma Arduino Nano 33 BLE Sense utilizada. Por último, se hace un breve análisis fonético de la lengua castellana, ya que es la base en la que se justifica el sistema VAD desarrollado.

En el capítulo 3 se plantean los objetivos que se persiguen, así como la metodología seguida para su consecución.

El capítulo 4 se presenta primeramente la base de datos utilizada y las manipulaciones llevadas a cabo sobre ella. Después se describen los algoritmos de detección de fonemas vocálicos y de detección de voz. A continuación, se analiza la implementación de dichos algoritmos en Arduino y, por último, se presentan las pruebas realizadas en la plataforma y los resultados obtenidos.

Finalmente, en el capítulo 5, se resumen las conclusiones alcanzadas y se recogen posibles líneas de desarrollo futuras.

2. Contexto y estado del arte

2.1 Sistemas VAD

Los sistemas VAD buscan determinar el principio y el final de una sección de voz dentro de una señal que contiene un discurso hablado. Es una etapa fundamental en los sistemas de procesamiento de voz que permite separar las secciones de voz del ruido de fondo.

Las aplicaciones de los VAD incluyen sistemas de audio conferencia, cancelación de eco, codificación de voz, reconocimiento de voz, identificación de hablantes, telefonía de manos libres y, en sistemas UMTS, son fundamentales para permitir el ajuste de la tasa de bits y mejorar la calidad de codificación. También se usan en algunas aplicaciones de procesamiento en las que se eliminan las secciones sin presencia de voz.

Desde el primer sistema de reconocimiento de voz presentado en 1952 por Bell Laboratories llamado “Audrey” (Automatic Digit Recognition) se han desarrollado cientos de métodos que se pueden clasificar en dos grupos: basados en umbrales o basados en modelos estadísticos.

Los primeros extraen características de la señal (en el dominio del tiempo y/o la frecuencia) como energía, zero crossing rate, espectro, espectro de la entropía… y las comparan con umbrales de referencia, de manera que si se superan dichos umbrales se determina que hay presencia de voz en la señal.

Dentro de esta categoría la literatura recoge numerosos ejemplos de uso. Así, por ejemplo, (Davis, Nordholm, & Togneri, 2006) usan la varianza del espectro con un umbral variable basado en estadísticas del ruido presente en la señal. (Tanyer & Ozer, 2000) combinan un umbral de energía adaptativo junto con medidas de la periodicidad y la tasa de paso por cero de la señal. (Woo, Yang, Park, & Lee, 2000) proponen un algoritmo basado en parámetros del logaritmo de la energía en un VAD aplicable en el ámbito de los vehículos a motor. (Yoo, Lim, & Yook, 2015) aprovechan los formantes de la voz humana, esto es, los picos de intensidad en el espectro del sonido, para detectar los sonidos de las vocales. (Shuyin, Ying, & Buhong, 2009) usan la propiedad de autocorrelación entre la sección de voz actual y las secciones cercanas. (Ramírez, Segura, Benítez, Torre, & Rubio, 2004) se basan en la divergencia espectral de largo plazo (LTSD) entre el ruido y el habla.

Los segundos usan modelos estadísticos para discriminar las zonas con y sin voz. La mayoría de ellos se basan en entrenar clasificadores usando características de la señal de audio tanto en el dominio del tiempo como en el de la frecuencia tales como, la varianza,

coeficientes de variación, percentiles de la distribución de probabilidad de la señal y coeficientes cepstrales en la frecuencia de Mel (MFCC, “Mel Frequency Cepstral Coefficients”).

Este segundo grupo también tiene una nutrida representación en la literatura. (Mousazadeh & Cohen, 2013) extraen la matriz de similitud de un modelo de agrupamiento espectral a partir de un conjunto de entrenamiento y lo combinan con un modelo de mezclas gaussianas (GMM, “Gaussian mixture model”) para determinar la razón de verosimilitud. (Kinnunen, 2007) tratan la detección como un problema de clasificación binaria y lo resuelven por medio de un modelo SVM (“Support Vector Machine”). (Chang, Kim, & Mitra, 2006) combinan tres modelos, GMM, complejo de Laplace y densidad de probabilidad Gamma, para mejorar los resultados obtenidos con cada uno de ellos individualmente. (Wu & Zhang, 2011) combinan una primera etapa basada en la detección de energía seguida de un modelo GMM sobre un registro de varias observaciones. (Bao & Zhu, 2012) presentan un VAD basado en reconocimiento de fonemas en mandarín usando GMM y un modelo oculto de Markov (HMM, “Hidden Markov Model”). (Hwang, Jeong, Oh, & Kim, 2017) desarrollan un VAD usando un HMM de doble nivel.

Dentro de este segundo grupo, se ha producido en los últimos años un importante progreso en técnicas basadas en aprendizaje profundo. Así, por ejemplo, (Mateju, Červa, Zdánský, & Málek, 2017) presentan un sistema basado en una red neuronal profunda (DNN, “Deep Neural Network”) entrenada sobre una base de señales de audio artificialmente modificado con los niveles de relación señal-ruido (SNR, “Signal to Noise Ratio”) deseados. (Jang, Ahn, Seo, & Jang, 2017) desarrollan un DNN con dos capas ocultas y MFCC como característica de entrada. (Hwang, Park, & Chang, 2016) describen un arreglo de DNNs entrenadas en distintos escenarios de ruido y usan una técnica de clasificación acústica del entorno para elegir en cada momento la DNN más apropiada. (Kang & Kim, 2016) emplean una DNN más robusta frente al ruido gracias a un esquema de aprendizaje multitarea. (Kim, Kim, Lee, Park, & Hahn, 2016) investigan un VAD para detección de vocales basado en red neuronal recurrente con memoria de corto y largo plazo (LSTM-RNN, “Long short term memory recurrent neural network”).

Como se puede ver la variedad de técnicas desarrolladas a lo largo del tiempo es muy amplia. Su mejor o peor desempeño están fuertemente relacionadas con las condiciones SNR de las señales a analizar, siendo este uno de los principales problemas al que se enfrenta cualquier sistema VAD.

2.2 TinyML

Tal y como se recoge en la web de la fundación TinyML (www.tinyml.org), TinyML se puede definir como un campo del aprendizaje automático que incluye hardware, algoritmos y software capaces de realizar análisis de datos de sensores de dispositivos con muy poca potencia (típicamente en el rango de los mW) posibilitando su empleo en dispositivos alimentados por batería que pueden estar permanentemente conectados.

El ejemplo más relevante de su empleo se encuentra en el campo de los microcontroladores. Se estima que en el mundo hay instalados más de 250 mil millones de microcontroladores y su campo de aplicación no ha dejado de crecer. La posibilidad de implementar técnicas de ML en este tipo de dispositivos presentes en todas partes abre un importante abanico de aplicaciones posibles para el análisis de datos. Dado que los modelos de ML que se implementan en estos dispositivos son extremadamente ligeros (unos pocos KBs) pueden operar continuamente con un efecto muy pequeño en la duración de la batería de los equipos.

En la evolución de esta tecnología se pueden distinguir tres etapas:

- IA en la nube: los modelos son entrenados y almacenados en la nube. Los dispositivos IoT se conectan al servidor, envían los datos y éste devuelve el resultado de la inferencia. Este esquema implica tránsito de información por la red y latencia en la transmisión.

- IA en la frontera: el entrenamiento se realiza en la nube y, para la inferencia, el modelo entrenado se implementa en un dispositivo intermedio, de manera que el dispositivo IoT se comunica localmente para obtener el resultado de la inferencia. Se evita así la transmisión de datos por la red y su correspondiente latencia.

- IA en el microcontrolador: en aquellos casos en que los costes vinculados a la anterior solución no sean asumibles el modelo se debe ejecutar directamente en el dispositivo IoT. Este planteamiento es el más eficiente y el de menor coste.

Las principales ventajas de TinyML son:

- Consumo energético muy bajo.

- Baja latencia: al realizar la inferencia localmente se evitan los retardos asociados al envío y recepción de información a los servidores.

- Se necesita poco ancho de banda: el grueso del trabajo se realiza localmente y las transmisiones de datos se reducen
- Privacidad: los datos no se envían ni almacenan en servidores remotos
- Bajo coste: el coste de los microcontroladores es sensiblemente inferior al de las CPU/GPU de los servidores. Así mismo la reducción de la trasmisión de datos redunda en una disminución de costes operativos y de infraestructura. Por último, el coste energético también se reduce enormemente (tanto por el procesamiento como por la trasmisión).

La aplicación más conocida y extendida del TinyML es el reconocimiento de palabras clave. Se usa, por ejemplo, en los dispositivos de Amazon, Apple y Google para iniciar la interlocución con sus servicios en la nube. La literatura recoge algunos ejemplos de esta aplicación: (Chen, Parada, & Heigold, 2014) describen una DNN para la detección de palabras clave, (Gruenstein, Alvarez, Thornton, & Ghodrat, 2017) presentan una arquitectura en cascada con procesadores digitales de señales (DSP, “digital signal processor”) y (Zhang, Suda, Lai, & Chandra, 2017) usan MFCC sobre una red neuronal convolucional (CNN, “convolutional neural network”).

Si bien ésta es la aplicación más conocida, existe una amplia variedad de aplicaciones que incluyen, entre otras, reconocimiento de actividades humanas (Chavarriaga, Sagha, Calatroni, & Digumarti, 2013), detección acústica de anomalías (Koizumi, y otros, 2019), mantenimiento predictivo (Susto, Schirru, Pampuri, McLoone, & Beghi, 2014) y detección visual de objetos (Chowdhery, Warden, Shlens, Howard, & Rhodes, 2019).

Los primeros desarrollos TinyML exigían la optimización manual de los modelos a implementar en el microcontrolador elegido, de manera que cualquier cambio de dispositivo implicaba un trabajo completo de adaptación. (David, y otros, 2021) presentan en 2020 una primera versión de Tensor Flow Lite Micro, una biblioteca basada en Tensor Flow para ejecutar modelos de aprendizaje profundo en sistemas embebidos, permitiendo a los desarrolladores abstraerse del microcontrolador específico a usar, ofreciendo portabilidad y flexibilidad. Esta biblioteca es la que se ha usado para el desarrollo del presente TFM.

2.3 Arduino

Arduino es una plataforma electrónica de código abierto basada en hardware y software de uso sencillo. Las tarjetas de Arduino incluyen un microcontrolador que posibilita al usuario su programación, para realizar lectura de entradas y activación de salidas según sus

necesidades. Implementada inicialmente en el ámbito académico, gracias a su sencillez y a la gran cantidad de recursos disponibles aportados por la comunidad de usuarios, su uso se ha generalizado usándose en muchos casos como una plataforma de desarrollo rápido de nuevos prototipos.

En 2019 Arduino presentó la placa Arduino Nano 33 BLE Sense (Figura 1). Se trata de una placa que incorpora un procesador nRF52840 de Nordic Semiconductors de 32-bit con arquitectura ARM® Cortex®-M4 CPU que funciona a 64 MHz e incluye 1 MB de memoria Flash y 256 KB RAM.

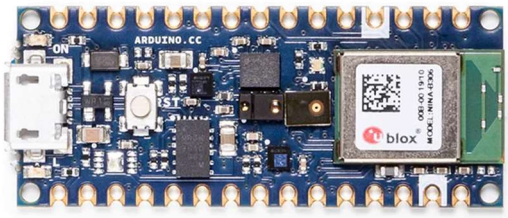

Figura 1: Placa Arduino Nano 33 BLE Sense (Fuente: Arduino)

Incorpora así mismo una gran variedad de sensores: sensor inercial de 9 ejes, sensor de temperatura y humedad, sensor barométrico, micrófono y sensor de luz/color/proximidad/gestos. Por último, incluye posibilidad de conexión por Bluetooth® y Bluetooth® Low Energy. La Figura 2 recoge la ubicación de los distintos sensores en la placa.

El objetivo de Arduino con el desarrollo de esta placa es la posibilidad de ejecutar aplicaciones de inteligencia artificial usando TinyML y, por ese motivo, se ha escogido como base para el desarrollo de este TFM.

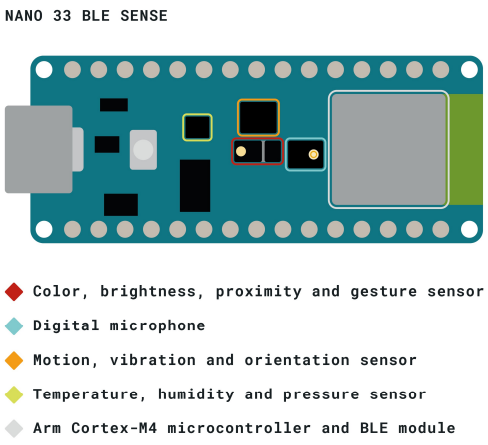

Figura 2: Distribución de sensores en Arduino Nano 33 BLE Sense (Fuente: Arduino)

2.4 Fonética en el análisis de voz

La fonética es la parte de la gramática que estudia los mecanismos de producción, transmisión y percepción de la señal sonora que constituye el habla. Es un campo que se encuentra en la base de numerosos procesos empleados en el análisis de voz. Uno de los objetos de estudio de la fonética es el fonema.

Los fonemas son abstracciones usadas para estudiar una lengua humana a nivel fonético-fonológico. Para establecer lo que constituye un fonema se requiere que exista una función diferenciadora y, por ese motivo, se relacionan los fonemas con los sonidos del habla que permiten distinguir palabras. Así, los fonemas /c/ y /l/ son fonemas del castellano porque existen palabras como /codo/ y /lodo/ que tienen significado distinto y su pronunciación únicamente es distinta en dos sonidos que se relacionan con dos fonemas diferentes.

Estructuralmente el fonema pertenece a la lengua y el sonido al habla. La palabra <paso>, consta de cuatro fonemas (/p/, /a/, /s/, /o/) que en el habla se corresponden con cuatro sonidos a los que se denominan fonos. Un fonema es una abstracción del fono efectivamente pronunciado en el acto de hablar y sirve para representar a una familia de fonos que los hablantes asocian a un sonido concreto durante la percepción o producción del habla. Los fonos representados por un mismo fonema se denominan alófonos. Un mismo fonema puede ser pronunciado con distintos alófonos dependiendo del hablante específico o de la posición relativa del fonema dentro de la palabra.

La simplicidad del concepto del fonema es el motivo de su interés en el ámbito del análisis de voz y la existencia de los alófonos, potencialmente infinitos para cada fonema, representa la principal dificultad a la hora de utilizarlos.

Uno de los usos típicos es el reconocimiento de idioma. Dado que cada idioma tiene un conjunto de fonemas diferentes resulta una aproximación interesante para llevar a cabo la diferenciación. (Toledano, y otros, 2007) presentan un sistema de reconocimiento de idiomas basado en una etapa de reconocimiento de fonemas seguido de una etapa de modelado de idioma. (Campbell, Richardson, & Reynolds, 2007) plantean un sistema basado en una etapa de conversión de habla a texto seguida de una etapa de modelado de idioma basado en SVM.

Otros ejemplos del empleo de fonemas en análisis de voz son su uso para reconocimiento de emociones (Ringeval & Chetouani, 2007) o la detección de vocales por medio de modelado de agrupamientos de fonemas (Luengo, Navas, Sánchez, & Hernáez, 2009).

El presente TFM persigue aprovechar las características específicas del conjunto de fonemas del castellano para desarrollar un sistema VAD. (Pérez, 2003) presenta los resultados de un recuento de frecuencia de fonemas en un corpus oral de lectura de noticias en la televisión chilena y los compara con los resultados de los recuentos llevados a cabo por otros cuatro autores. Dicha comparativa está recogida en la Tabla 1.

Tabla 1: Distribución porcentual de la frecuencia de los fonemas del español

	Alarcos Llorach	Navarro Tomás	Zipf Rogers	Guirao García	Pérez	Prom.
/i/	8,60	4,76	4,20	6,59	7,46	6,32
/e/	12,60	11,75	12,20	14,99	14,13	13,13
/a/	13,70	13,00	14,06	13,27	12,31	13,27
/o/	10,30	8,90	9,32	10,75	9,28	9,71
/u/	2,10	1,92	1,76	2,79	3,05	2,32
/p/	2,10	3,04	2,92	2,68	2,58	2,66
/t/	4,60	4,82	4,46	4,48	4,92	4,66
/t/	3,80	4,23	3,84	4,31	3,94	4,02
/b/	2,50	2,54	3,26	3,08	1,92	2,66
/d/	4,00	5,00	5,06	4,00	4,84	4,58
/g/	1,00	1,04	1,02	1,11	0,94	1,02
/f/	1,00	0,72	0,72	0,53	0,75	0,74
/z/	1,70	2,23	1,74			1,89
/s/	8,00	8,50	8,12	9,39	9,61	8,72
/x/	0,70	0,51	0,58	0,70	0,74	0,65
/ch/	0,40	0,30	0,30	0,40	0,32	0,34
/y/	0,40	0,40	2,40	0,72	0,69	0,92
/m/	2,50	2,40	2,98	3,17	2,62	2,73
/n/	2,70	2,94	5,94	7,14	7,78	5,30
/ñ/	0,20	0,36	0,36	0,24	0,24	0,28
/N/	3,70	4,69				4,20
/r/	2,50	2,40	5,90	5,40	6,19	4,48
/rr/	0,60	0,80	1,04	0,39	0,64	0,69
/R/	4,50	3,51				4,01
/l/	4,70	5,46	5,20	3,88	5,05	4,86
/ll/	0,50	0,60	0,60			0,57

Fuente: (Pérez, 2003)

Tomando como referencia los valores promedio de la última columna, se puede ver que los fonemas vocálicos representan un 47,41% del conjunto, esto es, prácticamente la mitad de los fonos emitidos por un hispanohablante corresponden a un fonema vocálico.

Por otro lado, las sílabas en castellano se estructuran siempre en torno a una vocal. La Tabla 2 muestra los tipos de sílabas en castellano tal y como recoge (Moreno, Taboada, Muñoz-Basols, & Lacorte, 2017) donde V corresponde a un sonido vocálico y C corresponde a un sonido consonántico.

Tabla 2: Tipos de sílabas en castellano

V	**A**-na
CV	u-**no**
VC	**an**-tes
VCC	**ins**-tru-ye
CVC	an-**tes**
CVCC	**cons**-pi-ra-ción
CCV	**tra**-zo
CCVC	**prés**-ta-mo
CCVCC	**trans**-por-te

Fuente: (Moreno, Taboada, Muñoz-Basols, & Lacorte, 2017)

(Moreno Sandoval, Toledano, Curto, & Torre, 2006) hacen un inventario de frecuencia silábico sobre el corpus C-ORAL-ROM cuyo resumen se recoge en la Tabla 3. En ella se puede ver que algo más del 95% de las sílabas en castellano carece de una estructura con dos sonidos consonánticos consecutivos.

Tabla 3: Frecuencia relativa de sílabas en castellano

CV	51,35
CVC	18,03
V	10,75
VC	8,60
CVV	3,37
CVVC	3,31
CCV	2,96
CCVC	0,88

Fuente: (Moreno Sandoval, Toledano, Curto, & Torre, 2006)

De lo anterior se extrae como premisa para el diseño de un VAD en lengua castellana que la detección únicamente de sonidos vocálicos permite detectar aproximadamente el 50% de los sonidos emitidos por el hablante y que entre dos sonidos vocálicos no consecutivos lo más probable es que haya un sonido consonántico no detectado.

2.5 Conclusiones

En este capítulo se ha recogido la importancia que los sistemas VAD tienen en el ámbito de los sistemas de procesamiento de la voz. Así mismo, se ha ofrecido una visión de la proyección futura y el interés de la implantación de técnicas de aprendizaje automático en dispositivos IoT de bajo coste y bajo consumo. Se ha presentado Arduino como una plataforma muy útil para el prototipado rápido. Por último, se han revisado brevemente los conceptos relativos a la fonética y las peculiaridades que ésta tiene en el caso concreto de la lengua castellana.

Este TFM aprovecha esas peculiaridades para plantear un algoritmo VAD basado en la detección de sonidos vocálicos, lo que simplifica la detección y facilita su implementación en un entorno TinyML.

3. Objetivos y metodología

3.1. Objetivos

Este TFM persigue el desarrollo e implementación en un microcontrolador de un sistema VAD en lengua castellana basado en la detección de fonemas vocálicos capaz de realizar su función en un entorno de recursos limitados. Para alcanzar dicho objetivo general se plantean los siguientes objetivos específicos:

- Adaptar una base de datos existente y convertirla en un conjunto de datos aptos para las necesidades del presente desarrollo.
- Plantear diversos algoritmos de aprendizaje automático útiles para la detección de fonemas vocálicos, comparar sus resultados y seleccionar el más idóneo para su implementación en un microcontrolador.
- Desarrollar un algoritmo que permita inferir la presencia o ausencia de voz a partir únicamente de la detección de fonemas vocálicos aprovechando las características específicas de la lengua castellana y analizar su precisión y viabilidad.
- Adaptar ambos algoritmos a la plataforma Arduino, verificar su funcionalidad y analizar su comportamiento.

3.2. Metodología

La metodología planteada sigue de manera secuencial los objetivos específicos anteriormente expuestos:

1. **Preparación de la base de datos**
   La base de datos utilizada es el Multilingual LibriSpeech (MLS) dataset presentado por (Pratap, Xu, Sriram, Synnaeve, & Collobert, 2020) que consta de audios de textos leídos por distintos hablantes. Para su adaptación se seguirán los siguientes pasos:
   - Etiquetado de fonemas por medio del software Audacity v3.3.3 de (Audacity Team, 1999). Dicho etiquetado será realizado de manera manual y determinará el inicio y final de cada fonema presente en las distintas locuciones. El etiquetado es necesario ya que se van a emplear técnicas de aprendizaje supervisado. Este etiquetado permitirá construir una base de

datos que conste de la señal de audio de cada fonema pronunciado y su identificación correspondiente.

- Análisis de la base de datos resultante, caracterizando cualitativa y cuantitativamente los fonemas etiquetados.
- Comparación de las diversas técnicas de análisis de las señales de audio, tanto en el dominio del tiempo como en el dominio de la frecuencia.
- Elección de la técnica más apropiada para su uso, dados los requerimientos de la aplicación que se persigue y la naturaleza de los datos de los que se dispone.
- Extracción de los parámetros que caracterizan cada uno de los fonemas etiquetados anteriormente haciendo uso de la técnica elegida. Estos parámetros constituirán la entrada al algoritmo de detección de fonemas vocálicos y se utilizarán en su fase de entrenamiento. De esta forma se habrá obtenido una base de datos de trabajo que consta de los parámetros de cada fonema y su correspondiente identificación como vocal o no vocal. Esta última parte del desarrollo será llevado a cabo usando lenguaje Python de (Python Software Foundation, 2001).

2. **Algoritmos de detección de fonemas vocálicos**

   Una vez elaborada la base de datos se procederá al desarrollo del algoritmo responsable de la detección de los fonemas vocálicos siguiendo las siguientes etapas:

   - Selección de algoritmos de clasificación. Se realizarán pruebas con tres algoritmos de clasificación, concretamente, Random Forest, SVC y red neuronal.
   - Selección de hiperparámetros apropiados para cada algoritmo
   - Entrenamiento con los datos etiquetados previamente.
   - Comparativa de resultados. Se realizará una comparativa de la capacidad de clasificación de los tres algoritmos sobre un conjunto de datos de validación, examinando diversas métricas en paralelo y teniendo también en cuenta los tiempos necesarios para su ejecución.
   - Análisis de la naturaleza de los errores que cometen al realizar la clasificación de los fonemas, tanto en la distinción de fonemas vocálicos y no vocálicos como en los tipos de fonemas que conllevan mayor dificultad de clasificación, a fin de buscar posibles soluciones o estrategias de mitigación.

- Selección de algoritmo. En base a esta comparativa, se determinará cuál es de ellos resulta más apropiado para llevar a cabo su posterior implementación en un microcontrolador.

Este desarrollo será llevado a cabo usando lenguaje Python de (Python Software Foundation, 2001).

3. Algoritmo VAD

Una vez definido el algoritmo de detección de fonemas vocálicos se buscará la estrategia que permita inferir la presencia de voz a partir de la detección de fonemas vocálicos únicamente. Para ello se llevarán a cabo los siguientes procesos:

- Análisis de la composición fonética del habla en lengua castellana. Empleando estudios sobre composición fonética de la lengua castellana y realizando análisis directamente sobre la base de datos construida se extraerán características relevantes que se puedan aprovechar para la detección de voz a partir de la detección de fonemas vocálicos.
- Planteamiento de estrategias de detección. En base al análisis anterior se buscará una estrategia que permita inferir de forma aproximada la presencia de voz a partir de la detección de fonemas vocálicos.
- Desarrollo del algoritmo VAD. Se llevará a cabo la combinación del algoritmo de detección de fonemas vocálicos y el algoritmo que infiere la presencia de voz. Dicha combinación constituirá el algoritmo VAD definitivo.
- Análisis de errores de detección del algoritmo VAD. Se llevarán a cabo pruebas de validación del algoritmo desarrollado sobre archivos de audio con zonas de voz/no voz etiquetadas para determinar su capacidad de detección y su tasa de error. Se caracterizarán así mismo, dichos errores extrayendo las conclusiones pertinentes.
- Análisis de comportamiento frente a perturbaciones. Se realizarán pruebas para determinar el comportamiento del algoritmo ante señales perturbadas por diferentes fuentes de ruido externas.
- Análisis de comportamiento ante otros idiomas. Se realizarán pruebas para determinar el comportamiento del algoritmo ante fuentes de audio en otros idiomas.

Este desarrollo también será llevado a cabo usando lenguaje Python de (Python Software Foundation, 2001).

4. Implementación en Arduino

Una vez definido el algoritmo VAD se llevará a cabo su adaptación a la plataforma Arduino Nano 33 BLE Sense haciendo uso de TensorFlow Lite para microcontroladores (David, y otros, 2021). Para ello se llevarán a cabo los siguientes pasos:

- Análisis de la implementación de (Tang, 2019) para un sistema de reconocimiento de palabras clave. Se realizará un estudio en profundidad de la implementación llevada a cabo por (Tang, 2019) en la misma plataforma de Arduino. Se examinará el código planteado con el fin de adaptarlo al algoritmo VAD desarrollado, reduciendo de este modo el tiempo de desarrollo y aprovechando el uso de un sistema funcional validado en cuanto a la captura de audio y su preprocesamiento.
- Adaptación del código de procesamiento de audio. Se llevarán a cabo las modificaciones pertinentes sobre el código de (Tang, 2019) para que genere los datos necesarios para la entrada al algoritmo de detección de fonemas vocálicos a partir de la señal de sonido captada por el micrófono de la placa de Arduino.
- Exportación del modelo de TensorFlow para la detección de fonemas vocálicos desarrollado previamente por medio de TensorFlow Lite para microcontroladores (David, y otros, 2021). Está exportación permitirá obtener un archivo binario directamente implementable en Arduino.
- Adaptación del código de inferencia de presencia de voz a partir de la detección de fonemas vocálicos. Se adaptará el código desarrollado para que pueda ser directamente ejecutado en el microcontrolador.
- Adaptación del código de respuesta ante la presencia de voz. Se llevarán a cabo las modificaciones pertinentes sobre el código de (Tang, 2019) de manera que la placa del microcontrolador señalice visualmente las detecciones de fonemas vocálicos, las inferencias de presencia de voz y las ausencias de voz.
- Pruebas de funcionamiento. Una vez implementado se realizarán pruebas para verificar su funcionalidad y comportamiento.

Este desarrollo será llevado a cabo usando lenguaje Python de (Python Software Foundation, 2001) para la exportación del modelo y la aplicación Arduino IDE para su adaptación a la plataforma del microcontrolador.

4. Desarrollo de algoritmo de detección de voz

4.1. Base de datos

La base de datos empleada para el desarrollo es el Multilingual LibriSpeech (MLS) dataset de (Pratap, Xu, Sriram, Synnaeve, & Collobert, 2020). Se trata de un corpus multilingüe adecuado para investigaciones sobre el habla. Está construido a partir de audiolibros de LibriVox y contiene 8 idiomas (inglés, alemán, holandés, español, francés, italiano, portugués y polaco) incluyendo unas 44.500 horas de audio en inglés y unas 6.000 horas para el resto de los idiomas. Para el caso concreto del español, hay disponibles unas 940 horas de audio en formato FLAC con una frecuencia de muestreo de 16kHz. Los audios están clasificados atendiendo al género del hablante y segmentados en archivos de entre 10 y 20 segundos. Los archivos están organizados en grupos de entrenamiento, desarrollo y test. En este caso, se ha usado parte del contenido del grupo de test que incluye 10 horas de audio repartido entre 10 hombres y 10 mujeres.

Para el desarrollo preliminar se han seleccionado 8 archivos (4 mujeres y 4 hombres) para llevar a cabo el etiquetado de fonemas de forma manual por medio del software Audacity.

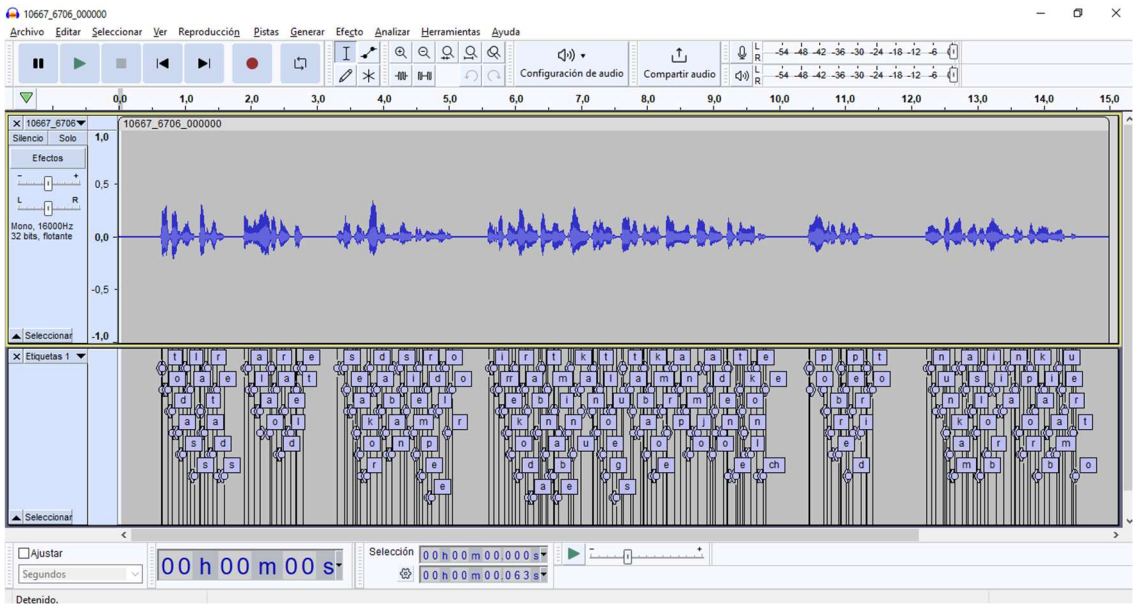

Figura 3: Etiquetado de fonemas en Audacity©

La Figura 3 muestra uno de los archivos completamente etiquetado dentro de la aplicación. La parte superior muestra la evolución de la señal en el tiempo y en la parte inferior se pueden ver las etiquetas correspondientes a los distintos fonemas.

Para mayor claridad, en la Figura 4 se ha recogido una sección de un segundo del mismo archivo. Para cada fonema se marca el principio y el final, lo que permie generar marcas de tiempo que se usarán posteriormente en el procesado.

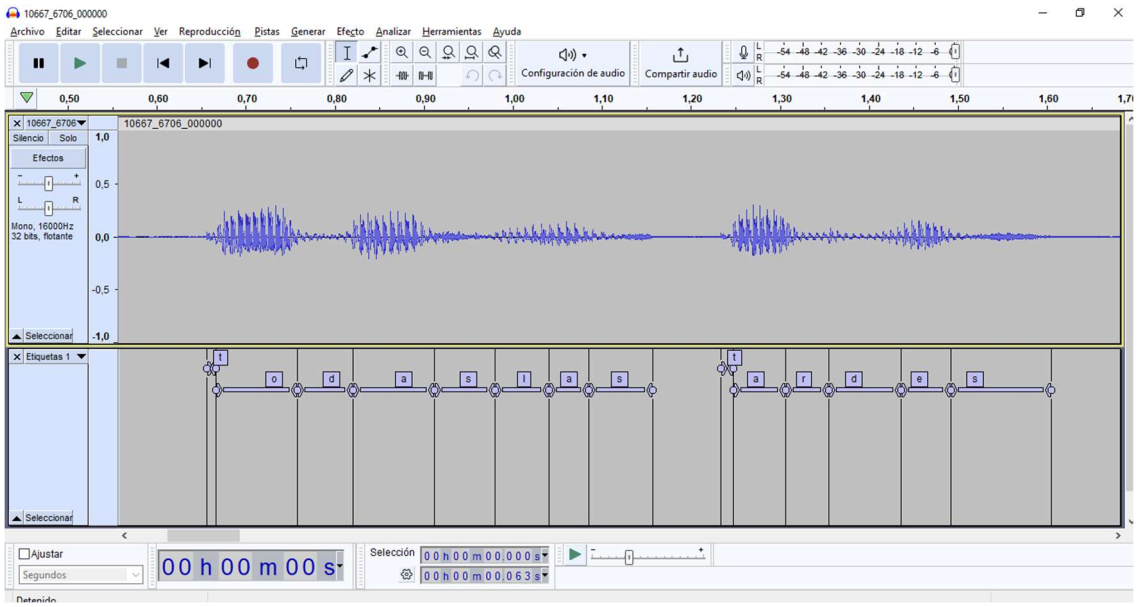

Figura 4: Detalle de etiquetado en Audacity©

Una vez finalizado el etiquetado, con el software Audacity se genera un archivo de texto con el tiempo de inicio, el tiempo de finalización y la etiqueta asignada a cada fonema.

4.2. Detección de fonemas vocálicos

Combinando la información de los archivos etiquetados se realiza un análisis de los fonemas presentes en el conjunto de grabaciones. Se han etiquetado un total de 1.313 fonemas cuya distribución se puede ver en la Figura 5.a. Se puede comparar con los valores recogidos en la Tabla 1 de distribución porcentual de la frecuencia de los fonemas del español, resultando unos porcentajes para los fonemas vocálicos muy parecidos a los recogidos en el estudio de (Pérez, 2003). El total de fonemas vocálicos etiquetados es de 619 por lo que se tiene una frecuencia relativa del 47,14%, prácticamente idéntica a la presentada en el apartado 2.4.

Para poder desarrollar el algoritmo de detección de fonemas vocálicos se hace necesario analizar este subconjunto de manera específica. La Figura 5.b recoge la distribución de la longitud de los fonemas vocálicos atendiendo al número de muestras de la señal en cada una de ellas. Se obtiene un valor medio de 1.147,25 muestras y un máximo y mínimo de 3.733 y 248 muestras respectivamente. Dado que la frecuencia de muestreo usada en las grabaciones es de 16kHz, eso ofrece un tiempo medio de 71,7 milisegundos por cada fonema vocálico, durando el fonema más largo 233,31 milisegundos y el más corto 15,5 milisegundos.

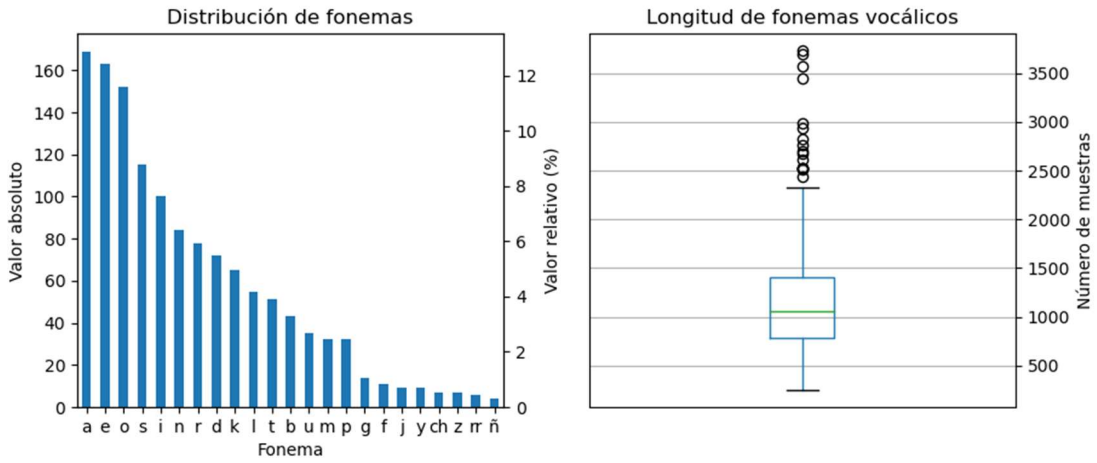

Figura 5 (a, b): Distribución de fonemas y número de muestras en los fonemas vocálicos

Con fines ilustrativos, se han recogido en la Figura 6 las señales correspondientes al fonema vocálico más corto y el más largo, además de una sección ampliada de este último.

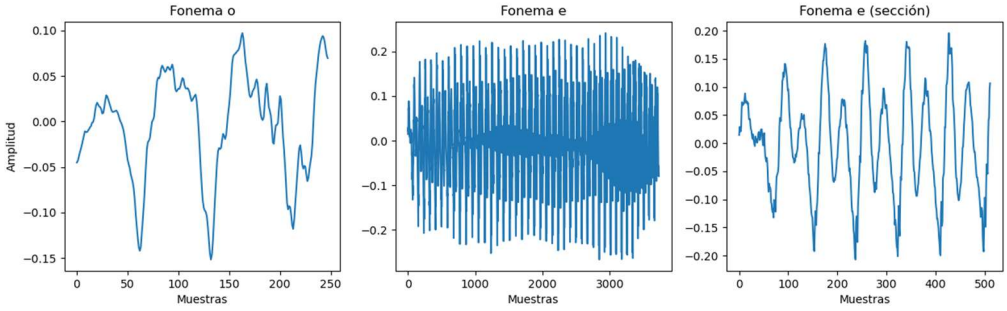

Figura 6: Señales de fonemas vocálicos

La primera consideración necesaria al plantear el algoritmo de detección de fonemas vocálicos son las características de la señal que se van a usar como entradas al sistema.

Una primera opción sería usar directamente la información de audio muestreada. En la Figura 6 se pueden observar las distintas formas de onda para los fonemas /o/ y /e/. Sería posible plantear un algoritmo que intentara extraer las características propias de cada una de ellas para diferenciarlos. Este es el enfoque que usan (Oord, y otros, 2016) para plantear su red neuronal profunda WaveNet. Esta opción no es aplicable en este caso ya que el coste computacional y de almacenamiento serían inasumibles para un microcontrolador.

La segunda opción se basa en el procedimiento más extendido a la hora de procesar señales de audio, el empleo de la transformada de Fourier para el análisis de las señales en el dominio de la frecuencia.

De manera simplificada, se puede decir que una señal se puede descomponer como una suma de señales senoidales, cada una de ellas con una frecuencia, amplitud y desfase

característicos. La transformada de Fourier es la herramienta matemática que permite llevar a cabo dicha descomposición.

Las señales utilizadas en el ámbito digital constituyen un caso especial dadas sus características, ya que se trata de señales discretas en tiempo discreto, esto es, son señales que pueden tomar un número finito de valores en instantes concretos de tiempo. Estas señales se obtienen al llevar a cabo dos procesos sobre las señales reales que se encuentran en la naturaleza de manera que puedan ser utilizadas en los entornos informáticos. Un primer proceso es la cuantización, con la que se aproximan los infinitos valores presentes en la realidad a un número finito de valores representables digitalmente. Un segundo proceso es el muestreo, con el que se usan los valores de la señal en determinados instantes, normalmente equiespaciados, desechándose el resto de los valores.

La transformada de Fourier aplicada a este tipo de señales se denomina transformada de Fourier en tiempo discreto (DTFT, “Discrete Time Fourier Transform”) y tiene la peculiaridad de tratarse de una señal continua en la frecuencia (toma valores en cualquier frecuencia) y periódica.

El empleo de la DTFT en un sistema informático obliga a llevar a cabo un proceso de discretización en la frecuencia análogo al que se ha comentado anteriormente respecto a las señales naturales en el tiempo. Esa discretización conduce a la obtención de la transformada de Fourier discreta (DFT, “Discrete Fourier Transform”) que constituye la base de trabajo en el dominio de la frecuencia en el ámbito computacional.

A partir de la DFT y haciendo uso de sus propiedades matemáticas se han desarrollado algoritmos eficientes para llevar a cabo su cálculo. Estos cálculos eficientes se conocen como transformada rápida de Fourier (FFT, “Fast Fourier Transform”).

Para trabajar en el dominio de la frecuencia con señales de gran duración, el método habitual consiste en llevar a cabo un enventanado de la señal, esto es, seleccionar un conjunto de muestras consecutivas de una determinada longitud, extraer las características en frecuencia de dicha ventana, desplazar la ventana un cierto número de muestras, de manera que la nueva ventana se solape parcialmente con la ventana previa y repetir el procedimiento hasta procesar toda la señal.

La Figura 7 muestra gráficamente el procedimiento y los dos valores necesarios para su definición: longitud de ventana y longitud de salto. En adelante, y con el fin de separar conceptos, se usará el término inglés “frame” para designar al conjunto de muestras

extraídas después de aplicar una ventana a la señal, esto es, a cada segmento enventanado.

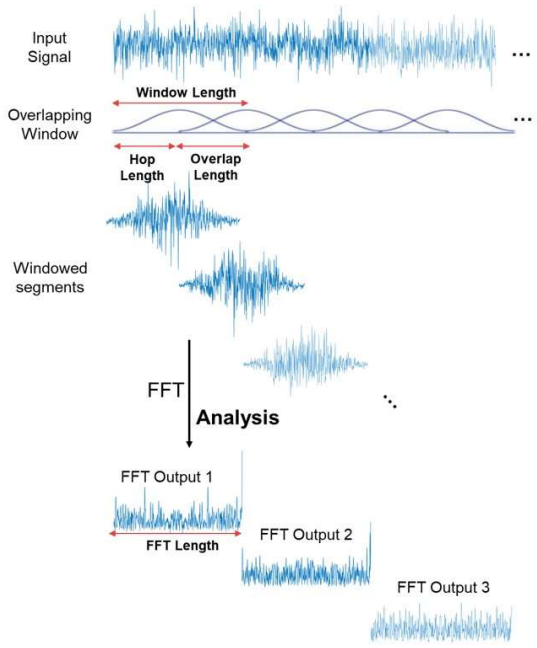

Figura 7: Enventanado y análisis en frecuencia (Fuente: https://es.mathworks.com/help/dsp/ref/dsp.stft.html)

Este procedimiento de enventanado seguido de cálculo de FFT se conoce como transformada de Fourier de tiempo reducido o STFT ("Short Time Fourier Transform") y es la base para la obtención del espectrograma, que permite ver de manera visual la evolución del contenido en frecuencia de la señal a lo largo del tiempo, mostrando en el eje de ordenadas las frecuencias (normalmente en escala logarítmica), en el eje de abscisas el tiempo y las amplitudes en cada frecuencia se muestran por medio de un diagrama de calor. La imagen izquierda de la Figura 9 muestra el espectrograma de una señal.

Otra opción posible, trabajando también en el dominio de la frecuencia, se basa en el empleo de los coeficientes cepstrales de las frecuencias de Mel (MFCC), que es la base de la inmensa mayoría de los sistemas que analizan el habla humana. Esta metodología trata de hacer una aproximación matemática al funcionamiento biológico del sistema auditivo humano, que recoge dos peculiaridades de nuestra percepción del sonido:

- Percepción logarítmica, esto es, para percibir el doble de sonido se necesitan aproximadamente 10 veces más de potencia sonora
- Incapacidad de distinguir frecuencias próximas, siendo este fenómeno más acusado en la zona de altas frecuencias que en la zona de bajas frecuencias.

Para recoger ambos efectos, después de realizar la STFT de la señal, se aplica una conversión pseudologarítmica y un agrupamiento de las frecuencias por medio de un banco de filtros como los mostrados en la Figura 8.

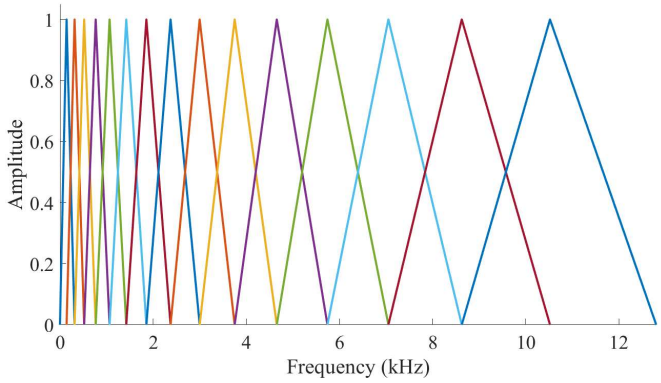

Figura 8: Banco de filtros de las frecuencias de Mel (Fuente: https://wiki.aalto.fi/display/ITSP/Cepstrum+and+MFCC)

Por último y, a fin de disminuir la correlación, se lleva a cabo una transformada discreta del coseno (DCT), resultando la conversión del espectrograma de partida en el gráfico MFCC final tal y como se puede ver en la Figura 9.

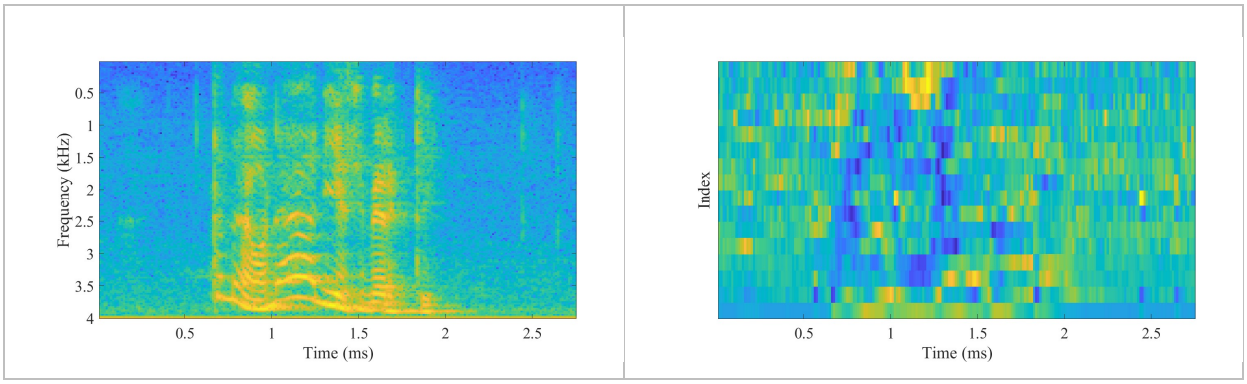

Figura 9: Espectrograma y MFCC (Fuente: https://wiki.aalto.fi/display/ITSP/Cepstrum+and+MFCC)

Habiendo descartado el análisis en el dominio del tiempo debido a la imposibilidad de implementarlo en un microcontrolador, se presenta a continuación el procedimiento seguido para llevar a cabo el análisis en el dominio de la frecuencia.

El desarrollo inicial se ha realizado empleando MFCC para la detección de fonemas vocálicos, pero, dado que para la fase de implementación en el microcontrolador se deseaba aprovechar el desarrollo presentado por (Warden & Situnayake, TinyML, 2019) tal y como se recoge en (Tang, 2019), finalmente se ha usado únicamente la información de la FFT por los motivos que se expondrán en el apartado 4.4. No obstante en la exposición que sigue se muestran ambos métodos con fines comparativos.

Teniendo en cuenta la información presentada sobre el tamaño de los fonemas vocálicos etiquetados y dado que el fonema vocálico etiquetado más corto tiene 248 muestras, se elige un tamaño de ventana de 256 muestras, lo que supone una duración de 16ms para una señal muestreada a 16kHz. Se ha redondeado a 256 muestras porque los algoritmos de cálculo FFT están optimizados para trabajar con un número de muestras que sea múltiplo de 2. Trabajando con ventanas de 256 muestras, el resultado de la transformada de Fourier de cada frame serán 128 valores complejos a partir de los que se extraerá el valor de amplitud. En cuanto a la longitud de salto se ha elegido un valor de 128 muestras lo que supone el 50% del tamaño de ventana y un tiempo de 8ms. Normalmente se suelen usar valores en torno al 30%, pero las pruebas realizadas no arrojan una mejora en el comportamiento de la detección y, sin embargo, usar una longitud de salto más corta implica realizar la FFT sobre un mayor número de frames lo que aumenta el coste computacional.

Se ha definido un tercer valor que corresponde con el número de coeficientes MFCC que se van a extraer. Dado que un paso previo a la obtención de estos coeficientes es la realización de la transformada de Fourier, y en este caso solo se dispone de 128 valores de transformada, eso limita el número de coeficientes MFCC y por ese motivo se ha elegido un valor bajo, ocho, concretamente. Nuevamente las pruebas realizadas no han arrojado mejoras en la detección incrementando el número de coeficientes

Con esos tres valores definidos se procede a realizar la STFT y a extraer los coeficientes MFCC por medio de la biblioteca Librosa en Python sobre los archivos de audio.

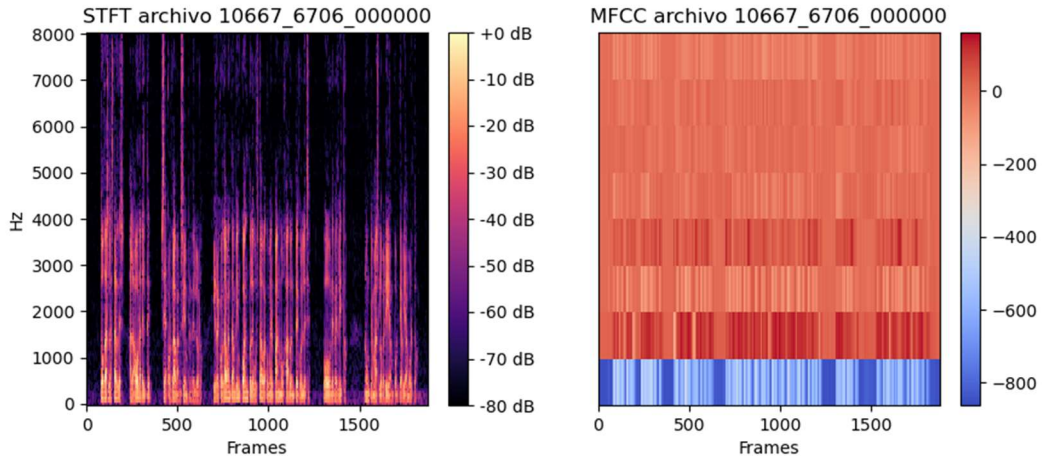

Figura 10: STFT y coeficientes MFCC

La Figura 10 recoge el resultado del procesamiento sobre uno de los archivos. El eje de abscisas recoge el orden de cada una de los frames en los que se subdivide la señal mediante el proceso de enventanado. Este archivo en concreto consta de 8.568 muestras lo que genera 1.875 frames, al aplicar un ancho de ventana de 256 muestras y una longitud de

salto de 128 muestras. La gráfica izquierda es el espectrograma de la señal y recoge los cambios de su contenido en frecuencia lo largo del tiempo. Dado que la señal de partida está muestreada a 16kHz, la frecuencia máxima que se puede registrar en el espectrograma es la mitad, 8kHz. La gráfica derecha presenta la información equivalente después de haber realizado el filtrado de Mel. Dado que se calculan 8 coeficientes, se pueden observar 8 bandas horizontales, que se corresponden con la evolución en el tiempo de cada uno de esos 8 coeficientes.

Los valores de FFT y MFCC así obtenidos son las características de entrada que se usan para los algoritmos de detección, pero es necesario asignar a cada frame una etiqueta que permita determinar si se corresponde o no con una vocal.

Esta asignación se logra superponiendo la información obtenida del etiquetado presentado en el apartado 4.1 con la secuencia de frames que se han definido en el proceso de enventanado utilizado para el cálculo de la STFT y los coeficientes MFCC.

La Figura 11 recoge esquemáticamente el método empleado. Se pueden observar dos fonemas consecutivos, /k/ y /o/, precedidos de un silencio. Para mayor claridad se han enmarcado los límites de los fonemas y del silencio. Se representan también los frames con la longitud de ventana y la longitud de salto elegidos, esto es, 256 y 128 muestras respectivamente. Esos valores hacen que los frames se superpongan unos a otros. Dentro de cada frame se muestra el fonema que se le asigna atendiendo a los siguientes criterios:

- Si todas las muestras del frame se encuentran dentro de una misma etiqueta de fonema de la base de datos de partida, se le asigna esa etiqueta
- Si las muestras del frame cubren varias etiquetas de fonema de la base de datos de partida, se le asigna una etiqueta de “mezcla”, representada arbitrariamente con el carácter ‘+’

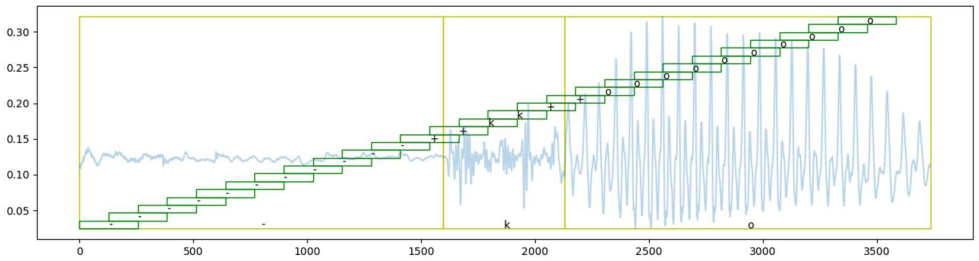

Figura 11: Etiquetado de frames

El resultado de la asignación de fonemas a cada frame quedaría para este ejemplo de la siguiente manera: para los primeros frames se asigna la etiqueta de silencio (identificada

con el carácter ‘-‘), seguidos de dos frames con etiqueta de mezcla, luego varios frames tipo /k/, otros dos frames de mezcla y el resto de frames tipo /o/.

La motivación de seguir este criterio de asignación es la siguiente. Todos los frames que quedan íntegramente dentro de un fonema no tienen por qué contener la misma información en frecuencia. Durante la locución del fonema, su señal evoluciona dependiendo del fonema que le precede y del fonema que le sucede, ya que se produce una adaptación a los distintos sonidos a emitir por parte de los elementos que intervienen en la fonación (faringe, laringe, cuerdas vocales, lengua, cavidad bucal…). Conceptualmente se puede simplificar diciendo que la locución del fonema implica una parte inicial “contaminada” del fonema precedente, una parte intermedia “pura” y una parte final “contaminada” por la preparación del fonema sucesor. La idea pues, es recoger toda esa variedad en el proceso de aprendizaje del algoritmo.

Dado que el objetivo es distinguir los fonemas vocálicos de cualquier otro sonido, ruido o silencio, solo se necesitan dos etiquetas del tipo “Vocal” / “No vocal”. No obstante, se ha preferido aprovechar el etiquetado completo de fonemas realizado en origen y conservar dicha información en el etiquetado de los frames, de manera que sea posible caracterizar de manera más pormenorizada los errores que cometen los algoritmos utilizados.

Aplicando este procedimiento a todos los frames en todos los archivos de audio se obtiene una base de datos que consta de un total de 15.131 frames. Cada frame contiene 128 valores correspondientes a su transformada de Fourier de 256 puntos, 8 valores correspondientes a los coeficientes MFCC y su etiqueta de fonema. Los fonemas resultantes se distribuyen en 4.284 frames vocálicos, 4.024 frames consonánticos, 3.895 frames de silencio o ruido (correspondientes a todas las regiones de la señal que no fueron etiquetadas con Audacity) y 2.928 frames de mezcla.

Dado que el interés fundamental es que los algoritmos aprendan las características de las señales vocálicas se ha reestructurado esta base de datos para definir la base de datos empleada en el entrenamiento. Se han seleccionado todos los frames vocálicos y se han añadido un número igual de frames no vocálicos. La composición de la parte no vocálica se ha repartido en un 25% de frames de mezcla, un 6,25% de frames de silencio y el resto frames consonánticos con una representación para cada consonante proporcional a la del conjunto de partida. El resultado es una base de datos para entrenamiento con 8.568 frames, mitad vocálicos y mitad no vocálicos y con la distribución mostrada en la Figura 12.

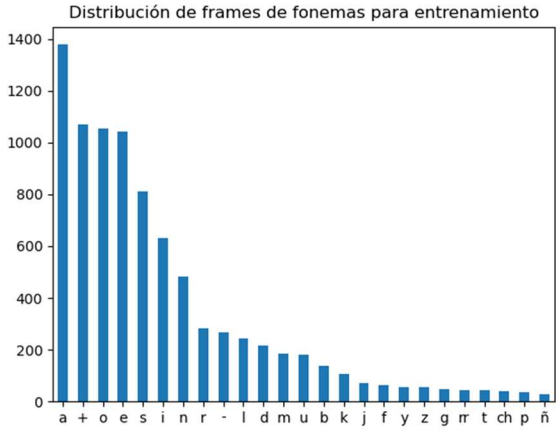

Figura 12: Distribución de fonemas en la base de datos de entrenamiento

Una vez definida la base de datos, se han planteado tres algoritmos para la tarea de clasificación: un clasificador tipo Random Forest, un clasificador tipo Support Vector Machine (SVC) y una red neuronal.

Tal y como se ha comentado, se han probado los algoritmos con los 128 parámetros de transformada de Fourier y en su forma transformada a través de los 8 coeficientes MFCC. En ambos casos se han obtenido resultados parecidos en cuanto a la precisión, pero el empleo de la transformada, al emplear un mayor volumen de parámetros, supone una mayor tendencia al sobreajuste y mayores requerimientos de almacenamiento y tiempo de proceso.

Los resultados obtenidos para el clasificador Random Forest con 20 estimadores y criterio de división tipo Gini son los recogidos en la Tabla 4 y, procediendo de manera análoga con el clasificador SVC, se obtienen los resultados mostrados en la Tabla 5.

Tabla 4: Informe de clasificación Random Forest

	MFCC			FFT			
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Support
No Vocal	0,86	0,84	0,85	0,86	0,84	0,85	857
Vocal	0,85	0,87	0,86	0,84	0,86	0,85	857
Accuracy			0,85			0,85	1.714
Macro Avg	0,85	0,85	0,85	0,85	0,85	0,85	1.714
Weighted Avg	0,85	0,85	0,85	0,85	0,85	0,85	1.714

Por último, se ha planteado una red neuronal en TensorFlow con una capa de entrada de 8 neuronas densamente conectadas con activación tipo Relu, una capa oculta con 16 neuronas densamente conectadas y activación Relu y una capa de salida con 2 neuronas densamente conectadas y activación Softmax, tal y como recogen la Tabla 6 y la Tabla 7 para el empleo de MFCC y FFT respectivamente.

Tabla 5: Informe de clasificación SVC

	MFCC			FFT			
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Support
No Vocal	0,87	0,82	0,85	0,82	0,83	0,82	857
Vocal	0,83	0,88	0,86	0,83	0,82	0,82	857
Accuracy			0,85			0,82	1.714
Macro Avg	0,85	0,85	0,85	0,82	0,82	0,82	1.714
Weighted Avg	0,85	0,85	0,85	0,82	0,82	0,82	1.714

Las pruebas realizadas con otras configuraciones han arrojado peores resultados en caso de disminuir el número de neuronas y un incremento en el sobreajuste en el caso de un mayor número de neuronas, por lo que se ha considerado que esta configuración es la más adecuada y ofrece una gran sencillez de cara a su posible implantación en un microcontrolador, ya que, al constar tan solo de 250 y 1.218 parámetros respectivamente, sus requerimientos de memoria son pequeños y sus tiempos de ejecución cortos.

Tabla 6: Estructura de red neuronal datos MFCC

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense     (Dense)            (None, 8)                 72

 dense_1   (Dense)            (None, 16)                144

 dense_2   (Dense)            (None, 2)                 34

=================================================================
Total params: 250
Trainable params: 250
Non-trainable params: 0
_________________________________________________________________
```

Tabla 7: Estructura de red neuronal datos FFT

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 Dense     (Dense)            (None, 8)                 1040

 dense_1   (Dense)            (None, 16)                144

 dense_2   (Dense)            (None, 2)                 34

=================================================================
Total params: 1,218
Trainable params: 1,218
Non-trainable params: 0
_________________________________________________________________
```

El entrenamiento de la red neuronal se ha llevado a cabo usando como métrica objetivo la exactitud y como función de pérdida Categorical Cross Entropy. Se han definido 50 epoch de entrenamiento y, para evitar el sobreajuste, se ha definido una interrupción anticipada en caso de no mejora del valor de la pérdida en el conjunto de validación durante dos epoch consecutivas. La Figura 13 recoge el proceso de entrenamiento para ambos modelos. En la parte superior, correspondiente al modelo MFCC, se puede observar que al llegar a 20 epoch se estanca la pérdida y el entrenamiento se detiene para evitar el sobreajuste.

Análogamente para el modelo FFT en la parte inferior el entrenamiento se detiene después de tan solo 15 epoch, ya que, al tener mayor cantidad de parámetros, el modelo tiende al sobreajuste antes. No obstante, como la precisión en el conjunto de entrenamiento y de validación son parecidas se puede concluir que no ha habido sobreajuste.

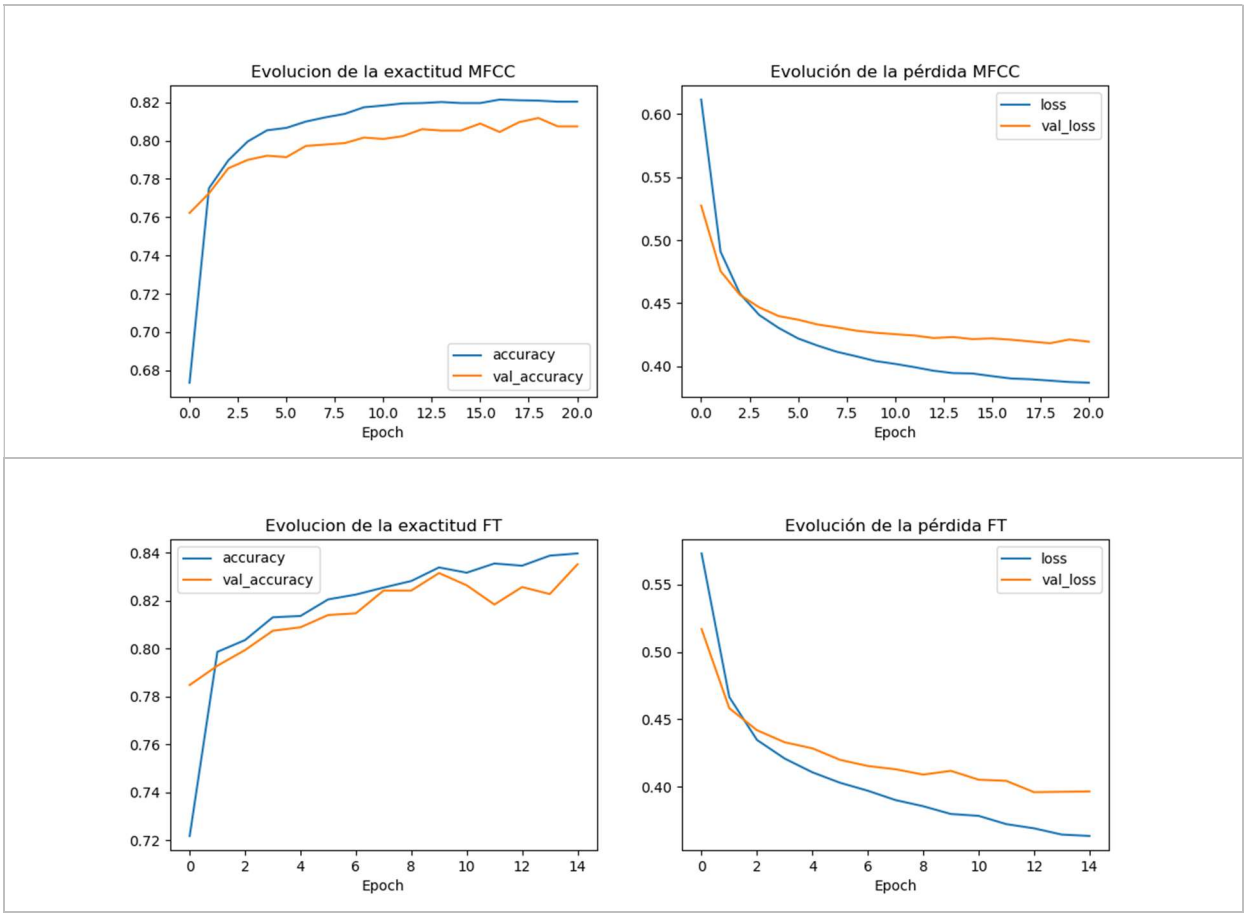

Figura 13: Progreso del entrenamiento de la red neuronal

Los resultados arrojados por la red neuronal están recogidos en la Tabla 8 y con fines ilustrativos se incluye la Figura 14 que recoge las matrices de confusión de los tres algoritmos. En ellas se puede ver fácilmente que el error más frecuente de los algoritmos, sobre todo en el caso de la red neuronal consiste en clasificar como vocales fonemas que no lo son.

Tabla 8: Informe de clasificación de la red neuronal

	MFCC			FFT			
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Support
No Vocal	0,86	0,79	0,83	0,87	0,79	0,83	857
Vocal	0,81	0,87	0,84	0,81	0,88	0,84	857
Accuracy			0,83			0,84	1.714
Macro Avg	0,83	0,83	0,83	0,84	0,84	0,84	1.714
Weighted Avg	0,83	0,83	0,83	0,84	0,84	0,84	1.714

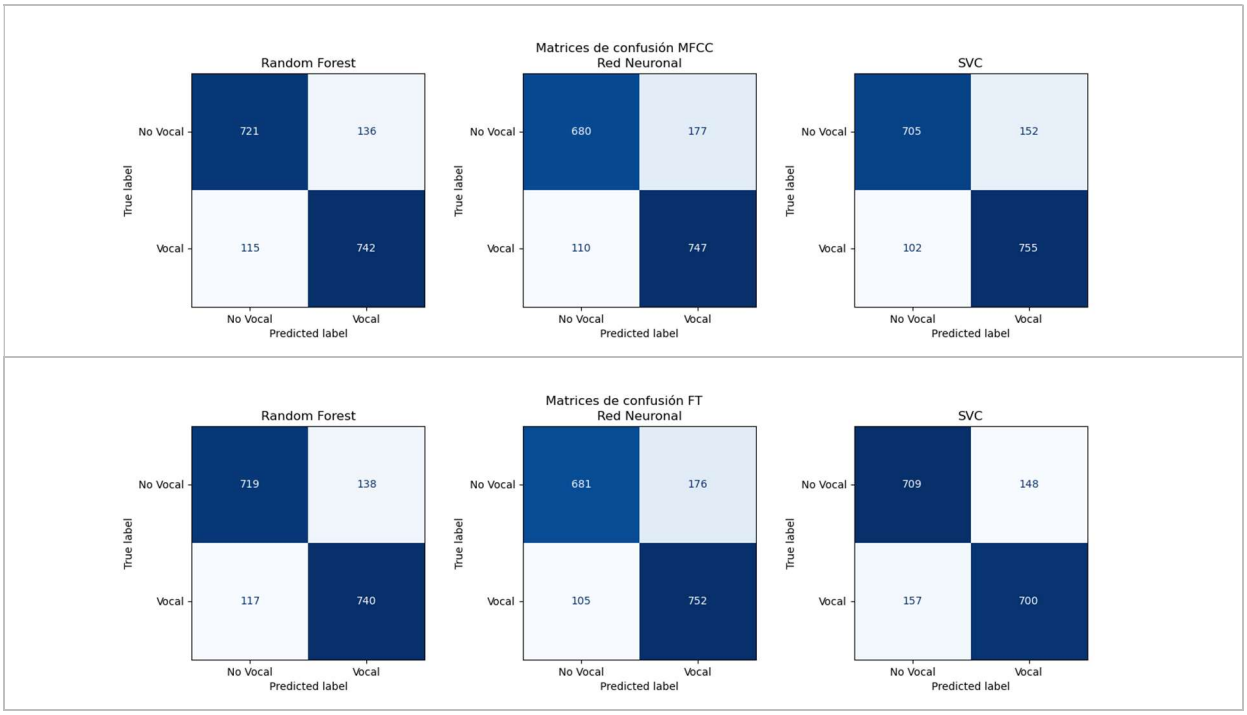

Figura 14: Matrices de confusión de los algoritmos

De la comparativa de los valores de las tres tablas con los informes de clasificación, se deduce que los tres algoritmos tienen un comportamiento muy parecido con una ligera desventaja por parte de la red neuronal. Pero a la hora de elegir el algoritmo a implementar en el microcontrolador es necesario tener en cuenta también el tiempo de procesamiento necesario. A tal fin, se ha llevado a cabo el cálculo del tiempo que necesita cada uno de los algoritmos en realizar las predicciones sobre el conjunto de datos de validación resultando los valores recogidos en la Tabla 9.

Tabla 9: Tiempos de procesamiento de algoritmos

	MFCC	FFT
Random Forest	0,584s	2,796s
Red Neuronal	0,369s	0,396s
SVC	2,925s	9,938s

El valor de estos tiempos no es relevante en sí mismo ya que es dependiente del tamaño del conjunto de validación y de la máquina en la que se ejecute, pero la comparativa entre ellos nos muestra que el algoritmo más rápido es la red neuronal. Se evidencia también el importante incremento del tiempo de ejecución al pasar de emplear MFCC a FFT, llegándose a multiplicar por un factor de 4 para Random Forest y SVC, aunque prácticamente no tiene efecto sobre la red neuronal. Lo anterior justifica el empleo de la red neuronal como algoritmo de detección de fonemas vocálicos a implementar en el microcontrolador.

Por último, y aprovechando que se disponen de las etiquetas del fonema de cada frame se ha realizado un análisis de los errores de clasificación en función del fonema real. La Figura 15 muestra el porcentaje de fonemas clasificados incorrectamente dentro del conjunto de datos de validación. Tal y como se ha observado en las matrices de confusión, el mayor número de errores corresponde a sonidos no vocálicos que se clasifican como vocálicos. Dentro de estos, uno de los grupos más relevantes corresponde a los fonemas de tipo mezcla probablemente debido a que un gran número de estos fonemas tendrán parte de contenido vocálico, lo que genera una información en frecuencia difícil de distinguir para los algoritmos de un fonema vocálico puro.

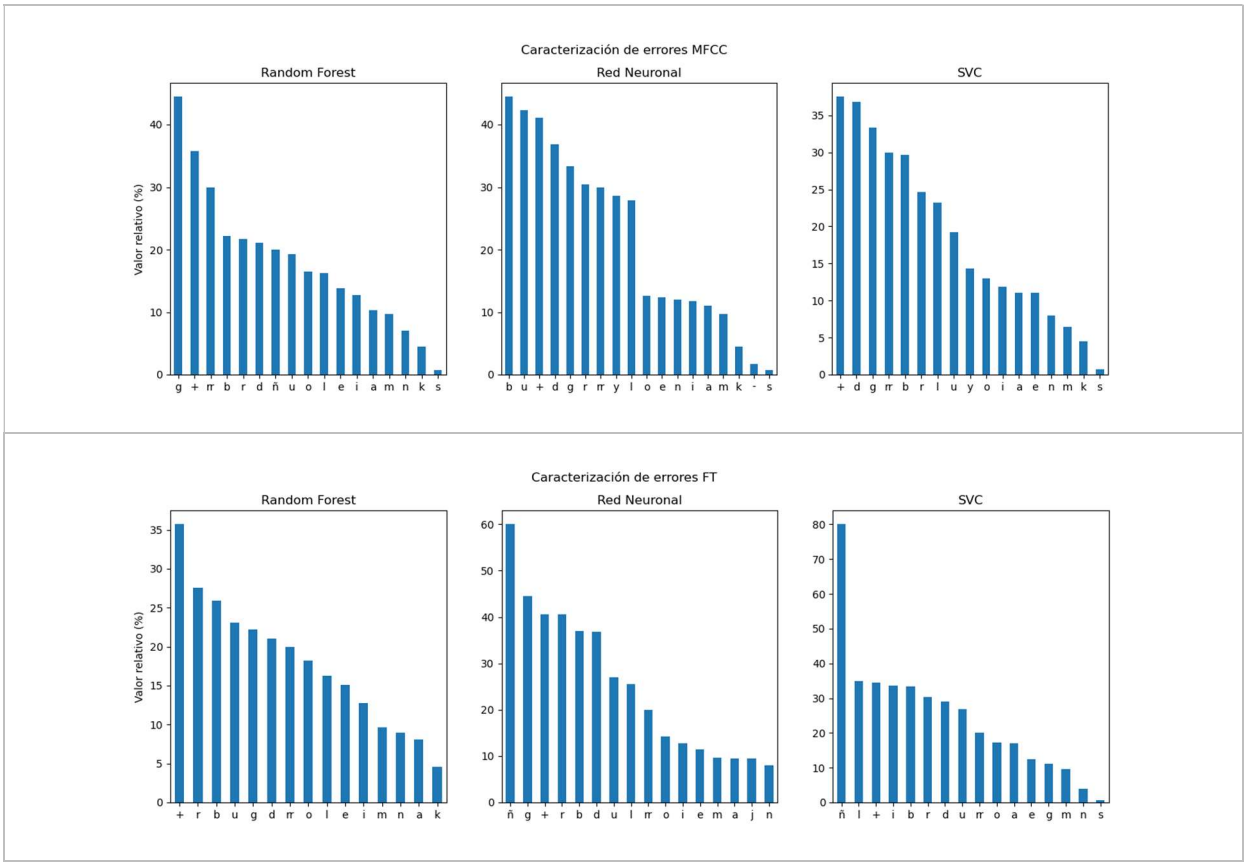

Figura 15: Caracterización de errores de predicción

4.3. Algoritmo VAD

Una vez desarrollado el algoritmo para detectar fonemas vocálicos se hace necesario establecer los criterios para determinar las regiones de la señal analizada en las que hay presencia de voz.

Para ello y, teniendo en cuenta lo expuesto en el punto 2.4, se ha desarrollado un sencillo algoritmo VAD basado en la premisa básica de que durante el acto del habla se produce

siempre una alternancia de tramos con contenido vocálico y tramos con contenido consonántico. Los tramos de contenido vocálico están formados por fonemas vocálicos consecutivos y, de igual forma, los tramos de contenido consonántico están formados por fonemas no vocálicos consecutivos. Con esta premisa, el funcionamiento del algoritmo VAD se reduce a localizar los tramos vocálicos, en los cuales la presencia de voz está garantizada, y suponer que la presencia de voz se extiende más allá de dicho tramo durante un cierto tiempo que se correspondería con un tramo consonántico.

A fin de determinar dicho tiempo se ha llevado a cabo una caracterización sobre la base de datos original, determinando las duraciones de los tramos vocálicos y consonánticos consecutivos y analizando la relación existente entre ellas. La distribución de esa relación queda reflejada en la Figura 16, arrojando un valor medio de 1,22 y un tercer cuartil de 1,64.

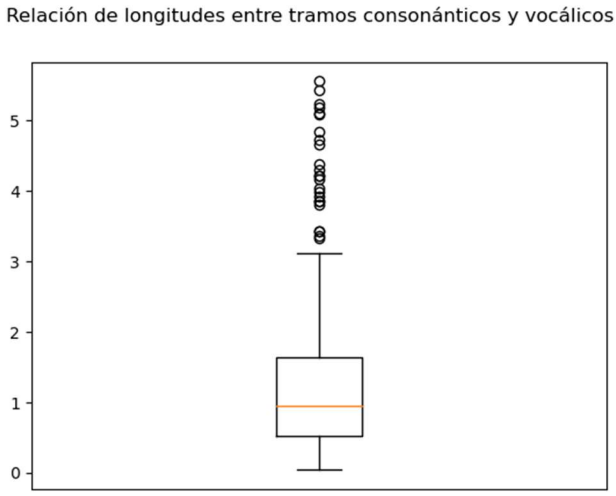

Figura 16: Proporción en la duración tramo consonante/vocal

En base a esos datos se ha optado por prolongar la detección de voz un valor 1,64 veces mayor que el último tramo vocálico detectado. El algoritmo VAD quedaría por tanto definido de la siguiente manera:

1. Al inicio de un tramo vocálico se activa la salida VAD.
2. Al finalizar un tramo vocálico, se supone que se inicia un tramo consonántico y se mantiene activa la salida VAD durante un tiempo 1,64 veces más largo que la longitud del tramo vocálico anteriormente detectado.
3. Al finalizar el tiempo correspondiente al hipotético tramo consonántico se desactiva la salida VAD.

El resultado de la ejecución del algoritmo VAD así definido sobre una porción de uno de los archivos etiquetados usados en el entrenamiento ofrece los resultados mostrados en la Figura 17 tanto usando MFCC como FFT. Se muestra la señal de audio en azul,

sobreimpresas en rojo las zonas que la red neuronal identifica como vocales y una marca de color verde en la zona superior que se corresponde con la salida del algoritmo VAD indicando las zonas en las que se estima la presencia de voz.

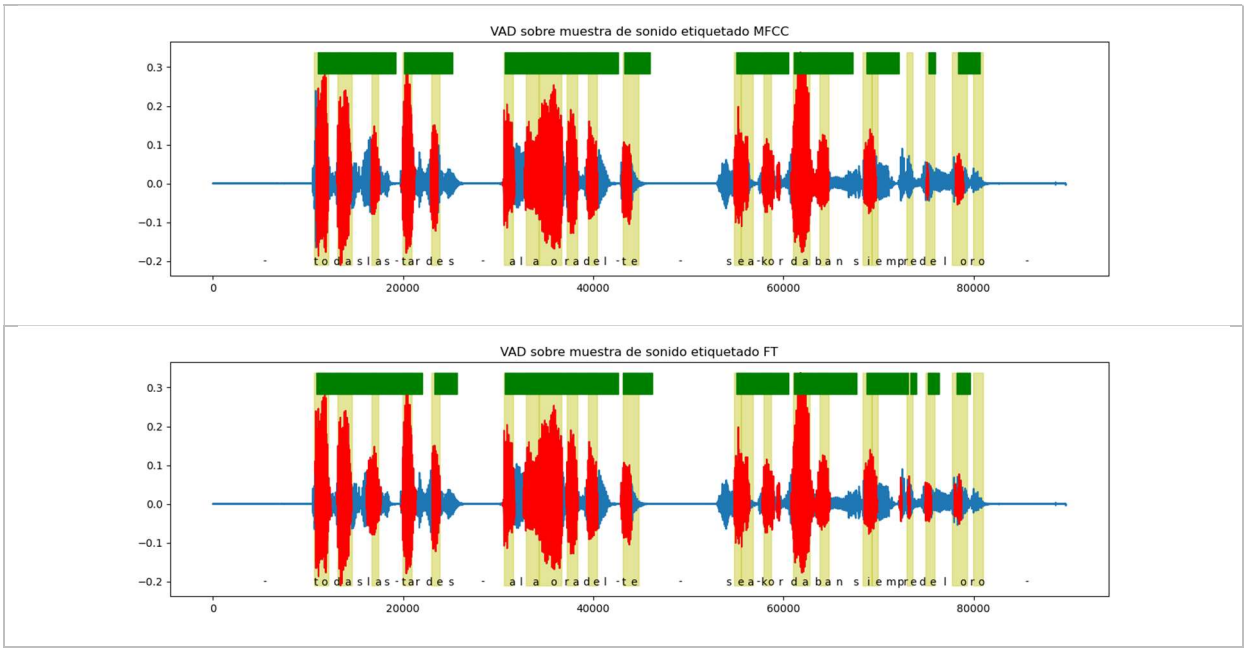

Figura 17: Algoritmo VAD sobre muestra etiquetada

Sobre esta porción de audio se ha llevado a cabo el análisis del error cometido por el algoritmo VAD. La Tabla 10 recoge el informe de clasificación y la Figura 18 muestra gráficamente la matriz de confusión para cada caso.

Tabla 10: Informe de clasificación del algoritmo VAD

	MFCC			FFT			
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Support
No Voz	0,67	0,88	0,76	0,69	0,85	0,76	26.618
Voz	0,93	0,79	0,85	0,92	0,81	0,86	54.384
Accuracy			0,82			0,83	81.002
Macro Avg	0,80	0,83	0,81	0,80	0,83	0,81	81.002
Weighted Avg	0,84	0,82	0,82	0,84	0,83	0,83	81.002

Los errores de clasificación se han determinado para cada muestra independientemente en la porción de audio de 81.002 muestras, teniendo en cuenta que gracias al etiquetado se sabe cuales pertenecen a zonas de voz y cuales no. Los errores de clasificación se pueden deber a:

- Retardo en la activación tras la detección de un fonema vocálico: la seña VAD se activa tras la detección de un primer frame clasificado como vocal, con lo que las muestras de ese primer frame no se identifican como voz.

- Fallos en la detección de fonemas vocálicos: los frames vocálicos no detectados implican no detectar voz y los frames no vocálicos incorrectamente clasificados como vocálicos puede provocar una falsa detección de voz en aquellos casos que la señal no se corresponda con un sonido consonántico (ruidos y sonidos no correspondientes al habla).

- Inicio de locución con fonemas consonánticos: si la palabra comienza como una consonante, el algoritmo VAD no activará la salida hasta que no detecte la primera vocal.

- Fin de locución en fonema vocálico: si la palabra finaliza en vocal y ésta es correctamente detectada, el algoritmo mantendrá activa la salida VAD durante el tiempo establecido por la ratio consonante/vocal que se ha establecido.

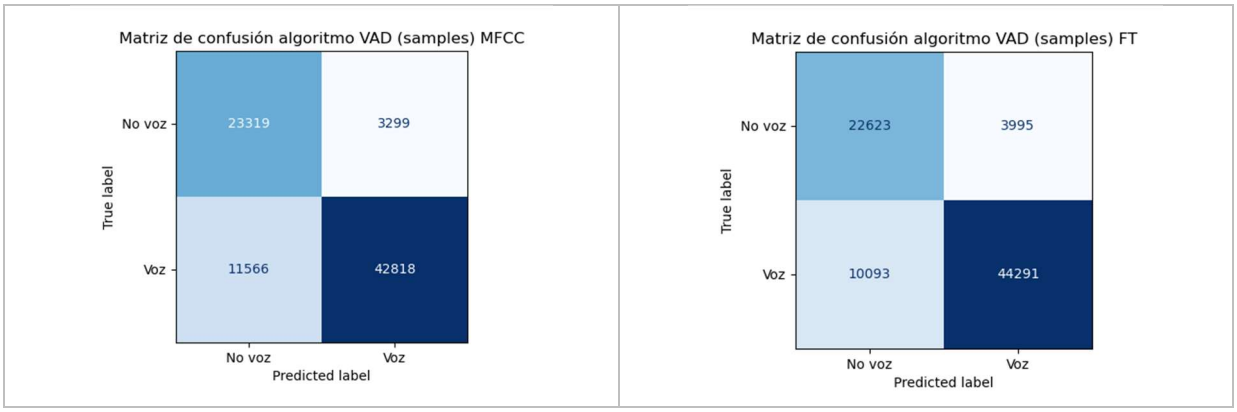

Figura 18: Matrices de confusión para el algoritmo VAD

Del análisis de estos resultados se puede concluir que la capacidad de detección de voz del algoritmo resulta algo limitada y el error más frecuente consiste en clasificar como “No voz” tramos de audio en los que existe voz. Por otro lado, la comparativa entre el empleo de MFCC y FFT parece indicar que, si bien en esta porción de audio el algoritmo basado en FFT ofrece un mejor comportamiento en su conjunto, tiene una mayor tendencia a clasificar como voz tramos en los que no existe locución y eso podría suponer que fuera menos robusto frente al ruido y otras perturbaciones.

A fin de verificar cómo se comporta el algoritmo frente a dichas perturbaciones se ha procedido a experimentar con una muestra de audio en castellano extraída de Multilingual LibriSpeech con una locución clara y con la misma muestra contaminada por una señal de ruido blanco, por una señal de sonido de fondo de un restaurante y por una señal de sonido con trinos de pájaros.

El resultado gráfico se recoge en la Figura 19 y Figura 20.

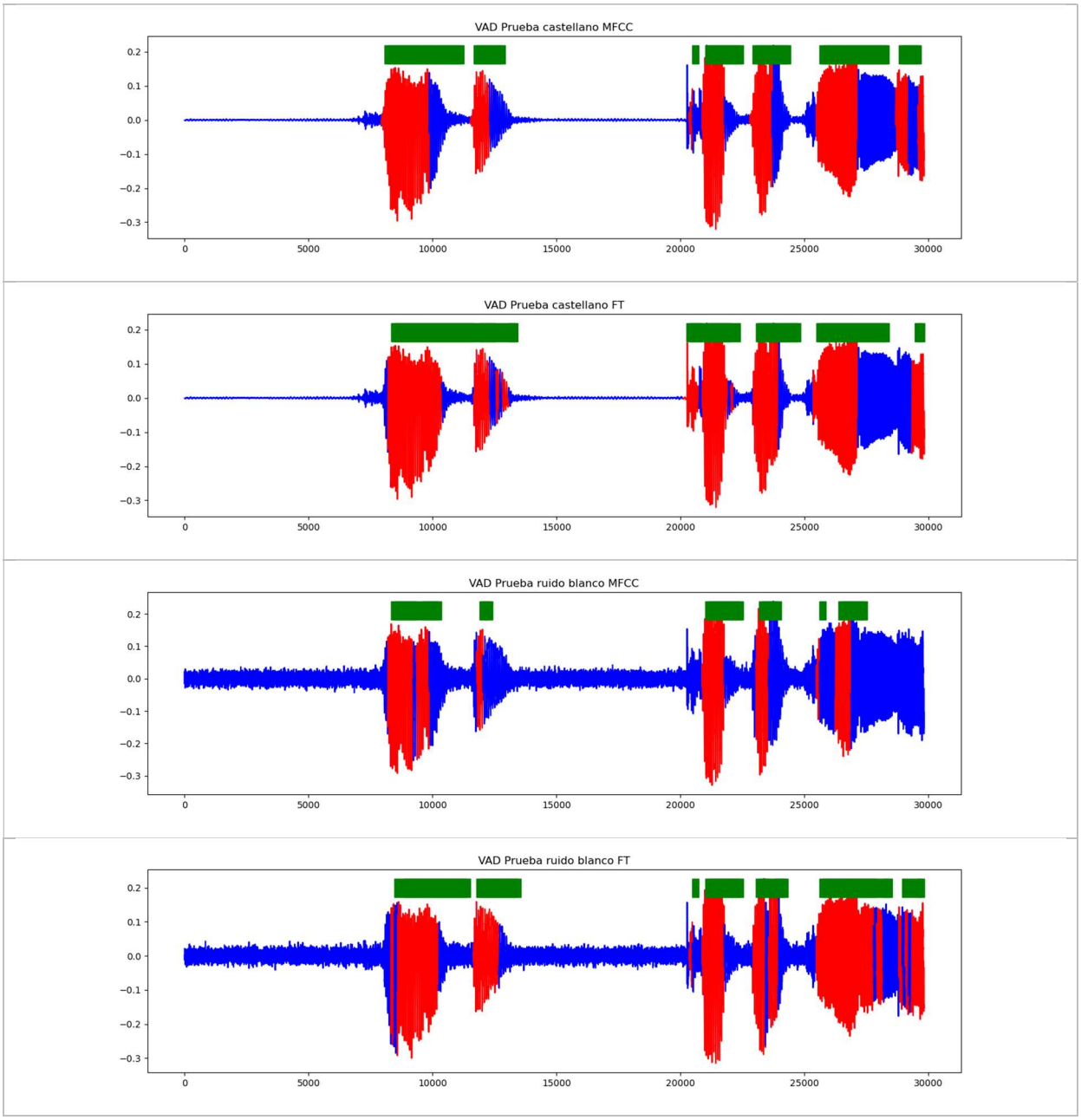

Figura 19: Pruebas algoritmo VAD (parte 1)

En los dos primeros gráficos de la Figura 19 se observa el comportamiento del algoritmo ante la señal en castellano sin contaminar, siendo muy similar en ambos con una pequeña ventaja para el caso de empleo de FFT. Los dos últimos gráficos de la Figura 19 muestran el comportamiento del algoritmo si se contamina la señal superponiendo un ruido blanco. Se aprecia que la detección en el algoritmo basado en MFCC sufre una caída importante y, sin embargo, el algoritmo basado en FFT, se comporta de manera más robusta. El motivo de este comportamiento se debe al hecho de que un ruido blanco presenta un espectro aproximadamente plano, de manera que el efecto que tiene sobre la FFT de la señal de partida es fundamentalmente una modificación en su amplitud variando poco su perfil. Pero al aplicar el banco de filtros a la FFT para obtener los MFCC, el efecto del ruido es desigual

en los distintos coeficientes, por lo que la distorsión de los MFCC respecto a los de la señal sin contaminar general problemas en el algoritmo VAD.

Esta aparente ventaja del algoritmo basado en FFT frente a un ruido blanco, se convierte en una desventaja cuando la perturbación consiste en un sonido con un contenido en frecuencia que no sea plano.

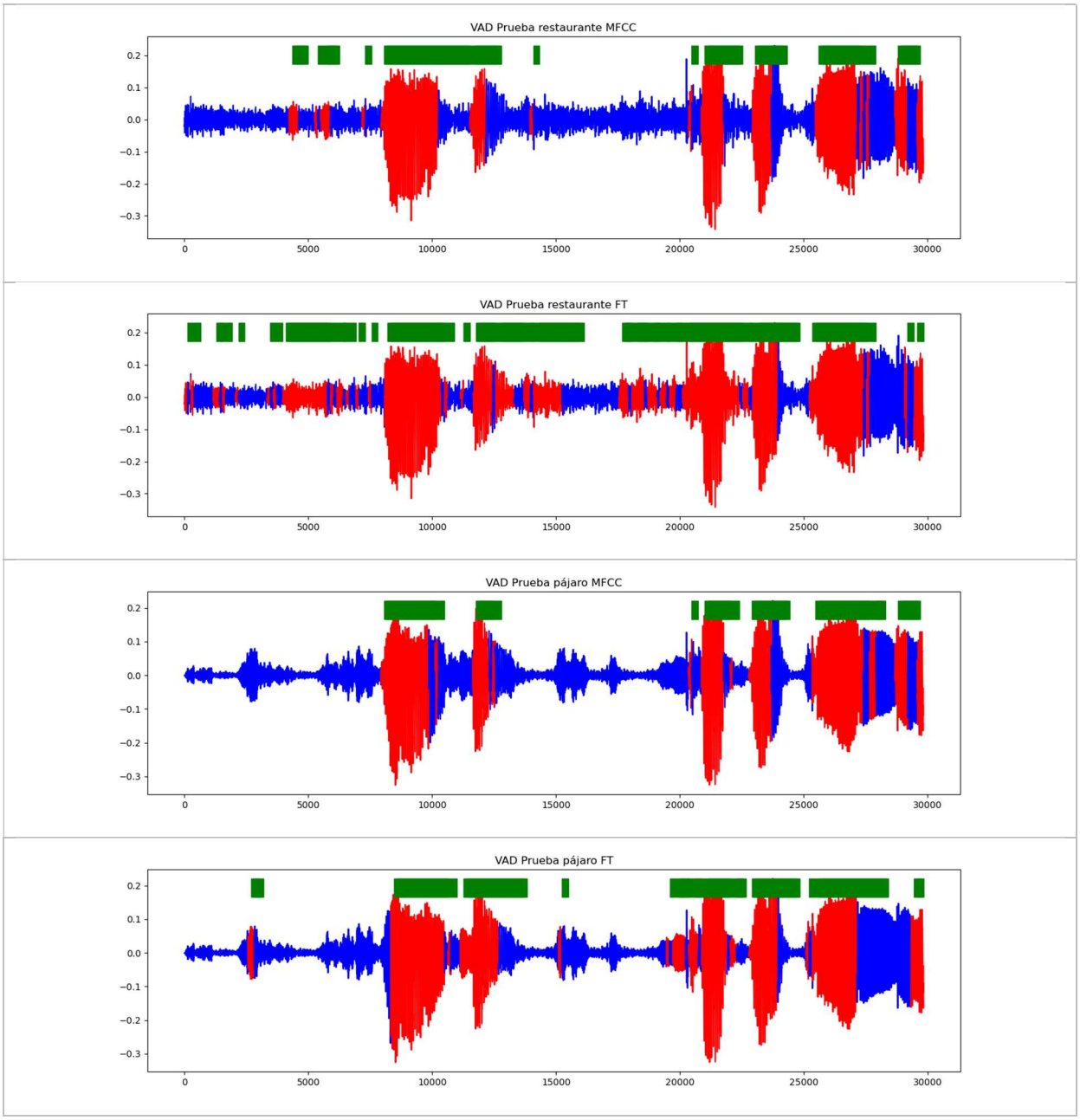

Figura 20:: Pruebas algoritmo VAD (parte 2)

En la Figura 20 se comprueba la mayor robustez del algoritmo basado en MFCC frente a un sonido de fondo de un restaurante y el trino de un pájaro, siendo este último especialmente llamativo ya que filtra todos los sonidos correspondientes al pájaro. En este caso, el banco de filtros aporta una mejora significativa en la detección del algoritmo VAD.

Por último, se ha realizado una prueba sobre un archivo de audio en otro idioma. Se ha usado un archivo de audio en polaco también extraído de Multilingual LibriSpeech. El resultado se recoge en la Figura 21. Como es lógico al variar el contenido fonético del idioma los resultados son significativamente peores.

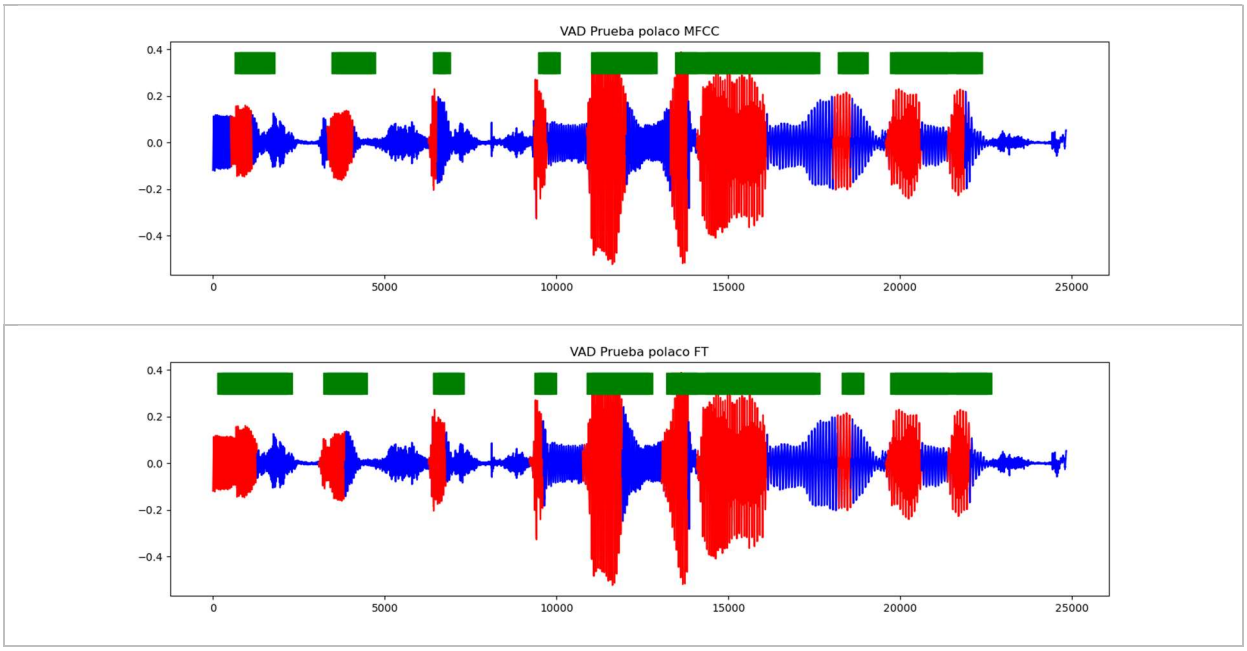

Figura 21: Prueba algoritmo VAD sobre locución en polaco

4.4. Implementación en Arduino

Tal y como se ha comentado en el apartado 4.2, aunque el desarrollo inicial se ha realizado empleando MFCC para la detección de fonemas vocálicos, finalmente para la fase de implementación en el microcontrolador se ha optado por la variante que hace uso de la FFT a fin de aprovechar parte del desarrollo presentado por (Warden & Situnayake, TinyML, 2019) tal y como se recoge en (Tang, 2019). Dicho desarrollo presenta una implementación sobre Arduino 33 BLE Sense de un sistema de reconocimiento de palabras clave (“yes” y “no”) basado en TensorFlow Lite para microcontroladores. La Figura 22 recoge los cinco componentes de que consta dicha aplicación: una función de captura de audio a través del micro incorporado en la placa, una función de extracción de las características de la señal de audio usadas como entrada a la red neuronal, un intérprete TF Lite que realiza la inferencia y establece la salida de la red neuronal, una función de reconocimiento de comando que establece si se ha detectado una palabra clave y, finalmente, una función que genera la respuesta apropiada en función del comando reconocido que, en este caso, consiste en encender un led verde si se reconoce la palabra “yes” y un led rojo si se reconoce la palabra “no”.

Esta implementación toma como punto de partida el modelo de reconocimiento de audio presentado por el equipo de (Tensorflow, 2020) y la base de datos (Warden, Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition, 2018), en la que se hace uso de espectrogramas y una red convolucional para llevar a cabo el reconocimiento. Desde ese punto de partida, realizan un modelo simplificado con tan solo una capa convolucional seguida de una capa densamente conectada y una salida softmax con un total de 16.652 parámetros que requieren menos de 17KB de memoria.

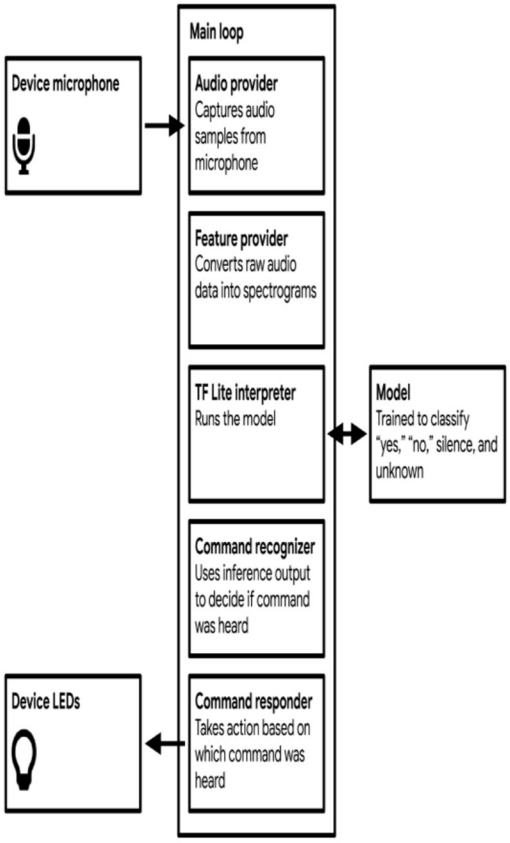

Figura 22: Componentes de la aplicación de reconocimiento de palabras clave. Fuente: (Warden & Situnayake, TinyML, 2019)

La entrada a la red neuronal se obtiene llevando a cabo un procesamiento de 1 segundo de señal de audio, al que se le aplica una STFT, seguido de un filtrado por medio de un banco de filtros similares a los de Mel con un promediado de frecuencias y, por último, un escalado logarítmico. El procedimiento es similar al que se sigue para la obtención de los MFCC tal y como se ha expuesto en el apartado 4.2 sin llevar a cabo la DCT final, y, tal y como indican los autores (Warden & Situnayake, TinyML, 2019) es el procedimiento que se ha usado internamente en Google en sus sistemas de reconocimiento de voz, aunque no se encuentre publicado.

Tres son los motivos por los que esta implementación no es válida para el algoritmo VAD que se analiza en este TFM. En primer lugar, al tratar con fonemas cuya duración se mide

en el orden de unos pocos milisegundos en lugar de los segundos que se usan en el reconocimiento de palabras, no tiene sentido aplicar una STFT para reconocimiento de fonemas ya que no se dispone de suficiente número de muestras. La única forma de incrementar el número de muestras sería elevar la frecuencia de muestreo, pero esto dispara los requerimientos de memoria y de procesamiento y no sería viable en un entorno Tiny. El no disponer de una imagen tipo espectrograma como entrada a la red neuronal hace que la opción de una red convolucional no sea interesante, motivo por el cual se ha optado por una red neuronal mucho más sencilla.

El segundo motivo que dificulta reutilizar completamente la implementación (Tang, 2019) para el algoritmo VAD tiene que ver con entrenamiento de la red neuronal. La red neuronal tendría que ser entrenada con una función de extracción de características del audio idéntica a la usada por el equipo de Tensorflow y ésta no es directamente accesible o, al menos, no sin llevar a cabo un cierto trabajo de ingeniería inversa. Aunque el desarrollo de Tensorflow presenta similitudes con los procedimientos normales de obtención de MFCC, no son idénticos, motivo por el que se ha descartado su empleo como características de entrada a la red neuronal.

El último motivo viene determinado por los tiempos de procesamiento. El equipo de Tensorflow, en el ejemplo desarrollado, realizan una STFT con ancho de ventana de 30ms y un desplazamiento de ventana de 20ms. En el algoritmo VAD planteado se analizan ventanas de 16ms pero con un desplazamiento de 8ms lo que supone un incremento en los tiempos de procesamiento necesarios. En estas circunstancias plantear el cálculo de los valores MFCC (incluyendo la etapa final DCT) para alimentar la red neuronal en el microcontrolador, además de una complicación técnica supondría una sobrecarga de procesamiento que haría inviable el funcionamiento del algoritmo VAD, toda vez, que la red neuronal debe ejecutarse al final del análisis de cada ventana.

Por lo anteriormente expuesto se ha adaptado el desarrollo de (Tang, 2019), modificando la red neuronal, simplificando la función de extracción de características de manera que solo se lleva a cabo el cálculo de la FFT y la obtención de sus módulos y adaptando las funciones de reconocimiento de comandos y de respuesta para la detección de fonemas vocálicos y de activación/desactivación de zonas VAD.

Para poder implementar redes neuronales en dispositivos pequeños, Tensorflow ha desarrollado la herramienta TFLite Converter que permite, a partir de un modelo neuronal completo, extraer una versión compacta que puede posteriormente ser ejecutada en el dispositivo de destino por medio de un intérprete. En el caso de que el dispositivo final sea un microcontrolador, TFLite Converter permite, además, aplicar un proceso de cuantización,

de manera que el modelo se almacena en enteros de 8 bits en lugar de los 32 bits sin cuantización, y las operaciones necesarias se realicen en punto fijo en lugar de en coma flotante. De esta forma se reduce el espacio necesario para almacenar el modelo en memoria en un factor de 4 y se reducen los tiempos de procesamiento al evitar el empleo de las operaciones en coma flotante.

Esta cuantización no supone pérdida en el rendimiento de los modelos. Para comprobarlo se ha llevado a cabo una comparativa de las métricas obtenidas para la red neuronal original, la red compacta TFLite sin cuantización y la red compacta TFLite con cuantización sobre el conjunto de datos de validación, cuyos resultados se recogen en la Tabla 11.

Tabla 11: Informe de clasificación modelos TFLite

	Original			Sin cuantización			Con cuantización			
	Precis	Recall	F1	Precis	Recall	F1	Precis	Recall	F1	Supp
No Vocal	0,83	0,80	0,81	0,83	0,80	0,81	0,82	0,80	0,81	857
Vocal	0,80	0,84	0,82	0,80	0,84	0,82	0,81	0,82	0,82	857
Accuracy			0,82			0,82			0,81	1.714
Macro Avg	0,82	0,82	0,82	0,82	0,82	0,82	0,81	0,81	0,81	1.714
Weighted Avg	0,82	0,82	0,82	0,82	0,82	0,82	0,81	0,81	0,81	1.714

Como se puede observar el comportamiento de los tres modelos es prácticamente idéntico. En cuanto al tamaño final de los modelos TFLite basados en FFT es de 6,74KB para el modelo sin cuantización y de tan solo 3,98KB para el modelo con cuantización. A partir del modelo TFLite con cuantización se extrae un archivo de texto plano que define un array en C con el que se puede compilar el modelo en binario que se implementa en el microcontrolador.

Las adaptaciones llevadas a cabo sobre la implantación original de (Tang, 2019) son las siguientes:

- El archivo de ejecución de Arduino VAD.ino, responsable de reservar las zonas de memoria necesarias tanto para el modelo TFLite como para las funciones de adquisición de audio y extracción de características ha requerido modificaciones mínimas siendo la más relevante la que define las operaciones presentes en la red neuronal, que en el caso del algoritmo VAD son AddRelu(), AddFullyConnected(), AddSoftmax() y AddQuantize(). Las tres primeras derivan de las características elegidas al definir la red del modelo y la última deriva del hecho de haber usado un modelo cuantizado.

- El archivo micro_features_model.cpp que contiene el array que define el modelo tal y como se ha extraído gracias a TFLite Converter y su posterior conversión a texto plano.

- En los archivos micro_features_micro_model_settings (.h y .cpp) se han modificado los parámetros relativos al proceso de enventanado, el tamaño del vector de entrada a la red neuronal y las categorías de salida (vocal y no vocal)

- En el archivo micro_features_micro_features_generator.cpp se ha modificado para escalar la salida calculada por la FFT de audio recibido y adaptarla a los valores de la FFT del conjunto de datos de entrenamiento. El valor de escalado se ha calculado experimentalmente comparando los resultados obtenidos en uno y otro caso, tomándose como valor final 25. Así mismo se ha realizado el ajuste para convertir los valores en enteros de 8 bits sin signo, tal y como se necesita para la entrada a la red neuronal TFLite.

- El archivo recognize_commands.cpp se ha simplificado para determinar simplemente el resultado de la inferencia del tipo de fonema recibido y para la aplicación del algoritmo VAD tal y como se ha definido en los apartados anteriores.

- El archivo arduino_command_responder.cpp se ha modificado para actuar sobre los leds de la placa, de manera que verde implica tramo vocálico (detectado por la red) y azul implica tramo consonántico (supuesto por el algoritmo VAD).

- El archivo frontend.c se ha modificado de manera que solo realiza el cálculo de la FFT y de su módulo.

- El archivo filterbank.c que contiene funciones usadas por el anterior se ha modificado con idéntico objetivo.

Todas estas adaptaciones se pueden consultar en el enlace presentado en el Anexo II.

A fin de verificar el correcto funcionamiento de la captura de datos de sonido y la extracción de características, esto es, el cálculo de la FFT, se ha aprovechado la posibilidad de comunicación por puerto serie de la placa de Arduino y la presencia en el IDE de Arduino de un sistema de representación gráfica de los valores recibidos por el puerto serie. Al final de cada ciclo de cálculo de FFT, se han transmitido los 129 valores calculados para su representación. Esta funcionalidad solo se ha utilizado durante la verificación y se ha eliminado de la versión definitiva, ya que dispara el consumo de recursos del microcontrolador e introduce un retraso que lo hace impracticable.

Primeramente, se ha realizado una prueba usando una onda sonora senoidal con una frecuencia de 4KHz. Dado que se usa una frecuencia de muestreo de 16KHz, la FFT

muestra la información para las frecuencias comprendidas entre 0 y 8KHz y, por tanto, una onda senoidal pura de 4KHz debe generar un único pico en la zona central.

Este pico se puede ver claramente en la Figura 23. En esta figura se representa en azul con valores crecientes entre 0 y 128, las posiciones del valor de FFT calculados en cada ciclo (value 1) y en naranja los valores de FFT efectivamente calculados (value 2).

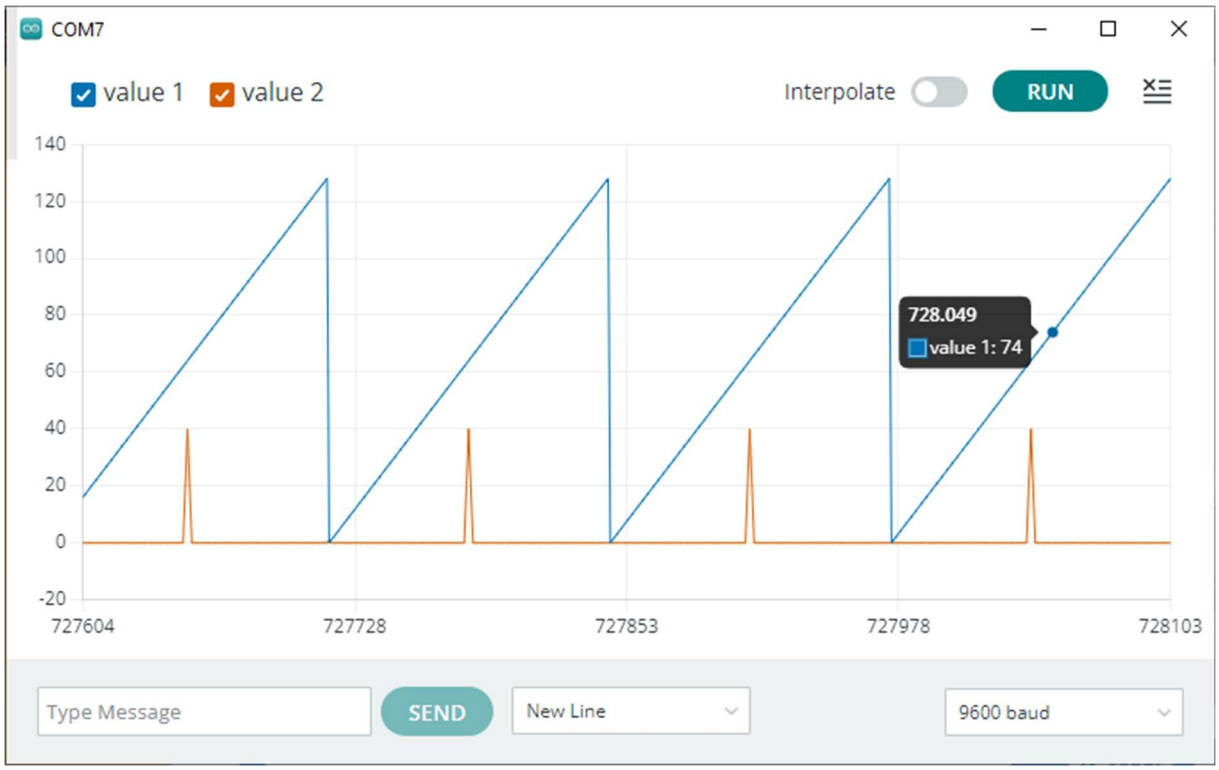

Figura 23: FFT para onda senoidal de 4KHz

Una segunda prueba realizada ha consistido en emplear una superposición de 3 ondas senoidales de 2, 4 y 6KHz respectivamente. El resultado recogido en la Figura 24 muestra la presencia de los tres picos correspondientes a las tres frecuencias mencionadas. La amplitud de cada una de las senoides queda reflejada por el valor de ordenadas del pico correspondiente.

Por último, se ha realizado una prueba con una locución del fonema /a/. El resultado se puede observar en la Figura 25. En este caso, en lugar de uno o varios picos, al tratarse de una señal con una mayor riqueza de frecuencias constituyentes se obtiene un perfil mucho más complejo. Este tipo de señales es lo que alimentará la entrada a la red neuronal para realizar la inferencia.

Figura 24: FFT para onda senoidal de 2KHz+4KHz+6KHz

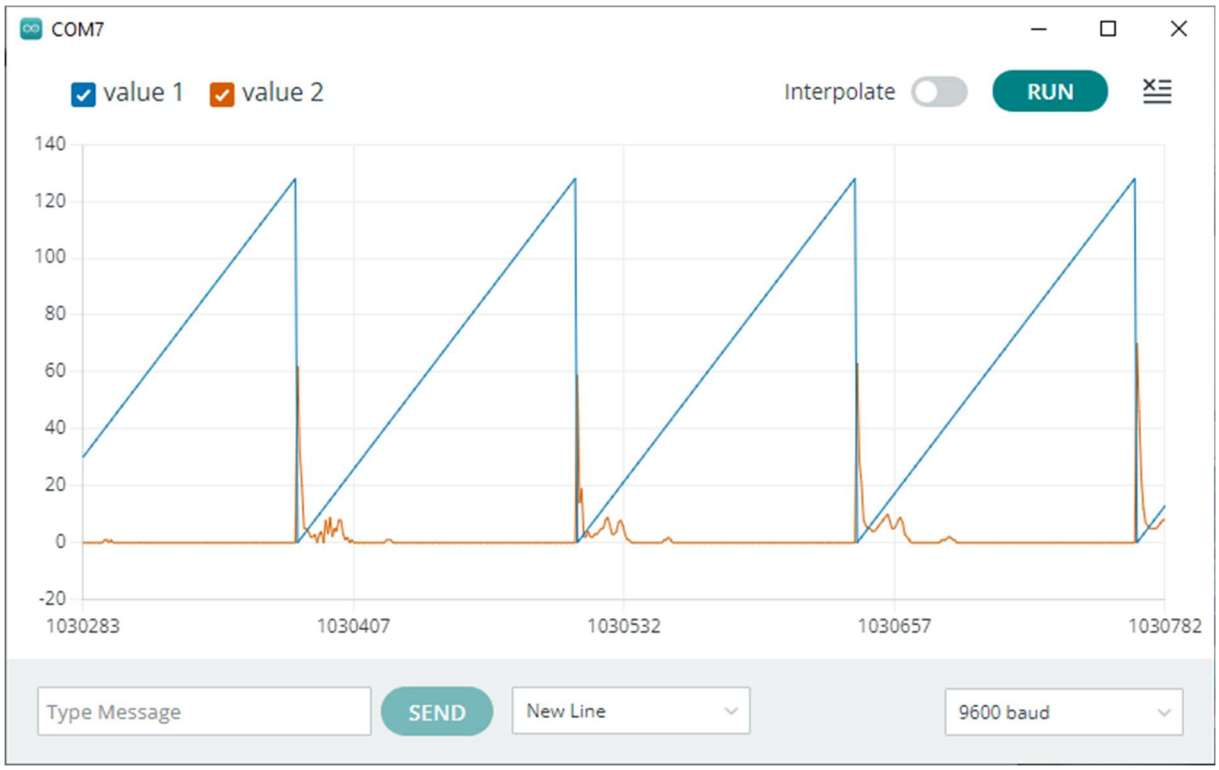

Figura 25: FFT del fonema /a/

Realizadas las modificaciones descritas anteriormente sobre el software original, compilado el programa y volcado en el microcontrolador la presencia de sonidos genera las respuestas mostradas en la Figura 26.

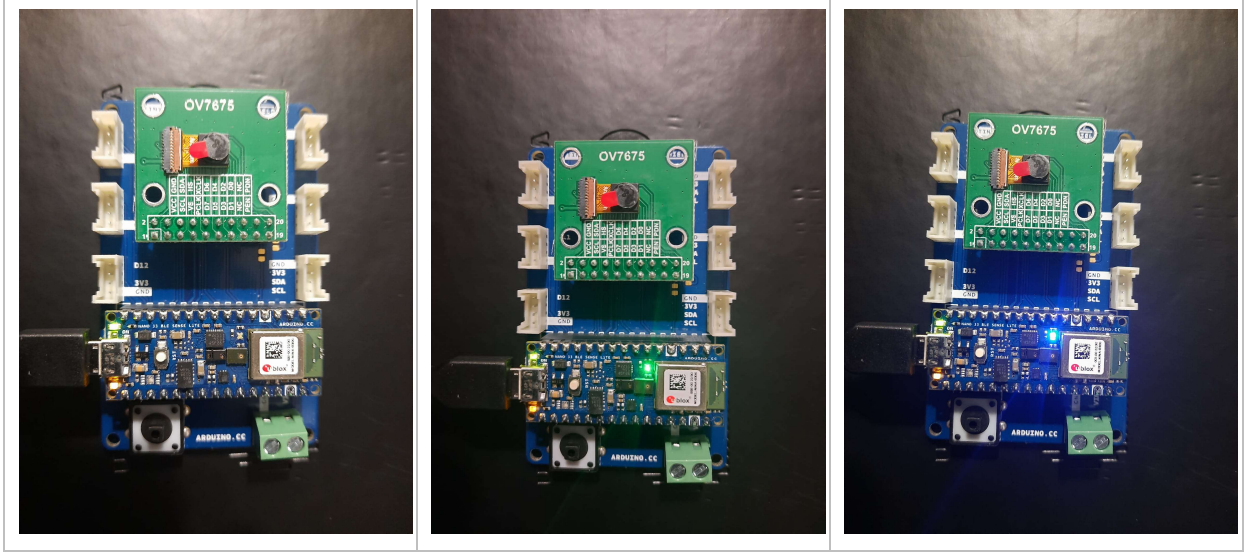

Figura 26: Placa Arduino ejecutando programa VAD (ausencia de voz, vocal detectada, consonante supuesta)

Las pruebas realizadas para verificar el funcionamiento arrojan conclusiones compatibles con las obtenidas al analizar el algoritmo VAD en el apartado 4.3. Se identifica como voz una gran variedad de sonidos no vocales, reafirmando la idea de la poca robustez del algoritmo frente al ruido.

Por otro lado, al enfrentarlo a locuciones limpias como las empleadas en la fase de entrenamiento se observa la introducción de un retardo importante en la respuesta ante los cambios en la señal de audio (voz/silencio), lo que lleva a la conclusión de que el procesamiento necesario es excesivo, bien por insuficiente capacidad de cálculo del microcontrolador o por una implementación no lo bastante eficiente.

5. Uso de datos y RGPD

Los datos usados para el desarrollo de este TFM proceden del Multilingual LibriSpeech (MLS) dataset de (Pratap, Xu, Sriram, Synnaeve, & Collobert, 2020) cuya licencia de uso, tal y como especifican sus autores, es Creative Commons CC BY 4.0 que permite copiar y redistribuir el material en cualquier medio o formato, así como remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente, siempre y cuando se indique claramente la atribución del material a sus autores.

Este dataset es un catálogo de archivos de audio, convenientemente clasificados, etiquetados y transcritos, extraídos de audiolibros del archivo LibriVox.

LibriVox es una iniciativa pública que pone a disposición de sus usuarios libros de dominio público, leídos por voluntarios y distribuidos por internet de forma pública y gratuita.

La voz debe ser considerada como un dato de carácter personal y, por tanto, sujeta a la normativa del RGPD, pudiendo incluso llegar a encuadrarse dentro de las categorías especiales de datos recogidas en el artículo 9 al tratarse de un dato biométrico que pude identificar unívocamente a una persona. No obstante, según el apartado 2.e de dicho artículo, el apartado 1 no se aplicará en aquellos casos en los que el tratamiento se refiera a datos personales que el interesado ha hecho manifiestamente públicos.

Dado que los narradores participan en el proyecto de Librivox de manera voluntaria y son conocedores de la naturaleza pública del mismo, se puede encuadrar dicha participación dentro del apartado 2.e del artículo 9 del RGPD.

Unido a lo anterior, hay que tener en cuenta que el Multilingual LibriSpeech (MLS) dataset no recoge ningún dato que identifique directamente a los narradores y, a su vez, en Librivox tampoco existen datos personales que permitan identificarles unívocamente. Por tanto, no es posible asignar un determinado archivo de voz a una persona física concreta o, al menos, no con los datos púbicamente disponibles.

Teniendo en cuanto lo anterior se puede concluir que los datos usados en el desarrollo de este TFM constituyen una información anónima y, por tanto, quedan fuera del ámbito de aplicación del RGPD.

Por otro lado, en el desarrollo de este TFM se ha aprovechado el desarrollo de software realizado por (Tang, 2019). Dicha implementación en Arduino tiene Copyright 2022 por los autores de TensorFlow y se ha desarrollado bajo licencia Apache 2.

Se trata de una licencia de software libre permisiva creada por la Apache Software Foundation (ASF). Esta licencia requiere la conservación del aviso de derecho de autor y el descargo de responsabilidad y permite al usuario del software la libertad de usarlo para cualquier propósito, para distribuirlo, modificarlo y distribuir versiones modificadas del software, bajo los términos de la licencia, sin preocuparse de las regalías.

Respetando lo establecido en la licencia, en el software modificado para el desarrollo de este TFM se ha mantenido el aviso de derecho de autor y descargo de responsabilidades del desarrollo original y el reconocimiento de la autoría original correspondiente a TensorFlow.

6. Conclusiones y trabajo futuro

6.1. Conclusiones

Este TFM ha perseguido el desarrollo e implementación en un microcontrolador de un sistema VAD en lengua castellana basado en la detección de fonemas vocálicos capaz de realizar su función en un entorno de recursos limitados. Se ha abordado desde una perspectiva de desarrollo completo incluyendo la selección y preparación de la base de datos, la elección y análisis de diversas soluciones para determinar aquella que resulta óptima y, finalmente, su adaptación a un entorno de ejecución con limitaciones específicas que han determinado las estrategias y caminos a seguir. A tal fin, se han establecido cuatro puntos de desarrollo específicos:

- Adaptar una base de datos existente y convertirla en un conjunto de datos aptos para las necesidades del presente desarrollo.

  Al no localizar bases de datos de audio con identificación de fonemas concretos, se ha elegido una base de datos con locuciones en castellano y se ha procedido a etiquetar manualmente el contenido fonético de las distintas grabaciones. Este proceso de etiquetado manual es muy laborioso y no está exento de dificultades, ya que definir las fronteras que separan dos fonemas en una locución es una tarea compleja y sujeta muchas veces a la interpretación del oyente. Esto introduce necesariamente errores que provocan problemas en la fase de entrenamiento de los algoritmos. La única forma de limitar este problema es usar una base de datos etiquetada suficientemente grande. Si bien, se ha llevado a cabo de manera exitosa la labor de etiquetado, permitiendo obtener una base de datos con la que trabajar, las limitaciones de tiempo inherentes al desarrollo de un TFM y la necesidad de no sacrificar la disponibilidad de éste para las etapas subsiguientes, ha obligado a trabajar con una base de datos con un contenido probablemente insuficiente.

- Plantear diversos algoritmos de aprendizaje automático útiles para la detección de fonemas vocálicos, comparar sus resultados y seleccionar el más idóneo para su implementación en un microcontrolador.

  Se han analizado las diversas opciones para extraer de las señales de audio características útiles al objetivo perseguido (MFCC, FFT) y se han usado para entrenar diversos modelos, analizando comparativamente el comportamiento de cada uno de ellos, tanto desde el punto de vista de su capacidad de acierto como

desde sus necesidades de procesamiento. Usando dicha comparativa y atendiendo a las limitaciones que plantea la última etapa del desarrollo, se ha elegido el algoritmo más adecuado. Los resultados finales ofrecen una exactitud ligeramente superior al 80%. Un resultado seguramente limitado por la disponibilidad de un conjunto de datos de entrenamiento no suficientemente extenso.

- Desarrollar un algoritmo que permita inferir la presencia o ausencia de voz a partir únicamente de la detección de fonemas vocálicos aprovechando las características específicas de la lengua castellana y analizar su precisión y viabilidad.

  Se ha realizado un análisis del contenido fonético para definir una estrategia que permita aproximar la presencia de voz en zonas sin contenido vocal. La estrategia elegida ha sido intencionadamente simple, para no provocar un incremento excesivo de los requerimientos de proceso, y a pesar de su sencillez permite obtener exactitudes por encima del 80% en señales de audio sin perturbar. El comportamiento, no obstante, se degrada rápidamente en presencia de perturbaciones, mostrando una de las grandes dificultades a las que se enfrentan los sistemas VAD, la contaminación por ruido.

- Adaptar ambos algoritmos a la plataforma Arduino, verificar su funcionalidad y analizar su comportamiento.

  A fin de simplificar los tiempos de desarrollo, se ha adaptado una implementación existente a los requerimientos del algoritmo desarrollado. Esto ha permitido obtener un sistema funcional que realiza la detección de voz con ciertas limitaciones. Se ha comprobado la existencia de retardos en la detección, así como una elevada tasa de fallos de detección.

Como conclusión general, este TFM ha permitido adquirir una visión de conjunto en el desarrollo de una solución práctica para la detección de voz, un aprendizaje de las técnicas empleadas para el análisis de señales de audio y un acercamiento a un campo con un prometedor futuro como es el del Tiny ML.

6.2. Líneas de trabajo futuro

Las limitaciones mostradas por el algoritmo desarrollado invitan a introducir mejoras como podrían ser:

- Ampliar la base de datos utilizada: introduciendo más hablantes, otras fuentes de audio con distintas calidades de grabación e incluso con perturbaciones, de manera que los algoritmos se puedan entrenar con datos más reales que les hagan más robustos.
- Mejoras en la implementación en Arduino: desarrollar una versión específica para la extracción de características en el microcontrolador, probablemente con una solución próxima al concepto de MFCC, con mayor coste computacional pero más robusta frente a las perturbaciones.
- Introducir mecanismos de análisis de error de clasificación en el microcontrolador: buscar métodos que permitan clarificar de manera precisa el grado de acierto de los algoritmos implementados dado que la simple inspección visual de las señales luminosas resulta claramente ineficaz. La dificultad radica en lograr esta tarea sin detraer recursos del microcontrolador que podrían provocar una peor respuesta del algoritmo de detección.
- Analizar la posibilidad de desarrollar soluciones análogas en otros idiomas en los que, por sus características fonéticas específicas, se puedan obtener mejores resultados de detección.

6. Bibliografía

Audacity Team. (1999). *Audacity® software is copyright © 1999-2023. The name Audacity® is a registered trademark*. Obtenido de https://audacityteam.org/

Bao, X., & Zhu, J. (2012). A novel voice activity detection based on phoneme recognition using statistical model. *EURASIP Journal on Audio, Speech, and Music Processing. 2012*.

Campbell, W., Richardson, F., & Reynolds, D. (2007). Language Recognition with Word Lattices and Support Vector Machines. *IEEE International Conference on Acoustics, Speech and Signal Processing 2007 (Vol 4)*, 989-992.

Chang, J.-H., Kim, N., & Mitra, S. (2006). Voice activity detection based on multiple statistical models. *IEEE Transactions on Signal Processing. 54.*, 1965-1976.

Chavarriaga, R., Sagha, H., Calatroni, A., & Digumarti, S. T. (2013). The opportunity challenge: A benchmark database for on-body sensor-based activity recognition. *Pattern Recognition Letters, 34(15)*, 2033-2042.

Chen, G., Parada, C., & Heigold, G. (2014). Small-footprint keyword spotting using deep neural networks. *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 4087-4091.

Chowdhery, A., Warden, P., Shlens, J., Howard, A., & Rhodes, R. (2019). Visual wake words dataset. *arXiv:1906.05721*.

David, R., Duke, J., Jain, A., Reddi, V. J., Jeffries, N., Li, J., . . . Warden, P. (2021). TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. *arXiv:2010.08678v3*.

Davis, A., Nordholm, S., & Togneri, R. (2006). Statistical voice activity detection using low-variance spectrum estimation and an adaptive threshold. *IEEE Transactions on Audio, Speech, and Language Processing, 14(2)*, 412-424.

Gruenstein, A., Alvarez, R., Thornton, C., & Ghodrat, M. (2017). A Cascade Architecture for Keyword Spotting on Mobile Devices. *31st Conference on Neural Information Processing Systems (NIPS 2017)*.

Hwang, I., Park, H.-M., & Chang, J.-H. (2016). Ensemble of deep neural networks using acoustic environment classification for statistical model-based voice activity detection. *Computer Speech & Language, Volume 38*, 1-12.

Hwang, Y., Jeong, M.-H., Oh, S.-R., & Kim, I.-H. (2017). Applying the Bi-level HMM for Robust Voice-activity Detection. *Journal of Electrical Engineering and Technology. 12*, 373-377.

Jang, I., Ahn, C., Seo, J., & Jang, Y. (2017). Enhanced Feature Extraction for Speech Detection in Media Audio. *Conference: Interspeech 2017*, 479-483.

Kang, T. G., & Kim, N. S. (2016). DNN-Based Voice Activity Detection with Multi-Task Learning. *IEICE Transactions on Information and Systems, 2016, Volume E99.D, Issue 2*, 550-553.

Kim, J., Kim, J., Lee, S., Park, J., & Hahn, M. (2016). Vowel based Voice Activity Detection with LSTM Recurrent Neural Network. *In Proceedings of the 8th International Conference on Signal Processing Systems (ICSPS 2016)*, 134-137.

Kinnunen, T. &. (2007). Voice Activity Detection Using MFCC Features and Support Vector Machine. *Int. Conf. on Speech and Computer (SPECOM07), Moscow, Russia, Vol. 2*, 556-561.

Koizumi, Y., Saito, S., Uematsu, H., Harada, N., Imoto, a., & Toy, K. (2019). ADMOS: A dataset of miniaturemachine operating sounds for anomalous sound detection. *In Proceedings of IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, 308-312.

Luengo, I., Navas, E., Sánchez, J., & Hernáez, I. (2009). Detección de vocales mediante modelado de clusters de fonemas. *Procesamiento del Lenguaje Natural, núm. 43*, 121-128.

Mateju, L., Červa, P., Zdánský, J., & Málek, J. (2017). Speech Activity Detection in online broadcast transcription using Deep Neural Networks and Weighted Finite State Transducers. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 5460-5464.

Moreno Sandoval, A., Toledano, D. T., Curto, N., & Torre, R. d. (2006). Inventario de frecuencias fonémicas y silábicas del castellano espontáneo y escrito. *IV Jornadas en Tecnología del Habla. Universidad de Zaragoza*, 77-81.

Moreno, N., Taboada, I., Muñoz-Basols, J., & Lacorte, M. (2017). *Introducción a la lingüística hispánica actual: teoría y práctica.* Londres: Routledge.

Mousazadeh, S., & Cohen, I. (2013). Voice Activity Detection in Presence of Transient Noise Using Spectral Clustering. *IEEE Transactions on Audio, Speech, and Language Processing, vol. 21, no. 6*, 1261-1271.

Oord, A. v., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., . . . Kavukcuoglu, K. (2016). WaveNet: A Generative Model for Raw Audio. *arXiv:1609.03499v2 [cs.SD] 19 Sep 2016*.

Pérez, H. (2003). Frecuencia de fonemas. *E-rthabla. N1*.

Pratap, V., Xu, Q., Sriram, A., Synnaeve, G., & Collobert, R. (2020). MLS: A Large-Scale Multilingual Dataset for Speech Research. *Interspeech 2020*, 2012.03411.

Python Software Foundation. (2001). *Copyright © 2001-2023*. Obtenido de https://www.python.org/

Ramírez, J., Segura, J. C., Benítez, C., Torre, A. d., & Rubio, A. (2004). Efficient voice activity detection algorithms using long-term speech information. *Speech Communication 42*, 271–287.

Ringeval, F., & Chetouani, M. (2007). Exploiting a Vowel Based Approach for Acted Emotion Recognition. *Conference: Verbal and Nonverbal Features of Human-Human and Human-Machine Interaction, COST Action 2102 International Conference, Patras, Greece*, 243-254.

Shuyin, Z., Ying, G., & Buhong, W. (2009). Auto-Correlation Property of Speech and its Application in Voice Activity Detection. *Conference: Education Technology and Computer Science, 2009. ETCS '09. First International Workshop onVolume: 3*, 265 - 268.

Susto, G. A., Schirru, A., Pampuri, S., McLoone, S., & Beghi, A. (2014). Machine learning for predictive maintenance: A multiple classifier approach. *IEEE Transactions on Industrial Informatics, 11(3)*, 812-820.

Tang, Y. (2019). *Tflite Micro Arduino Examples*. Obtenido de Tflite Micro Arduino Examples: https://github.com/tensorflow/tflite-micro-arduino-examples

Tanyer, S. G., & Ozer, H. (2000). Voice activity detection in nonstationary noise. *IEEE Transactions on speech and audio processing, 8(4)*, 478-482.

Tensorflow. (2020). *Tensorflow*. Obtenido de https://www.tensorflow.org/tutorials/audio/simple_audio

Toledano, D. T., Gonzalez-Dominguez, J., Abejon-Gonzalez, A., Spada, D., Mateos-Garcia, I., & Gonzalez-Rodriguez, J. (2007). Improved Language Recognition Using Better Phonetic Decoders and Fusion with MFCC and SDC Features. *8th Annual Conference of the International Speech Communication INTERSPEECH 2007*, 194-197.

Warden, P. (2018). Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv:1804.03209*.

Warden, P., & Situnayake, D. (2019). *TinyML.* O'Reilly Media, Inc.

Woo, K. H., Yang, T. Y., Park, K. J., & Lee, C. (2000). Robust voice activity detection algorithm for estimating noise spectrum. *Electronics Letters, 36(2)*, 180-181.

Wu, J., & Zhang, X.-l. (2011). An efficient voice activity detection algorithm by combining statistical model and energy detection. *EURASIP Journal on Advances in Signal Processing; New York Tomo 2011*, 1-10.

Yoo, I., Lim, H., & Yook, D. (2015). Formant-based robust voice activity detection. *IEEE ACM Trans, Audio Speech Lang. Process. 23 (12)*, 2238–2245.

Zhang, Y., Suda, N., Lai, L., & Chandra, V. (2017). Hello Edge: Keyword Spotting on Microcontrollers. *arXiv:1711.07128*.

Anexos

Anexo I. Jupyter Notebook – Desarrollo de algoritmo VAD

Desarrollo de un algoritmo de detección de voz en castellano con TinyML

Sep-23, Roberto Tejedor Ferrero

Máster Universitario en Análisis y Visualización de Datos Masivos
Escuela Superior de Ingeniería y Tecnología
Universidad Internacional de La Rioja

In [ ]:
```
'''
Desarrollo de un algoritmo de detección de voz en castellano con TinyML
Sep-23, Roberto Tejedor Ferrero
Máster Universitario en Análisis y Visualización de Datos Masivos
Escuela Superior de Ingeniería y Tecnología
Universidad Internacional de La Rioja
'''
import os
import math
import time
import librosa
import soundfile as sf
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import pandas as pd
import warnings

from sklearn.model_selection import *
from sklearn.tree import *
from sklearn.preprocessing import *
from sklearn.metrics import *
from sklearn.ensemble import *
from sklearn import tree
from sklearn.svm import SVC

from IPython.display import Audio

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Carga de datos

In [ ]:
```
# Listado de archivos etiquetados:
# extension .flac (audio), extesion .txt (etiquetas)
directorio = '.\\Train audios\\'  # Ubicación de archivos de audio/etiquetas
contenido = os.listdir(directorio)
archivos = []
for archivo in contenido:
    if '.flac' in archivo:
        archivos.append(archivo[:-5]) # Se extrae el nombre de cada archivo
archivos
```

In [ ]:
```
# Construcción de lista con la señal de cada fonema y su etiqueta
fonema_sample = []
```

```
for archivo in archivos:
    fonema_list = []
    file = open(directorio + archivo + ".txt", "r", encoding = 'utf-8')
    for line in file:
        fonema_list.append(line.split())
    file.close()
    signal, sr = sf.read(directorio + archivo + '.flac')
    for fonema in fonema_list:
        fonema_temp = []
        inicio = int(float(fonema[0]) * sr) # Muestra inicial del fonema
        fin = int(float(fonema[1]) * sr) # Muestra final del fonema
        fonema_temp.append(signal[inicio:fin]) # Señal correspondiente al fonema
        fonema_temp.append(fonema[2]) # Etiqueta del fonema
        fonema_sample.append(fonema_temp)
```

Análisis de datos

In [ ]:
```
# Consrtucción de dataframe de fonemas conteniendo la señal,
# su longitud (número de muestras) y la etiqueta
df_fonemas = pd.DataFrame(data = fonema_sample, columns = ['Señal', 'Fonema'])
df_fonemas['Longitud'] = df_fonemas['Señal'].apply(np.size)

# Construcción de dataframe con fonemas exclusivamente vocálicos
vocales = ['a', 'e', 'i', 'o', 'u']
df_vocal = df_fonemas.loc[df_fonemas['Fonema'].isin(vocales)]

fig, axes = plt.subplots(1, 2, figsize = (10, 4))

# Gráfica de distribución de fonemas
df_fonemas['Fonema'].value_counts().plot.bar(ax = axes[0])
axes[0].set_xlabel('Fonema')
axes[0].set_ylabel('Valor absoluto')
axes[0].set_title('Distribución de fonemas')
axes[0].set_xticklabels(axes[0].get_xticks(), rotation = 0)
twin_axes_0 = axes[0].twinx()
valor_rel = df_fonemas['Fonema'].value_counts() / len(df_fonemas['Fonema']) * 100
twin_axes_0 = valor_rel.plot.bar()
twin_axes_0.set_ylabel('Valor relativo (%)')
twin_axes_1 = axes[1].twinx()

# Boxplot de longitud de fonemas vocálicos
df_vocal.boxplot(column = ['Longitud'], ax = twin_axes_1)
axes[1].set_axis_off()
axes[1].set_xlabel(' ')
twin_axes_1.set_title('Longitud de fonemas vocálicos')
twin_axes_1.set_ylabel('Número de muestras')
plt.savefig("Plots\\Distribucion fonemas")
plt.show()

print('Total fonemas: ', len(df_fonemas['Fonema']))
print('Caracterización fonemas vocálicos:\n', df_vocal['Longitud'].describe())
```

In [ ]:
```
# Localización de índices de fonema vocálico de mayor y menor longitud
minimo = int(df_vocal['Longitud'].describe().min())
maximo = int(df_vocal['Longitud'].describe().max())
index_short_fonema = df_vocal.loc[df_vocal['Longitud'] == minimo].index
index_long_fonema = df_vocal.loc[df_vocal['Longitud'] == maximo].index
```

In [ ]:
```
fig, axes = plt.subplots(1, 3, figsize = (15, 4))

# Señal correspondiente al fonema vocálico más corto
```

```
axes[0].plot(df_vocal.loc[index_short_fonema]['Señal'].iat[0])
axes[0].set_title('Fonema ' +
                  df_fonemas.iloc[index_short_fonema]['Fonema'].iat[0])
axes[0].set_xlabel('Muestras')
axes[0].set_ylabel('Amplitud')

# Señal correspondiente al fonema vocálico más largo
axes[1].plot(df_vocal.loc[index_long_fonema]['Señal'].iat[0])
axes[1].set_title('Fonema ' + df_fonemas.iloc[index_long_fonema]['Fonema'].iat[0])
axes[1].set_xlabel('Muestras')

# Detalle de la señal correspondiente al fonema vocálico
# más largo (primeras 512 muestras)
axes[2].plot(df_vocal.loc[index_long_fonema]['Señal'].iat[0][0:512])
axes[2].set_title('Fonema ' +
                  df_fonemas.iloc[index_long_fonema]['Fonema'].iat[0] +
                  ' (sección)')
axes[2].set_xlabel('Muestras')
plt.savefig("Plots\\Señales fonemas")
plt.show()
```

Generación de base de datos con extracción de características

In [ ]:
```
# Se define la función que extrae las características de entrada al
# algoritmo a partir de las salida de la STFT.
# El proceso básico simplemente extrae los valores absolutos
def STFT_to_feat(D):
    feat = np.abs(D)
    return feat
```

In [ ]:
```
# Enventanado de señales y construcción de dataframe con FT, MFCC,
# etiqueta de fonema y tipo de sonido (Vocal / No vocal) para
# cada frame resultante del enventanado

# Parámetros para enventanado:
# WIN_LENGTH definida en función del fonema vocálico más corto
# HOP_LENGTH ajustada para lograr buen solapamiento sin exigir
# una elevada carga de proceso
WIN_LENGTH = 256
HOP_LENGTH = 128
# MFCC_VALUES define el número de coeficientes de MFCC que se van a calcular
MFCC_VALUES = 8

vocales = ['a', 'e', 'i', 'o', 'u']

flag_plot = True # Usado para graficar la STFT y MFCC del primer archivo de audio
# Dataframe que contendra FT, MFCC, etiqueta y tipo de sonido de cada frame
df_data = pd.DataFrame()

for archivo in archivos:
    signal, sr = sf.read(directorio + archivo + '.flac')
    # Cálculo de la STFT usando los parámetros de enventanado
    D = librosa.stft(signal, center=False, n_fft = WIN_LENGTH,
                     win_length = WIN_LENGTH, hop_length = HOP_LENGTH)

    # Almacenamiento de valores FT
    feat = STFT_to_feat(D)
    df_temp = pd.DataFrame(feat.T,
                           columns = ["FT" + str(i) for i in range(0, len(feat))])
```

```
# Gráfica de STFT del primer archivo (con conversión logarítmica en dB)
if flag_plot:
    fig, axes = plt.subplots(1, 2, figsize=(10, 4))
    warnings.filterwarnings("ignore")
    S_db = librosa.amplitude_to_db(np.abs(D), ref = np.max)
    img = librosa.display.specshow(S_db, x_axis ='frames',
                                   y_axis ='linear', ax = axes[0], sr = sr)
    warnings.filterwarnings("default")
    axes[0].set_title('STFT archivo ' + archivo)
    fig.colorbar(img, ax = axes[0], format = "%+2.f dB")

# Cálculo de MFCC usando los parámetros de enventanado
warnings.filterwarnings("ignore")
mfccs = librosa.feature.mfcc(y = signal, center = False, sr = sr,
                             n_mfcc = MFCC_VALUES, n_fft = WIN_LENGTH,
                             win_length = WIN_LENGTH, hop_length = HOP_LENGTH)
warnings.filterwarnings("default")
# Almacenamiento de valores MFCC. Se obtienen MFCC_VALUES parámetros
df_temp = pd.concat([df_temp,
                     pd.DataFrame(mfccs.T,
                                  columns = ["MFCC" +
                                             str(i) for i in \
                                             range(0,MFCC_VALUES)])],
                                  axis = 1)

if flag_plot: # Gráfica de MFCC del primer archivo
    warnings.filterwarnings("ignore")
    img = librosa.display.specshow(mfccs, x_axis ='frames',
                                   ax = axes[1], sr = sr)
    warnings.filterwarnings("ignore")
    fig.colorbar(img, ax = axes[1])
    axes[1].set_title('MFCC archivo ' + archivo)
    plt.savefig("Plots\\Espectro MFCC")
    plt.show()
    flag_plot=False

# Extracción de información del etiquetado de archivos
file = open(directorio + archivo + ".txt", "r", encoding = 'utf-8')
fonema_list=[]
for line in file:
    fonema_list.append(line.split())
file.close()

 # Array que contendrá la posición de la muestra inicial de cada fonema
fonema_inicio = np.empty((0), dtype = int)
# Array que contendrá la posición de la muestra final de cada fonema
fonema_fin = np.empty((0), dtype = int)
# Array que contendrá el tipo de fonema
fonema_tipo = np.empty((0), dtype = str)

# Se completa el etiquetado manual realizado en Audacity
# con las zonas no etiquetadas que se consideran silencio/ruido

if (int(float(fonema_list[0][0])*sr)) > 0:
    # Inicio primer fonema > 0 ==> Silencio inicial
    fonema_inicio = np.append(fonema_inicio, int(0))
    fonema_fin = np.append(fonema_fin, int(float(fonema_list[0][0]) * sr))
    fonema_tipo = np.append(fonema_tipo, '-')

# Primer fonema etiquetado
fonema_inicio = np.append(fonema_inicio,
                          (int(float(fonema_list[0][0]) * sr)))
fonema_fin = np.append(fonema_fin, int(float(fonema_list[0][1]) * sr))
fonema_tipo = np.append(fonema_tipo, fonema_list[0][2])
```

```
    # Resto de fonemas con silencios intermedios
    for fonema in fonema_list[1:-1]:
        inicio = int(float(fonema[0]) * sr)
        final = int(float(fonema[1]) * sr)
        if inicio > fonema_fin[-1]:
            # Fonemas no consecutivos ==> Silencio intermedio
            fonema_inicio = np.append(fonema_inicio, fonema_fin[-1])
            fonema_fin = np.append(fonema_fin, inicio)
            fonema_tipo = np.append(fonema_tipo, ('-'))
        fonema_inicio = np.append(fonema_inicio, inicio)
        fonema_fin = np.append(fonema_fin, final)
        fonema_tipo = np.append(fonema_tipo, fonema[2])

    # Silencio final
    if fonema_fin[-1] < int(len(signal) - 1):
        # Fin de última etiqueta antes de fin de archivo ==> Silencio final
        fonema_inicio = np.append(fonema_inicio, fonema_fin[-1])
        fonema_fin = np.append(fonema_fin, int(len(signal)-1))
        fonema_tipo = np.append(fonema_tipo, '-')

    # Se procede al etiquetado de los frames en función de su composición

    # Número de frames obtenido en el enventanado de la señal
    total_frames = (len(signal) - WIN_LENGTH) // HOP_LENGTH + 1

    # etiqueta_fonema: array que contendra el tipo de fonema de cada frame.
    # Se inicializa con fonema '+' que se corresponde a un fonema tipo "mezcla"
    etiqueta_fonema = np.full((total_frames), np.array('+')).astype('U2')

    # Array con índices del primer frame con inicio inmediatamente
    # después del inicio de cada fonema
    inicio_frame_index = np.ceil(fonema_inicio / HOP_LENGTH).astype(int)

    # Array con índices del último frame con final inmediatamente
    # antes del final de cada fonema
    final_frame_index = ((fonema_fin-WIN_LENGTH + 1) // HOP_LENGTH)

    # Todos los frames que queden completamente dentro de un fonema
    # se etiquetan con ese fonema
    for i in range(0, len(fonema_inicio)):
        if inicio_frame_index[i] <= final_frame_index[i]:
            for frame_index in range(inicio_frame_index[i],
                                     final_frame_index[i] + 1):
                etiqueta_fonema[frame_index] = fonema_tipo[i]

    df_temp['Etiqueta fonema'] = etiqueta_fonema
    df_data = pd.concat([df_data, df_temp])
    df_temp = df_temp.iloc[0:0] # Borrado del temporal
```

Muestra de proceso de enventanado

In [ ]:

```
# Muestra de proceso de enventanado y asignación de fonemas
# a cada frame sobre el último archivo leído

# Se define la zona de muestra
FOCUS_INI = 88000
FOCUS_END = 90000

# Se localizan los fonemas que quedan en la zona de muestra y
# se amplía la zona para albergarlos completos
```

```
inicio_focus = fonema_inicio[np.argwhere((fonema_inicio <= FOCUS_END) &
                                         (fonema_fin >= FOCUS_INI))].flatten()
fin_focus = fonema_fin[np.argwhere((fonema_inicio <= FOCUS_END) &
                                   (fonema_fin >= FOCUS_INI))].flatten()
inicio_muestra = int(inicio_focus[0])
fin_muestra = int(fin_focus[-1])
fin_focus = fin_focus-inicio_focus[0] # Se ajusta el origen a 0
inicio_focus -= inicio_focus[0]
tipo_focus = fonema_tipo[np.argwhere((fonema_inicio <= FOCUS_END) &
                                     (fonema_fin >= FOCUS_INI))].flatten()

fig, ax = plt.subplots(figsize = (16, 4))

muestra = signal[inicio_muestra:fin_muestra] # Porción de señal de muestra
altura = max(muestra) - min(muestra)
base = min(muestra)

# Marcado fonemas etiquetados
for xi, xf, fonema in zip(inicio_focus, fin_focus, tipo_focus):
    ax.add_patch(Rectangle((xi, base), xf - xi, altura,
                           color = 'y', fill = False))
    ax.text(xi + (xf - xi) // 2, base, fonema, size = 10)

# Señal completa
ax.plot(muestra, alpha = 0.3)

# Marcado y etiquetado de frames
total_frames = (len(muestra) - WIN_LENGTH) // HOP_LENGTH
for i in range(0, total_frames):
    inicio_frame = i * HOP_LENGTH
    fin_frame = (i * HOP_LENGTH) + WIN_LENGTH - 1
    # Se marcan los frames
    ax.add_patch(Rectangle((inicio_frame, base + i / total_frames * altura),
                           WIN_LENGTH, altura / total_frames,
                           color = 'g', fill = False))
    # Se etiqueta
    for xi ,xf , fonema in zip(inicio_focus, fin_focus, tipo_focus):
        if (inicio_frame >= xi) & (fin_frame <= xf):
            # Frame completo dentro de un fonema
            ax.text(inicio_frame + (WIN_LENGTH / 2),
                    base + i / total_frames * altura, fonema, size = 10)
        elif (inicio_frame>=xi) & (inicio_frame<xf):
            # Frame entre dos fonemas
            ax.text(inicio_frame + (WIN_LENGTH / 2),
                    base + i / total_frames * altura, '+', size = 10)
plt.savefig("Plots\\Enventanado")
plt.show()
```

In [ ]:

```
# Ampliación del dataframe con el tipo de sonido:
# silencio, mix, vocal o consonante
df_data['Sonido'] = df_data['Etiqueta fonema']
tipos_de_fonema = df_data['Etiqueta fonema'].value_counts().index.to_list()
consonantes = [fonema for fonema in tipos_de_fonema
               if (fonema not in vocales) & (fonema != '-') & (fonema != '+')]
df_data['Sonido'].mask(df_data['Etiqueta fonema'] == '-', 'Silencio',
                       inplace = True)
df_data['Sonido'].mask(df_data['Etiqueta fonema'] == '+', 'Mix', inplace = True)
df_data['Sonido'].mask(df_data['Etiqueta fonema'].isin(vocales), 'Vocal',
                       inplace = True)
df_data['Sonido'].mask(df_data['Etiqueta fonema'].isin(consonantes),
                       'Consonante', inplace = True)
df_data.drop_duplicates(inplace = True)
```

```
df_data=df_data.reset_index(drop = True)
df_data['Sonido'].value_counts()
```

Dataframe de entrenamiento

In [ ]:
```
# Preparación de dataframe de entrenamiento que contiene todas las vocales,
# un porcentaje de silencios, un porcentaje de fonemas mezcla y el resto
# consonantes con representación proporcional al origen
total_vocales = len(df_data.loc[df_data['Sonido'] == 'Vocal'])
muestra_silencio = total_vocales // 16
df_data_train = pd.concat([df_data.loc[df_data['Sonido'] == 'Vocal'],
                           df_data.loc[df_data['Sonido'] == 'Silencio'].
                               sample(muestra_silencio, random_state = 1)])
muestra_mezcla = total_vocales // 4
df_data_train = pd.concat([df_data_train,
                           df_data.loc[df_data['Sonido'] == 'Mix'].
                               sample(muestra_mezcla, random_state = 1)])
muestra_consonante = 2 * total_vocales - len(df_data_train)
pesos_consonantes = df_data.loc[df_data['Sonido'] == 'Consonante']. \
                            groupby('Sonido')['Sonido']. \
                            transform('count')
df_data_train = pd.concat([df_data_train,
                           df_data.loc[df_data['Sonido'] == 'Consonante'].
                               sample(muestra_consonante,
                                      weights = pesos_consonantes ,
                                      random_state = 1)])

# La columna Sonido tendrá valores Vocal o No vocal según el fonema que contenga
df_data_train['Sonido'].mask(df_data_train['Etiqueta fonema'] == '-', 'No Vocal',
                             inplace = True)
df_data_train['Sonido'].mask(df_data_train['Etiqueta fonema'] == '+', 'No Vocal',
                             inplace = True)
df_data_train['Sonido'].mask(df_data_train['Etiqueta fonema'].isin(vocales),
                             'Vocal', inplace = True)
df_data_train['Sonido'].mask(df_data_train['Etiqueta fonema'].isin(consonantes),
                             'No Vocal', inplace = True)
df_data_train['Sonido'].value_counts()
```

In [ ]:
```
# Se representa la distribución de fonemas en el dataframe de entrenamiento
df_data_train['Etiqueta fonema'].value_counts().plot.bar()
plt.title('Distribución de frames de fonemas para entrenamiento')
plt.xticks(rotation = 0)
plt.savefig("Plots\\Distribucion frames entrenamiento")
plt.show()
```

Algoritmos de detección de fonemas vocálicos

In [ ]:
```
# Para poder hacer uso de los algoritmos se transforman las clases de salida
# en valores numéricos.
categorias = df_data_train['Sonido'].value_counts().index
encoder = LabelEncoder()
encoder.fit(categorias)
```

In [ ]:
```
# Constante para determinar si se hace uso de los valores FT o MFCC para
# el entrenamiento y predicción de los algoritmos
USE_FT = False # Si False se usa MFCC
USE_SCALER = True # Usar/no usar estandarización

# Se asigna a "x" los valores correspondientes a los atributos
# y a "y" los correspondientes a la clase
```

```
if USE_FT:
    # Se entrena con los valores de FT
    x = df_data_train.iloc[:,0:(WIN_LENGTH//2+1)]
    extension='FT' # Definido para nombrar archivos de imágen
else:
    # Se entrena con los valores de MFCC
    x = df_data_train.iloc[:,-MFCC_VALUES-2:-2]
    extension='MFCC'
y = encoder.transform(df_data_train['Sonido'])
# Se normaliza el valor de los atributos en el caso de empleo de MFCC
# Las conversiones de tipo aplicadas en el algoritmo FT hacen que el escalado
# no aporte mejora y complica la implantación en el microcontrolador
if USE_SCALER:
    scaler = StandardScaler().fit(x)
else:
    scaler = StandardScaler(with_mean = False, with_std = False).fit(x)
x.iloc[:,:] = scaler.transform(x)
# Se divide el dataset en dos grupos: entrenamiento y validación
# conservando las proporciones de la clase objetivo
x_train, x_valid, y_train, y_valid = train_test_split(x, y, test_size = 0.20,
                                                      random_state = 1,
                                                      stratify = y)
```

In [ ]:
```
# Se construye un modelo de red neuronal
mod_redn = keras.Sequential([
    layers.Dense(8, activation = 'relu', input_shape = [len(x_train.keys())]),
    layers.Dense(16, activation = 'relu'),
    layers.Dense(len(categorias), activation = 'softmax')
  ])

# Se compila el modelo y se usa la precisión como métrica
mod_redn.compile(loss = 'sparse_categorical_crossentropy',
                 optimizer = 'Adam',
                 metrics = ['accuracy'])
mod_redn.summary()
```

In [ ]:
```
# Se define una funcion de parada si no hay mejora en el transcurso de 2 epochs
early_stop = keras.callbacks.EarlyStopping(monitor = 'val_loss', patience = 2)
history = mod_redn.fit(
  x_train, y_train,
  epochs = 50, validation_split = 0.2, callbacks = early_stop)
```

In [ ]:
```
# Se grafica la evolución de la precisión y la perdida durante el entrenamiento
hist = pd.DataFrame(history.history)

fig, axes = plt.subplots(1, 2, figsize = (12, 4))

hist[['accuracy','val_accuracy']].plot(ax = axes[0])
hist[['loss','val_loss']].plot(ax = axes[1])
axes[0].set_title('Evolucion de la exactitud ' + extension)
axes[1].set_title('Evolución de la pérdida ' + extension)
axes[0].set_xlabel('Epoch')
axes[1].set_xlabel('Epoch')
plt.savefig("Plots\\Entrenamiento red neuronal " + extension)
plt.show()
```

In [ ]:
```
# Se hacen predicciones con el conjunto de validación
# y se calcula el tiempo de proceso
inicio_rn = time.time()
prediccion_rn = mod_redn.predict(x_valid)
fin_rn = time.time()
time_rn = fin_rn-inicio_rn
```

In [ ]:
```
# Se define un algoritmo de clasificación Random Forest
mod_rdfr = RandomForestClassifier(n_estimators = 20, criterion = 'gini',
                                  random_state = 1)
# Se entrena con los datos de train y se realiza la predicción con los de test
inicio_rdfr = time.time()
prediccion_rdfr = mod_rdfr.fit(x_train, y_train).predict(x_valid)
fin_rdfr = time.time()
time_rdfr = fin_rdfr-inicio_rdfr # Tiempo de proceso

# Se define un algoritmo de clasificación SVC
mod_svc = SVC(random_state = 1)
# Se entrena con los datos de train y se realiza la predicción con los de test
inicio_svc = time.time()
prediccion_svc = mod_svc.fit(x_train, y_train).predict(x_valid)
fin_svc = time.time()
time_svc = fin_svc-inicio_svc # Tiempo de proceso
```

In [ ]:
```
# Tiempos de ejecución de la predicción sobre los datos de validación
print('Tiempo Random Forest (s): ', time_rdfr)
print('Tiempo Red Neuronal (s): ', time_rn)
print('Tiempo SVC (s): ', time_svc)
```

In [ ]:
```
# Para obtener la predicción de la red neuronal se selecciona el
# índice de la mayor probabilidad
clase_max_prob = pd.DataFrame(prediccion_rn).idxmax(axis = 1)

# Se recuperan las etiquetas de Type
y_valid_label = encoder.inverse_transform(y_valid)
prediccion_rdfr_label = encoder.inverse_transform(prediccion_rdfr)
prediccion_svc_label = encoder.inverse_transform(prediccion_svc)
clase_max_prob_label = encoder.inverse_transform(clase_max_prob)

# Se comparan las métricas de los modelos
print('Informe de clasificación Random Forest\n',
      classification_report(y_valid_label, prediccion_rdfr_label,
                            zero_division = 0))
print('Informe de clasificación SVC\n',
      classification_report(y_valid_label, prediccion_svc_label,
                            zero_division = 0))
print('Informe de clasificación Red Neuronal\n',
      classification_report(y_valid_label, clase_max_prob_label,
                            zero_division=0))
```

In [ ]:
```
# Se grafican las matrices de confusión
fig, (ax1, ax2 ,ax3) = plt.subplots(1, 3, figsize = (16, 4))
etiquetas = encoder.inverse_transform(range(0,len(categorias)))
ConfusionMatrixDisplay.from_predictions(y_valid_label, prediccion_rdfr_label,
                                        display_labels = etiquetas,
                                        ax = ax1, cmap = 'Blues',
                                        colorbar = False)
ax1.set_title("Random Forest")
ConfusionMatrixDisplay.from_predictions(y_valid_label, clase_max_prob_label,
                                        display_labels = etiquetas,
                                        ax = ax2, cmap = 'Blues',
                                        colorbar = False)
ax2.set_title("Red Neuronal")
ConfusionMatrixDisplay.from_predictions(y_valid_label, prediccion_svc_label,
                                        display_labels = etiquetas,
                                        ax = ax3, cmap = 'Blues',
                                        colorbar = False)
ax3.set_title("SVC")
plt.suptitle('Matrices de confusión ' + extension)
```

```
plt.savefig("Plots\\Matrices confusion todos modelos " + extension)
plt.show()
```

In [ ]:
```
# Se caracterizan los errores de predicción viendo
# los tipos de fonema real en las clasificaciones erróneas
error_index_rdfr = x_valid.loc[(prediccion_rdfr_label != y_valid_label)].index
err_rdfr = df_data_train.loc[error_index_rdfr]['Etiqueta fonema'].value_counts()
error_index_rn = x_valid.loc[(clase_max_prob_label != y_valid_label)].index
err_rn = df_data_train.loc[error_index_rn]['Etiqueta fonema'].value_counts()
error_index_svc = x_valid.loc[(prediccion_svc_label != y_valid_label)].index
err_svc = df_data_train.loc[error_index_svc]['Etiqueta fonema'].value_counts()
freq_fonem = df_data_train.loc[x_valid.index]['Etiqueta fonema'].value_counts()

fig, axes = plt.subplots(1,3,figsize=(16, 5))

left, right = err_rdfr.align(freq_fonem, join = "outer", axis = 0)
(left / right * 100).dropna().sort_values(ascending = False).\
                                    plot.bar(ax = axes[0], rot = 0)
axes[0].set_title("Random Forest")
axes[0].set_ylabel('Valor relativo (%)')

left, right = err_rn.align(freq_fonem, join="outer", axis=0)
(left / right * 100).dropna().sort_values(ascending = False).\
                                    plot.bar(ax = axes[1], rot = 0)
axes[1].set_title("Red Neuronal")

left, right = err_svc.align(freq_fonem, join="outer", axis=0)
(left / right * 100).dropna().sort_values(ascending = False).\
                                    plot.bar(ax = axes[2], rot = 0)
axes[2].set_title("SVC")

plt.suptitle('Caracterización de errores ' + extension)
plt.savefig("Plots\\Errores todos modelos " + extension)
plt.show()
```

Algoritmo VAD

In [ ]:
```
# El habla implica alternancia de secciones de fonemas consonánticos y vocálicos
# Definición de zonas de cada tipo y relación entre ellas
# Los silencios entre palabras no se consideran

# Array que contendrá las longitudes de los tramos de fonemas vocálicos
long_fonema_voc = np.empty((0), dtype = int)
# Array que contendrá las longitudes de los tramos de fonemas consonánticos
long_fonema_con = np.empty((0), dtype = int)
long_vocal = 0
long_consonante = 0

if df_fonemas.iloc[0]['Fonema'] in vocales:
    vocal = True # Primer fonema vocal
else:
    vocal = False # Primer fonema consonante

# Se comparan fonemas consecutivos. Si son del mismo tipo
# se incrementa la longitud del tramo y si son de distinto tipo
# se almacena la longitud y se cambia el tipo de tramo
for fonema, longitud in zip(df_fonemas['Fonema'], df_fonemas['Longitud']):
    if (fonema in vocales) & vocal:
        long_vocal += longitud
    elif (fonema in vocales) & (not vocal):
        vocal = True
        consonante = False
```

```
        long_vocal = longitud
        long_fonema_con = np.append(long_fonema_con, long_consonante)
    elif (fonema not in vocales) & vocal:
        vocal = False
        consonante = True
        long_fonema_voc = np.append(long_fonema_voc, long_vocal)
        long_consonante = longitud
    else: # (fonema not in vocales) & (not vocal)
        long_consonante += longitud

# Se calculan la relaciones de las longitudes
# de tramos consonánticos y vocálicos consecutivos
if len(long_fonema_voc) >= len(long_fonema_con):
    ratio_con_voc = long_fonema_con / long_fonema_voc[:len(long_fonema_con)]
else:
    ratio_con_voc = long_fonema_con[:len(long_fonema_voc)] / long_fonema_voc
```

In [ ]:
```
# Boxplot de las relaciones calculadas
plt.boxplot(ratio_con_voc)
plt.suptitle('Relación de longitudes entre tramos consonánticos y vocálicos')
plt.xticks([])
plt.savefig("Plots\\Proporcion consonante-vocal")
plt.show()
pd.Series(ratio_con_voc).describe()
```

In [ ]:
```
# La constante RATIO_CONS_VOCAL establece cuanto tiempo se mantiene activa
# la salida VAD después de la última detección de un frame vocálico. Su valor
# se elige en base a la relación de longitudes entre tramos consonánticos
# y vocálicos determinada anteriormente
RATIO_CONS_VOCAL = 1.64
```

Prueba de algoritmo VAD sobre archivo etiquetado

In [ ]:
```
# Se carga el archivo, sus etiquetas y
# se completa el etiquetado con los silencios
archivo = archivos[0]
signal, sr = sf.read(directorio + archivo + '.flac')

# Extracción de información del etiquetado de archivos
file = open(directorio + archivo + ".txt", "r", encoding = 'utf-8')
fonema_list = []
for line in file:
    fonema_list.append(line.split())
file.close()

# Array que contendrá la posición de la muestra inicial de cada fonema
fonema_inicio = np.empty((0), dtype=int)
# Array que contendrá la posición de la muestra final de cada fonema
fonema_fin = np.empty((0), dtype=int)
# Array que contendrá el tipo de fonema
fonema_tipo = np.empty((0), dtype=str)

# Se completa el etiquetado manual realizado en Audacity
# con las zonas no etiquetadas que se consideran silencio/ruido
if (int(float(fonema_list[0][0]) * sr)) > 0:
    # Inicio primer fonema > 0 ==> Silencio inicial
    fonema_inicio = np.append(fonema_inicio, int(0))
    fonema_fin = np.append(fonema_fin, int(float(fonema_list[0][0]) * sr))
    fonema_tipo = np.append(fonema_tipo, '-')

# Primer fonema etiquetado
```

```
fonema_inicio = np.append(fonema_inicio, (int(float(fonema_list[0][0]) * sr)))
fonema_fin = np.append(fonema_fin, int(float(fonema_list[0][1]) * sr))
fonema_tipo = np.append(fonema_tipo, fonema_list[0][2])

# Resto de fonemas con silencios intermedios
for fonema in fonema_list[1:-1]:
    inicio = int(float(fonema[0]) * sr)
    final = int(float(fonema[1]) * sr)
    if inicio > fonema_fin[-1]:
        # Fonemas no consecutivos ==> Silencio intermedio
        fonema_inicio = np.append(fonema_inicio, fonema_fin[-1])
        fonema_fin = np.append(fonema_fin, inicio)
        fonema_tipo = np.append(fonema_tipo, ('-'))
    fonema_inicio = np.append(fonema_inicio, inicio)
    fonema_fin = np.append(fonema_fin, final)
    fonema_tipo = np.append(fonema_tipo, fonema[2])

# Silencio final
if fonema_fin[-1] < int(len(signal) - 1):
    # Fin de última etiqueta antes de fin de archivo ==> Silencio final
    fonema_inicio = np.append(fonema_inicio, fonema_fin[-1])
    fonema_fin = np.append(fonema_fin, int(len(signal) - 1))
    fonema_tipo = np.append(fonema_tipo, '-')

# Extracción de una muestra de audio con sus etiquetas

# Se define la zona de muestra
FOCUS_INI = 0
FOCUS_END = 86000

# Se localizan los fonemas que quedan en la zona de muestra y
# se amplía la zona para albergarlos completos
inicio_focus = fonema_inicio[np.argwhere((fonema_inicio <= FOCUS_END) &
                                         (fonema_fin >= FOCUS_INI))].flatten()
fin_focus = fonema_fin[np.argwhere((fonema_inicio <= FOCUS_END) &
                                   (fonema_fin >= FOCUS_INI))].flatten()
inicio_muestra = int(inicio_focus[0])
fin_muestra = int(fin_focus[-1])
fin_focus = fin_focus - inicio_focus[0] # Se ajusta el origen a 0
inicio_focus -= inicio_focus[0]
tipo_focus = fonema_tipo[np.argwhere((fonema_inicio <= FOCUS_END) &
                                     (fonema_fin >= FOCUS_INI))].flatten()
muestra = signal[inicio_muestra:fin_muestra] # Porción de señal de muestra
```

In [ ]:

```
# Algoritmo detección de fonemas vocálicos - Red neuronal
warnings.filterwarnings("ignore")

# Array que contiene el inicio de los frames que la red neuronal
# identifica como vocal
voc_pred_inicio = np.empty((0), dtype = int)
total_frames = (len(muestra) - WIN_LENGTH) // HOP_LENGTH
for i in range(0, total_frames):
    # Señal dentro del frame
    frame = muestra[i * HOP_LENGTH:i * HOP_LENGTH + WIN_LENGTH]
    if USE_FT: # FT del frame
        D = librosa.stft(frame, center = False, n_fft = WIN_LENGTH,
                         win_length = WIN_LENGTH, hop_length = HOP_LENGTH)
        feat = STFT_to_feat(D)
        # Predicción
        tipo = encoder.inverse_transform( \
            [mod_redn.predict(scaler.transform(feat.T),verbose = 0).argmax()])
    else: # MFCC del frame
        mfccs = librosa.feature.mfcc(y = frame, center = False, sr = sr,
```

```
                                    n_mfcc = MFCC_VALUES, n_fft = WIN_LENGTH,
                                    win_length = WIN_LENGTH,
                                    hop_length = HOP_LENGTH)
        # Predicción
        tipo = encoder.inverse_transform( \
            [mod_redn.predict(scaler.transform(mfccs.T),verbose = 0).argmax()])

    if tipo == 'Vocal':
        # Si se identifica vocal se almacena el inicio del frame
        voc_pred_inicio = np.append(voc_pred_inicio, i * HOP_LENGTH)

warnings.filterwarnings("default")
```

In [ ]:

```
VAD_TRANSP = 1
VOC_TRANSP = 1

# Array que contendrá los inicios de zonas VAD
vad_inicio_arr = np.empty((0), dtype = int)
# Array que contendrá los finales de zonas VAD
vad_fin_arr = np.empty((0), dtype = int)

fig, ax = plt.subplots(figsize = (16, 4))

altura = max(muestra) - min(muestra)
base = min(muestra)
tope = max(muestra)
# Marcado fonemas etiquetados destacando vocales
for xi, xf,fonema in zip(inicio_focus, fin_focus, tipo_focus):
    if (fonema in vocales):
        ax.add_patch(Rectangle((xi, base), xf - xi, altura,
                               alpha = 0.4, color = 'y'))
    ax.text(xi + (xf - xi) // 2, base, fonema, size = 10)

# Señal completa
ax.plot(muestra)

# La primera zona VAD se inicia al final del primer frame
# identificado como vocal
vad_inicio = voc_pred_inicio[0] + WIN_LENGTH
# Inicio de la primera zona de detección vocálica
voc_det_inicio = voc_pred_inicio[0]
# Longitud de zona vocálica consecutiva detectada
voc_length = 0
# Inicio del anterior frame vocálico
inicio_previo = 0
for ini_voc in voc_pred_inicio[1:]:
    if ((ini_voc-inicio_previo) == HOP_LENGTH): # Frames vocálicos consecutivos
        # Se amplia la zona de detección de vocal con un HOP_LENGTH
        voc_length += HOP_LENGTH
    else: # Frames vocálicos no consecutivos
        # Se calcula la anchura de la zona VAD proporcionalmente
        # a la anchura de zona vocálica detectada
        vad_wide = (voc_length + WIN_LENGTH) +  \
                    int(RATIO_CONS_VOCAL * (voc_length + WIN_LENGTH))
        if (vad_inicio + vad_wide) >= (ini_voc + WIN_LENGTH):
            # Se interrumpe la zona VAD si excede la nueva zona vocálica
            vad_wide = ini_voc + WIN_LENGTH-vad_inicio
        # Se almacenan inicio/fin de zona VAD para análisis error
        vad_inicio_arr = np.append(vad_inicio_arr, vad_inicio)
        vad_fin_arr = np.append(vad_fin_arr, vad_inicio + vad_wide)
        # Marcado de zonas por algoritmo VAD
        ax.add_patch(Rectangle((vad_inicio, tope - (altura) / 10), vad_wide,
                               (altura) / 10, alpha = VAD_TRANSP, color = 'g',
```

```
                                zorder = 10))
        # La señal identificada como vocal se colorea de rojo
        if inicio_previo+WIN_LENGTH>=ini_voc:
            # Se interrumpe la zona vocálica en el frame actual
            # en caso necesario
            voc_det_fin = ini_voc
        else:
            voc_det_fin = inicio_previo + WIN_LENGTH
        ax.plot(np.array(range(voc_det_inicio, voc_det_fin)),
                muestra[voc_det_inicio:voc_det_fin], 'r' ,alpha = VOC_TRANSP)
        # La zona VAD se inicia en el nuevo tramo vocálico
        vad_inicio = ini_voc + WIN_LENGTH
        # Se almacena el inicio del nuevo tramo de detección vocálica
        voc_det_inicio = ini_voc
        voc_length = 0
    inicio_previo = ini_voc

if voc_length >= 0: # Mostrar los últimos frames vocálicos
    vad_wide = (voc_length + WIN_LENGTH) + int(RATIO_CONS_VOCAL *
                                               (voc_length + WIN_LENGTH))
    # Se almacenan inicio/fin de zona VAD para análisis error
    vad_inicio_arr = np.append(vad_inicio_arr, vad_inicio)
    vad_fin_arr = np.append(vad_fin_arr, vad_inicio + vad_wide)
    # Marcado de zonas por algoritmo VAD
    ax.add_patch(Rectangle((vad_inicio, tope - (altura) / 10), vad_wide,
                           (altura) / 10, alpha = VAD_TRANSP, color = 'g',
                           zorder = 10))
    # La señal identificada como vocal se colorea de rojo
    voc_det_fin = inicio_previo + WIN_LENGTH
    ax.plot(np.array(range(voc_det_inicio, voc_det_fin)),
            muestra[voc_det_inicio:voc_det_fin], 'r' ,alpha = VOC_TRANSP)
ax.set_title("VAD sobre muestra de sonido etiquetado " + extension)
plt.savefig("Plots\\VAD sobre muestra " + extension)
plt.show()
```

Análisis del error de detección del algoritmo VAD

In [ ]:

```
# Análisis de error de detección VAD

# Inicio de samples etiquetados como NO silencio ==> Presencia de voz
inicio_voz = inicio_focus[np.argwhere(tipo_focus != '-')].flatten()
# Fin de samples etiquetados como NO silencio ==> Presencia de voz
fin_voz = fin_focus[np.argwhere(tipo_focus != '-')].flatten()
# Total samples con voz
voz_real = (fin_voz - inicio_voz).sum()

posicion_ant = 0
tp = 0 # Verdadero positivo
tn = 0 # Verdadero negativo
fp = 0 # Falso positivo
fn = 0 # Falso negativo
voz_on = False # Frame con voz real
vad_on = False # VAD activado
limite_superior = np.array([vad_fin_arr[-1], fin_voz[-1], vad_inicio_arr[-1],
                            inicio_voz[-1]]).max() + 1

# Se analiza el archivo de principio a fin
while not (inicio_voz[0] == fin_voz[0] == vad_inicio_arr[0] ==  \
           vad_fin_arr[0] == limite_superior):

    hilera = np.array([vad_fin_arr[0], fin_voz[0],
                       vad_inicio_arr[0], inicio_voz[0]])
```

```
    # El índice determina si se trata de un inicio/final de voz real/VAD
    tipo = hilera.argmin()
    # Posición del sample del cambio VAD o voz real
    posicion_act = np.array(hilera).min()

    # Se asigna el error/acierto a quien corresponda en
    # función del estado voz real/VAD
    tramo = posicion_act - posicion_ant
    posicion_ant = posicion_act
    if vad_on & voz_on: # Verdadero positivo
        tp += tramo
    elif (not vad_on) & voz_on: # Falso negativo
        fn += tramo
    elif vad_on & (not voz_on): # Falso positivo
        fp += tramo
    else:        # not vad_on & not voz_on - Verdadero negativo
        tn += tramo

    # Se actualizan los estados voz real/VAD y se elimina
    # la posición inicio/final de voz real/VAD que se acaba de analizar
    if tipo == 0: # Fin vad
        vad_on = False
        vad_fin_arr = vad_fin_arr[1:]
    elif tipo == 1: # Fin voz
        voz_on = False
        fin_voz = fin_voz[1:]
    elif tipo == 2: # Inicio VAD
        vad_on = True
        vad_inicio_arr = vad_inicio_arr[1:]
    else:        # Inicio voz
        voz_on = True
        inicio_voz = inicio_voz[1:]

    # Cuando se han eliminado todos los elementos de una lista
    # se rellena con un valor 1 unidad superior al máximo para
    # evitar un error de array vacio
    if len(vad_inicio_arr) == 0:
        vad_inicio_arr = np.append(vad_inicio_arr, limite_superior)
    if len(vad_fin_arr) == 0:
        vad_fin_arr = np.append(vad_fin_arr, limite_superior)
    if len(inicio_voz) == 0:
        inicio_voz = np.append(inicio_voz, limite_superior)
    if len(fin_voz) == 0:
        fin_voz = np.append(fin_voz, limite_superior)
```

In [ ]:

```
# Informe de clasificación del algoritmo VAD y su matriz de confusión
# para cada sample de sonido
fig, axe = plt.subplots(1, 1, figsize = (16, 4))
y_true = np.append(np.zeros(tn + fp), np.ones(fn + tp))
y_pred = np.append(np.append(np.zeros(tn), np.ones(fp)),
                   np.append(np.zeros(fn), np.ones(tp)))
labels = ['No voz', 'Voz']
print('Informe de clasificación algoritmo VAD\n',
      classification_report(y_true, y_pred, target_names = labels,
                            zero_division = 0))
ConfusionMatrixDisplay.from_predictions(y_true, y_pred, display_labels = labels,
                                        cmap = 'Blues', colorbar = False,
                                        ax = axe, values_format = 'd')
axe.set_title("Matriz de confusión algoritmo VAD (samples) " + extension)
plt.savefig("Plots\\VAD Matriz confusión " + extension)
plt.show()
```

Pruebas sobre archivos de audio no etiquetados

In [ ]:
```
# Definición de función del algoritmo VAD
def VAD_alg(signal, plt_name):
    altura = max(signal) - min(signal)
    base = min(signal)
    tope = max(signal)

    fig, ax = plt.subplots(figsize = (16, 4))

    vocal_on = False # Tramo vocálico en curso
    vad_length = 0

    warnings.filterwarnings("ignore")

    total_frames = (len(signal) - WIN_LENGTH) // HOP_LENGTH
    for i in range(0, total_frames):
         # Señal dentro del frame
        frame = signal[i * HOP_LENGTH:i * HOP_LENGTH + WIN_LENGTH]

        if USE_FT: # FT del frame
            D = librosa.stft(frame, center = False, n_fft = WIN_LENGTH,
                             win_length = WIN_LENGTH, hop_length = HOP_LENGTH)
            feat=STFT_to_feat(D)
             # Predicción
            tipo = encoder.inverse_transform(  \
                        [mod_redn.predict(scaler.transform(feat.T),
                                          verbose = 0).argmax()])
        else: # MFCC del frame
            mfccs = librosa.feature.mfcc(y = frame, center = False, sr = sr,
                                         n_mfcc = MFCC_VALUES,
                                         n_fft = WIN_LENGTH,
                                         win_length = WIN_LENGTH,
                                         hop_length = HOP_LENGTH)
            # Predicción
            tipo = encoder.inverse_transform(
                        [mod_redn.predict(scaler.transform(mfccs.T),
                                          verbose = 0).argmax()])

        if (tipo == 'Vocal') & (not vocal_on):
            # Si se identifica vocal e inicio de tramo vocálico
            vocal_on = True
            vad_length = 0
        elif (tipo == 'Vocal') & vocal_on: # Dentro de tramo vocálico
            vad_length += RATIO_CONS_VOCAL
        elif (tipo == 'No Vocal') & vocal_on: # Fin de tramo vocálico
            vocal_on = False
            vad_length += RATIO_CONS_VOCAL
        else:
            # (tipo == 'No Vocal') & not Vocal_on - Fuera de tramo vocálico
            vocal_on = False
        # Marcado de zonas por algoritmo VAD
        if vad_length > 1:
            ax.add_patch(Rectangle((i * HOP_LENGTH, tope - (altura) / 10),
                                   WIN_LENGTH, (altura) / 10, color = 'g',
                                   zorder=10))
            vad_length -= 1

        # La señal identificada como vocal se colorea de rojo
        # y la no vocálica en azul
        if vocal_on:
            color = 'r'
```

```
        else:
            color = 'b'
        ax.plot(np.array(range(i * HOP_LENGTH, i * HOP_LENGTH + WIN_LENGTH)),
                signal[i * HOP_LENGTH:i * HOP_LENGTH + WIN_LENGTH], color)
    ax.set_title("VAD " + plt_name)
    plt.savefig("Plots\\" + plt_name)

    warnings.filterwarnings("default")
```

In [ ]:
```
# Prueba sobre archivo en castellano
signal, sr = sf.read('.//Test audios//castellano.flac')
signal = signal[0:30000] # Se usa una porción de audio
VAD_alg(signal, "Prueba castellano " + extension) # Se aplica el algoritmo VAD
Audio(data = signal, rate = sr)
```

In [ ]:
```
# Efecto de la adición de ruido blanco
wn = np.random.normal(loc = np.mean(signal), scale = np.std(signal) * 0.2,
                      size = len(signal)) # Señal de ruido blanco
wn_signal = wn + signal # Se superpone al audio original
# Se aplica el algoritmo VAD
VAD_alg(wn_signal, "Prueba ruido blanco " + extension)
Audio(data = wn_signal, rate = sr)
```

In [ ]:
```
# Efecto de la adición de sonido de ambiente
rest_signal, sr = sf.read('.//Test audios//restaurante.wav')
sr = 16000
# Se superponen las señales previo ajuste de la frecuencia de muestreo
# y su intensidad
rest_signal_adj = rest_signal[range(0, len(rest_signal), 3), 0]
noisy_signal = signal + rest_signal_adj[range(0, len(signal))] * 0.3
 # Se aplica el algoritmo VAD
VAD_alg(noisy_signal, "Prueba restaurante " + extension)
Audio(data = noisy_signal, rate = 16000)
```

In [ ]:
```
# Efecto de la adición de sonido de canto de pájaros
bird_signal, sr = sf.read('.//Test audios//birds.wav')
sr = 16000
# Se superponen las señales previo ajuste de la frecuencia de muestreo
# y su intensidad
bird_signal_adj = bird_signal[range(0, len(bird_signal), 3), 0]
forest_signal=signal+bird_signal_adj[range(0, len(signal))] * 0.1
VAD_alg(forest_signal,"Prueba pájaro " + extension) # Se aplica el algoritmo VAD
Audio(data = forest_signal, rate = 16000)
```

In [ ]:
```
# Prueba sobre archivo en polaco
polaco_signal, sr = sf.read('.//Test audios//polaco.opus')
polaco_signal = polaco_signal[25000:50000] # Se usa una porción de audio
VAD_alg(polaco_signal, "Prueba polaco " + extension) # Se aplica el algoritmo VAD
Audio(data = polaco_signal, rate = 16000)
```

Exportación de modelo a TFLite para microcontroladores

In [ ]:
```
mod_redn.save('.\\Model' + extension) # Guardado del modelo entrenado

# Conversión del modelo con TFLite sin cuantización
converter = tf.lite.TFLiteConverter.from_saved_model('.\\Model' + extension)
tflite_model = converter.convert()

# Guardado del modelo TFLite sin cuantización
```

```
with open('model' + extension + '.tflite', 'wb') as f:
    f.write(tflite_model)

# Conversión del modelo con TFLite con cuantización
converter = tf.lite.TFLiteConverter.from_saved_model('.\\Model' + extension)
# Se establecen optimizaciones por defecto que incluyen cuantización
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.uint8   # Entrada a la red entero sin signo
converter.inference_output_type = tf.uint8  # Salida de la red entero sin signo
# Se define una función que proporciona datos extraidos del bloque de validación
# como un dataset representativo para que el conversor pueda hacer los ajustes
rep_data = x_valid.sample(500)
def representative_dataset_generator():
    for i in range(0, len(rep_data)):
        yield [rep_data.iloc[i,:].to_numpy(dtype='float32')]
converter.representative_dataset = representative_dataset_generator
# Se hace la conversión
tflite_model = converter.convert()
# Guardado del modelo TFLite con cuantización
with open('model_quantized' + extension + '.tflite', 'wb') as f:
    f.write(tflite_model)
```

In [ ]:

```
# Función de inferencia para modelo TFLite
def run_tflite_model(modelo, datos):

    # Se inicializa el intérprete
    interprete = tf.lite.Interpreter(modelo)
    # Se asigna memoria
    interprete.allocate_tensors()
    # Índices de entrada y salida
    input_details = interprete.get_input_details()[0]
    output_details = interprete.get_output_details()[0]

    # Array de predicciones
    predicciones = np.zeros(len(datos), dtype = int)
    for i in range(0,len(datos)):
        x = datos.iloc[i,:].to_numpy(dtype = 'float32')

        # Si los datos estan cuantizados se reescalan los datos
        # de entrada a uint8
        if (input_details['dtype'] == np.uint8) | \
            (input_details['dtype'] == np.int8):
            input_scale, input_zero_point = input_details["quantization"]
            x = (x / input_scale + input_zero_point).astype('uint8')
        interprete.set_tensor(input_details["index"], [x])
        interprete.invoke()
        output = interprete.get_tensor(output_details["index"])[0]
        predicciones[i] = output.argmax()

    return predicciones
```

In [ ]:

```
# Comparación de métricas de los modelos
orig_model = pd.DataFrame(prediccion_rn).idxmax(axis = 1)
# Se recuperan las etiquetas de Type
y_valid_label = encoder.inverse_transform(y_valid)

orig_model_label = encoder.inverse_transform(orig_model)
prediccion_tflite_label = encoder.inverse_transform(  \
                            run_tflite_model('model' + extension + '.tflite',
                                             x_valid))
prediccion_tflite_q_label = encoder.inverse_transform(  \
                              run_tflite_model('model_quantized' + extension +
```

```
                                                '.tflite', x_valid))

# Se comparan las métricas de los modelos
print('Informe de modelo original\n',
          classification_report(y_valid_label, orig_model_label,
                                zero_division = 0))
print('Informe de modelo sin cuantizacion\n',
          classification_report(y_valid_label, prediccion_tflite_label,
                                zero_division = 0))
print('Informe de modelo con cuantizacion\n',
          classification_report(y_valid_label,prediccion_tflite_q_label,
                                zero_division = 0))
```

Anexo II. Archivos de código para el microcontrolador

Los archivos con el código necesario para su implementación en la plataforma de Arduino se pueden descargar desde el siguiente repositorio:

https://github.com/RobertoTF/Spanish-VAD.git