PROCESAMIENTO DE VOZ PARA MEJORAR LA PRONUNCIACIÓN

Mayra Alexandra Castrosqui Florián

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE GRADO

Febrero 2017

Director: Adrián Riesco Rodríguez

Codirector: Enrique Martín Martín

Resumen

El aprendizaje de la pronunciación de idiomas extranjeros mediante programas informáticos se encuentra en pleno desarrollo en la actualidad. Sin embargo, se trata de un campo que no se encuentra muy desarrollado en el software libre. La mayoría de aplicaciones de aprendizaje de idiomas que ofrecen esta utilidad tienen software propietario.

Por esta razón, en este proyecto haremos un esfuerzo por investigar y llevar a cabo una de las vías para conseguir esta utilidad. La aproximación que le daremos a este problema será la comparación de audios con las voces de un nativo y un estudiante. Se tratarán las dos voces para facilitar su comparación, esta comparación nos servirá a los estudiantes para saber si nuestra pronunciación se aproxima a la de un nativo y podernos enfocar en qué aspectos mejorar.

Para ello, veremos cómo hemos de tratar señales de voz y qué tipo de filtros usar para limpiarlas. Después aplicaremos la transformada de Fourier y construiremos espectrogramas con el objetivo de introducir el tratamiento de imágenes para la simplificación de la información y al final poder hacer la comparación de los datos con la información destacada. Este proceso se implementará como una aplicación Java.

Palabras clave Comparación del habla, pronunciación de idiomas, filtros de señales de voz, espectrograma, detección de blobs, comparación de matrices.

Abstract

Learning of foreign language pronuntiation through computer programs is in full development nowadays. Nevertheless, it is a field not very developed as free software. Most of the existing language learning programs which offer this utiliry are propietary software.

For this reason, in this project we will make an effort to investigate and acomplish one of the ways to achieve this utility. The approach we will give to this problem will be the comparison of the audio given by a native speaker with that of a student. Both voices will be treated to facilitate the comparison, a comparison which will help the students to know if their pronuntiation is close to that of a native speaker, and will let them focus on which aspects to improve.

For this, we will see how we must treat voice signals and what types of filters to use in order to clean them. Next, we will apply Fourier transform and build spectrograms with the purpose of introducing image treatment for information simplification, to get to do the comparison between the data and the highlighted information. This process will be implemented as a Java application.

Keywords Speech comparison, language pronunciation, voice signal filter, spectrogram, blob detection, matrix comparison

Índice General

Índice de Figuras

Capítulo 1

Introducción

Veamos cuáles han sido las motivaciones y la idea inicial del proyecto.

1.1 Motivación y objetivos

El saber idiomas cada vez es más importante, sobre todo en el mundo laboral. Hoy en día, la gran mayoría de personas sabe inglés y quiere seguir perfeccionándolo o quiere aprender un idioma nuevo. A veces el hecho de tener un buen nivel en un idioma puede ser un requisito obligatorio para un puesto o puede abrirte puertas haciéndote destacar frente a otro candidato con características similares. Por eso podemos ver, en el día a día, muchos anuncios y publicidad de academias de idiomas, campamentos de inmersión lingüística y, cómo no, aplicaciones móviles, aplicaciones de escritorio y aplicaciones web para aprender y mejorar en muchos idiomas de manera autónoma y organizada.

Estas aplicaciones ofrecen muchas y muy variadas utilidades para el usuario. Sistemas para aprender vocabulario, comprensión lectora, práctica de gramática y práctica de pronunciación son algunas de las funcionalidades que ofrecen. Pero la mayoría de estos productos son privados y no hay mucho desarrollo en código abierto, alguna de las razones de esto es que es muy complejo, es un tema muy extenso, muchas formas de aproximación y, por tanto, no merezca la pena ofrecerlo gratis.

Sobre todo el campo de verificar la pronunciación, solo algunas aplicaciones en el mercado la ofrecen, suelen ser de pago y a varias les queda mucho por mejorar su precisión. Por lo que se trata de un área en desarrollo actual. Para saber más sobre la situación actual de este software en el mercado, ver el capítulo 3.

Con la idea de una aplicación dedicada al aprendizaje de idiomas, decidimos llevar a cabo una propia pero con software libre y código abierto. La utilidad principal de esta aplicación sería la ayuda en la pronunciación.

El proyecto incluye, a partir de un guión [2], la investigación y el desarrollo de una aplicación a partir de cero. Se eligió el guión mencionado ya que mostraba señales de ser factible y poder llegar a resultados satisfactorios. Sabíamos que es solo una de las muchas posibles soluciones para el problema planteado de comparar voces para ayudar en la pronunciación, pero una buena forma de empezar en este campo.

He podido realizar este trabajo gracias a unos fundamentos sólidos ganados en el grado. El trabajo en solitario dio cabida a que pudiera ver todos los detalles que formaron parte de elaborar el software final: la planificación, la investigación y la

aplicación de todo lo recopilado. Fue de mucha ayuda haber cursado la asignaturas de programación y matemática aplicada. Esto aportó al proyecto velocidad en muchas etapas del trabajo, en particular a las métricas para matrices usadas y a la estructura del código del proyecto. Para la creación del proyecto Java, la asignatura de Tecnología de la programación fue una base más que suficiente. Gracias al empleo de la programación orientada a objetos aserto que ha quedado un código entendible, lo que facilitará que sea extendido por algún tercero. En cuanto a las matemáticas aplicadas, nos hemos topado con series y funciones de transferencia cuando investigamos cómo funcionaban los filtros de señales, y con la transformada de Fourier cuando queríamos obtener las magnitudes de la señal por frecuencia y por tiempo. Todo esto lo hemos podido enfrentar con moderada facilidad gracias a lo visto en el grado.

1.2 Estructura de la memoria

Esta memoria se centra en explicar el paso a paso de la implementación de la aplicación dando una base teórica de los pasos que se llevan a cabo y por qué. El proyecto se encuentra en GitHub bajo licencia GPLv3 LearnLanguage [1].

En el capítulo 2 podremos ver todas las etapas por las que se pasó para construir la aplicación final. Para cada etapa se hace un exploración teórica de los conceptos o técnicas que entran en juego y luego se detalla la implementación, tratando de hacer hincapié en el porqué de las cosas. Al final del capítulo veremos cómo han ido los resultados de las pruebas que hemos realizado. Para poder desplegar el proyecto hay una guía de usuario o implementador en Github.

Al final, en capítulo 3, veremos las conclusiones a las que llegamos, los problemas con los que tuvimos que enfrentarnos, la situación actual de este tipo de software y el trabajo futuro.

Capítulo 2

Diseño e implementación

En esta sección explicaremos detenidamente los distintos pasos que se llevaron a cabo para obtener la aplicación final.

Como se ha indicado en la introducción, la tarea fundamental de nuestra aplicación consiste en comparar dos audios, que servirá para comparar la pronunciación de un nativo y la de un estudiante. La aplicación está enfocada al tratamiento y comparación del sonido y puede ser incluida por cualquier otra aplicación con interfaz de usuario, ya sea una aplicación móvil, de escritorio u *online*. Teniendo esto en cuenta se diseñó la aplicación.

Al principio, se decidió que se programara una aplicación Android con las principales funcionalidades que tienen las aplicaciones de idiomas actuales, es decir, poder trabajar tanto pronunciación como escritura y nuevo vocabulario. Se empezó el proyecto con la parte de pronunciación pero nos topamos con problemas de bibliotecas incompatibles. Al ser tratamiento de señal de voz, la mayoría de las aproximaciones a este problema están implementadas en Python o Matlab. Se estuvo averiguando sobre compatibilidades pero como se intentaba avanzar con la parte funcional y visual a la vez en un lenguaje nuevo, se perdió mucho tiempo. Al final, al tratarse de una sola persona la que desarrollaría este proyecto, se decidió que nos ocuparíamos solo del procesamiento interno de la pronunciación y no del *frontend*.

Dado que ya no teníamos que desarrollar en Android Studio, una de las decisiones fue programar en lenguaje Java, ya que es el lenguaje con el que tengo más dominio y experiencia. Además, Java tiene muchas bibliotecas y recursos que hemos tratado de aprovechar. Nuestro objetivo ahora es realizar un software que pueda ser utilizado con el *frontend* que se desee, ya sea una aplicación móvil o web. El código puede encontrarse en GitHub, en LearnLanguage (`https://github.com/Aryalexa/LearnLanguage`) y es código abierto y libre.

Para desarrollar la parte de tratamiento de voz para mejorar la pronunciación, se propuso una guía de trabajo. Esta guía se encuentra en How to detect how similar a speech recording is to another speech recording [2].

Daremos en primer lugar una visión general de todo el programa. Cuando una persona use el programa se grabará diciendo una frase indicada y la aplicación dirá si la pronunciación ha sido suficientemente similar a la pronunciación de referencia almacenada en el sistema, que en general corresponderá a la de un nativo. Por ello, el primer paso es obtener un audio directamente del micrófono del medio que

esté usando el usuario. El sonido capturado se guardará digitalmente como una secuencia de intensidades que contienen toda la información de audio; este contiene muchas frecuencias no útiles para nuestro objetivo de comparar voces pronunciando la misma frase. Por ello habrá una etapa de filtrado. Después se hará una transformación de Fourier a la señal dividida en segmentos, que nos dará el valor de los impulsos por frecuencia de los segmentos. Aprovecharemos que se encuentra en el dominio de la frecuencia para quitar el sonido de fondo. Y, dado el espectrograma resultante de la transformación limpia, nos quedaremos con la matriz bidimensional asociada, donde cada valor representa la intensidad del impulso por tiempo (segmento) y frecuencia. Esta matriz se puede interpretar como una imagen donde el color en cada punto se consigue a partir del elemento correspondiente en la matriz. Una vez que tenemos la imagen, se hace una detección de *blobs* o regiones para distinguir las zonas más destacadas o sobresalientes de la imagen. Con esto conseguiremos una matriz de los valores con más contraste, lista para ser comparada con otra matriz de la misma clase, teniendo en cuenta que las dimensiones de las matrices pueden variar ya sea por la velocidad o volumen del hablante.

En el resto de la sección daremos los detalles de estos pasos, dando en primer lugar una explicación teórica y desarrollando después los principales aspectos de implementación.

2.1 Capturando la voz

Cuando el usuario grabe su voz para ser procesada debemos capturar el audio en un formato que nos facilite su tratamiento. Grabar en formato de audio *raw* nos da total control y visualización de los datos capturados. Cuando emitimos el sonido para ser grabado emitimos una señal analógica (continua) que el micrófono captura y guarda en forma de señal digital (discreta). La transformación de esta señal analógica a digital se hace a través de un proceso llamado muestreo [3]. En procesamiento de señales, el muestreo es la reducción de una señal continua a una señal discreta. Consiste en tomar muestras de una señal analógica a una frecuencia o tasa de muestreo (*sample rate*) constante, para cuantificarlas y codificarlas posteriormente [4]. La cuantificación consiste en atribuir un valor finito (discreto) de amplitud a cada muestra, un valor dentro de un conjunto específico de valores que luego es codificado en bits.

En la figura 2.1 podemos observar cómo se toman muestras discretas S_i de la señal continua $S(t)$ cada T unidades de tiempo. Por tanto, el sonido se graba en muestras discretas y la velocidad con la que se toman estas muestras se llama *sample rate* o frecuencia de muestreo.

El método más común de representar sonido analógico digitalmente es la modulación por impulsos codificados (PCM por sus siglas en inglés *Pulse-code modulation* [5]). PCM es un procedimiento de modulación que transforma una señal analógica en una secuencia de bits. Además, es la forma estándar de audio digital en ordenadores, discos compactos, telefonía digital y otras aplicaciones similares [5].

En un flujo PCM la intensidad de una señal analógica es muestreada regularmente en intervalos uniformes, y cada muestra es cuantizada al valor más cercano dentro de un rango de pasos digitales.

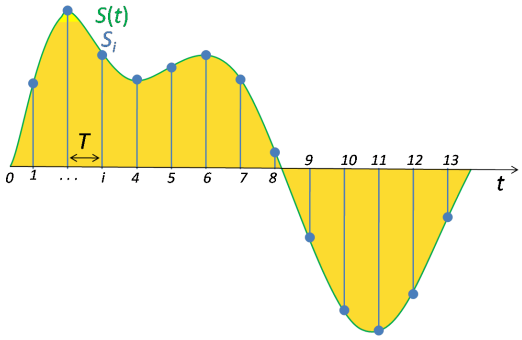

FIGURA 2.1: Muestreo de una señal continua para obtener una señal discreta. [10]

Para determinar cómo será la señal digital y medir su fidelidad a la señal analógica, el flujo PCM tiene dos propiedades básicas:

- Frecuencia de muestreo: el número de veces por segundo que se toma una muestra
- *Bit depth*: el número de posibles valores digitales usados para representar cada muestra.

¿Qué valores le daremos a estas variables?

En cuanto a la frecuencia de muestreo, hay que tener en cuenta los límites de audición o rango de frecuencias que un humano percibe, que se encuentran entre los 20 y 20K Hz. Por esto, en música y grabaciones en general se muestrea a 44.1 kHz, es decir, se toman 44.1 mil muestras de la señal analógica por segundo.

Por otro lado, las señales de voz o *speech*, que solo contienen la voz humana, se suelen tomar menos muestras por segundo. La mayoría de fonemas se encuentra entre 100 y 4K Hz, permitiendo un *sample rate* de 8 kHz [6] [7] [8] [9].

Usaremos un ratio de 44.1 kHz por ser más común y bastante utilizado. Pero tendremos en cuenta más adelante el rango de frecuencias en que se mueve la señal de la voz.

Para explicar por qué se escoge como frecuencia el doble del máximo valor del rango usaremos el teorema de muestreo de Nyquist-Shannon [11], que dice que al tomar muestras de una señal con una frecuencia que sea el doble de la frecuencia máxima de la señal, dichas muestras contendrán toda la información necesaria para reconstruir la señal original. Es decir, se podrá reconstruir la señal analógica a partir de la digital. Y por eso siempre se toma ese criterio para elegir la frecuencia de muestreo.

Y en cuanto al *bit depth*, es decir, el número de bits de información para cada muestra que se corresponde directamente con el valor cuantificado de la muestra, elegiremos uno común. Los *bit depth* más comunes para PCM son 8, 16, 20 o 24 bits por muestra. Nosotros usaremos un *short* con signo que son 16 bits. Un *bit depth* de 16-bit nos dará 65536 niveles.

Otra variable que debemos configurar cuando grabamos audio es el número de canales, en este caso un solo canal o monocanal será suficiente, ya que no estamos

buscando la calidad de las pistas de música. Un canal de audio es un canal o pista donde los elementos grabados tienen su propio área en la grabación, cuando escuchamos una grabación con varios canales, todos ellos suenan simultáneamente. Cuando se almacena música se usan dos canales (estéreo) y más de dos en el caso de películas para conseguir un mejor efecto.

En cuanto al formato del archivo de audio que obtenemos, como hemos dicho, intentaremos que sea formato *raw* o PCM, es decir, una secuencia de palabras de 16 bits que representan la intensidad de cada muestra tomada de la señal.

En cuanto a la implementación, dos de los formatos contenedores de audio más conocidos, WAV y AIFF, utilizan el formato PCM. Nosotros utilizaremos WAVE (*Waveform Audio File Format*) conocido como WAV que es, en pocas palabras, un *raw* con cabecera donde se tiene el tamaño del archivo, el tamaño de la cabecera, el formato de audio (PCM), la frecuencia de muestreo, número de canales (1 o 2 ya que WAV es solo para audio digital), el número de bits por muestra y otros campos como se muestra en la imagen 2.2. Quitar esa cabecera y obtener los datos en crudo es muy simple: tan solo debemos descartar los primeros 44 bits y guardamos el resultado en un **`.raw`**. Nos quedamos con lo el campo `data` que es el audio en bruto y se encuentra codificado con sistema *little-endian*, esto quiere decir que el byte más significativo de los dos que forman las palabras de 16 bits es el segundo y, por tanto, se almacena en primer lugar.

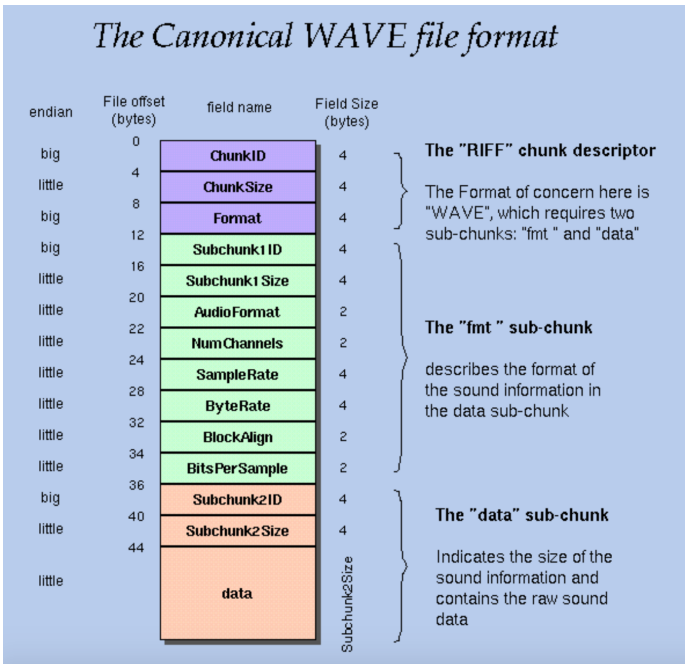

FIGURA 2.2: Formato de archivos WAVE [12].

En resumen los ajustes que usaremos para capturar audio en formato WAV serán los siguientes:

- Codificación 16-bit PCM.
- Un solo canal o monocanal.
- Frecuencia de muestreo: 44.1 kHz

- Sistema *little-endian* para almacenar los bytes.

En nuestro caso, hemos llevado a cabo esta captura de voz con la aplicación de captura y edición de audio Audacity [13] , que es software libre y multiplataforma. En Audacity puedes especificar las configuraciones que indicamos en la descripción teórica que hemos presentado anteriormente; al leer un fichero raw será necesario indicar estos ajustes.

La forma de obtener el archivo de audio wav o raw es abierta. Usar Audacity es lo que nosotros hemos hecho pero se pueden obtener de muchas otras formas. Por ejemplo, un dato interesante es que todo dispositivo Android soporta la captura de audio con 1 canal a 44.1 kHz en codificación 16-bit PCM.

Ya tenemos hecho el primer paso, hemos conseguido los datos de audio en un formato que podemos manipular. El audio grabado está en bruto, lo siguiente será quitarle el ruido y frecuencias no útiles.

2.2 Filtrando el audio

La segunda parte de nuestro procedimiento consiste en aplicar filtros para quedarnos con una versión más limpia de la voz que hemos grabado en el paso anterior. Como queremos que esta aplicación pueda adaptarse a varios interfaces de usuario (aplicaciones web, aplicaciones móviles y de escritorio) el sonido se podrá recoger de muchas maneras. Las frecuencias de voz que contienen información relevante están dentro de un rango relativamente estrecho y los micrófonos recogen un rango mayor. Por tanto, cuando grabamos sonido se recogen muchas frecuencias que no son útiles para el tratamiento de voz. A través de un filtro pasa banda se podrán rechazar las frecuencias que se encuentran fuera del rango útil.

2.2.1 Filtros

Un filtro electrónico es cualquier medio que una señal atraviesa modificando la naturaleza de esta [15]. Los filtros son utilizados en el procesamiento de señales. Discriminan componentes de frecuencia de la señal, descartando o destacando las frecuencias requeridas pudiendo modificar la amplitud o la fase de la señal.

Un filtro puede ser analógico o digital. Nos centraremos en los filtros digitales porque estamos trabajando con una señal digital.

2.2.2 Filtros digitales

Un filtro digital [15] es un algoritmo que tiene como entrada un señal digital, es decir, una secuencia discreta de valores, y otra señal digital como salida. Se trata de operaciones matemáticas que alteran el espectro o el contenido frecuencial de la señal entrante, pudiendo modificar su amplitud o su fase. Se suelen utilizar para fortalecer o atenuar frecuencias, mejorar la calidad de la señal o para síntesis de sonido logrando efectos auditivos.

En Procesamiento digital de señales (DSP por sus siglas en inglés) los filtros digitales se usan sobre los valores numéricos asociados a las muestras tomadas de señales analógicas. Siempre se puede hacer una conversión de señal analógica a

digital y viceversa. Nosotros ya tenemos nuestra señal de audio en muestras y usaremos los filtros digitales.

Es interesante destacar que los filtros digitales muestran, en general, un mejor desempeño que los filtros analógicos, que operan con señales continuas y suelen consistir en un dispositivo hardware. Algunas de las ventajas de los filtros digitales sobre los analógicos son que los filtros digitales son programables y, por tanto, no tienen tantas restricciones como los segundos, se pueden conseguir características muy extremas e incluso hacer que se adapten a la señal según vaya siendo procesada. Son más fáciles de diseñar e implementar, y no les afectan las condiciones del ambiente.

Por estos motivos, entre otros, cuando se quiere filtrar una señal analógica y seguir obteniendo una señal analógica, se hace una conversión A/D tomando muestras de la señal analógica de entrada para obtener una señal digital (valores discretos). Después se le aplica un filtro digital para posteriormente hacer otra conversión D/A y volver a obtener una señal continua a partir de la señal digital ya filtrada. REF

En la figura 2.3 se aprecia cómo se utilizan los filtros digitales para procesar señales analógicas con ayuda de conversores A/D (*ADC*) y conversores D/A (*DAC*).

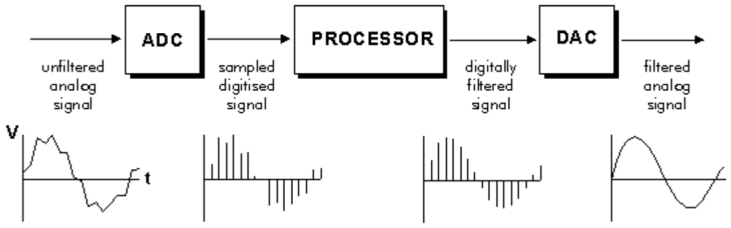

FIGURA 2.3: Proceso típico en DSP(*Digital signal processing*)

Recordemos que nuestro objetivo principal es ser capaces de comparar dos señales de audio. En esta fase inicial usaremos un filtro digital para quedarnos con un rango de frecuencias útiles. Las frecuencias que podemos percibir los humanos se encuentran entre 200 Hz y 20000 Hz. Y específicamente la voz humana se encuentra en un rango aún más pequeño como vimos en la sección anterior y puede observarse en la figura 2.4.

Los límites de la voz están entre 200-800 Hz y 3000-3500 Hz de frecuencia. Y serán valores como estos los que serán las frecuencias de corte en nuestro filtro.

2.2.3 Tipos de filtros digitales

Veremos a los filtros digitales como funciones que reciben una secuencia de números o valores (la señal de entrada) y produce una nueva secuencia de valores (la señal de salida filtrada). Habrá varios tipos de filtros según el criterio que se escoja. Esta información nos ayudará a decidir qué filtros usaremos para nuestro propósito.

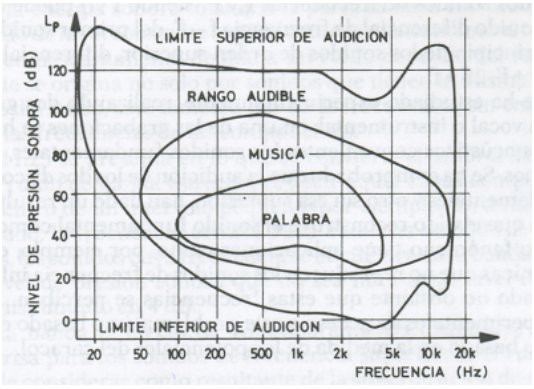

FIGURA 2.4: Límites de audición humanos. [9]

Según la respuesta en frecuencia

Es decir, de acuerdo a la parte del espectro que dejan pasar y que atenúan. De esta forma los filtros se puede seleccionar qué frecuencias quieren alterar y cuáles bloquear. Según esto podemos diferenciar 4 tipos básicos:

- Filtro paso bajo o corte alto. Deja pasar las frecuencias que están por debajo de una determinada frecuencia, por tanto, atenúa las frecuencia altas.
- Filtro paso alto o corte bajo. Deja pasar las frecuencias que están por encima de una determinada frecuencia, por tanto, para las frecuencia bajas.
- Filtro pasa banda. Es la combinación de un filtro paso bajo y uno paso alto, este filtro deja pasar las frecuencias que están en una determinada banda de frecuencia, es decir, entre dos frecuencias determinadas.
- Filtro rechaza banda. También es combinación de un paso bajo y uno paso alto, pero este deja pasar la mayor parte de las frecuencias sin alterar y atenúa las de un rango especificado, es decir, bloquea las frecuencias de la banda determinada por dos frecuencias.

Los dos primeros tipos se definen por su frecuencia de corte. Y los dos últimos tipos de filtros se definen por su frecuencia central y su ancho de banda.

La frecuencia de corte es aquella se encuentra en el límite entre la banda pasante y la atenuada. Las frecuencias bien por encima o por debajo de ella se ven atenuadas por un factor de $1/\sqrt{2}$ (aproximadamente 0.707).

Llamamos banda pasante (*passband*) al rango de frecuencias que no se verán afectadas por el filtro y simplemente pasarán. Mientras que la banda atenuada (*stopband*) son aquellas frecuencias que el filtro bloquea. Las transiciones entre la banda pasante y la banda de corte no son generalmente limpias en los filtros reales. Existe, por tanto, una banda de transición.

Expresión general de un filtro y clasificación por respuesta al impulso

Podemos ver la ecuación de un filtro en función del número de muestra (n):

$$y(n) = \sum_{k=0}^{N} b_k \cdot x(n-k) - \sum_{k=1}^{M} a_k \cdot y(n-k) \tag{2.1}$$

El filtro se ve definido por la elección de los coeficientes a y b. Y depende tanto de las muestras de entrada anteriores como de las muestras ya filtradas. El número de muestras anteriores que el filtro utiliza y mezcla con la nueva muestra de entrada nos da el orden del filtro.

Si la salida $y(n)$ solo se calcula con las muestras de entrada se dice que el filtro definido es no-recursivo. Por otro lado, un filtro recursivo es aquel que también utiliza los valores de salida calculados antes.

Estos dos filtros también son conocidos como FIR (Respuesta finita al impulso o *Finite Impulse Response*) e IIR (Respuesta infinita al impulso o *Infinite Impulse Response*), respectivamente. Estos términos son la clasificación de los filtros según su respuesta al impulso. La respuesta al impulso de un filtro digital es la secuencia de salida cuando el filtro tiene como entrada la señal impulso o entrada unitaria, siendo un esta una muestra con el valor 1 seguida de más muestras pero con valor cero [16].

Filtros FIR Los filtros FIR cuando tienen como entrada la señal impulso producen como salida un número finito de términos. Para obtener las salidas $y(n)$ se retrasa la señal de entrada hasta ese momento y se combina con la nueva señal de entrada $x(n)$. Se trata de un combinación lineal y solo están implicadas señales de la entrada, tanto la presente como las pasadas. Si vemos la ecuación anterior 2.1, para los filtros FIR queda de la siguiente forma

$$y[n] = b_0 \cdot x[n] + b_1 \cdot x[n-1] + b_2 \cdot x[n-2] + \cdots + b_N \cdot x[n-N]$$

Solo están presentes los coeficientes b_i que determinan el comportamiento del filtro. Se tienen $N+1$ coeficientes y N es el orden filtro. De hecho, la respuesta del filtro FIR a la señal impulso

$$x = \{1, 0, 0, 0, 0, 0, 0, \ldots\}$$

es la señal

$$y = \{b_0, b_1, b_2, b_3, \ldots, b_N, 0, 0, 0, \ldots\}$$

donde se pueden ver los coeficientes y la finitud.

Filtros IIR Los filtros IIR producen una salida infinita, esto se produce por la recursión en su ecuación. Ya que tanto a como b son no nulos y la ecuación siempre es de la forma

$$\begin{aligned} y[n] = {} & b_0 \cdot x[n] + b_1 \cdot x[n-1] + b_2 \cdot x[n-2] + \cdots + b_N \cdot x[n-N] \\ & - a_1 \cdot y[n-1] - a_2 \cdot y[n-2] - a_3 \cdot y[n-3] - \cdots - a_M \cdot y[n-M] \end{aligned} \tag{2.2}$$

Es decir, para la respuesta $y[n]$ se tiene una combinación lineal de entradas hasta ese momento, la señal de entrada actual y señales de salida anteriores. Se tienen $N+1$ términos del coeficiente b y M términos del coeficiente a. El orden de un filtro IIR es el máximo entre N y M.

Se dice que se retroalimenta, por eso son también llamados recursivos o de *feedback*. Y por ese motivo se dice que su respuesta es teóricamente infinita, siempre habrá términos de salida que sirvan entrada en la fórmula. Pero en realidad, a términos prácticos, la respuesta de casi todos los filtros IIR se reduce a cero en tiempo finito.

FIR vs IIR. Función de transferencia La función de transferencia de ambos filtros se calcula con el uso de la transformada Z sobre la respuesta al impulso. Lo que teníamos, una función cuyo dominio discreto es el tiempo, con la aplicación de esta transformación, ahora es una función con dominio la frecuencia y cuya variable es compleja. De esta manera, vemos la respuesta en impulso como respuesta en frecuencia. Esta forma de representar los filtros nos da información sobre los ceros y polos de la ecuación del filtro.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_M z^{-M}} \tag{2.3}$$

Los filtros IIR y FIR se distinguen en los siguientes elementos:

- Tiempo de cómputo. Mirando las ecuaciones 2.2 y 2.3 parece que los filtros IIR o recursivos requieren realizar más cálculos ya que dependen tanto de términos de la señal de entrada como de términos ya calculados de la salida. Pero, en general, ocurre lo contrario. Para conseguir la respuesta según unas ciertas especificaciones, los filtros IIR necesitan un menor número de coeficientes que los filtros FIR. Es decir, el filtro IIR requiere un orden menor para conseguir las especificaciones, haciendo que el cómputo sea más rápido, la memoria requerida, el número de operaciones y el tiempo de cómputo es menor.

- Estabilidad. Si vemos la función de transferencia de ambos filtros se ve que un filtro IIR puede tener polos y por tanto, inestabilidad, mientras que un FIR sólo tiene ceros, y por eso, siempre es estable y no entra en oscilación

- Respuesta de fase lineal. La fase es la relación que se hay entre un punto de la señal de entrada y su imagen por el filtro. La fase es así una función que idealmente es lineal. Pero esta linealidad solo se consigue realmente con los filtros FIR. La respuesta de fase de los filtros IIR no es lineal, sobre todo cerca a la banda de transición, pero se pueden hacer buenas aproximaciones. Además, si estas bandas de transición son muy cortas, es decir se hacen cortes muy abruptos, los FIR pueden requerir un gran número de retardos. Mientras que los IIR pueden ofrecer cortes muy marcados.

Estas son las diferencias más notables, pero con la tecnología actual la diferencia entre los dos tipos es poco perceptible.

Los filtros que usaremos

Sabemos que los filtros pasa banda se pueden construir a partir de los filtros paso bajo. Lo que queremos es desechar las frecuencias que se salen del rango especificado y por eso usaremos un pasa banda. Habrá cortes que atenúen frecuencias muy bajas y muy altas. El filtro lowpass que usaremos como base para el bandpass debe intentar hacer un corte limpio. Este filtro electrónico ideal del que hablamos es el llamado filtro *brick-wall*, el cual deja pasar por completo la banda de paso y atenúa

el resto con una transición muy abrupta. El nombre se debe a la forma que tiene su función de transferencia: constante hasta que se llega a la frecuencia de corte donde la intensidad de la señal tiende a menos infinito casi de inmediato.

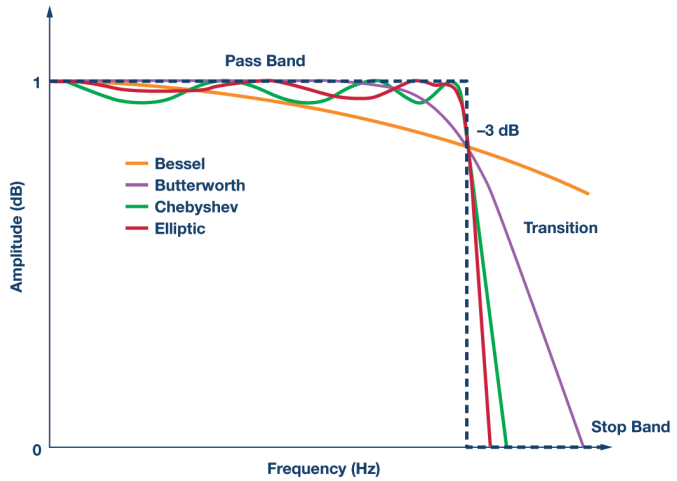

FIGURA 2.5: Diagramas de Bode a partir de la función de transferencia de los filtros Bessel, Butterworth, Chebyshev y elíptico

Los filtros electrónicos cuyo comportamiento más se asemejan a este filtro ideal son Butterworth, Chebyshev I, Chebyshev II, elíptico y Bessel, podemos ver su diagrama de Bode en al figura 2.5. Estos son la versión digital de sus homónimos analógicos.

Casi todos los filtros analógicos son IIR, sin embargo los filtros digitales pueden ser FIR o IIR. Recordamos que la presencia de la retroalimentación en el filtro hace que este sea IIR.

Entre las opciones que tenemos elegimos usar el filtro Butterworth, que va a preservar la parte de la señal que queremos.

2.2.4 IMPLEMENTACIÓN

Programar un filtro conlleva dos pasos. Primero, teniendo claro que nuestro filtro digital es IIR, sabemos que debemos conseguir los coeficiente a y b que definirán a nuestro filtro. Y segundo, una vez tenemos los coeficientes podemos aplicarlos a cualquier entrada que le demos a nuestro filtro y obtendremos la señal filtrada.

Para obtener los coeficientes que definen al filtro lo que hicimos fue documentarnos sobre cómo lo implementan en bibliotecas de lenguajes como Python y Matlab. Al principio se intentó aprovechar que Python tiene métodos específicos para el filtrado de señales pero como en esos momentos se intentaba hacer una aplicación android en Android Studio, trabajar desde ese entorno con Python no resultó muy fructuoso. Se estudiaron Jython [17] y SL4A (*Script Layer for Android*) [18] como posibles puentes. Pero resultaron un poco complicados para alguien que no tiene experiencia en Android. Como la final se optó por una aplicación Java se implementó el cálculo de los coeficientes y el filtrado en una clase de Java.

Para calcular ambos coeficientes se necesita usar números complejos así que se añadió la biblioteca `commons-math3-3.6.1` y se utilizó en concreto la clase `org.apache.commons.math3.complex.Complex`. Por otro lado, ambos coeficientes dependen tanto de la frecuencia de muestreo f_s, las frecuencias de corte y del orden del filtro. Estos datos serán parámetros que se especifican al crear el filtro.

Cuando le pasemos estos parámetros a nuestro constructor del filtro, hay que hacer un pequeño preprocesamiento. Lo que se quiere es tener en cuenta el Teorema de muestreo de Nyquist-Shannon y hacer que los cortes estén entre 0 y 1. Lo que haremos es usar la mitad de f_s como el máximo de las frecuencias captadas f_m según el teorema mencionado y después calcular las razones de los cortes, es decir la frecuencia de corte dividido entre f_m.

El orden del filtro condicionará el número de términos que tendrán a y b. Para conseguir un filtro Butterworth se realizan una serie de operaciones que pueden ser consultadas en el código fuente de ese método en Pyhton [19] , obtendremos $2*N+1$ términos para a y b siendo N el orden del filtro.

Una vez que tenemos los coeficientes calculados ya podemos aplicar la ecuación 2.1 para conseguir la salida y. De esta manera obtenemos nuestra señal filtrada.

2.3 Transformación

En este momento del proceso tenemos la señal digitalizada y filtrada con los rangos de frecuencia elegidos. Queda usar la transformación de Fourier para construir el espectrograma y trabajar con él. Veamos en qué consiste esa transformación.

La señal digital con la que estamos trabajando se encuentra todavía en el dominio del tiempo, un ejemplo en la figura 2.6. Tenemos las intensidades representadas con 16 bits a lo largo del tiempo, lo que nos da el rango de valores de -32768 a +32767. Lo que queremos es cambiar el dominio de tiempo a frecuencia.

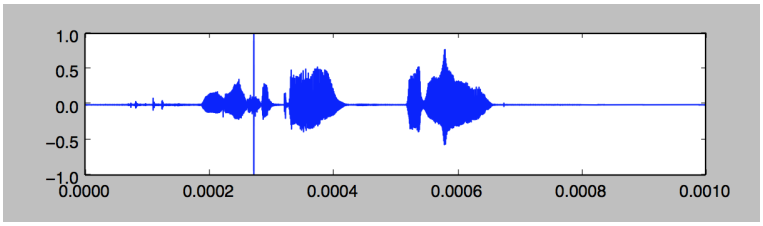

FIGURA 2.6: Intensidad de la onda a través del tiempo.

Para este análisis de frecuencias utilizaremos la **Transformada de Fourier**. La transformada de Fourier convierte una señal continua en el tiempo en su espectro de frecuencias, descomponiendo la señal en distintas frecuencias.

La transformada de Fourier de una función en el tiempo es una función compleja con dominio la frecuencia, donde los módulos de sus valores complejos representan la cantidad de las frecuencias presentes en la función inicial; y los argumentos representan la fase en una sinusoide básica en esa frecuencia. La transformada de Fourier

es reversible, siempre se puede hacer el cambio de un dominio a otro.

¿Qué frecuencias son las detectadas por la transformada de Fourier? La señal contiene muchas frecuencias que pueden ser detectadas. Si queremos detectar una frecuencia f particular debemos tener en cuenta la frecuencia de muestreo f_s de nuestra señal. Recordamos que el teorema de Shannon nos dice que dada una señal, esta no puede contener frecuencias por encima de la mitad de su frecuencia de muestreo. Es decir, si f es una frecuencia contenida en la señal entonces $-f_s/2 \leq f \leq f_s/2$. Si por el contrario $f_s \leq 2f$, esto es, la frecuencia elegida es muy alta, se dará lo que llamamos *aliasing*. Por lo que la frecuencia detectable más alta es justo la mitad de la frecuencia de muestreo f_s y se la llama frecuencia de Nyquist o de doblado ($f_s/2$) [11]. De esta forma, una señal muestreada con frecuencia de muestreo f_s, por el teorema de Shannon, los componentes de su espectro que pueden ser extraídos para frecuencias f que cumplen $|f| \leq f_s/2$.

Nuestra señal no es continua, es discreta, por lo que utilizaremos la **Transformada de Fourier Discreta (DFT)**, que requiere que la entrada sea discreta y de duración finita. Como resultado obtendremos una secuencia con la misma longitud que es una función de la frecuencia con valores complejos.

La DFT recibe como entrada una secuencia de muestras equiespaciadas como la que tenemos. Para nuestra señal discreta con frecuencia de muestreo f_s, la DFT nos devuelve una secuencia con la misma longitud, donde sus componentes también están equiespaciados para las distintas frecuencias detectadas. Si la señal está compuesta por n muestras, los valores de la transformada estarán separados por $\Delta f = f_s/n$ Hz. A este valor se le llama resolución espectral o de frecuencia es la mínima frecuencia detectada [**?**]. Siempre puede ser menor si reducimos la frecuencia de muestreo f_s o trabajamos con una señal más duradera en el tiempo.

La Transformada de Fourier Discreta (DFT) se puede calcular de manera eficiente con el algoritmo **FFT (*Fast Fourier Transform*)**. Este algoritmo fue publicado en 1965 por J. Cooley y J. Tuckey [20] y tiene diversas aplicaciones con señales. Esta técnica hace que el algoritmo consiga una complejidad de $O(n \log n)$ cuando aplicando la definición se tiene una complejidad de $O(n^2)$, donde n es el tamaño de la entrada. El algoritmo FFT puede ser aplicado a secuencias con longitud que no sea potencia de 2. Pero el algoritmo más usado es aquel en el que divide la transformada en 2 con la mitad de tamaño en cada paso y por lo tanto solo acepta como entrada secuencias de longitud potencia de 2. Veremos que cuando apliquemos el algoritmo esta no será una limitación muy grande.

Como hemos visto, la secuencia de números complejos que nos da el FFT tendrá una longitud igual a la de nuestra señal discreta con dominio en el tiempo, mientras que las frecuencias detectadas irán desde $-f_s/2$ a $f_s/2$. La segunda mitad de sus componentes corresponde a las frecuencias negativas y contiene los conjugados complejos de la primera mitad positiva de frecuencias, por lo que no añade nueva información.

De hecho, la secuencia de longitud n construida por el FFT está formado por un primer elemento que es la amplitud de la señal para la frecuencia $f = 0$, seguido de $n/2$ elementos para las frecuencias

$$f_s/n, 2f_s/n, \ldots, (n/2)f_s/n = f_s/2$$

siendo esta última frecuencia la resolución espectral, el elemento para esta frecuencia tiene parte imaginaria nula por lo que coincide con el elemento para la frecuencia $-f_s/2$. Los últimos $n/2-1$ elementos se corresponden con las frecuencias negativas ordenadas de menor a mayor:

$$(n/2-1)f_s/n, \ldots, -2f_s/n, -f_s/n$$

Si suponemos que nuestra frecuencia de muestreo f_s es 44.1 kHz y el número de muestras igual a la longitud de la secuencia FFT es $n = 1024$, entonces las frecuencias para las que se calcula la amplitud en la señal serán:

$$\begin{aligned} 0 &: 0 * 44100/1024 = 0.0 \text{ Hz} \\ 1 &: 1 * 44100/1024 = 43.1 \text{ Hz} \\ 2 &: 2 * 44100/1024 = 86.1 \text{ Hz} \\ 3 &: 3 * 44100/1024 = 129.2 \text{ Hz} \\ &\ldots \\ 511 &: 511 * 44100/1024 = 22006.9 \text{ Hz} \end{aligned}$$

Cabe resaltar que las frecuencias útiles son, en efecto, las $n/2$ primeras. La última amplitud útil para aplicaciones prácticas es la asignada a la frecuencia $(n/2-1)$-ésima, 22006.9 Hz en nuestro ejemplo. La magnitud para la frecuencia $n/2 * f_s/n$ o frecuencia de Nyquist, 22050 Hz en el ejemplo, no es usada generalmente. Ya que, en casos prácticos, para evitar el *aliasing* no se utilizan las señales a frecuencia $f_s/2$ y superiores.

Sin embargo, si aplicamos DFT con el algoritmo FFT a toda nuestra señal, nos encontramos que perdemos toda la información con respecto al tiempo, sabemos la magnitud de la señal con respecto a la frecuencia pero no sabemos en qué momento se están dando esas magnitudes, por lo que para calcular la magnitud haremos uso de una ventana deslizante, esto es, dividir la señal total en partes o *chunks* formados por un número fijo de muestras. Y aplicar la transformada de Fourier discreta a cada una, así sabremos las magnitudes por frecuencia en cada *chunk*. El tamaño de estos *chunks* será de alguna potencia de 2 (256, 512 o 1024 muestras equivalente a 512,1024 o 2048 bytes) que es la longitud que debe tener el input del FFT.

De esta forma obtenemos el espectrograma que es una representación visual de tres dimensiones de la señal de audio, podemos ver un ejemplo en la figura 2.7. En un espectrograma se puede observar en el eje vertical las frecuencias, en el eje horizontal el tiempo (los *chunks*) y en cada punto la magnitud de la intensidad de la señal en ese tiempo y esa frecuencia. De esta forma, una línea horizontal representa un tono puro, es decir una señal con una sola frecuencia y una línea vertical significa una ráfaga de ruido blanco, es decir, una señal que tiene la misma intensidad en frecuencias distintas.

El objetivo de utilizar el espectrograma es conseguir una representación gráfica del audio. Esta es la representación de la que partimos para comparar posteriormente dos señales de audio.

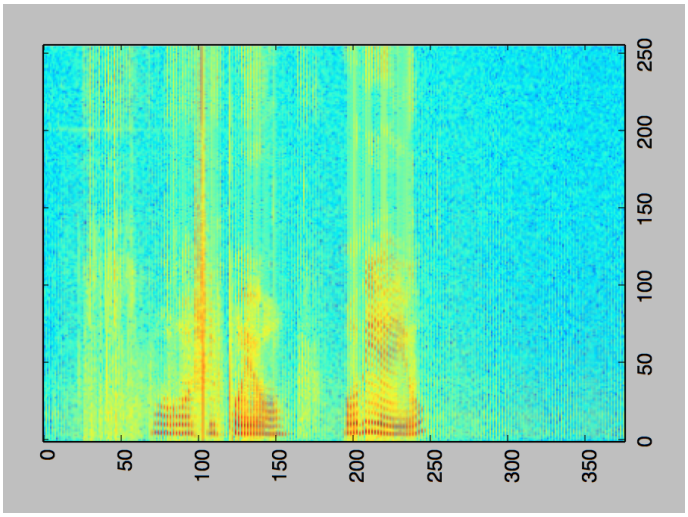

FIGURA 2.7: Espectrograma con los valores calculados a partir de la figura 2.6. Con tamaño de *chunk* 512.

2.3.1 IMPLEMENTACIÓN

En cuanto a la implementación, lo primero creamos una clase nueva donde implementaremos la aplicación del algoritmo FFT a la señales de longitud 2^m (una potencia de 2). Usaremos el método que pertenece a la biblioteca que ya hemos incluido en la sección anterior. En concreto usaremos:
`org.apache.commons.math3.transform.FastFourierTransformer.transform`

Esta función recibe como parámetros la secuencia de valores de cada *chunk* convertidos en una array de números complejos y el tipo de transformación (directa para pasar del dominio del tiempo al de la frecuencia o inversa). Esta función devuelve un array de complejos con la transformada de ese *chunk*.

Conseguimos así un array de dos dimensiones que representa el espectrograma. La primera dimensión es el número de *chunks*, la segunda el número de muestras por *chunk* y en cada elemento de la matriz estará la intensidad representada por un número complejo.

Por la definición de Transformada de Fourier el módulo de estos números complejos es la magnitud de la señal en las frecuencias indicadas. Estos módulos son los valores que buscamos para la representación gráfica de nuestro audio. Al tratarse de los módulos, la segunda parte de la transformada no se utiliza por estar formado por los conjugados complejos de la primera mitad. Los conjugados tienen el mismo módulo.

Al final de estos pasos tenemos un array bidimensional que representa el espectrograma con las magnitudes de la señal dado un *chunk* y una frecuencia positiva. Por lo que, la primera dimensión es el número de *chunks* en los que dividimos la señal, y la segunda dimensión es la mitad del número de muestras o de la longitud de la secuencia FFT ($n/2$).

El siguiente paso es la sustracción del ruido de fondo. Para ello, siguiendo el guión, se halla la media de las magnitudes por *chunk* y luego nos quedamos con el mínimo de estas medias. Para ello hay que ir calculando el módulo o magnitud de cada intensidad en el espectrograma y luego hallar el mínimo de estas. Acordamos asumir que esto representa el sonido de fondo así que convertimos todos los elementos del espectrograma a su forma polar, hallando su magnitud y su argumento. Luego sustraemos de las magnitudes el mínimo calculado y después recuperamos la forma de parte real y parte imaginaria. El resultado de este último proceso era que se eliminaba casi por completo la señal, por lo que, obtenemos una señal más clara si nos saltamos este paso y pasamos al siguiente.

2.4 Detección de *blobs*

En visión artificial, una disciplina científica dedicada a la compresión de imágenes del mundo real para ser analizadas digitalmente, se procesan imágenes y vídeos, se les extrae información para que su contenido pueda ser tratado por un ordenador. A partir de una imagen entendida como una secuencia de píxeles se busca entenderla para algún propósito, como puede ser el reconocimiento de objetos, la detección de eventos o restauración de imágenes, entre otros. Se utilizan técnicas como detección de movimiento, comparando *frames* en un vídeo; detección de presencia de objetos aplicando *background subtraction*, detección de objetos usando un umbral de brillo; extracción de características o *features*; reconocimiento de patrones; etc. El procesamiento de imágenes por ordenador permite el uso de algoritmos para estas tareas. Nuestro objetivo en esta etapa de nuestra aplicación será la detección de *blobs*.

Un *blob* se puede entender como una región de interés, una zona en la imagen cuyos puntos son semejantes. En visión artificial, los métodos de detección de *blobs* se utilizan para detectar en imágenes digitales propiedades que pueden variar por regiones como el brillo o el color. La detección de *blobs* es un método rápido que se puede utilizar para el rastreo, reconocimiento de objetos o incluso detectar la piel de una persona. Estas tareas simples pueden ayudarnos en tareas más elaboradas como detección de rostros, rastreo de varios objetos en un vídeo, etc.

Nosotros usaremos el detector de *blobs* para encontrar las partes más significativas de nuestro sonido. Haremos esto a partir de la imagen que puede obtenerse del resultado de la transformación de Fourier, el espectrograma. Serán estas partes las que nos ayudarán a comparar un muestra de voz con otra. Para detectar estos *blobs* se utiliza distintos métodos [21]. Están los métodos diferenciales, los basados en extremos locales y otros basados en la segmentación de la imagen. Los primeros se basan en las derivadas de la función con la que se discrimina con respecto a la posición y las segundas, se basan en mínimos y máximos locales de esta función.

El modo más fácil de aproximarse a la detección de *blobs* es usando un algoritmo de segmentación, donde la imagen queda dividida en particiones llamadas segmentos. Lo que se busca es simplificar el contenido de la imagen encontrando contornos o fronteras en los objetos presentes. Se le asigna una etiqueta a los pixeles del mismo segmento ya que comparten algún rasgo, estos píxeles deben de ser similares por alguna característica visual como el color o la intensidad. El resultado de la segmentación nos da las zonas que llamamos *blobs* y se consiguen gracias a la

definición de un umbral. Este umbral decide si el píxel procesado cumple el requerimiento o no para formar parte del *blob*.

Hablando de forma general, no solo para imágenes 2D, un *blob* o *metaball* es un objeto n-dimensional definido por una función continua y diferenciable con n variables. En visión artificial, las más usadas son las bidimensionales y las tridimensionales. En 3D, un *blob* es una isosuperficie, es decir, una superficie tridimensional de valor constante. Está formado por todos aquellos puntos (x, y, z) que, dada una función real f, cumple $f(x, y, z) = 0$. Mientras que en 2D, se trata de un isolínea o curva de nivel, es decir, una curva que conecta los puntos (x, y) en que la función tiene un mismo valor constante.

Una vez conocemos esta función f, definimos un umbral que discriminará los puntos según el valor que toman con f. Los puntos (x, y, z) cuya imagen queda por debajo del umbral formarán un conjunto y aquellos que quedan por encima formarán otro conjunto. Esto dividirá todo el espacio tridimensional en dos conjuntos, podemos imaginarnos un sólido y el espacio vacío que deja como los dos conjuntos. La superficie continua que separa estos dos conjuntos es lo que llamamos isosuperficies. Que la función sea continua y diferenciable quiere decir que no tiene cortes ni esquinas. Los *blobs* pueden tomar distintas formas según la función elegida. Lo análogo pasa con un espacio bidimensional, resultando una curva que separa dos zonas de la imagen, podemos ver un ejemplo de isolíneas en la figura 2.8

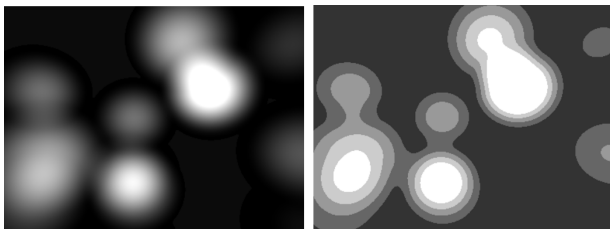

FIGURA 2.8: Isolíneas y *blobs*. Una imagen y las isolíneas encontradas para distintos umbrales. [22]

Un detector de *blobs* simple [?] Supongamos que estamos buscando *blobs* de un color particular, es decir, las zonas de la imagen que sean del color pedido. Se va analizando píxel a píxel ¿es suficientemente similar al color especificado? Entra en juego el primer umbral, el que mide la distancia entre el color del píxel actual y dicho color. Cuando la distancia es menor al umbral, el pixel pertenece a un *blob*. La distancia entre colores RGB se puede tratar como una distancia en un espacio tridimensional.

El siguiente paso es saber a qué *blob* pertenece, puede ser a uno nuevo o a uno de los ya existentes. Para saberlo debemos medir la distancia entre este píxel tratado como punto en un espacio bidimensional y todos los *blobs*. Usamos un segundo umbral, con este mediremos si el píxel está lo suficientemente cerca de un *blob* como para pasar a pertenecer a él. Si se decide que forma parte del *blob*, este se redimensionará según la posición del nuevo píxel. Cuando hayamos recorrido todos los píxeles de la imagen habremos obtenido un número de *blobs* que representan las zonas que más se parecen al color seleccionado.

2.4.1 IMPLEMENTACIÓN

A. Del espectrograma al detector de *blobs*

El último resultado que tenemos hasta ahora es una matriz de valores complejos que representan el espectrograma del sonido inicial. Siendo su posición en la matriz los valores tiempo y frecuencia y su valor es la intensidad del sonido. Lo que tenemos que hacer es interpretar estas intensidades como colores; para ello calculamos el módulo de los valores complejos de la matriz obteniendo valores reales positivos. Después calculando el mínimo y el máximo de estos valores los cambiaremos a escala 0-255 de números enteros, que es el rango de valores para los tres colores RGB que se usan en las imágenes.

Representación de imágenes En primer lugar fue necesario estudiar el detector de *blobs* que vamos a utilizar, conocer su entrada y su salida. El detector elegido es una implementación hecha para Processing [23]. Processing es un lenguaje diseñado para aprender a programar en el contexto de la visualización, además es un *sketchbook* para crear software donde se programa con código Processing. Este lenguaje es muy parecido a Java y fácil de traducir y al ser dedicado a tratar imágenes y videos, tiene bibliotecas integradas muy útiles para esto.

Como el detector de *blobs* en Processing recibe como parámetro una imagen, debemos saber cómo interpreta dicha imagen, qué conjuntos de datos la representan y cuáles de estos datos son útiles para el detector. En Processing existe una clase `Picture` para representar imágenes y puede ser creado a partir de una imagen digital. Esta clase tiene muchos atributos y funciones de los cuales el detector de blob solo un par:

- Un array de enteros donde cada elemento representa un píxel y tiene el valor del color de ese píxel.
- Dos valores que son las dimensiones alto y ancho de la imagen, donde número total de píxeles es igual a alto*ancho.

Por lo que en nuestro proyecto necesitaremos una clase análoga con solo estos atributos. Para representar los colores es suficiente un tipo int en Java, es decir, 4 bytes. Con 1 byte que son 8 bits se representan los 256 valores que puede tomar el rojo, verde o azul del formato RGB. Esto suma 3 bytes de los 4 que tiene un entero, el byte restante Processing lo utiliza para representar el canal alfa, es decir, utiliza un formato RGBA. Este canal extra de información se usa para la opacidad y su nombre se inspira en la letra griega usada en la fórmula de la interpolación $\alpha A + (1 - \alpha)B$. Si su valor está al mínimo se interpreta que el píxel es totalmente transparente y no se ve, y por el contrario, si el canal alfa está al máximo se trata de un píxel totalmente opaco. Los valores intermedios hacen posible dar un grado de transparencia al píxel. Lo que vamos a construir nosotros es una imagen tradicional opaca, por lo que el canal alfa valdrá 255.

El orden en el que se escriben los valores en Processing es RGBA, como muestra la figura 2.9.

En resumen, si entendemos cada dígito como un valor hexadecimal (4 bits) podemos representar un píxel de nuestro array como un entero con valor `0x`$\alpha\alpha$`RRGGBB` donde `0x`$\alpha\alpha$ siempre será `0xFF`.

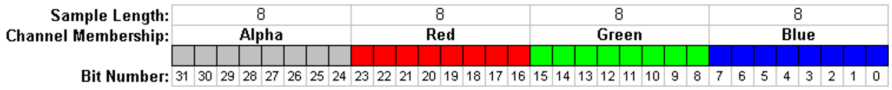

FIGURA 2.9: Formato ARGB para colores RGBA. 4 bytes son los necesarios para su representación. [24]

Construcción de la imagen A partir de la matriz de valores reales positivos, que son las magnitudes de los valores del espectrograma, hay que crear una matriz de colores. Con este objetivo en mente, lo primero que haremos es transformar un array de doubles en un array de enteros con valor entre 0 y 255, incluidos. Para ello, calculamos el mínimo y el máximo de los valores que toman las magnitudes en el espectrograma y escalamos con una simple regla de proporcionalidad. Pero los valores siguen siendo reales, los convertimos en enteros con un simple redondeo hacia abajo.

Ya tenemos un valor entero entre 0 y 255 (incluidos), lo siguiente es transformar este valor en un color gris. Será gris ya que tanto el rojo, el verde y el azul tendrán este valor. De esta forma crearemos una imagen en escala de grises a partir del espectrograma.

Si la magnitud de un elemento del espectrograma pasa a valer `0xCD` en hexadecimal, después de todo el proceso mencionado, haremos que el color del píxel correspondiente sea `0xFFCDCDCD`.

Este cálculo solo se trata de sumas y desplazamientos, es muy simple:

```
0xFFCDCDCD = 0xFF000000 + 0xCD0000 + 0xCD00 + 0xCD
= (0xFF«24) + (0xCD«16) + (0xCD«8) + 0xCD
```

Con este último cálculo ya tenemos la entrada preparada para empezar el procesamiento que hará el detector de *blobs*.

B. *Blob Processing*

El detector de *blobs* que utilizaremos en nuestra aplicación se basa en un detector de bordes que segmenta la imagen si encuentra contraste entre un píxel y los que lo rodean. La forma de medir si encontramos suficiente contraste es mediante un umbral o *threshold*.

Thresholding En segmentación, una vez tenemos la función a comparar lo siguiente es definir el valor del umbral que dividirá la imagen. En nuestro caso lo que hace el detector es sumar los tres valores del color: rojo, verde y azul. Y lo compara con un umbral que puede variar entre 0 y 255*3. Para elegir este número hay que tener en cuenta que si el umbral es bajo se encontrarán más bordes y el resultado se puede ver muy afectado por el ruido y se pueden encontrar bordes que no son útiles para nuestro propósito. Por el contrario, si el umbral es muy alto se pueden perder bordes o quedar incompletos.

En el caso de los espectrogramas, las imágenes son siempre de una manera concreta. Para estudiar cómo afectan los distintos umbrales en esta clase de imágenes, lo que haremos será definir un número de niveles, digamos N. Entonces tendremos N distintos niveles donde los umbrales serán proporcionales a $0, 1/N, \ldots, (N-1)/N$.

(x,y)	(x+1,y)
(x,y+1)	(x+1,y+1)

FIGURA 2.10: Píxeles estudiados en la detección de *blobs* dado un píxel (x, y).

Blob detection Vamos a intentar comprender cómo está funcionando el detector de *blobs* de la biblioteca que hemos elegido usar. Como hemos mencionado antes, la función que se utiliza para comparar los colores es la suma de los canales RGB. $f(r, g, b) = r + g + b$ Por lo que vamos a calcular este valor para todos los píxeles y guardarlos en una variable para su posterior uso. Además vamos a llevar también una variable donde marcaremos qué píxeles han sido ya visitados y cuáles no. Al principio todos están sin visitar.

El proceso empieza a recorrer la imagen píxel a píxel, visitando toda una fila para luego seguir por la siguiente. Para cada píxel lo que hace es calcular lo que llama índice del cuadrado. Recoge las comparaciones del píxel tratado y de 3 de su alrededor. En concreto, si estamos trabajando con el píxel (x, y), se tienen en cuenta el siguiente en su fila y los dos píxeles de la fila siguiente que se encuentran justo debajo de los dos anteriores, como se muestra en la figura 2.10.

Lo que se busca es que haya un contraste entre los colores de estos píxeles, por lo que si todo se encuentran por debajo o por encima del umbral, significa que son similares y no nos interesa. El caso opuesto sí, por lo que si hay por lo menos un píxel del conjunto que no sea igual al resto con respecto al umbral, se llevará el caso al siguiente paso para ver si debe formar un borde.

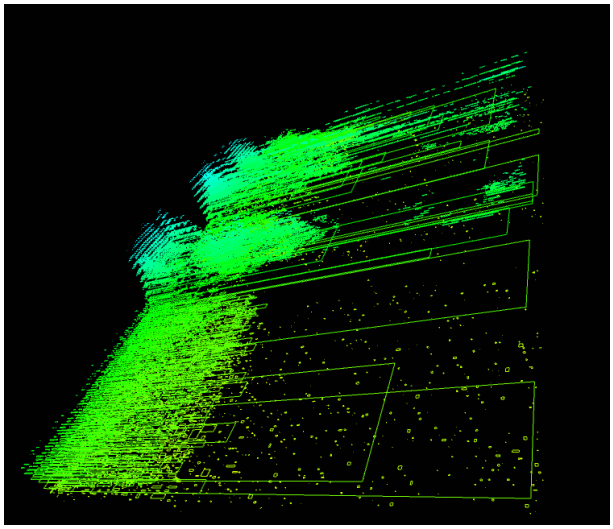

FIGURA 2.11: Blobs rectangulares en un espectrograma. Es el resultado del detector de *blobs* aplicado al espectrograma de la imagen 2.7

A grandes rasgos, cuando hay probabilidad de nuevo *blob*, pasamos a buscarlo. Para encontrar un *blob* b, se empieza por el píxel estudiado (x, y) y se decide si debe formar parte de b. Esta decisión depende del umbral elegido. Si es así se recalcula

las dimensiones del *blob b*: `xmin`, `xmax`, `ymin` y `ymax` se actualizan. Después se examinan los píxeles vecinos para ver si ellos también son parte de *b*, para ello se hace una llamada recursiva: se examina de la misma forma a los vecinos del píxel actual y se examina a los vecinos de estos últimos. Cada vez que se explora un nuevo píxel se marca como visitado. Al final tenemos definidos varios *blobs* dado un umbral definido.

Nosotros ejecutamos este proceso para todos los umbrales definidos en la etapa de *thresholding*. Por lo que para cada nivel, tenemos un conjunto de *blobs* detectados. Por la naturaleza de los espectrogramas, no tenemos regiones detectadas que sean grandes o significativas por sí solas. Por lo que nos quedaremos con los `xmin`, `xmax`, `ymin` y `ymax` de todos los *blobs* detectados en cada nivel. Esta es la información que nos será útil en el siguiente paso.

2.5 Comparación

Esta es la última etapa de nuestra aplicación y de nuestro recorrido. A partir del procesamiento que ha pasado la señal de voz del inicio vamos a obtener una matriz. De esta forma la comparación de voces se ha reducido a comparar matrices.

Los valores del espectrograma serán la base para la matriz que represente la pronunciación grabada. Para cada valor que toma el umbral en la detección de *blobs*, se formará una matriz que será el espectrograma que hemos contruido pero recortada según lo que valgan las variables `xmin`, `xmax`, `ymin` y `ymax`. Para ellos hemos construido los métodos `recortarX` y `recortarY` que recortan la matriz. Esto significa que dado un umbral, podemos obtener matrices de distintas dimensiones para cada archivo de audio que se usará en la comparación.

Pero vamos a querer que, si dos matrices se van a comparar, sean de dimensiones iguales. Esto lo conseguiremos gracias a la interpolación lineal. En nuestro proyecto Java una matriz está formada por un array con los elementos de la matriz y dos enteros: alto (número de filas) y ancho (el número de columnas), ya que nuestras matrices son de dos dimensiones (como los espectrogramas y las imágenes). Para igualar las dimensiones de dos matrices, hallamos la mayor altura y la mayor anchura, y posteriormente interpolamos ambas matrices haciendo que su alto y ancho coincidan con los máximos calculados.

Una vez tenemos dos matrices que comparten dimensiones, podemos pasar a los métodos de comparación de matrices. En primer lugar, mediremos la similitud entre las matrices con la similitud del coseno. Esta medida sirve para determinar la similitud entre vectores no nulos, nos da el valor del coseno del ángulo entre los vectores independientemente de la magnitud de los vectores. La similitud de dos vectores A y B viene dada por

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|}$$

Analicemos, lo que esto significa, el coseno del ángulo entre A y B varía entre -1 y 1, si este es positivo los vectores forman un ángulo agudo, si es 0 son perpendiculares y si es negativo forman un ángulo obtuso. Teniendo que si vale 1, tienen la misma dirección y si vale -1, son opuestos.

Para nuestros cálculos, consideraremos a la matriz como un solo vector de alto*ancho elementos. Según hemos construido el espectrograma, solo tenemos elementos positivos, por lo que el rango del coseno del ángulo se reduce a ser de 0 a 1, y el ángulo varía de 0 a $\pi/2$. Lo cual nos permite decir que la distancia y la similitud entre A y B son:

$$\text{distancia} = \frac{\cos^{-1}(\text{similitudCoseno})}{\pi/2}$$

$$\text{similitud} = 1 - \text{distancia}$$

El resto de métodos estarán basados en el cálculo de distancias entre matrices. Extraeremos una matriz de la otra y mediremos la diferencia con las correspondientes normas de matrices. Las normas aplicadas serán:

- Norma infinito, que es simplemente la máxima suma absoluta de las columnas de la matriz.
$$\|A\|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}|$$
- Norma 1, que es simplemente la máxima suma absoluta de las filas de la matriz.
$$\|A\|_\infty = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}|$$
- Norma Frobenius, que es equivalente a la norma 2 en vectores de una dimensión.
$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$$

Lo siguiente será hacer pruebas y valorar los resultados. Probaremos 3 frases de distintos idiomas:

- Japonés: *Itadakimasu* (gracias por la comida; comamos)
- Coreano: *bogo sip-eoyo, eodiyeyo?* (te echo de menos, ¿dónde estás?; quiero verte, ¿dónde estás?)
- Español: *Ellos vinieron a casa mientras estabas fuera*.

Tendremos una muestra maestra para cada frase y 3 intentos de una persona pronunciando la frase, ver figuras 2.12, 2.13, y 2.14.

FIGURA 2.12: Audio máster y 3 de prueba en japonés

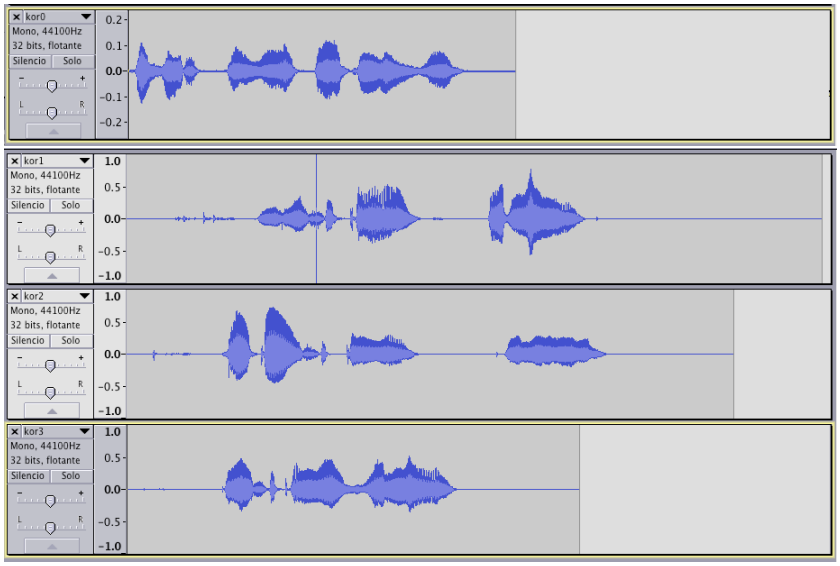

FIGURA 2.13: Audio máster y 3 de prueba en coreano

Veamos los resultados obtenidos si comparamos cada frase con los 9 intentos guardados.

Primero explicaremos el formato de los resultados que hemos adjuntado en la memoria en el apéndice A. Para cada comparación guardamos la similitud, la distancia infinito, la distancia uno y la distancia Frobenius para cada umbral. El umbral define el grado de similitud de color entre los píxeles que se pide, mientras más alto el umbral es más estricta la formación de *blobs*. Estamos trabajando con 12 umbrales o niveles:

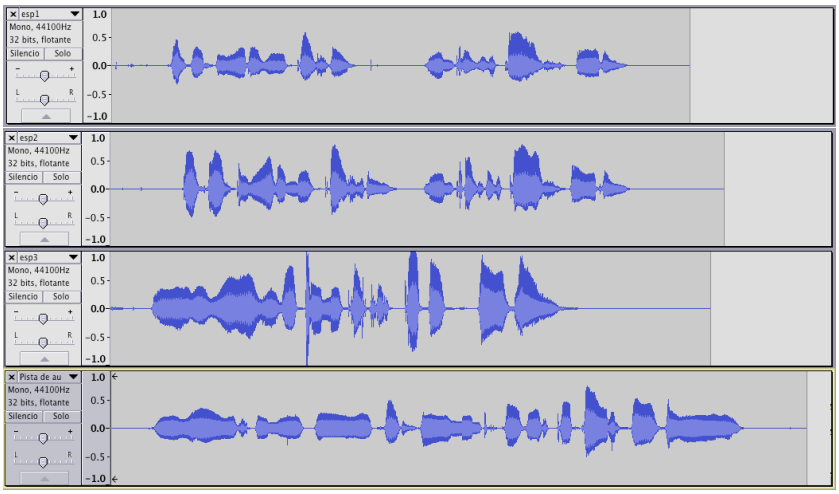

FIGURA 2.14: Audio máster y 3 de prueba en español

- 25.0%
- 31.25%
- 37.5%
- 43.75%
- 50.0%
- 56.25%
- 62.5%
- 68.75%
- 75.0%
- 81.25%
- 87.5%
- 93.75%

Para la similitud del coseno, mientras el valor sea más cercano a 1, más parecidas serán las muestras de audio. Como tomamos la similitud para varios niveles, hemos calculado la media de los 12 valores. Para las distancias, por el contrario, mientras más se acerquen a cero, más similares serán los audios. En este caso, comparamos los resultados de los intentos para ver cuál ha tenido más éxito. Una forma de verlo, es ver cuál de los intentos ha obtenido más mínimos comparando con los demás.Veamos casos más concretos.

Cuando comparamos el máster de japonés con sus 3 intentos, según la similitud del coseno el tercer intento es el más similar. Si miramos la distancia infinito, los valores del tercer audio son menores que el resto, esto también significa que el tercer intento ha tenido un mejor resultado para esta distancia. Además la distancia 1 y la de Frobenius están de acuerdo con esta decisión.

¿Qué pasa si hablamos en otro idioma cuando el programa espera esta frase en japonés? La similitud del coseno, nos da resultados que no muestran mucha diferencia comparados con el caso japonés-japonés. Sin embargo, las tres distancias muestran, en su mayoría, valores superiores a la comparación con la misma frase, lo que se traduce a más diferentes. Por tanto, más similitud conseguida para las mismas frases gracias a las distancias. La excepción ha sido el tercer intento de coreano, que muestra resultados de ser muy similar. si observamos sus muestras, podemos ver que efectivamente, son muy parecidas.

Otra de las pruebas es con coreano. Primero se comparan los 3 intentos de decir la frase elegida y luego otros idiomas con frases distintas. En este caso, los valores más altos en la similitud del coseno dan la mejor similitud, por poco, a la comparación coreano-coreano. La distancia 1 deja claro que la frase en español es muy distinta a la coreana pero la japonesa no tanto; los resultados son muy similares. Frobenius sí que nos ayuda a confirmar que las frases en coreano son las más parecida, las diferencia con el resto de frases es significativa.

Para el español hemos guardado, en los intentos 2 y 3, versiones natural e intencionadamente distorsionadas de la frase. Para las cuatro comparaciones, se muestras mejores resultados, como esperábamos, del intento 1.

Capítulo 3

Conclusiones

En la actualidad, en el campo de verificación de la pronunciación, hay poco software abierto. Casi todo el software que un usuario puede usar con esta funcionalidad es privado. La mayoría de estos software ofrecen una subscripción para el uso de su producto que suele ser una aplicación de aprendizaje de idiomas bastante completo, incluyendo muchas otras actividades además del refuerzo en la pronunciación. Algunas de estas aplicaciones son:

- Duolingo [25].
- Rosetta Stone y su tecnologia TruAccent [26]. Esta empresa compró Tellmemore de Auralog [27].
- Mango [28].
- Saundz [29].
- saundz [30].
- Babbel [31].

En este proyecto hemos visto el paso a paso del procesamiento del habla, cómo se captura la voz, las configuraciones necesarias según las aplicaciones que se le quieran dar. Cómo es suficiente solo un canal y cuál debe ser la frecuencia de muestreo necesaria para capturar la voz. Hemos visto que hay distintos filtros para las señales digitales, que nosotros necesitamos un filtro paso banda y el filtro Butterworth es una buena opción porque conserva las distintas intensidades de la onda. Hemos visto qué frecuencias del resultado de la transformada de Fourier son las necesarias para la creación del espectrograma, las frecuencias positivas. Y aprendimos que la detección de blobs en imágenes se puede hacer buscando lo que es más similar a una referencia o buscando los contrastes en toda la imagen haciendo que los blobs se formen solos a partir de la creación de bordes.

En contraste con el guión [2]:

- La sustracción del sonido de fondo del modo que se describe no muestra buenos resultados.
- La comparación de matrices deja no es muy precisa si se trata de distintas frases comparadas.
- Al tratarse de espectrogramas de frases de palabras, elegir el blob más grande no es buena opción, ya que se pierde mucha información. Optamos por eso elegir la unión de los blobs.

También investigamos si aplicaciones de reconocimiento de música nos podrían ser útiles, como Shazam [32], que lo que hace es buscar una canción, dada una muestra, en una base de datos con millones de canciones. El proceso en este caso es muy parecido al nuestro: construye un espectrograma y luego aplica una extracción de características diferente, al final consigue un código con los *features* importantes de la canción y lo guarda en una tabla clave-valor [33]. Esto nos demuestra que una versión de la aproximación he hemos tomado es usada en una de las aplicaciones más conocidas a nivel mundial.

El trabajo, al estar escrito en Java, puede ser utilizado en una gran diversidad de sistemas operativos y hardware. El proyecto ha llevado a cabo las etapas individuales exitosamente pero al final de todo el recorrido es indiscutible ver que aún faltan cosas por pulir y hacerlas más a la medida de los datos, es decir, adaptarnos aún más a su naturaleza, por ejemplo, lo espectrogramas que tratamos con visión artificial no son imágenes que el ojo humano está acostumbrado a ver y con las que esta área de investigación trabaja, por lo que la detección del *feature* principal de la imagen se podría practicar otros métodos de extracción de *features* [34] o otra implementación del detector de blobs.

3.1 Dificultades encontradas

Como hemos mencionado anteriormente, nos topamos con muchos muros difíciles de enfrentar. Muchos de ellos los superamos tomando caminos alternativos.

El primer problema fue lo ambicioso del proyecto, el tiempo necesario para terminar un proyecto de estas características supera el tiempo el de un TFG y por tanto no pudo llevarse a cabo. Hay muchos caminos que nos habría gustado explorar más y llevar a cabo más pruebas.

Al principio del proyecto, se quería implementar una aplicación Android preparada para la interacción con el usuario. Con imágenes a tiempo real y más utilidades complementarias. Pero al tratarse de sola una persona con el tiempo limitado, y las dificultades que surgían para adaptar el guión [2] a una aplicación Android, se decidió implementar solo el procesamiento de voz y preparar el software de tal manera que se pudiera utilizar en cualquier plataforma y ser utilizado dentro de otras aplicaciones.

Más en concreto, una de las dificultades encontradas fue al principio, en la fase de investigación para empezar el desarrollo. Al ser tratamiento de señal de voz, lo que más se encuentra buscando sobre este tema son soluciones Python o Matlab. Se estudiaron Jython [17] y SL4A (Script Layer for Android) [18] como posibles enlaces pero el paso de datos de un lenguaje a otro lo complicaba innecesariamente. Más adelante surgió la pregunta de cómo enfrentarse al resultado que daba el detector de *blobs* y el entendimiento del código fuente que usamos. Al final, se solucionó como demostramos en el capítulo 2.

3.2 Trabajo futuro

Este trabajo, al tratarse de una posible vía de muchas para implementar el procesamiento y la comparación de pronunciaciones, el software tiene mucha cabida para

su mejora. Inspirándonos en los programas que existen en la actualidad y en la experiencia y toma de decisiones hecha durante el desarrollo de la aplicación, aquí tenemos algunas posibles extensiones o mejoras:

- Representación de la señal de onda y del espectrograma en tiempo real para comparar los sonidos cuando se haya optado por una interfaz de usuario.
- Comparar con varias muestras hechas por nativos en cada comparación y no solo con una.
- Estudiar el caso de pasar un reconocimiento de voz para verificar las palabras usadas además de comparar la entonación.
- Estudiar la comparación de señales de voz por correlación.

Apéndice A

Resultados de la pruebas

Incluimos el resultado de las pruebas. Estos han sido analizados y podemos ver su interpretación al final del capítulo 2.

japanese

jap0-jap1	jap0-jap2	jap0-jap3	
cosine similarity			
0,810847429	0,81526514	0,8292891	
0,814985922	0,803160716	0,836300385	
0,818677125	0,803974921	0,835998398	
0,813853739	0,804423409	0,82957254	
0,817561272	0,821427359	0,836230018	
0,819255481	0,810063266	0,831533623	
0,806133275	0,807903597	0,833627644	
0,803834195	0,809936695	0,827836174	
0,795788791	0,808295865	0,819486771	
0,825601411	0,816461538	0,83007818	
0,823707756	0,805984438	0,840893818	
0,824349414	0,828190897	0,854073003	
0,814549651	0,81125732	0,833743305	0,833743305

jap0-esp2	jap0-esp3	jap0-esp4
cosine similarity		
0,818731215	0,819600712	0,81938087
0,824557305	0,816352851	0,829677345
0,837347844	0,827401755	0,842476085
0,827057504	0,819285443	0,826838227
0,850024807	0,83467447	0,850643038
0,852784229	0,840805725	0,852617921
0,837112098	0,832689892	0,83521913
0,83782668	0,835262445	0,832221511
0,823083545	0,828440125	0,826085723
0,835641475	0,860595461	0,839857302
0,83240235	0,852452331	0,839475822
0,837212025	0,853955928	0,848015956
0,834481757	0,835126428	0,836875744

dist inf			
226,9682891	119,7979324	142,0099005	119,7979324
210,795722	146,7526165	112,4661152	112,4661152
245,7167615	154,2238309	114,0479064	114,0479064
212,6117601	141,0849085	108,6344501	108,6344501
231,9382874	142,627184	112,7729544	112,7729544
273,1557256	125,7876484	97,53681194	97,53681194
271,3466003	129,7291836	84,05360147	84,05360147
292,3076046	89,70806976	67,80539234	67,80539234
321,0394348	92,15962409	90,75671462	90,75671462
88,65882927	69,28706235	69,07837741	69,07837741
30,81308311	25,5675289	21,12612917	21,12612917
18,2059651	16,13481341	14,82971645	14,82971645

dist inf		
157,7221964	153,0210396	215,4094746
183,261855	153,5426739	165,89155
182,3958969	156,4531408	168,6657245
192,8550481	159,0048583	213,8820532
173,7517792	145,6316197	193,3628957
137,6311918	169,2764313	125,460638
133,834302	154,7651435	143,1766296
125,6139883	146,8803709	173,0871572
141,3033519	139,3681378	173,3821334
51,76887121	35,85603338	76,32970611
41,24964233	22,20920332	51,18456185
18,70195801	14,00843385	16,16463302

Dist 1			
40,56140665	20,17676598	26,48780073	20,17676598
41,39279048	20,02786049	21,86450355	20,02786049
39,8572789	22,02078721	18,69998765	18,69998765
36,47769685	20,09712905	19,13180819	19,13180819
35,98640621	17,99592864	18,35119792	17,99592864
40,23212714	19,02891286	18,64363035	18,64363035
40,35327324	19,60580444	19,58760061	19,58760061
46,51897421	19,00629201	18,50362669	18,50362669
49,64139239	27,31683011	20,8433105	20,8433105
21,89440731	13,72715422	17,66429903	13,72715422
18,20326097	10,56486794	11,28280451	10,56486794
21,15270643	10,36757746	12,05488188	10,36757746

Dist 1		
36,39861854	48,21985871	44,91630055
34,41827961	35,05694936	39,69441858
34,75336429	31,39969326	48,09999102
33,85680429	35,24288922	40,73748705
28,35170943	28,06677852	36,10174771
31,67568034	29,6641639	39,14893763
30,56590021	25,94342688	40,94877491
26,40378904	24,09604981	39,89797946
29,55584687	29,98463418	42,29264969
33,71284561	26,06727038	42,15014703
28,86484381	24,13669705	42,20621231
27,32375615	22,41421818	36,21496407

dist Frobenius			
46,11069307	33,23121729	31,02706832	31,02706832
43,83319232	34,34095176	29,9413275	29,9413275
43,43325252	34,02731669	30,31575147	30,31575147
43,07151419	32,43771006	29,70966981	29,70966981
44,2297485	31,41709357	30,28203138	30,28203138
44,33994134	34,58520349	32,07293618	32,07293618
46,39495002	31,51283606	28,45614385	28,45614385

dist Frobenius		
46,45791503	45,01328379	49,37785393
44,31169745	45,98812887	46,06196406
42,0026231	44,20131213	43,41836116
44,15523091	45,6662571	47,2102991
37,62743166	41,15147776	40,1343353
37,92028883	40,01735383	40,30346335
37,91675958	37,52530663	41,62772373

japanese

48,8480196	28,63590109	26,87107051	26,87107051	32,75508325	32,97287349	41,78249013
59,04871533	34,1497433	29,5253134	29,5253134	35,52511563	34,27390875	44,07965075
32,93010073	23,89266585	26,06094582	23,89266585	26,55545204	20,57253667	40,09522489
17,20113007	13,06197835	11,56846629	11,56846629	23,39511423	16,64107306	31,04484296
12,57051247	9,588253103	10,27339742	9,588253103	15,72600618	14,01130372	16,9409513

	jap0-kor1	jap0-kor2	jap0-kor3	
	cosine similarity			
	0,795675735	0,792867138	0,822515825	
	0,818562691	0,794861203	0,851495441	
	0,824240511	0,800166801	0,855953964	
	0,814184057	0,793347562	0,850957695	
	0,823267475	0,800116751	0,85854316	
	0,835684318	0,809869384	0,868167692	
	0,826783298	0,811053353	0,864883929	
	0,815860427	0,810246274	0,844341981	
	0,816337945	0,81636118	0,844273949	
	0,826871678	0,818919222	0,863850798	
	0,839051866	0,816872889	0,865760185	
	0,796068994	0,823036622	0,866301146	
0,836875744	0,819382416	0,807309865	0,854753814	0,854753814

	dist inf			
153,0210396	167,5850357	154,2183306	230,2603929	154,2183306
153,5426739	134,8223006	172,0517559	150,3475403	134,8223006
156,4531408	135,7571933	168,7635998	183,069152	135,7571933
159,0048583	155,2382222	178,2955472	178,2495254	155,2382222
145,6316197	135,951248	166,8610359	175,8566281	135,951248
125,460638	119,4567622	128,7338669	127,9075062	119,4567622
133,834302	115,3033875	131,6881302	120,0375659	115,3033875
125,6139883	126,5110127	90,17971059	119,8660071	90,17971059
139,3681378	122,0153822	60,44075819	141,8083423	60,44075819
35,85603338	104,8067038	41,46705684	34,34428807	34,34428807
22,20920332	27,36657553	28,48011075	19,85489665	19,85489665
14,00843385	19,05636103	17,9865045	14,39398966	14,39398966

	Dist 1			
36,39861854	48,32123743	44,13412302	22,19064159	22,19064159
34,41827961	52,21898792	44,27831329	20,04286606	20,04286606
31,39969326	38,86017671	39,64031863	20,20741917	20,20741917
33,85680429	42,62289741	44,42847106	21,90250216	21,90250216
28,06677852	39,5161181	41,08066007	20,5372616	20,5372616
29,6641639	36,74983813	39,22354343	17,77800063	17,77800063
25,94342688	26,2812435	29,70985509	17,10084284	17,10084284
24,09604981	29,55491176	30,25134196	17,52782402	17,52782402
29,55584687	26,1268956	26,48901255	17,67995742	17,67995742
26,06727038	24,24161357	34,01043969	11,2524493	11,2524493
24,13669705	19,7857567	30,82911437	11,14418873	11,14418873
22,41421818	15,90160086	18,46462876	12,71375932	12,71375932

	dist Frobenius			
45,01328379	57,55164757	54,3455512	36,43208758	36,43208758
44,31169745	47,48701869	58,17967322	31,33321477	31,33321477
42,0026231	49,27275751	56,28676847	31,35890555	31,35890555
44,15523091	54,19871114	59,62569598	30,78782718	30,78782718
37,62743166	49,19980977	53,07522925	30,7462461	30,7462461
37,92028883	47,26965901	51,48466524	28,83728398	28,83728398
37,52530663	34,04334847	40,82493275	26,6863335	26,6863335

japanese

32,75508325	35,84664906	37,05453085	27,50871803	27,50871803
34,27390875	35,40319394	28,29539318	28,09675859	28,09675859
20,57253667	33,61871267	25,21190029	14,57075791	14,57075791
16,64107306	15,24371941	18,77246746	10,89678301	10,89678301
14,01130372	13,10417162	13,55510501	11,07481903	11,07481903

kor0-kor1	kor0-kor2	kor0-kor3	
cosine similarity			
0,763086769	0,782715187	0,81500838	
0,761363766	0,765146959	0,784022378	
0,772651902	0,766800605	0,783740236	
0,758662603	0,759909335	0,805091327	
0,757651751	0,780327128	0,783019541	
0,781179209	0,768019083	0,803775126	
0,774971234	0,762402275	0,79326541	
0,770727689	0,767001374	0,792833298	
0,806298317	0,798068713	0,822350441	
0,82608174	0,791175291	0,851111324	
0,825673182	0,788442461	0,863445451	
0,803515785	0,819524098	0,865052862	
0,783488662	0,779127709	0,813559648	0,813559648

kor0-jap0	kor0-jap1	kor0-jap2
cosine similarity		
0,753518606	0,77800227	0,777387253
0,748980142	0,770897377	0,788443207
0,793969346	0,784131629	0,789614811
0,784034952	0,772311878	0,792416281
0,778266429	0,770009679	0,78899197
0,769095317	0,782707674	0,801353534
0,770120222	0,781651645	0,788482037
0,772462211	0,781574699	0,77957297
0,775553028	0,781912368	0,796799546
0,805378645	0,795865519	0,795255689
0,801210928	0,795815095	0,802843903
0,800651472	0,785285312	0,859780127
0,779436775	0,781680429	0,796745111

dist inf			
157,8427219	85,5479632	136,8750412	85,5479632
126,0701539	120,1252085	137,9939985	120,1252085
144,1689339	126,7455081	127,9171425	126,7455081
163,1938584	94,44496835	145,7349371	94,44496835
161,3957952	105,1389897	147,1318304	105,1389897
114,1349462	93,90101182	102,9038077	93,90101182
101,0135827	86,85433384	97,61131243	86,85433384
92,58295041	71,10441017	86,97870884	71,10441017
72,37177708	44,72517296	93,58342826	44,72517296
57,96504287	50,74812239	36,19616401	36,19616401
30,10180851	36,11924466	28,06447774	28,06447774
6,096876281	5,77816795	6,81465776	5,77816795

dist inf		
159,4589041	116,1298469	113,4484197
172,8406598	118,7428397	110,4342861
120,3256526	99,63481765	108,2716297
194,0403875	133,4565903	132,3637724
189,8318421	132,0555566	115,8717825
147,0121779	108,4534066	104,5282301
140,8558721	81,06193549	63,98904112
143,9701006	63,43039586	66,77965821
139,5517741	64,86366646	66,50988993
111,7542698	77,00382044	62,2342437
48,07239138	30,99271674	38,92837113
6,862517687	7,63826762	5,646428652

Dist 1			
40,08302006	35,75222921	27,40405543	27,40405543
42,23331056	38,87852086	27,81673187	27,81673187
41,93724489	41,20796584	27,35133904	27,35133904
40,24915204	37,96213332	29,44334472	29,44334472
39,99546887	33,14378516	28,0816919	28,0816919
43,48898025	37,63995314	32,24937194	32,24937194
28,03842029	36,79145239	30,85599477	28,03842029
29,08050572	35,76356383	21,99452221	21,99452221
16,76054907	20,87725556	20,132615	16,76054907
20,37091561	29,23373325	18,53554944	18,53554944
23,38336621	28,95293942	21,63726193	21,63726193
19,58282882	19,3217659	15,41153076	15,41153076

Dist 1		
39,06437882	25,14923212	24,4703707
39,79643594	24,31617022	24,30829814
32,17513126	22,20779791	22,21171797
30,11158477	27,10876892	22,82391557
35,63397144	23,81765547	25,39857508
39,9252909	29,40495179	24,65142039
40,09878048	23,07191716	20,19668854
34,49491675	21,31150689	21,41668252
35,4411223	21,41567682	17,91725772
20,39425174	24,06083959	19,68295783
19,5117203	28,78768957	30,12205351
15,28382039	20,16990145	19,09247316

dist Frobenius			
51,56043068	37,44010795	44,6035087	37,44010795
46,7088793	49,33185529	47,08361492	46,7088793
51,17357082	50,28272572	47,93771523	47,93771523
51,97953948	45,47684912	49,80252344	45,47684912
51,38741957	40,21158169	52,25304643	40,21158169
51,29843131	46,98124859	40,43918579	40,43918579
40,50809707	41,31214912	39,63461657	39,63461657

dist Frobenius		
45,84997285	33,67679425	40,38938453
53,39009574	38,77430418	39,36190429
34,75354517	30,37953386	39,08259019
44,31236598	37,06500167	40,63296883
45,76521689	39,17016204	39,63654337
51,73142596	39,68086901	37,04933979
48,96150889	29,89346582	29,14235009

korean

38,70695114	34,91737224	36,58497932	34,91737224	46,37506399	26,89212085	29,91480101
28,09174972	23,34138211	31,40362061	23,34138211	43,94983179	26,36900379	27,34931453
24,58381231	29,44522298	21,3599724	21,3599724	26,62801505	25,6943739	27,88339887
21,63243128	25,53305422	19,41131003	19,41131003	17,19254424	20,67831228	26,46851348
8,641702834	8,100831413	7,612837701	7,612837701	8,613114349	8,920911888	8,109179688

	kor0-esp2	or0-esp3	or0-esp4	
	cosine similarity			
	0,774960232	0,76117031	0,76985079	
	0,782369647	0,777490153	0,776887349	
	0,779910401	0,774505802	0,776447218	
	0,771754074	0,767556681	0,766599311	
	0,772076071	0,771498785	0,767174358	
	0,794893252	0,790880338	0,789013591	
	0,794851545	0,78688292	0,793678412	
	0,796835383	0,784595078	0,796391021	
	0,792771152	0,784342322	0,794322792	
	0,799700193	0,81720137	0,801820231	
	0,805596486	0,813516378	0,810711574	
	0,823461078	0,836867972	0,830589713	
0,796745111	0,79076496	0,788875676	0,789457196	0,79076496

	dist inf			
113,4484197	135,043392	87,95414742	150,3768599	87,95414742
110,4342861	123,3408974	122,3933847	142,5153617	122,3933847
99,63481765	111,5030152	106,5354196	128,6471703	106,5354196
132,3637724	134,285415	142,5128444	159,47866	134,285415
115,8717825	134,2071439	142,0004545	158,3258405	134,2071439
104,5282301	94,18993476	99,10945156	115,7976064	94,18993476
63,98904112	96,78041628	112,5582454	101,127204	96,78041628
63,43039586	80,86739861	79,99843747	96,9647379	79,99843747
64,86366646	75,22456247	80,14110972	83,20552423	75,22456247
62,2342437	47,08126844	41,71376256	74,13949582	41,71376256
30,99271674	29,62427945	34,02714398	36,62869039	29,62427945
5,646428652	7,183759232	7,056933011	6,101297469	6,101297469

	Dist 1			
24,4703707	36,9919981	35,97837255	49,45132125	35,97837255
24,30829814	39,13518658	37,85060077	44,90709526	37,85060077
22,20779791	37,52219777	37,24544087	47,47676055	37,24544087
22,82391557	38,63509897	40,69028344	49,86295711	38,63509897
23,81765547	37,09589877	37,96046413	50,21712601	37,09589877
24,65142039	41,76961083	47,18892672	55,34531513	41,76961083
20,19668854	42,29722111	41,38991468	54,8988134	41,38991468
21,31150689	31,4080424	36,08658069	35,44793508	31,4080424
17,91725772	29,46574111	28,35022847	40,94285809	28,35022847
19,68295783	28,14023964	29,65641803	42,03720581	28,14023964
19,5117203	30,38418283	31,90015937	34,34946315	30,38418283
15,28382039	26,38379901	23,27627389	37,35716692	23,27627389

	dist Frobenius			
33,67679425	57,06255646	47,80771328	62,95161451	47,80771328
38,77430418	54,29138488	55,16147898	60,00154999	54,29138488
30,37953386	51,01763199	52,33534737	56,22816645	51,01763199
37,06500167	59,02510481	59,7323329	65,14798638	59,02510481
39,17016204	58,98322212	59,03369704	65,04667488	58,98322212
37,04933979	46,45664919	49,31404416	52,88295463	46,45664919
29,14235009	46,2923597	47,07641981	50,61182585	46,2923597

korean				
26,89212085	40,39285939	43,30797396	47,71439898	40,39285939
26,36900379	38,20016933	41,06004015	43,94564126	38,20016933
25,6943739	30,49755717	27,92595515	40,50002272	27,92595515
17,19254424	24,86012604	27,34891042	28,85657225	24,86012604
8,109179688	10,33233615	9,444946494	11,15558875	9,444946494

esp1-esp2	esp1-esp3	esp1-esp4	
cosine similarity			
0,822641475	0,801569899	0,822860709	
0,801345552	0,792812393	0,799840004	
0,816260667	0,808916203	0,811771963	
0,802338634	0,797650732	0,800184779	
0,801979449	0,797228098	0,8005974	
0,820382565	0,799319971	0,799742282	
0,80457146	0,823290359	0,801801904	
0,829350765	0,817059418	0,79945312	
0,82710804	0,831499386	0,803517418	
0,839437056	0,827693603	0,815149852	
0,852089683	0,822165042	0,828660001	
0,848806155	0,829294421	0,839154673	
0,822192625	0,812374961	0,810227842	0,822192625

esp1-jap1	esp1-jap2
	cosine similari
0,781321483	0,774080052
0,787230543	0,791345307
0,78458393	0,77568375
0,784126667	0,773737997
0,779461515	0,787041353
0,784073929	0,788503415
0,790191026	0,773925329
0,793637814	0,771901643
0,797110116	0,77339283
0,812050708	0,812172636
0,799046339	0,816522657
0,809422287	0,815580746
0,791854697	0,787823976

dist inf			
145,3418793	123,5937754	173,2274316	123,5937754
142,2499721	141,9169067	176,7382848	141,9169067
163,1937888	152,0126961	189,7437868	152,0126961
129,213706	156,8182661	131,77201	129,213706
132,9590804	155,1784986	132,5220781	132,5220781
113,2257607	119,9029829	122,4172356	113,2257607
129,2442222	125,1621248	119,9867766	119,9867766
99,44856329	112,5299697	109,0171794	99,44856329
92,24114633	85,93142372	111,8779484	85,93142372
59,46214476	79,53435156	79,24785634	59,46214476
32,2439948	61,0316864	35,48315745	32,2439948
8,049137835	7,635113004	8,062826905	7,635113004

dist inf	
263,1608237	132,022834
241,7672547	138,2417511
276,0518859	139,2793049
238,5765212	123,0992488
227,9255317	134,2871179
239,4118433	135,3873937
225,6244679	114,1699763
228,0826233	114,8094424
220,5952136	101,3271754
119,1824726	72,76059161
59,60874704	37,02517284
8,873604182	7,390077977

Dist 1			
27,91345407	39,8935788	37,06597076	27,91345407
35,95554925	35,14048796	38,62041082	35,14048796
32,67043826	32,53894676	49,8212631	32,53894676
32,67686451	33,57056402	41,49523117	32,67686451
33,01430425	35,56417314	41,3924425	33,01430425
26,49196054	31,25718393	38,48287548	26,49196054
28,94450072	25,32806408	43,23971213	25,32806408
26,6354045	29,84330974	38,89012208	26,6354045
25,70091092	31,55497779	44,60871187	25,70091092
27,96195931	35,67324331	34,70753059	27,96195931
21,77456391	35,25759538	30,28681706	21,77456391
23,14784812	29,81966125	34,23982009	23,14784812

Dist 1	
34,4522941	38,1331934
38,5419762	39,37950724
37,40231527	42,95972073
35,17230361	40,16860306
37,89102007	42,63651074
40,7196699	38,43733605
35,52584949	41,39658914
36,05628505	39,4347472
39,96593783	37,64665043
33,35869168	26,74196138
33,1393798	36,46906785
27,75939548	35,85205015

dist Frobenius			
44,99875692	45,62033905	48,68426358	44,99875692
48,94629234	50,66934779	53,05666877	48,94629234
49,71565386	51,03513758	54,32241015	49,71565386
48,65775488	49,51608829	53,00275786	48,65775488
48,69979725	49,49610353	52,85917603	48,69979725
42,80488775	47,49103461	50,52335447	42,80488775
44,28826582	41,08432403	48,84332668	41,08432403

dist Frobenius	
48,40226264	48,93994178
47,76001292	47,15803011
48,8927066	48,66991982
45,14491493	46,74141043
45,70437154	47,27011327
47,87545758	47,25902125
43,57973388	44,36098749

spanish

38,73968424	42,72780788	47,06089456	38,73968424	42,45896015	42,32160264
35,29032118	38,10636441	41,89986834	35,29032118	42,06776195	37,95332185
29,61322975	37,04849188	37,84784534	29,61322975	31,76677481	29,1440011
20,80302327	29,61351557	26,13790434	20,80302327	24,20327855	24,84268516
9,493903603	10,89481912	10,83131644	9,493903603	11,17713736	10,79648194

spanish

esp1-jap3ty		esp1-kor1	esp1-kor2	esp1-kor3	
		cosine similarity			
0,789612692		0,814587017	0,769259761	0,816398027	
0,808359529		0,802173289	0,788402398	0,833420296	
0,797443486		0,805561026	0,774553442	0,829204232	
0,803651453		0,789356352	0,791812291	0,827621342	
0,806406525		0,788749209	0,772397487	0,831957851	
0,805857183		0,789566232	0,776743751	0,83297196	
0,799129526		0,796576994	0,761232299	0,830756177	
0,802595957		0,800609293	0,769048757	0,830691502	
0,815673012		0,821946185	0,796119034	0,83590469	
0,808791897		0,820259809	0,783203479	0,847121446	
0,815427657		0,819308783	0,791561512	0,854018528	
0,84839896		0,831588004	0,796322909	0,852990592	
0,808445656	0,808445656	0,806690183	0,780888093	0,83525472	0,83525472

		dist inf			
110,6261204	110,6261204	153,2847558	152,9496932	162,7464372	152,9496932
98,23237209	98,23237209	128,699921	174,7037004	143,9830021	128,699921
108,3172791	108,3172791	153,7655243	152,0061641	138,0032114	138,0032114
106,4326582	106,4326582	154,0678393	156,4274838	124,7495009	124,7495009
95,50206297	95,50206297	150,035166	140,1650959	140,9685662	140,1650959
102,703549	102,703549	138,0620666	115,3335786	127,8006815	115,3335786
98,16480728	98,16480728	145,9829584	130,729052	123,6478853	123,6478853
83,2852176	83,2852176	136,8926287	123,8561776	105,6694147	105,6694147
63,45193137	63,45193137	86,67057069	71,0762207	108,5219707	71,0762207
74,70550525	72,76059161	84,11826006	84,29567125	75,07945451	75,07945451
52,18277092	37,02517284	47,39295879	44,80214986	44,63144604	44,63144604
7,460970744	7,390077977	6,698070048	7,53323756	6,332681695	6,332681695

		Dist 1			
40,13377298	34,4522941	29,91386291	35,27650175	40,34929974	29,91386291
37,96085797	37,96085797	32,21586448	38,05398751	33,45887576	32,21586448
35,84592225	35,84592225	30,04733308	30,78457107	30,69753556	30,04733308
33,04898594	33,04898594	37,32750587	34,77723306	32,79682412	32,79682412
39,12086111	37,89102007	36,35701229	33,82475057	31,60804338	31,60804338
35,67081749	35,67081749	34,65938594	32,0156622	31,68767604	31,68767604
33,78114496	33,78114496	28,10200936	40,6188498	30,61952881	28,10200936
30,70594353	30,70594353	26,6162232	34,67860775	26,46591666	26,46591666
32,81584203	32,81584203	30,17425669	32,64913834	29,05858326	29,05858326
33,62377437	26,74196138	26,92928576	40,69917736	37,00809659	26,92928576
43,25744013	33,1393798	36,40636008	37,4416361	34,25604344	34,25604344
34,03289886	27,75939548	28,63430026	29,98428419	27,09911416	27,09911416

		dist Frobenius			
49,40471169	48,40226264	42,5208845	49,11080152	57,39820372	42,5208845
44,3930431	44,3930431	42,03275167	51,12236196	44,08464987	42,03275167
48,20007353	48,20007353	44,49445688	48,62002687	48,0796948	44,49445688
43,84372865	43,84372865	50,16954234	47,46445689	46,18189956	46,18189956
42,86446887	42,86446887	49,51658117	46,49835729	45,41719078	45,41719078
45,48749338	45,48749338	47,24462551	45,10968048	43,38672222	43,38672222
42,3633879	42,3633879	41,52395393	46,61276608	41,89174574	41,52395393

38,48079104	38,48079104	39,07902694	42,74110228	39,53907409	39,07902694
32,05178539	32,05178539	33,46673689	34,59697172	37,33967091	33,46673689
31,95884226	29,1440011	31,32099157	44,43523674	32,3554433	31,32099157
32,76137116	24,20327855	28,99165784	31,54390859	25,18367613	25,18367613
11,15767669	10,79648194	10,13326335	11,64252492	10,58601881	10,13326335

Bibliografía

[1] Repositorio del projecto https://github.com/Aryalexa/LearnLanguage

[2] Algorithm - How to detect how similar a speech recording is to another speech recording, https://stackoverflow.com/questions/17010516/how-to-detect-how-similar-a-speech-recording-is-to-another-speech-recordin

[3] Muestro digital, https://es.wikipedia.org/wiki/Muestreo_digital

[4] Cuantificación, https://es.wikipedia.org/wiki/Cuantificación_digital

[5] *Pulse-code modulation* (PCM) https://en.wikipedia.org/wiki/Pulse-code_modulation

[6] Señal de voz, https://es.wikipedia.org/wiki/Se~nal_de_voz

[7] Voice Acoustics: an introduction, http://newt.phys.unsw.edu.au/jw/voice.html

[8] Umbrales de la audición, http://www.eumus.edu.uy/docentes/maggiolo/acuapu/umb.html

[9] Límites de audición, https://sites.google.com/site/lasondasyelsonido/ontaminacion-acustica/limites-de-audicion

[10] Record, play and visualize raw audio data in Android, https://www.newventuresoftware.com/blog/record-play-and-visualize-raw-audio-data-in-android

[11] Teorema de muestreo, https://es.wikipedia.org/wiki/Teorema_de_muestreo_de_Nyquist-Shannon

[12] WAVE PCM soundfile format, http://soundfile.sapp.org/doc/WaveFormat/

[13] Audacity http://www.audacityteam.org/

[14] Filtro electrónico, https://es.wikipedia.org/wiki/Filtro_electrónico

[15] Filtro digital, https://es.wikipedia.org/wiki/Filtro_digital

[16] Respuesta al impulso, http://www.atmel.com/Images/doc2527.pdf

[17] Jython, http://www.jython.org/

[18] SL4A, https://github.com/damonkohler/sl4a

[19] Butterworth filter, https://github.com/scipy/scipy/blob/v0.18.1/scipy/signal/filter_design.py#L1861-L1932

[20] Cooley, James W.; Tukey, John W. (1965). "An algorithm for the machine calculation of complex Fourier series". Mathematics of Computation. 19 (90): 297–301. ISSN 0025-5718. doi:10.1090/S0025-5718-1965-0178586-1

[21] Detección de blobs, https://en.wikipedia.org/wiki/Blob_detection

[22] Metaballs (also known as: Blobs), http://www.geisswerks.com/ryan/BLOBS/blobs.html

[23] Processing, https://processing.org/

[24] RGB, https://en.wikipedia.org/wiki/RGBA_color_space

[25] Duolingo, https://es.duolingo.com/

[26] Rosetta stone, http://www.rosettastone.com/speech-recognition

[27] Tellmemore, http://www.edukwest.com/rosetta-stone-acquires-tell-me-more/

[28] Mango, https://blog.mangolanguages.com/intuitive-language-contruction-part-ii-pronunciation/

[29] Saundz, http://saundz.com/the-saundz-way-of-teaching-english-pronunciati

[30] VowelViz, http://completespeech.com/vowelvizpro/

[31] Babbel, https://blog.babbel.com/tech-background-babbel-speech-recognition

[32] Shazam, https://www.shazam.com/es

[33] , How Shazam works, https://laplacian.wordpress.com/2009/01/10/how-shazam-works/

[34] Feature extraction, https://en.wikipedia.org/wiki/Feature_extraction