UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

PreBird: Biblioteca Open Source de Python para el pre-procesamiento de audio, orientado a canto de aves

TESINA

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Liber Adrián Hernández Abad

DIRECTOR DE TESINA

Dr. Ivan Vladimir Meza Ruiz

Ciudad Universitaria, Cd. Mx., 2021

Índice

Introducción

En la vida moderna la computación se encuentra prácticamente en todas las áreas imaginables, desde las ciencias hasta las artes, y las aplicaciones computacionales en cada una de estas áreas son también un objeto de estudio porque estas pueden llegar a ser amplias, profundas y diversas. Las computadoras son herramientas multidisciplinarias que también han encontrado su lugar en la biología, en el estudio de la vida, y en una de sus disciplinas: la Bioacústica.

Entendiendo esto, el Departamento de Ciencias de la Computación del IIMAS (Instituto De Investigaciones En Matematicas Aplicadas Y En Sistemas por sus siglas) de la UNAM crea un programa de investigación en el cual los alumnos de carreras afines (Biología y Computación) pueden participar y realizar servicio social en él; dicho programa de investigación se llama *Sistemas inteligentes para la biología* (clave 2019-12/53-901).

La meta de este programa, entre otras, es construir una Inteligencia Artificial que sea capaz de reconocer especies de aves a través de sus cantos.

La importancia del reconocimiento de cantos de aves es su utilidad, la cual va desde censos, obtención de densidad poblacional, comprensión de su distribución en un ecosistema (Richards, 1981) así como el estudio de las migraciones y comprensión de la conducta de las especies. También se usan los cantos para identificar nuevas especies y las relaciones de aves con el macrosistema (Tubaro, 1999). Todo esto puede resultar muy benéfico en los campos de la biología, bioacústica y ornitología.

Para construir la Inteligencia Artificial que reconoce la especie de ave a través de su canto es necesario, a grandes rasgos, pasar por los siguientes pasos:

- Obtención del canto de un ave: El trabajo de campo hecho por los biólogos, donde utilizan herramientas para grabar audio donde se aprecie el canto de un ave o múltiples aves, sin importar la especie a la que pertenezcan.

- Catalogación de cantos: Regularmente realizado por las mismas personas que grabaron los audios. Asocian datos a los audios obtenidos, incluyendo nombres de las especies escuchadas en el audio, lugar donde se grabó y la fecha.
- Preprocesamiento del canto: Mediante técnicas de procesamiento de señales, como operaciones morfológicas, se elimina la mayor cantidad de ruido posible del audio y se recorta el audio para obtener uno nuevo donde solo es percibido el canto del ave.
- Obtención del espectrograma del canto de ave: El resultado de este proceso es una imagen con el espectrograma del canto de un ave. En este espectrogama son observables las frecuencias en las que canta el ave.
- Entrenamiento de red neuronal: Mediante técnicas de Deep Learning (Rama de la Inteligencia Artificial dedicada al aprendizaje automático) se realiza reconocimiento de imágenes. La CNN (Red Neuronal Convolucional por sus siglas en inglés) procesa una gran cantidad de espectrogramas hasta que es capaz de distinguir entre cuál espectrograma corresponde al canto de un ave y cuáles no.
- Reconocimiento del canto de ave: Finalmente, el producto debería ser una combinación de lo mencionado anteriormente: Un programa que sea capaz de determinar la especie de un ave a través del audio de su canto.

Al leer estos pasos se pueden intuir múltiples problemas a resolver como lo son la recuperación de muestras de datos para la Inteligencia Artificial, la codificación para *machine learning*, el entrenamiento de la red neuronal y su calibramiento, etc. También queda abierta la pregunta de cómo la CNN puede no solo distinguir que un audio es el canto de un ave, sino a qué especie pertenece dicho canto. La solución desarrollada en esta tesina concierne al procesamiento de audio, por lo tanto resuelve los problemas que conciernen al paso llamado Preprocesamiento del canto de ave y obtención del espectrograma. En este paso se generará un espectrograma para su identificación en la red neuronal, lo cual da como resultado el reconocimiento de la especie de ave que emitió el canto. Durante el proceso también se explicará cómo se resuelve el problema donde la CNN distingue las especies de aves.

Actualmente Python es uno de los lenguajes de programación más usados en el mundo, manejando el paradigma de programación orientada a objetos, este lenguaje ocupa el primer lugar en búsquedas de

tutoriales para su aprendizaje en el motor de búsqueda de Google (Carbonnelle, 2020). Tan solo en abril de 2019 existían 8.2 millones de desarrolladores que codificaban en Python, sobrepasando incluso a la comunidad de codificadores más grande anterior: Java, la cual tenía 7.6 millones de codificadores en la misma fecha (Tung, 2019). Adicionalmente a todo esto, también existe una biblioteca en Python que permite trabajar con CNN de forma eficaz, esta biblioteca se llama Pytorch. Por otra parte, Python es un lenguaje de sintaxis simple, reduciendo el tiempo de codificación contra otros lenguajes y además existe un catálogo amplio de bibliotecas que satisfacen todas las necesidades del proyecto, tales como Numpy y Matplotlib.

Tomando en cuenta los datos mencionados en el párrafo anterior, resulta conveniente pensar en una solucion codificada en Python para los problemas que se presentan en el programa de Sistemas inteligentes para la biología a la hora de llevar a cabo una Inteligencia Artificial como la que se busca construir. La forma en la que se ofrece la solución es en una biblioteca de Python de código abierto, que contenga todas las funciones necesarias para procesar audio y dar el formato más adecuado para una CNN que procese su espectrograma. Esta solución será abordada en este texto, explicando todo el desarrollo y los resultados finales.

El capítulo consecuente a esta Introducción será el Marco Teórico, donde se abordarán una justificación al proyecto desde el punto de vista de conservación de las aves así como la teoría necesaria que lleva al reconocimiento de espectrogramas mediante Redes Neuronales (pasando desde la digitalización del canto, la obtención de su espectrograma hasta el reconocimiento de imágenes) y una descripción de la herramienta que permite publicar la biblioteca de Python: Python Package Index. Después los siguientes capítulos serán los experimentos con los que se probará la funcionalidad de la biblioteca y los resultados obtenidos. Finalmente, se abordarán las conclusiones del proyecto donde también se describe la expectativa del alcance de este y posibles mejoras para siguientes versiones de la biblioteca y sugerencias para que la Inteligencia Artificial final tenga un rendimiento más alto.

Objetivos

General

Desarrollar mediante técnicas de Inteligencia Artificial una CNN que sea capaz de reconocer e identificar cantos de aves para auxiliar trabajos relacionados a la Bioacústica.

Específico

Desarrollar una biblioteca para Python, de código abierto, para el pre-procesamiento de audio que sirva como herramienta para proyectos en el área de la bioacústica en el Departamento de Ciencias de la Computación en el IIMAS.

Marco Teórico

En este capítulo se encuentra la teoría con la que se entendió y realizó el proyecto, el estado del arte en donde podemos ubicar el contexto para su utilidad y comprende de 5 temas fundamentales:

1. **Reconocimiento de aves:** En este se aborda la importancia del reconocimiento e identificación de los cantos de aves, por qué importa saber a qué especie pertenece cada canto y cómo ayuda esto a la preservación de las aves, así como el lugar que ocupan las especies de este grupo de animales en las listas de peligro de extinción.

2. **Procesamiento de señales de audio:** Explicación del cómo se digitalizan los cantos de aves y cómo se procesa el audio para trabajar con él en el proyecto. También se menciona un poco sobre el rango de frecuencias importantes para el proyecto.

3. **Espectrogramas de cantos de aves:** La parte fundamental de la biblioteca es que sea capaz de generar espectrogramas limpios que sean útiles para una Red Neuronal Convolucional y reconocimiento de imágenes.

4. **Deep learning, Redes Neuronales Convolucionales y Reconocimiento de imágenes:** A pesar de que en la biblioteca elaborada en este proyecto no se trabaja con Redes Neuronales es importante mencionar y comprender la teoría de estos temas porque en el servicio social para el IIMAS el procesamiento de audio y generación de espectrogramas es solo una parte, previa a esto. Entender qué sigue después de lo que realiza la biblioteca ayuda a optimizarla con respecto al proyecto.

5. **Python Package Index:** Breve explicación sobre la herramienta con la que la biblioteca puede ser pública para la comunidad de Python y también con la que se facilita la distribución a más personas que deseen trabajar en el tema.

Reconocimiento de cantos de aves

Hasta hace poco las aves se consideraban uno de los grupos de animales mejor estudiados con el aproximado de 10.000 especies a nivel mundial (SEO BirdLife, 2016), sin embargo, en el año de 2016 la revista científica PLoS ONE publicó un artículo donde se encontró evidencia de que existen alrededor de 18.000 especies de aves (Barrowclough *et al.*, 2016). De este gran grupo de animales, 1 de cada 8 especies se encuentra en algún nivel de amenaza de extinción.

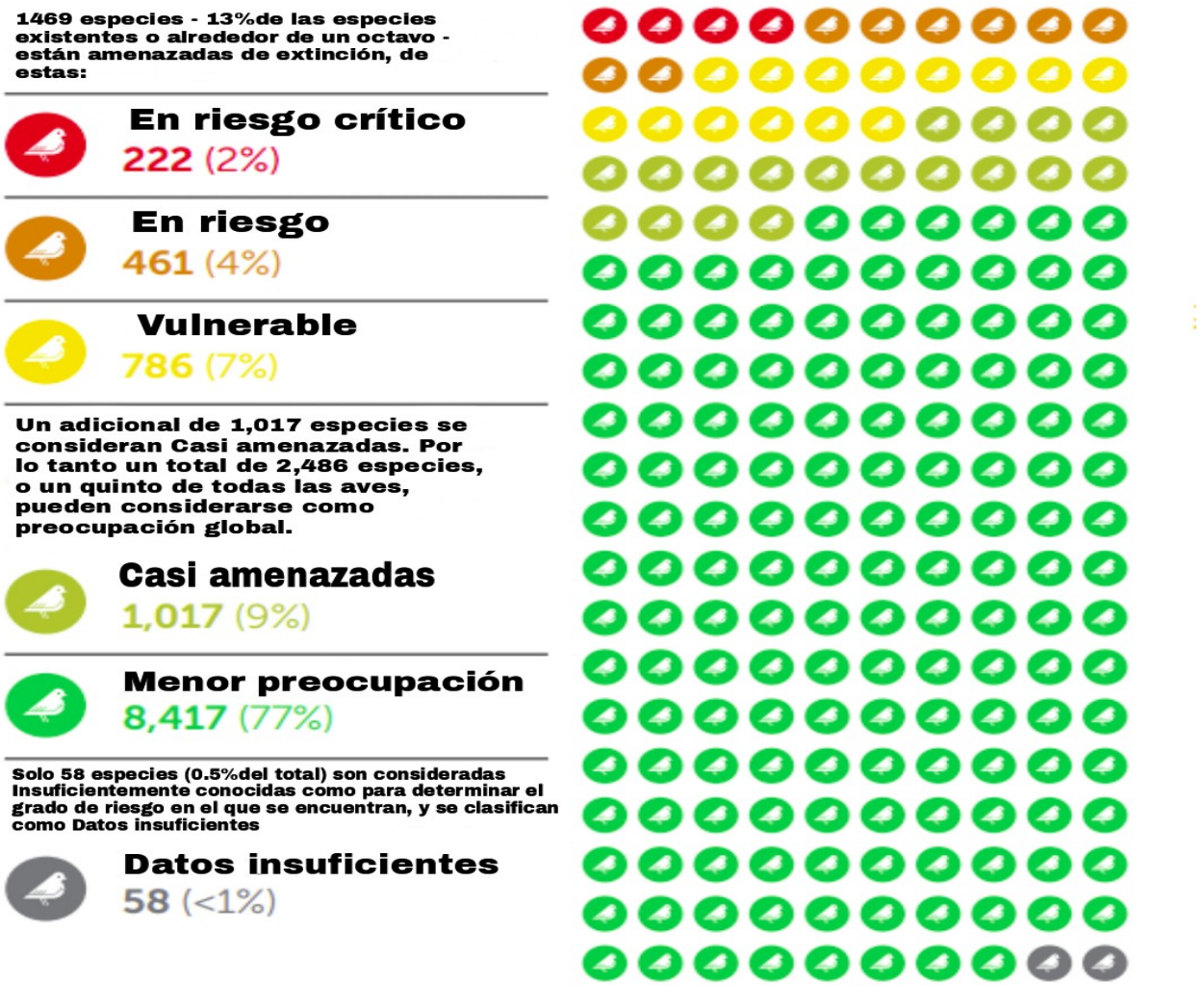

Figura 1. Niveles de amenaza de extinción de aves considerando 10,961 especies reconocidas. (Traducido y tomado de Birdlife International, 2018)

Adicional a las aves amenazadas hoy en día, se estima que un total de 182 especies (161 especies con certeza) se han extinto desde el año 1500, de las cuales 19 fueron tan solo en los últimos 25 años del siglo XX y 4 posibles extinciones a partir del año 2000. La rapidez de las extinciones incrementa con la pérdida de hábitats naturales por actividades humanas como la principal causa (Birdlife International, 2017).

México alberga aproximadamente el 10% de especies de aves a nivel mundial con 1120 (más de las que existen en el territorio combinado de Estados Unidos y Canadá), de las cuales otro 10% (102 especies) son endémicas del país, es decir, casi una de cada diez aves mexicanas viven exclusivamente en el país (Berlanga *et al.,* 2015). Inevitablemente existen especies mexicanas en algún tipo de amenaza de extinción, según la Lista Roja de la IUCN (2013) hay 110 especies de aves de México amenazadas.

Es así como ha surgido la necesidad de metodologías que permitan el monitoreo de poblaciones de aves. Ralph *et al.* describen múltiples métodos de monitoreo, estos se auxilian de la escucha de cantos de aves, por mencionar algunos, los cantos de aves son útiles para el método de Censo de búsqueda intesiva y en el Método de búsqueda de nidos (Ralph *et al.,* 1996).

Con las tecnologías de hoy en día es posible procesar los cantos de aves de forma digital, obteniendo grabaciones más limpias que permitan identificaciones más eficientes. La digitilización de los cantos de aves no solo permite una aproximación distinta al manejo de la información recopilada, también abre la posiblidad de procesos computacionales como el Reconocimiento de imágenes para automatizar el trabajo y para el desarrollo de herramientas auxiliares en el trabajo de campo. De esto proviene la importancia del correcto procesamiento del audio.

Procesamiento de señales de audio

El punto de partida del procesamiento de señales de audio es disponer de los audios, en este caso los cantos de aves. Obtener una señal análoga del mundo y grabarla, y es en este mismo proceso donde las herramientas actuales de grabación discretizan las señales por su cuenta, lo cual resulta en la digitalización de la señal. El resultado final de la grabación es un archivo de audio en formato wav.

El formato de audio conveniente para los fines de procesamiento es cualquiera que no esté comprimido o que esté comprimido sin pérdida (lossless), es decir que existe menor pérdida de información de la señal análoga (señal original), lo cual permite trabajar con mayor exactitud en obtención de tiempos, amplitud y frecuencias de la señal. El formato wav cumple con esto (aunque también puede contener audio con pérdidas) y la calidad de audio es alta. Es destacable mencionar que los archivos de audio suelen ser pesados por la cantidad de información que poseen, por lo tanto el procesamiento también es más lento y trabajar con grandes cantidades de wavs requiere grandes cantidades de recursos computacionales (memoria y tiempo).

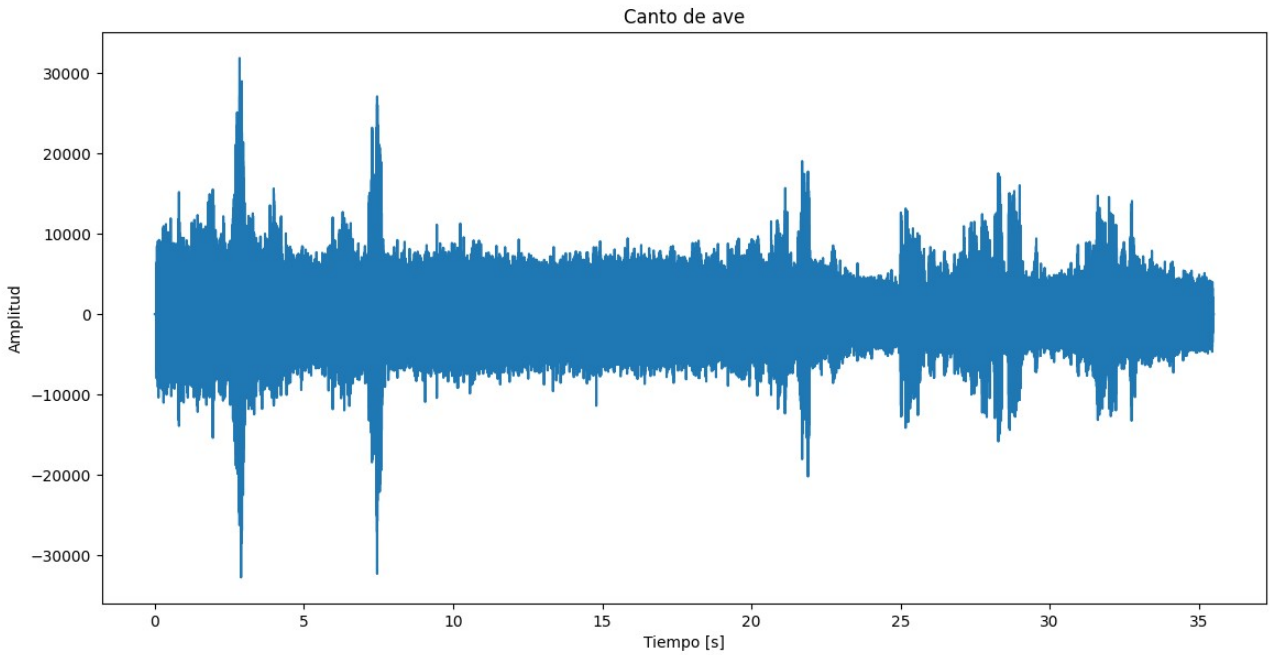

Figura 2. Canto de ave digitalizado. En el eje de las abscisas se encuentra el tiempo y en el de las ordenadas la amplitud de la señal. Gráfica generada en Python.

En la figura 2 es observable la señal digitalizada, en una gráfica de amplitud contra tiempo. No obstante esta información no es tan valiosa por sí sola ya que a simple vista no podemos determinar la frecuencia del canto del ave. Las aves son capaces de emitir cantos que van desde los 2,000 hasta los 7,000 Hz, esto quiere decir que todos los cantos de aves son audibles para el ser humano puesto que el rango de frecuencia audible que poseemos está entre los 20 y 20,000 Hz. Conocer la frecuencia del canto de un ave es importante ya que las especies cantan en frecuencias distintas, articulando sílabas y frases distintas mediante un órgano llamado siringe (Gordillo *et al.*, 2013). Las sílabas y frases emitidas

por las aves forman figuras distintas en un espectrograma y la frecuencia a la que suenan determinan el lugar de las figuras en él, como se puede observar en la Figura 3.

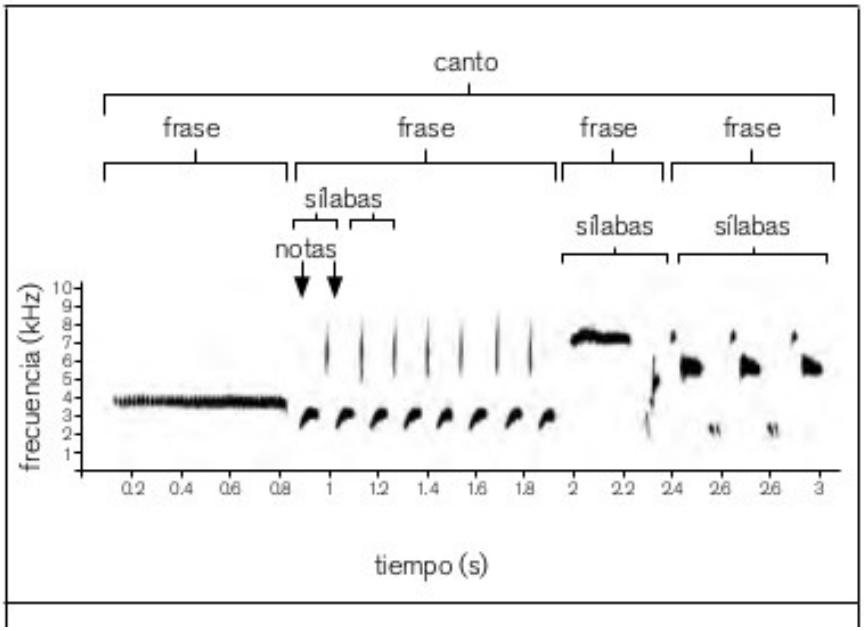

Figura 3. Espectrograma del gorrión melódico *(Melospiza melodía)* en donde se representa la división del canto en notas, sílabas, frases y canto. Tomado de *Estructura y evolución de las vocalizaciones de las aves* (Gordillo *et al.*, 2013).

Espectrogramas de cantos de aves

Un espectrograma es una representación gráfica de la frecuencia de una señal contra el tiempo en la que se manifiesta. Existe más de un método para obtener el espectrograma de una señal, el empleado en la biblioteca es con la STFT (Transformada de Fourier de Tiempo Corto por sus siglas en inglés); en este método se toman segmentos, o ventanas, de la función variante en el tiempo y a estos segmentos se les aplica la Transformada de Fourier. El resultado de transformar cada segmento es una línea vertical en la imagen, que a su vez representa una magnitud en frecuencia en un momento específico del tiempo. Se utiliza STFT en particular porque la función *spectrogram* de la biblioteca Scipy, que obtiene el espectrograma de la función, tiene definido este método.

Es importante comprender por qué la STFT se aplica de segmento en segmento: Por definición, la Transformada de Fourier se aplica a funciones invariantes en el tiempo y periódicas, pero en las aplicaciones de la vida real no trabajamos siempre con funciones que cumplan estas características

(como lo son las generadas por los cantos de aves) por lo que segmentar pequeños fragmentos de la señal es tratarlos individualmente como funciones que pudieran ser invariantes en el tiempo y periódicas.

No obstante esta no es la única dificultad a superar: Cuando el número de periodos de la señal discreta fragmentada no es un número par entonces los extremos del fragmento de señal son discontinuos. La discontinuidad en la señal provoca que los componentes de los extremos de la señal sean convertidos al dominio de la frecuencia como altas frecuencias, las cuales no corresponden a la señal original. Para solucionar este problema se usan funciones de ventaneo. Las funciones de ventaneo se multiplican finitamente a través del tiempo, acercándose al 0 en los extremos. En la Figura 4 se aprecia la función de ventaneo utilizada en esta solución: Ventana de Hann.

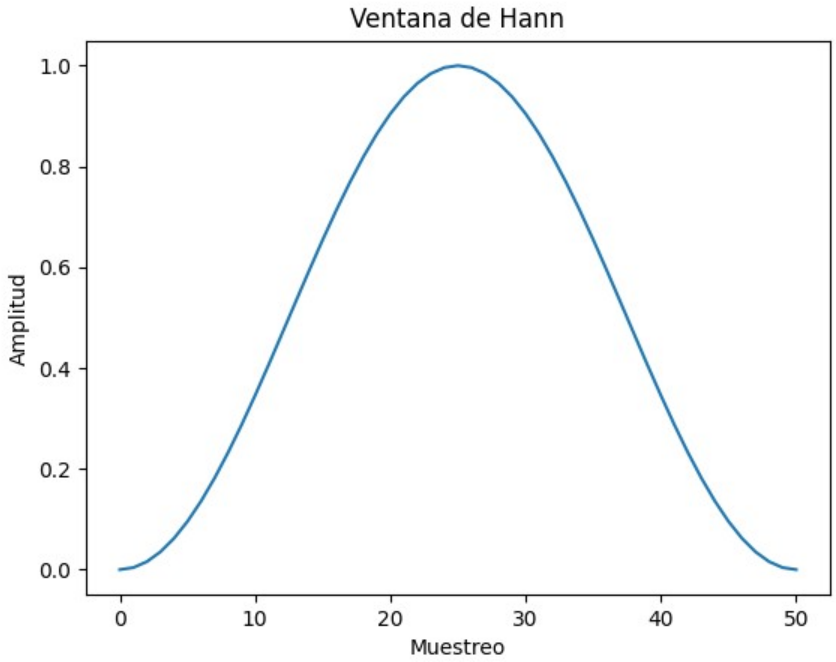

Figura 4. Ventana de Hann. Figura generada en Python.

A pesar de que no existe un método exacto para elección de funciones de ventaneo, hay algunos motivos por los cuales la ventana de Hann es la más adecuada: Los espectrogramas resultantes tienen menor actividad en frecuencias altas cuando se usa esta ventana (se utilizaron todas las funciones de ventana disponibles como Tukey y Triangular por mencionar algunas) y esto se debe a que esta función

evita la fuga espectral reduciendo la energía en ambos límites de la señal, en los que aproxima los valores a 0, es decir, cuando se hacen los saltos entre ventana y ventana el principio y el final de cada segmento de la señal pueden no coincidir, lo cual sería tomado como discontinuidad de señal en la Transformada de Fourier (recordando que esta aplica para señales periodicas) y se vería reflejado como frecuencias altas inexistentes en el espectrograma por lo que aproximar el principio y final de la señal a 0 evita esto. Así mismo también es la más utilizada para propósitos generales.

Cuando se generan espectrogramas también hay que tomar en cuenta un aspecto más: El sobrelape. Se explicó previamente que la función de ventana se aplica por segmentos a toda la señal. Este concepto se refiere a aplicar la siguiente función de ventana no al final de la ventana anterior sino sobrepuesta. Al sobreponer las funciones de ventana se reduce la fuga espectral y a mayor número de sobrelapes se aumenta la nitidez del espectrograma, recordando que aplicando la STFT a un segmento de la señal se obtiene una parte del espectrograma. En la biblioteca se utiliza el sobrelape predefinido por la función *spectrogram* de Scipy cuyo factor es el de la longitud de cada segmento dividida entre 8, en otras palabras, por cada segmento se iniciará el siguiente ventaneo un octavo después del principio del anterior, al cual se le aplicará posteriormente la Transformada de Fourier para obtener su valor en el dominio de la frecuencia.

Tras tomar en cuenta estas consideraciones es entonces cuando se procesan y obtienen los espectrogramas mediante Python como en la Figura 5. Sin embargo existe una consideración adicional a realizar: Se ha mencionado el auxilio de otra biblioteca de Python, llamada Scipy, para la obtención de espectrogramas, la cual corta el rango de frecuencia obtenido hasta los 20,000 Hz automáticamente porque este es el rango máximo audible para los humanos. Tomando en cuenta que prácticamente todas las aves cantan en frecuencias menores a los 10,000 Hz se toma la decisión de truncar el espectrograma hasta esa frecuencia para tener imágenes donde se observan claramente las figuras formadas por las notas, sílabas y frases de un canto. Además se descarta del espectrograma ruido del ambiente que suena a frecuencias mayores, lo cual producía espectrogramas con figuras no importantes en la parte alta, esto entorpecía el proceso de reconocimiento de imágenes. Se considera que hay frecuencias mayores a 10,000 Hz que son parte del canto del ave que pueden aparecer en el espectrograma pero esto se debe a la resonancia en el ambiente y se descartan; con el umbral determinado se observan las frecuencias fundamentales del canto de un ave claramente.

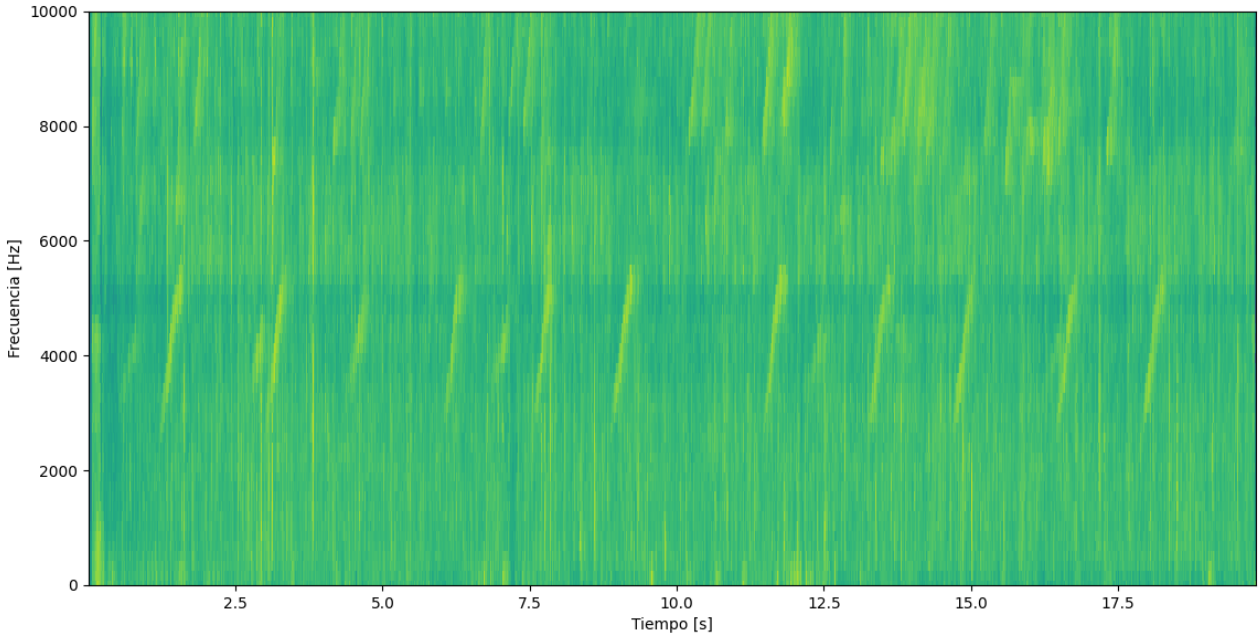

Figura 5. Espectrograma de un canto de ave digitalizado, obtenido con las funciones de Scipy. Figura generada en Python.

El espectrograma como el de la Figura 5 es casi el resultado final de procesar el audio. Más adelante, en la sección de Metodología, se explican los cambios que sufre este espectrograma antes de pasar a ser usado para el reconocimiento de imágenes.

Deep learning, Redes Neuronales Convolucionales y Reconocimiento de imágenes

Deep learning es una rama de Machine learning, ambas pertenecientes al campo de la Inteligencia Artificial. Deep learning usa modelos con múltiples capas para lograr este objetivo: Modelos matemáticos computacionales que sean capaces de realizar predicciones y ayudar en la toma de decisiones mediante el aprendizaje previo con conjuntos de datos similares.

En los sistemas de aprendizaje tradicionales se enfocan los esfuerzos para que todos los componentes funcionen correctamente y al final tenga éxito en la predicción o decisión. Esto es lo que se conoce como una Red Neuronal, compuesta por múltiples neuronas conectadas entre sí donde hay neuronas de entrada (activadas mediante sensores que perciben el entorno) y otras neuronas que se activan a través

de conexiones con peso de neuronas previamente activas. Algunas neuronas pueden influenciar su entorno a través de acciones predefinidas (Schmidhuber, 2015). Para obtener el comportamiento deseado todas las neuronas en una Red Neuronal deben conectarse de tal forma que el resultado sea el esperado a la entrada, asignando el valor necesario a cada neurona, a este problema de asignación y modificación de valores hasta que el sistema de aprendizaje tenga éxito se le conoce como *Fundamental credit assignment problem* que fue tratado por Minsky en 1969.

En Deep learning existe un modelo más complejo al explicado previamente de Red neuronal: Redes Neuronales Convolucionales (CNN). Las CNN son clave en el avance y aplicaciones de visión computacional y por lo tanto en el reconocimiento de imágenes. En una CNN existen múltiples capas profundas de procesos donde la entrada en este caso es un conjunto de pixeles correspondientes a una sección de una imagen; tienen múltiples capas ocultas y cada una puede especializarse en una tarea como el reconocimiento de bordes o curvas, las capas ocultas se clasifican en capa de convolución, capa de agrupación (*pooling layer*) y un *fully connected clasiffier* (Clasificador totalmente conectado). Se considera al *fully connected classifier* como un perceptrón que procesa las características obtenidas en capas anteriores, es decir, un conector que une los resultados de capas anteriores. Un perceptrón es un clasificador binario. Por otro lado, la capa de agrupación es donde ocurre la operación llamada *max pooling*, la cual consiste en reducir en la medida de lo posible el número de muestras de una entrada recibida, es decir, reducir las dimensiones, lo cual da lugar a predecir o asumir sobre las características en una región de la entrada. Esto se puede observar en la Figura 6. Los pixeles seleccionados de la imagen pasan por una operación de convolución (de ahí el nombre de este tipo de Red Neuronal) y después por todas las capas hasta que se tiene una salida (Valueva *et al.*, 2020). La lectura de los componentes en la salida es lo que determina la decisión o predicción del sistema.

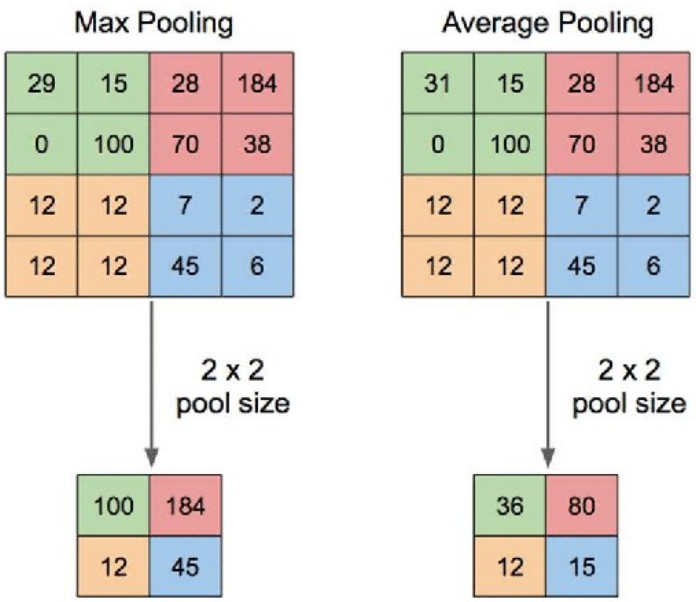

Figura 6. Representación de la operación *max pooling*. Tomado de *Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail* (Yani *et al.*, 2019).

Mediante el uso de CNN se puede aplicar el reconocimiento de imágenes. El reconocimiento de imágenes es una aplicación de estos modelos matemáticos donde al tener una imagen como entrada se determinan características u objetos en ella que permiten reconocerla como parte de una categoría. Una categoría, por ejemplo, sería la categoría de animales cuadrúpedos y una más específica sería la de perros. En el procesamiento de imágenes tradicional se utilizan filtros o kernels para resaltar características de una imagen como bordes. En una CNN se realizan múltiples convoluciones entre las capas y estas operaciones se realizan tomando una porción de la imagen para convolucionar con un kernel, el cual es de menor tamaño a la imagen. El uso de múltiples capas, y por consecuente, de más kernels da como resultado la extracción de más características en una imagen, refinando el procesamiento de la imagen para su identificación o categorización. El uso de kernels (o filtros) se ejemplifica en la Figura 7 con el Kernel de Prewitt para detección de bordes. Finalmente, la representación de toda una CNN se puede apreciar en la Figura 8.

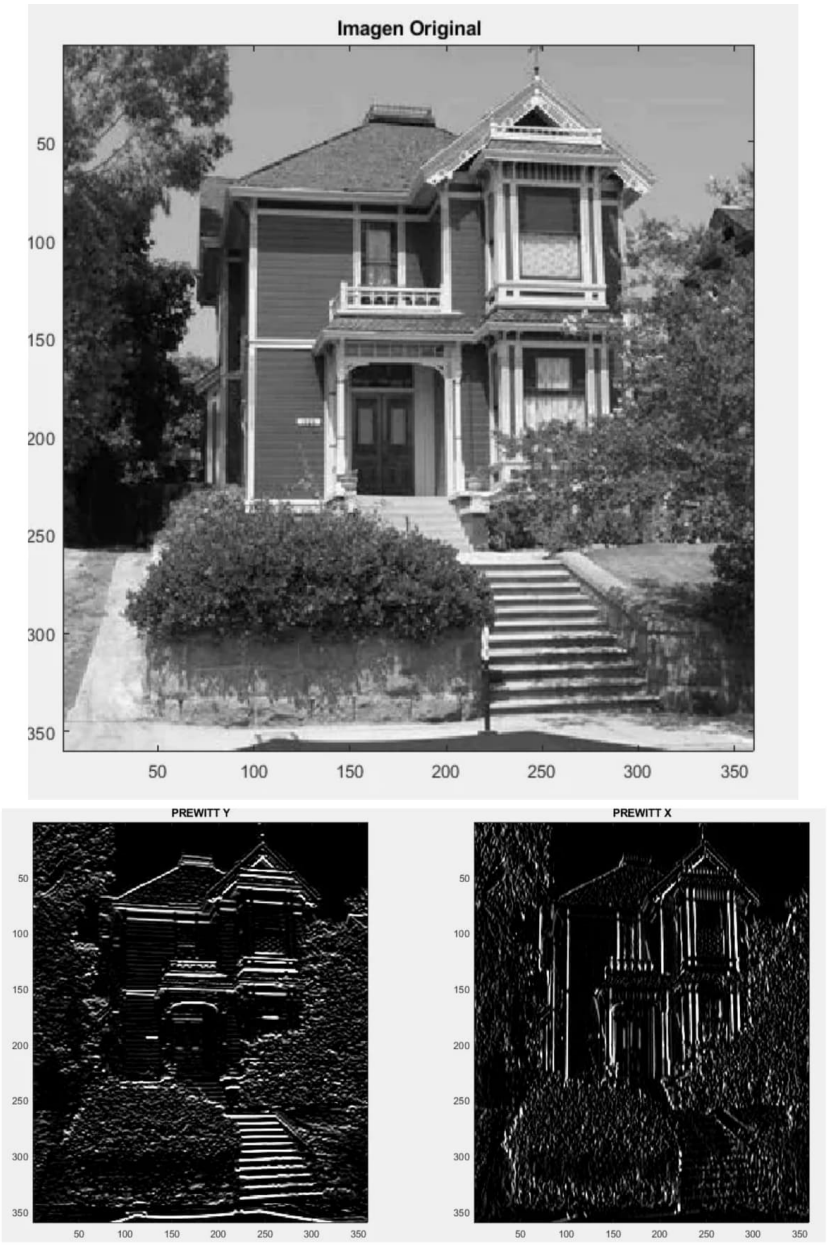

Figura 7. Arriba la imagen original, abajo a la izquierda la imagen original con Kernel de Prewitt aplicado en el eje Y, abajo a la derecha la imagen original con Kernel de Prewitt aplicado en el eje X. Figura generada mediante Matlab.

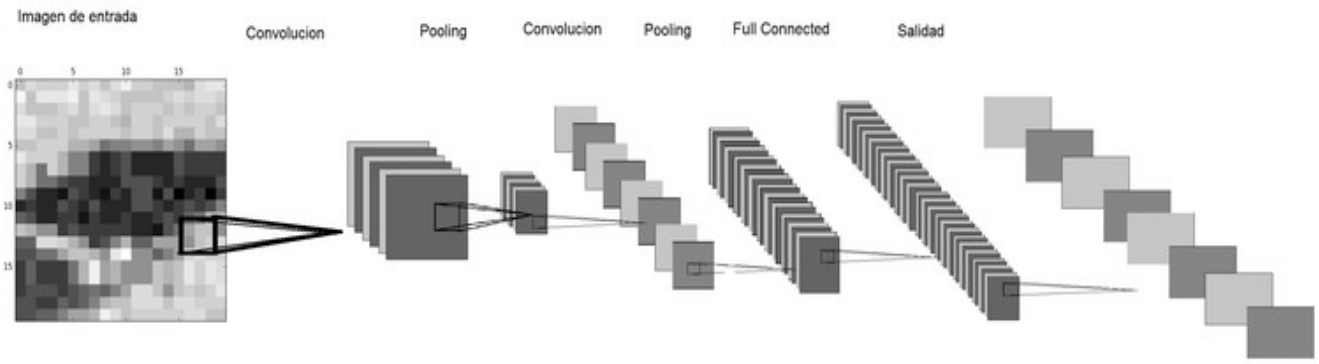

Figura 8. Representación de una CNN donde se observa la entrada, operación de convolución y las capas hasta llegar a la salida. Tomada de *Clasificación automática de coberturas del suelo en imágenes satelitales utilizando redes neuronales convolucionales: Un caso aplicado en Parques Nacionales Naturales de Colombia* (Suárez *et al.*, 2016)

En la Figura 8 se observa el proceso de una CNN que reconoce imágenes de izquierda a derecha: Primeramente se selecciona una región de pixeles de la imagen, esta se convoluciona con un kernel y pasa a la capa de *pooling*, donde se reduce el tamaño de la entrada con la operación *max pooling* y pasa a la siguiente capa, a pesar de que en la Figura 8 solo se muestran dos veces, este proceso de convolución y pooling puede repetirse a lo largo de un mayor número de capas. Después de pasar por estas capas sigue la capa donde se unen los resultados de las anteriores (la previamente mencionada *fully connected clasiffier*) para finalmente obtener una salida con pixeles clasificados.

Aún cuando en la biblioteca desarrollada no se utilizan CNN es importante comprenderlas, así como las aplicaciones que tienen, porque como se ha mencionado con anterioridad el objetivo de procesar el audio y obtener los espectrogramas es tratar a estos como imágenes, que pasarán como conjuntos de datos a una CNN para entrenarla, y que aprenda de manera no supervisada a distinguir cuáles espectrogramas pertenecen al canto de la especie de un ave mediante el reconocimiento de imágenes.

En otras palabras, el procesamiento de audio para obtención de espectrogramas es solo una parte en el esquema final en el servicio social realizado en el IIMAS para el programa de Sistemas Inteligentes para la biología, previo al uso de CNN.

Python Package Index

Finalmente, comprender toda esta teoría es lo que permitió realizar el proyecto y desarrollar la biblioteca, sin embargo existe una herramienta más usada en el proyecto. De abreviación PyPi, Python Package Index es un repositorio de software para el lenguaje de programación Python y actúa como herramienta para encontrar e instalar software desarrollado por la comunidad programadora de Python. Se usa para distribuir el software por aquellos programadores que hacen los paquetes.

Es esta herramienta con la cual se hace público el software desarrollado en Python para este proyecto y al igual que con muchos otros paquetes, se puede instalar con los comandos de pip, el cual es el instalador de paquetes para Python y permite instalar paquetes de PyPI y otros índices.

El proceso con el cual se publicará se revisará a detalle en Documentación y publicación de la biblioteca de la sección de Metodología de este texto. No obstante, a grandes rasgos consiste en estructurar el software en clases, crear los archivos necesarios para PyPi, los cuales tienen las especificaciones de otras bibliotecas necesarias, la licencia, versión, nombre del paquete, entre otros atributos.

Es destacable mencionar que se eligió una licencia de software libre por lo que cualquiera que quisiera tiene acceso al código (esto quiere decir que el repositorio en el que se encuentra el código es público) y podría tomarlo para modificarlo y darle otros usos o incluso para optimización. También se requiere crear un README.md el cual es un archivo que da una breve introducción textual al software. Otra característica notable de PyPi es que permite darle mantenimiento al software y lanzar actualizaciones a través de versiones.

Metodología

En esta sección se describirá el funcionamiento de la biblioteca, el cómo realiza sus funciones. Para llevar a cabo la descripción de cada función se agrupará a cada una en alguna de las 4 clases creadas, haciendo uso del paradigma de la programación orientada a objetos, de acuerdo a su utilidad. Las 4 clases mencionadas son: *process*, *spec*, *display* y *result*. La Figura 9 representa un esquema de la biblioteca preBird, con las 4 clases contenidas y las funciones que se diseñaron para cada una de ellas.

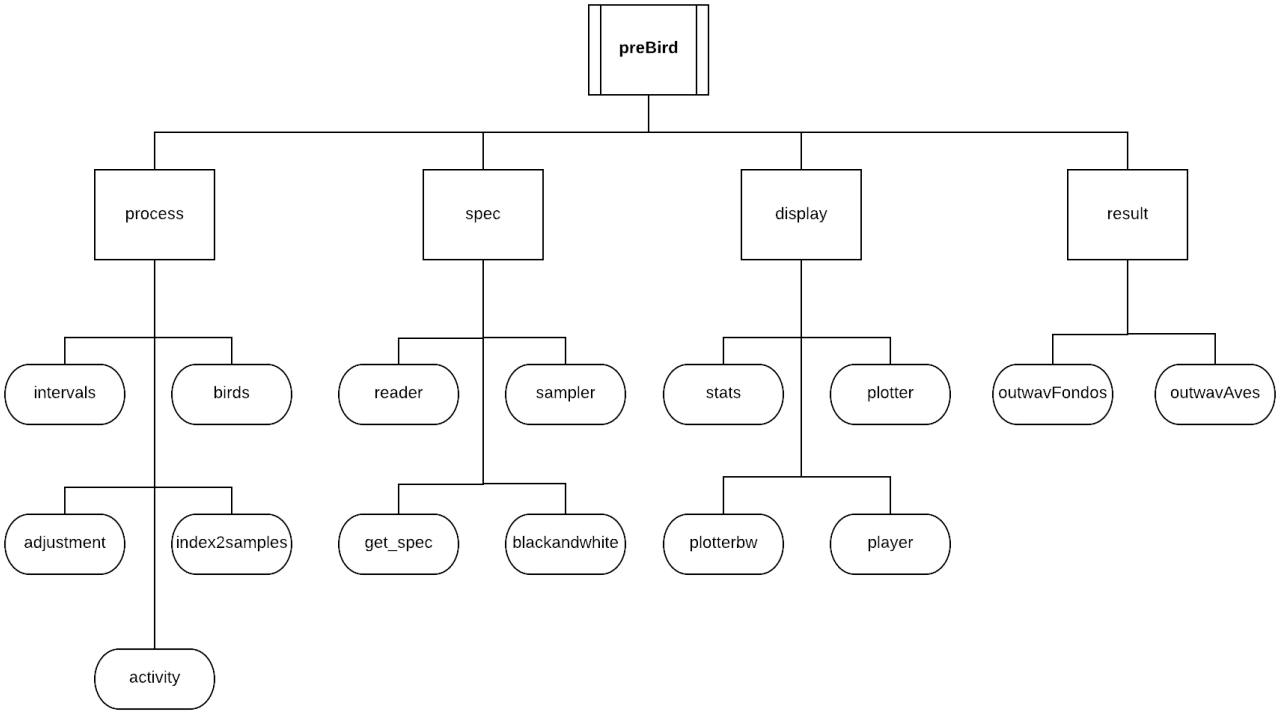

Figura 9. Diagrama que representa las clases y funciones de la biblioteca preBird. Figura generada mediante Lucidchart.

Al final de la sección se describe a detalle el proceso con el cual se publicó la biblioteca mediante Python Package Index, desde la creación de un repositorio en GitHub hasta la configuración de los archivos para su publicación.

La clase *spec* de la biblioteca

Esta clase cuenta con 4 funciones, todas dirigidas a la obtención de espectrogramas a partir de los audios utilizados por el usuario. Estas funciones son *reader, sampler, get_spec* y *blackandwhite*

reader

Parámetros de entrada: *route, lim*
Salida: *out*
Bibliotecas externas usadas: *glob*

Función donde *route* es una cadena con la ruta hacia donde se encuentran los wavs a ser utilizados. Las funciones de la biblioteca *glob* de Python se encargan de encontrar todos los nombres de archivos con cierta extensión en una ruta. La salida de esta función es una lista *out* con los nombres de todos los archivos de extensión wav en la ruta proporcionada.

sampler

Parámetros de entrada: *names*
Salida: *out_sr, out_s*
Bibliotecas externas usadas: *scipy*

En esta función se utiliza la lista de nombre de wavs obtenida en *reader* y mediante las funciones de *scipy* se obtiene el número de muestras por segundo, estos se guardan en *out_sr*, y los datos de los wav, que se guardan en *out_s*.

get_spec

Parámetros de entrada: *samples, samples_rates*
Salida: *f, t, s*
Bibliotecas externas usadas: *numpy, sklearn*

Esta función se encarga de obtener los espectrogramas de cada wav, utilizando ambas salidas de la función *sampler*. Utiliza la función *spectrogram* de *scipy* para ello. Una vez obtenido se normaliza

mediante la biblioteca *sklearn* y finalmente, se le aplica una operación de logaritmo para realzar valores que se encuentren más cercanos al 0 y para reducir los valores que se encuentran más alejados, lo que da como resultado una imagen más limpia y clara.

Los espectrogramas se guardan en la lista *s* mientras que las listas *f* y *t* contienen las frecuencias y el tiempo para cada espectrograma.

blackandwhite

Parámetros de entrada: *spectrograms*
Salida: *bwspecs*
Bibliotecas externas usadas: *numpy, sklearn*

Función para el análisis espectral, facilita la visualización de la señal en el espectrograma. Usa los espectrogramas obtenidos en *get_spec* y los normaliza mediante la biblioteca *sklearn*, después de esto selecciona todos aquellos pixeles que superen el umbral del 60% del máximo valor en el espectrograma. Los que superen el umbral se volverán 1 y los que no se volverán 0. Esto da como resultado un espectrograma con solo los valores de 1 y 0. Cada uno de estos nuevos espectrogramas se guarda en *bwspecs*.

La clase *process* de la biblioteca

Esta clase cuenta con 5 funciones: *adjustment, activity, intervals, index2samples* y *bird*. Cada una de las funciones en esta clase se dedica a realizar operaciones y procesos lógicos que afectan a los espectrogramas.

adjustment

Parámetros de entrada: *spectograms*
Salida: *spectograms*
Bibliotecas externas usadas: *morphology*

Función con una entrada *spectograms* que corresponde a una lista de espectrogramas, y una salida del mismo nombre, que contiene los mismos espectrogramas pero con aplicación de operaciones

morfológicas de dilatación y erosión. Dichas operaciones son aplicadas mediante la biblioteca *morphology* de Python.

La dilatación agrega más pixeles alrededor de los pixeles más aislados y la erosión elimina zonas con muy pocos pixeles, obteniendo un espectrograma con mayores valores de interés. Tras varias pruebas se determinó que los mejores espectrogramas eran el resultado de dos dilataciones y una erosión. Las funciones morfológicas utilizadas pertenecen a la biblioteca *skimage*.

activity

Parámetros de entrada: *specT, percentage, show*
Salida: *activities*
Bibliotecas externas usadas: *numpy, matplotlib*

Función clasificadora donde la entrada *specT* es una lista de espectrogramas. Además posee dos parámetros opcionales: *percentage* y *show*. Donde *percentage* es un porcentaje de umbral de corte para clasificar valores y su valor default es del 60% (es decir 0.6) mientras que *show* es un booleano cuyo valor default es *False* y si llega a ser *True* activa en la función un visualizador del resultado que ayuda a observar la actividad seleccionada. Esto último es mostrado con las funciones de la biblioteca *matplotlib*.

La salida de esta función es una lista que corresponde al eje del tiempo de cada espectrograma que indica si hubo actividad o no la hubo en ese instante del tiempo.

Funciona seleccionando el valor máximo del espectrograma, este valor máximo es multiplicado por *percentage*. Después, usando la función *where* de *numpy* remplaza por 1.0 todos los valores superiores al 60% del máximo valor del espectrograma y por 0.0 a cualquier valor debajo de este umbral. Con esto nos aseguramos de que los valores marcados como 1.0 son aquellos de interés en el espectrograma pues corresponden a fragmentos donde la actividad fue mayor.

Después de esto, se suman todos los valores del eje y para cada valor del eje x, resultando en un arreglo de una sola dimensión en el eje x, el eje del tiempo. Una vez que se tiene esta suma, se usa una vez más

la función *where* para remplazar los valores mayores o iguales a 1.0 por 1.0 y los demás a 0 lo que da como resultado una lista de una sola dimensión por cada espectrograma donde en el eje del tiempo hay un 0 si no hubo actividad en ese instante y 1 si esta existió.

intervals

Parámetros de entrada: *specs, show*
Salida: *indicesAve, indicesFondo*
Bibliotecas externas usadas: *numpy, matplotlib*

Función donde *specs* es una lista con la suma en el eje del tiempo de los espectrogramas obtenidos, es decir, la salida de la función *activity*. Ambos conjuntos de salida, *indicesAve* e *indicesFondo* contienen una lista de pares de números para cada espectrograma, donde el primer número del par indica un índice en el espectrograma donde inicia un segmento y el segundo indica dónde termina este segmento en el espectrograma. Estos segmentos indicados por el par del número corresponden a los segmentos del espectrograma donde hubo canto de ave o donde no lo hubo. Los segmentos con canto de ave van a la lista *indicesAve* y los segmentos sin canto de ave van a la lista *indicesFondo.*

Para lograr este resultado se lee cada uno de los espectrogramas de entrada, y mediante funciones de *numpy* se hace una diferencia de cada elemento de *specs* con el elemento en la siguiente posición, lo cual arroja un vector lleno de valores -1, 0 y 1; lógicamente esto quiere decir que donde existe un -1 es donde existe el final de uno de los segmentos marcados en el eje del tiempo y donde hay 1 es donde comienza dicho segmento. Los 0 significan los segmentos donde existe el mismo valor pero no son de nuestro interés. Después de obtener este vector se obtienen las posiciones donde haya -1 para una lista y 1 para otra lista. Son estas posiciones obtenidas las que determinan qué índices en el espectrograma original corresponden a inicios y finales de un segmento de actividad, que puede ser de canto de ave o de ruido de fondo. Finalmente se escriben estos valores en *indicesAve* e *indicesFondo*.

Si se activa a *show* se muestra una gráfica que contiene el resultado de diferenciar el vector *specT.* Esto es conveniente para depurar el programa y ajustarlo de ser necesario, también para comprender audios que presenten irregularidades.

index2samples

Parámetros de entrada: *index, samples, specs*
Salida: *convertedindex*
Bibliotecas externas usadas: Ninguna

En esta función *index* es una de las dos posibles salidas de la función *intervals*, *samples* es una de las salidas obtenidas en la función *get_spec* al igual que *specs*. La salida de esta función se llama *convertedindex* y contiene una lista índices que corresponden a las muestras discretizadas de cada *sample* en vez de al espectrograma.

Funciona con operaciones sencillas, iterando cada *sample*, multiplica cada inicio y final de *index* por la longitud de *sample* y lo divide entre la longitud de *index* para hacer la conversión de índice en espectrograma a índice en wav.

bird

Parámetros de entrada: *convertedindex, samples*
Salida: *indices*
Bibliotecas externas usadas: *numpy*

Función sencilla, donde *convertedindex* son los índices obtenidos previamente en la función *index2samples* y *samples* los muestreos de los audios que el usuario esté utilizando obtenidos en la función *get_spec*. La salida de esta función es una lista llamada *índices* cuyos valores corresponden a segmentos del wav correspondiente, estos segmentos son los tiempos en donde se escucha la actividad del canto de un ave. También se usa esta función para obtener los segmentos donde no hay canto de ave.

Para lograr esto itera sobre cada *sample*, toma los segmentos indicados por *convertedindex* y genera una lista, en donde usando los valores discretizados de cada *sample* inserta en su lugar un número en la discretización para inicio y uno para fin, colocándolos en forma de vector con la función *hstack* de *numpy*. Finalmente, la lista de índices son pares de números discretos de cada sample en los que se comienza a escuchar la actividad o no actividad del audio y en los que termina.

La clase *display* de la biblioteca

Las funciones de esta clase se orientan a mostrar los resultados y otros datos de interés provenientes de las funciones de las otras clases. Consta de 4 funciones: *stats, plotter, plotterbw* y *player*.

stats

Parámetros de entrada: *specs*
Salida: Ninguna
Bibliotecas externas usadas: *numpy*

Muestra en consola las estadísticas por cada espectrograma en una lista. El espectrograma puede ser cualquiera de los que pertencen a otras funciones de la biblioteca. Las estadísticas que muestra son la media, el máximo y el valor del umbral de corte.

plotter

Parámetros de entrada: *frequencies, times, spectrograms, specific, i*
Salida: Ninguna
Bibliotecas externas usadas: *matplotlib*

Visualizador gráfico que muestra espectrogramas, utilizando las funciones de *matplotlib* donde *frequencies, times* y *spectrograms* son los valores obtenidos de *get_spec*, siendo spectrograms una lista que puede contener espectrogramas modificados en otras funciones (con la excepción de *blackandwhite*).

El parámetro opcional *specific* (cuyo valor es False por default) indica a la función si quiere mostrar un solo espectrograma muestra de toda la lista de espectrogramas, de lo contrario mostrará la gráfica de todos los espectrogramas. Si este valor se vuelve True entonces deberá tomar el valor de *i* para saber cuál espectrograma de la lista tomar, es decir, un número entero y su valor por default es el primero de la lista, o sea el espectrograma de la posición 0.

plotterbw

Parámetros de entrada: *frequencies, times, spectrograms, specific, i*
Salida: Ninguna
Bibliotecas externas usadas: *matplotlib*

El funcionamiento es idéntico al de *plotter*, la diferencia es que esta función visualiza el espectrograma en blanco y negro, por lo cual es ideal para los espectrogramas que salgan de la función *blackandwhite*.

Visualizador gráfico que muestra espectrogramas, utilizando las funciones de *matplotlib* donde *frequencies* y *times* son los valores obtenidos de *get_spec*, y donde *spectrograms* es una lista que contiene los espectrogramas obtenidos en la función *blackandwhite*.

El parámetro opcional *specific* (cuyo valor es False por default) indica a la función si quiere mostrar un solo espectrograma muestra de toda la lista de espectrogramas, de lo contrario mostrará la gráfica de todos los espectrogramas. Si este valor se vuelve True entonces deberá tomar el valor de *i* para saber cuál espectrograma de la lista tomar, es decir, un número entero y su valor por default es el primero de la lista, o sea el espectrograma de la posición 0.

player

Parámetros de entrada: *wavs, lim, start*
Salida: Ninguna
Bibliotecas externas usadas: *playsound*

Permite al usuario escuchar los wavs contenidos en una lista. En esta función *wavs* es una lista con las rutas donde se encuentran los archivos con esta extensión para ser reproducidos y *lim* es un entero que representa el número de archivos a reproducir. Existe además el parámetro opcional *start* cuyo valor default es 0, si se especifica otro valor entonces la reproducció empezará desde dicho valor en la lista de wavs.

La clase *result* de la biblioteca

Las funciones en esta clase están orientadas a entregar los resultados finales del procesamiento realizado por las otras clases. Consta de solo 2 funciones: *outwavAves* y *outwavFondos*.

outwavAves

Parámetros de entrada: *index, route*
Salida: *wavs*
Bibliotecas externas usadas: *scipy*

Utiliza los índices *index* obtenidos en la función *bird* para cortar los segmentos indicados en el wav. Una vez que sabe los segmentos que utilizará los escribe en forma de archivo de audio de formato wav en la ruta *route* especificada por cada uno de los espectrogramas. Esto da como resultado que a partir de un archivo de audio donde se escucha un canto de ave con ruido de fondo se obtenga otro donde solo se escucha el fragmento del tiempo que contiene el canto de ave.

outwavFondos

Parámetros de entrada: *index, route*
Salida: *wavs*
Bibliotecas externas usadas: *scipy*

Funciona de la misma manera que *outwavAves*. Realiza el mismo procedimiento descrito en dicha función, las diferencia son que la cadena con la que guarda los archivos de audio y la ruta *route* son distintas pero con un formato parecido. La ruta es especificada por el usuario al momento de llamar la función y los archivos se guardan en el siguiente formato:

Fondo#.wav

Donde el caracter # es sustituido por un número entero que se encuentra entre el rango de 0 y el número de archivos de audio a generar. Siguiendo esta lógica los archivos de la función *outwavAves* se guardan de la siguiente manera:

Ave#.wav

Uso y flujo de la biblioteca

Una vez revisadas las funciones y su utilidad es también importante saber cómo se relacionan entre sí para hacer uso correcto de esta biblioteca, a continuación, en la Figura 10, se muestra un diagrama de flujo donde se conectan entre sí las funciones representando el funcionamiento y cómo las salidas de algunas funciones son entradas de otras.

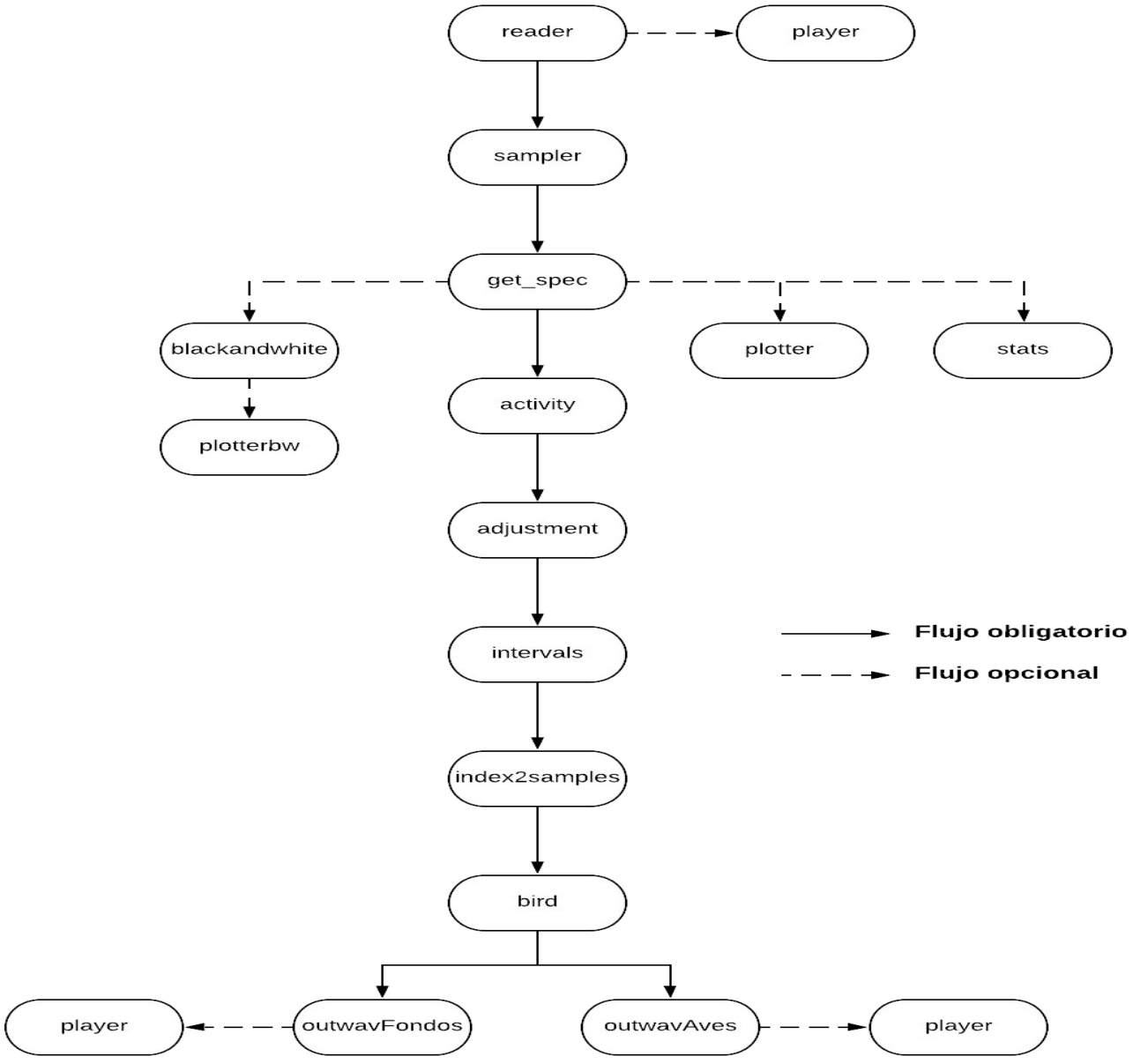

Figura 10. Diagrama de flujo que representa cómo se conectan las funciones de la biblioteca preBird. Figura generada mediante Lucidchart.

Según se detalla en la Figura 10, la primer función a utilizarse es *reader* puesto que es la que adquiere los archivos que se procesarán en las siguientes funciones (estos archivos pueden escucharse opcionalmente mediante la función *player*), posteriormente la salida de *reader* pasa a ser la entrada de la función *sampler* que obtiene la información de los archivos de audio para los siguientes pasos. La salida de *sampler* proporciona los datos suficientes para obtener un espectrograma, por ello su salida es la entrada de *get_spec*, función que se encarga de pasar al dominio de la frecuencia.

A partir de los espectrogramas se puede ir a 3 caminos distintos para el análisis: *blackandwhite, stats* y *plotter*. En la función *blackandwhite* se pasa la información de la matriz espectral a términos de unos y ceros para una visualización binomial de la actividad, es por ello que el siguiente paso de esta función es *plotterbw* que permite dicha visualización. Otro de los caminos posibles es obtener las estadísticas del espectrograma mediante la función *stats*. Finalmente se puede optar por únicamente ver el espectrograma tal y como se encuentra.

Una vez que se obtienen los espectrogramas de los audios se pasa a la función *activity* que los procesa y determina mediante un umbral qué valores son de importancia en el eje del tiempo, dictando en qué instantes del tiempo hubo canto de ave y en cuáles no. La salida de *activity* pasa a ser la entrada de *intervals* que, por decirlo de una forma, indica en qué momentos del espectrograma inicia la actividad y en qué momentos finaliza. La salida de *intervals* se convierte ahora en la entrada de *index2samples* que obtiene estos segmentos de actividad en términos del muestreo del audio.

Finalmente el procesamiento del audio entra en su última etapa, donde la función *bird* toma como entrada la salida de *index2samples* y une los segmentos en 2 categorías: Todos los segmentos de actividad y todos los segmentos de no actividad. Es entonces cuando la salida de cada categoría pasa a como entrada a otra función que puede ser *outwavAves* u *outwavFondos*, funciones que se encargan de pasar a formato wav la información de cada segmento, dando como resultados archivos de audio donde se escucha exclusicamente ruido de fondo o canto de ave. Opcionalmente estos archivos pueden escucharse mediante la función *player*.

En el siguiente capítulo “Experimentos y resultados” se demuestra cómo se usa paso a paso, mostrando el proceso explicado en esta sección.

Publicación de la biblioteca

El trabajo realizado hasta ahora tendría menos impacto para el Departamento de Ciencias de la computación del IIMAS si no se encontrara en Python Package Index (PyPi). Gracias a PyPi cualquiera puede utilizar las funciones programas mediante el simple comando que se muestra a continuación:

pip install prebird

Al usar este comando el usuario instala la biblioteca para su uso con Python, esto también puede realizarse dentro de un ambiente virtual. A continuación se enlistan los pasos seguidos para subir la biblioteca a PyPi:

1. Crear un directorio que se subirá a la plataforma Github. Este directorio debe contener una carpeta con el nombre del paquete y con los siguientes archivos:
    - Archivo con extensión .py que contiene el código programado para la biblioteca, en este caso el archivo es *prebird.py*.
    - Un archivo llamado *__init__.py* que contiene una declaración de las clases contenidas en la biblioteca

- En el directorio que se sube al repositorio de Github, afuera de la carpeta del paquete, también deben existir archivos necesarios para PyPi y son los siguientes:
    - *setup.py*: Archivo de configuración que proporciona la información principal a PyPi: Nombre del paquete, versión, licencia, descripción, autor, email del autor, url del repositorio, url de descarga, palabras clave para búsqueda del paquete, bibliotecas externas a instalar y los clasificadores.
    - *setup.cfg*: Indica el nombre del archivo con descripción, es decir, *README.md*
    - *LICENSE.txt*: Contiene todas las especificaciones de la licencia elegida para el proyecto, *prebird* tiene una licencia de clase *Academic Free License* y su etiqueta es *afl-3.0*. Esta licensia permite que cualquiera tenga acceso al código publicado y lo modifique, es decir, es una licensia open source.
    - *README.md*: Contiene la descripción de la biblioteca.

El repositorio que alimenta el paquete preBird es el que se encuentra en la siguiente liga, accessible para todo público: https://github.com/RoyFocker/preBird

2. Crear una distribución fuente mediante la siguiente línea de código:

python setup.py sdist

3. Instalar el paquete *twine* de Python y ejecutar la siguiente instrucción:

*twine upload dist/**

Con esto la biblioteca será subida a PyPi y podrá instalarse con el comando *pip*.

Vale la pena mencionar que para darle mantenimiento al código hay que actualizar los archivos necesarios en el repositorio y cambiar la versión, después actualizar la versión en el archivo *setup.py* y repetir los pasos 2 y 3 descritos previamente. Los datos referentes a la versión como la url de descarga y el número de versión se obtienen mediante Github. Adicionalmente, para llevar a cabo una nueva versión de código se utiliza Github para crear una nueva versión de la siguiente forma:

- Ir a la url donde se encuentra el repositorio.
- Hacer click en la sección de versiones.
- Seleccionar “Crear nueva versión”.
- Proporcionar el número de versión, el nombre de esta y una breve descripción.
- Dar click en “Publicar versión”.

Una vez seguido este proceso Github nos proporciona una nueva url de descarga, que habrá que actualizar en el archivo *setup.py*.

Documentación de la biblioteca

Python posee un módulo propio para generar la documentación de archivos, este módulo es llamado *pydoc* y mediante este fue que un archivo html es generado. Este archivo html puede abrirse como página web de manera local y contiene la información de las clases y métodos contenidos en cada una. Para generar la documentación primeramente hay que escribir comentarios con descripciones para todo cada clase y cada método, estos comentarios se encapsulan entre 3 comillas como se ve en este ejemplo:

“””Este es un comentario encapsulado”””

A esto se le conoce como Docstring y van al principio del código a describir, *pydoc* se encarga de reunirlas y presentarlas en un archivo html. Para generar la documentación hay que utilizar el comando que se presenta a continuación:

python3 ruta/al/modulo/modulo.py -w nombre_del_modulo

Después es necesario generar el archivo html con el siguiente comando:

pydoc3 -w nombre_del_modulo

En la biblioteca *prebird* solo existe un módulo, del mismo nombre por lo que estos pasos solo se siguieron una sola ocasión. El archivo html con la documentación resultante luce como en la Figura 11.

Figura 11. Archivo HTML generado ubicado en servidor local. Abierto y mostrado mediante Google Chrome.

La Figura 11 muestra una descripción del módulo prebird, los módulos externos que utiliza, las clases dentro del módulo, las 4 clases y los métodos de cada una, además en esta figura se observa cómo luce una de las clases y sus métodos, la clase *display* en este caso. El archivo html se encuentra en el repositorio de Github.

Experimentos y resultados

En esta sección se llevan a cabo 3 demostraciones distintas para mostrar el funcionamiento de las funciones en las 4 clases de la biblioteca. Para todas las pruebas se importó cada uno de los módulos de la biblioteca *prebird* así como también se importó *argparse* para codificar un módulo *main* que pueda leer argumentos desde la consola, todo esto se muestra en la Figura 12.

```
from prebird import process
from prebird import spec
from prebird import display
from prebird import result
import argparse

def main():
    parser = argparse.ArgumentParser()

    # Parametros requeridos
    parser.add_argument(
        "DIR",
        default=None,
        type=str,
        help="Directorio con archivos wav a procesar",
    )
    parser.add_argument(
        "--num_wavs",
        default=0,
        type=int,
        help="Numero de wavs a procesar [0] process all",
    )
    parser.add_argument(
        "--percentage",
        default=0.6,
        type=float,
        help="Umbral de corte con respecto a porcentange del maximo",
    )

    args = parser.parse_args()

if __name__ == '__main__':
    main()
```

Figura 12. Importaciones para las pruebas y el módulo main preparado para recibir argumentos desde consola. Captura de pantalla de Sublime Text.

Los argumentos que el programa recibirá son los siguientes:

- *DIR*: La ruta al directorio donde se encuentran los archivos wav con cantos de ave.
- *num_wavs*: El número de archivos a procesar, para los fines de prueba solo existen 50 wavs.

- *percentage*: Umbral de corte para los espectrogramas, a partir de este umbral se decide qué es actividad y qué no lo es.

Demostración 1

En esta prueba se procesa únicamente 1 archivo de audio y solo se usan las funciones de flujo obligatorio descritas en la Figura 10. Se muestra la salida de cada una de las funciones utilizadas. El código para llevar a cabo esto se guarda en un archivo llamdo *test1.py* y es el mostrado en la Figura 12 más el que se muestra en la Figura 13. Adicionalmente se utiliza el valor defaulta de 0.6 del parámetro *percentage*.

```
specs_names= spec.reader(args.DIR,args.num_wavs)
print("\nSalida de reader:")
print(specs_names)
samples_rates,samples=spec.sampler(specs_names)
print("\nSalida 1 (muestras usadas para cada audio) de sampler")
print(samples_rates)
print("\nSalida 2 (datos de audio) de sampler")
print(samples)
frequencies,times,specs=spec.get_spec(samples,samples_rates)
print("\nSalida 1 (Frecuencias) de get_spec:")
print(frequencies)
print("\nSalida 2 (Tiempos) de get_spec:")
print(times)
print("\nSalida 3 (Espectrogramas) de get_spec:")
print(specs)
specprueba=specs[:]
specprueba=process.adjustment(specprueba)
specprueba=process.activity(specprueba,args.percentage,show=False)
print("\nSalida de activity")
print(specprueba)
specprueba=process.adjustment(specprueba)
print("\nSalida de adjustment")
print(specprueba)

indicesAve,indicesFondo=process.intervals(specprueba,show=False)
print("\nSalida 1 (Indices con canto de ave) de intervals")
print(indicesAve)
print("\nSalida 2 (Indices con ruido de fondo) de intervals")
print(indicesFondo)

indicesAve=process.index2samples(indicesAve,samples,specs)
print("\nSalida de index2samples para canto de ave:")
print(indicesAve)
ave=process.bird(indicesAve,samples)
print("\nSalida de bird para cantos de ave")
print(ave)
indicesFondo=process.index2samples(indicesFondo,samples,specs)
print("\nSalida de index2samples para ruido de fondo:")
print(indicesFondo)
fondo=process.bird(indicesFondo,samples)
print("\nSalida de bird para ruido de fondo")
print(fondo)
wavsAve=result.outwavAves(ave,'/home/royfocker/samples/out/bird/')
wavsFondo=result.outwavFondos(fondo,'/home/royfocker/samples/out/background/')
```

Figura 13. Código del archivo *test1.py*. Captura de pantalla de Sublime Text.

Los resultados de correr este código se observan en la Figura 14 y la Figura 15 a continuación.

```
royfocker@royfocker:~$ python3 test1.py /home/royfocker/samples/ --num_wavs 1
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34257.wav...

Salida de reader:
['/home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34257.wav']

Salida 1 (muestras usadas para cada audio) de sampler
[44100]

Salida 2 (datos de audio) de sampler
[array([0, 0, 0, ..., 0, 0, 0], dtype=int16)]

Salida 1 (Frecuencias) de get_spec:
[array([    0.      ,   172.265625,   344.53125 ,   516.796875,
         689.0625  ,   861.328125,  1033.59375 ,  1205.859375,
        1378.125   ,  1550.390625,  1722.65625 ,  1894.921875,
        2067.1875  ,  2239.453125,  2411.71875 ,  2583.984375,
        2756.25    ,  2928.515625,  3100.78125 ,  3273.046875,
        3445.3125  ,  3617.578125,  3789.84375 ,  3962.109375,
        4134.375   ,  4306.640625,  4478.90625 ,  4651.171875,
        4823.4375  ,  4995.703125,  5167.96875 ,  5340.234375,
        5512.5     ,  5684.765625,  5857.03125 ,  6029.296875,
        6201.5625  ,  6373.828125,  6546.09375 ,  6718.359375,
        6890.625   ,  7062.890625,  7235.15625 ,  7407.421875,
        7579.6875  ,  7751.953125,  7924.21875 ,  8096.484375,
        8268.75    ,  8441.015625,  8613.28125 ,  8785.546875,
        8957.8125  ,  9130.078125,  9302.34375 ,  9474.609375,
        9646.875   ,  9819.140625,  9991.40625 , 10163.671875,
       10335.9375  , 10508.203125, 10680.46875 , 10852.734375,
       11025.      , 11197.265625, 11369.53125 , 11541.796875,
       11714.0625  , 11886.328125, 12058.59375 , 12230.859375,
       12403.125   , 12575.390625, 12747.65625 , 12919.921875,
       13092.1875  , 13264.453125, 13436.71875 , 13608.984375,
       13781.25    , 13953.515625, 14125.78125 , 14298.046875,
       14470.3125  , 14642.578125, 14814.84375 , 14987.109375,
       15159.375   , 15331.640625, 15503.90625 , 15676.171875,
       15848.4375  , 16020.703125, 16192.96875 , 16365.234375,
       16537.5     , 16709.765625, 16882.03125 , 17054.296875,
       17226.5625  , 17398.828125, 17571.09375 , 17743.359375,
       17915.625   , 18087.890625, 18260.15625 , 18432.421875,
       18604.6875  , 18776.953125, 18949.21875 , 19121.484375,
       19293.75    , 19466.015625, 19638.28125 , 19810.546875,
       19982.8125  , 20155.078125, 20327.34375 , 20499.609375,
       20671.875   , 20844.140625, 21016.40625 , 21188.671875,
       21360.9375  , 21533.203125, 21705.46875 , 21877.734375,
       22050.      ])]
```

Figura 14. Parte 1 de la salida de *test1.py*.

```
Salida 2 (Tiempos) de get_spec:
[array([2.90249433e-03, 7.98185941e-03, 1.30612245e-02, ...,
       3.54873469e+01, 3.54924263e+01, 3.54975057e+01])]

Salida 3 (Espectrogramas) de get_spec:
[array([[-13.02585  , -13.02585  , -13.02585  , ...,  -0.3003397,
         -2.5173569, -13.02585  ],
       [-13.02585  , -13.02585  , -13.02585  , ...,  -0.203619 ,
         -1.9848099, -13.02585  ],
       [-13.02585  , -13.02585  , -13.02585  , ...,  -1.0763178,
         -2.6817808, -13.02585  ],
       ...,
       [-13.02585  , -13.02585  , -13.02585  , ...,   5.056952 ,
          4.0483794, -13.02585  ],
       [-13.02585  , -13.02585  , -13.02585  , ...,   5.3901443,
          3.7729244, -13.02585  ],
       [-13.02585  , -13.02585  , -13.02585  , ...,   5.101883 ,
          3.165423 , -13.02585  ]], dtype=float32)]

Salida de activity
[array([0., 0., 0., ..., 1., 1., 0.])]

Salida de adjustment
[array([0., 0., 1., ..., 1., 1., 1.])]

Salida 1 (Indices con canto de ave) de intervals
[[(array([1]), 6989)]]

Salida 2 (Indices con ruido de fondo) de intervals
[<zip object at 0x7f3c465dab80>]

Salida de index2samples para canto de ave:
[[(12136, 84819804)]]

Salida de bird para cantos de ave
[array([2062, 2287,   80, ...,    0,    0,    0], dtype=int16)]

Salida de index2samples para ruido de fondo:
[[(0, 12136), (84819804, 84819804)]]

Salida de bird para ruido de fondo
[array([    0,     0,     0, ..., -1624, -1613,   237], dtype=int16)]
royfocker@royfocker:~$
```

Figura 15. Parte 2 de la salida de *test1.py*.

En estas dos Figuras se describe a qué función pertenece cada dato o set de datos. Es observable que la mayoría son datos numéricos de *numpy*. Finalmente, la salida final del programa son dos archivos de audio en formato wav, ubicados en los directorios que se pueden observar en la Figura 13. Estos archivos se muestran en las Figuras 16 y 17.

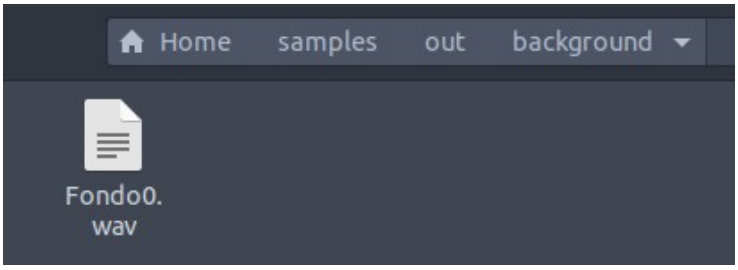

Figura 16. Archivo wav producido para ruido de fondo al correr el programa.

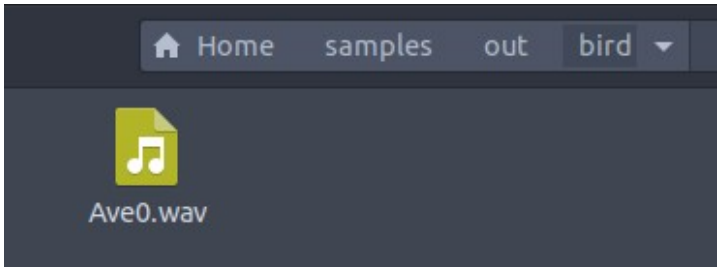

Figura 17. Archivo wav producido para canto de ave al correr el programa.

El archivo wav utilizado en esta prueba fue detectado todo como canto de ave al utilizar un umbral del 60%, por lo que el audio de ruido de fondo resulta en un archivo vacío, por lo tanto luce distinto gráficamente al de canto de ave.

Demostración 2

En esta prueba se procesan 10 archivos de audio y se usan algunas de las funciones de flujo opcional descritas en la Figura 10: *plotter*, *stats* y *player*. Se muestra únicamente la salida de las funciones utilizadas que no se encuentran en el Experimento 1. El código para llevar a cabo esto se guarda en un archivo llamdo *test2.py* y es el mostrado en la Figura 12 más el que se muestra en la Figura 18. Adicionalmente se utilizará un umbral más alto con valor de 0.8 para clasificar actividad mayor al 80% del valor máximo del espectrograma. Finalmente los parámetros de *show* para las funciones de *activity* e *intervals* cambian su valor a True para observar las ayudas gráficas que proporcionan.

```
specs_names= spec.reader(args.DIR,args.num_wavs)
samples_rates,samples=spec.sampler(specs_names)
frequencies,times,specs=spec.get_spec(samples,samples_rates)

print("\nComienza muestreo de gráficas de la funcion plotter")
display.plotter(frequencies,times,specs)

print("\nComienza muestreo de estadisticas de la funcion stats")
display.stats(specs)

specprueba=specs[:]
specprueba=process.adjustment(specprueba)

print("\nVisualizador de la función activity")
specprueba=process.activity(specprueba,args.percentage,show=True)

print("\nVisualizador de la función intervals")
indicesAve,indicesFondo=process.intervals(specprueba,show=True)

indicesAve=process.index2samples(indicesAve,samples,specs)
ave=process.bird(indicesAve,samples)

indicesFondo=process.index2samples(indicesFondo,samples,specs)
fondo=process.bird(indicesFondo,samples)

wavsAve=result.outwavAves(ave,'/home/royfocker/samples/out/bird/')

print("\nAsi es como se ve la consola cuando reproduce audio mediante la funcion player")
display.player(wavsAve,1)

wavsFondo=result.outwavFondos(fondo,'/home/royfocker/samples/out/background/')
```

Figura 18. Código del archivo *test2.py*. Captura de pantalla de Sublime Text.

Los resultados de correr este código se muestran en las Figuras 19 y 20 a continuación.

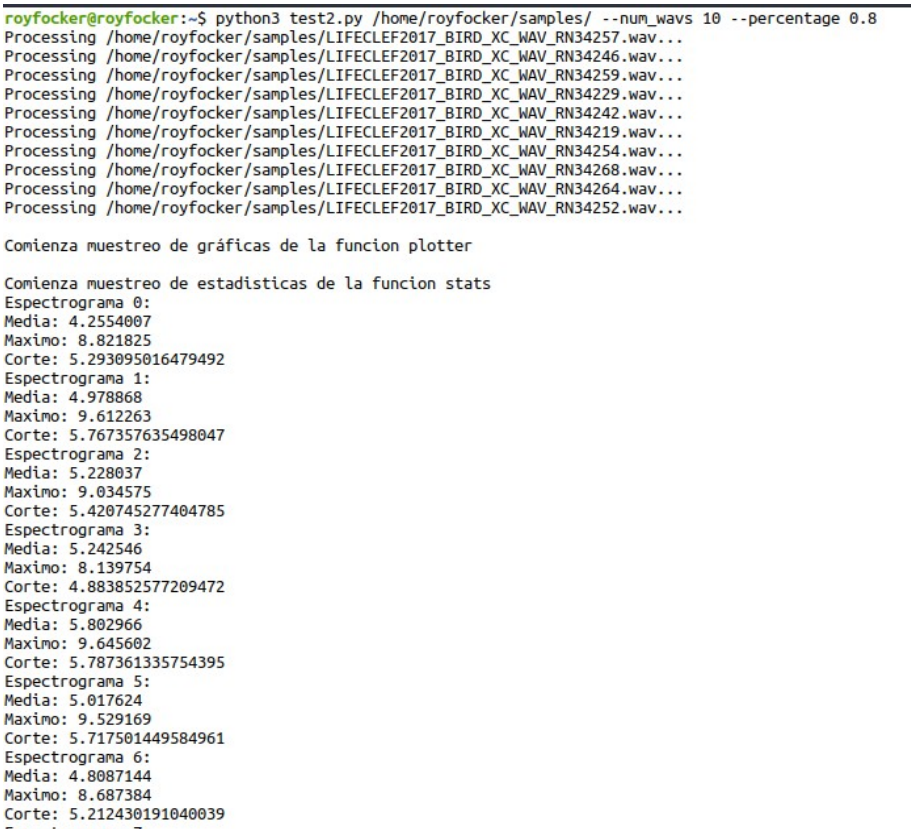

Figura 19. Parte 1 de la salida de *test2.py*.

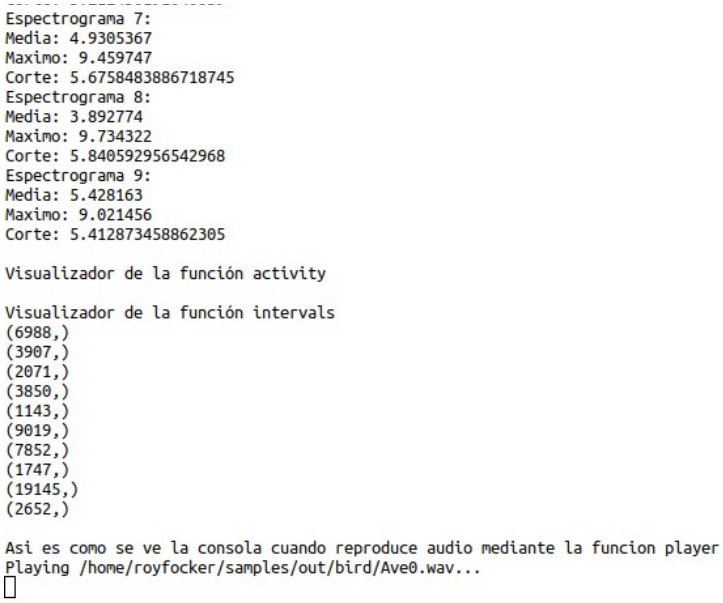

Figura 20. Parte 2 de la salida de *test2.py*.

En la Figura 19 se lee “Comienza muestreo de gráficas de la función plotter”, en esta parte la biblioteca *matplotlib* comienza a mostrar gráficas para los espectrogramas. En la Figura 21 se muestra uno de los espectrogramas gráficamente. Una particularidad del espectrograma de esta Figura es que no detectó frecuencias más allá de los 8000 Hz. por lo que muestra en blanco ese rango de frecuencias en el tiempo.

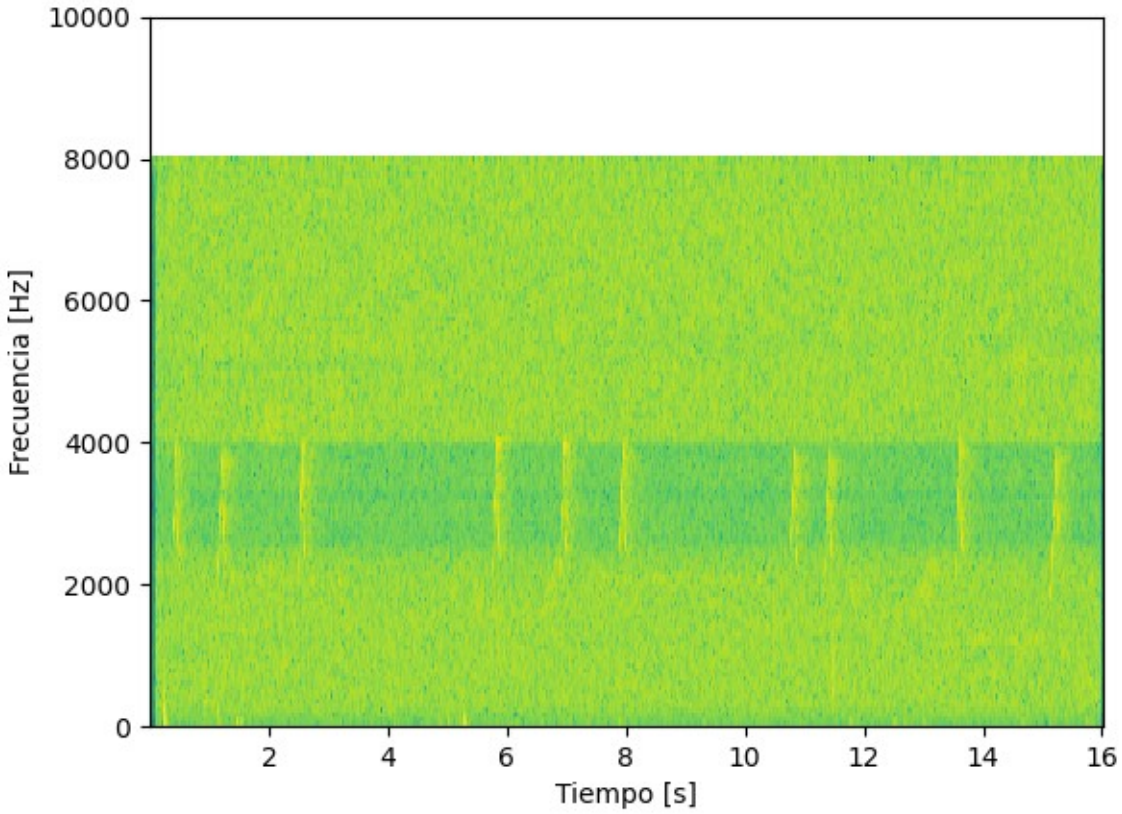

Figura 21. Gráfica de un espectrograma al correr *test2.py*. Gráfica generada mediante la biblioteca *matplotlib*.

Así mismo, en la Figura 20 se lee “Visualizador de la función activity”, en esta parte el programa invoca otra vez a la biblioteca *matplotlib* pero esta vez para mostrar de forma gráfica la suma de valores en el eje del tiempo, donde si hubo actividad el valor en ese instante del tiempo es 1 y si no hubo el valor en ese tiempo será 0. Todo esto se observa en la Figura 22.

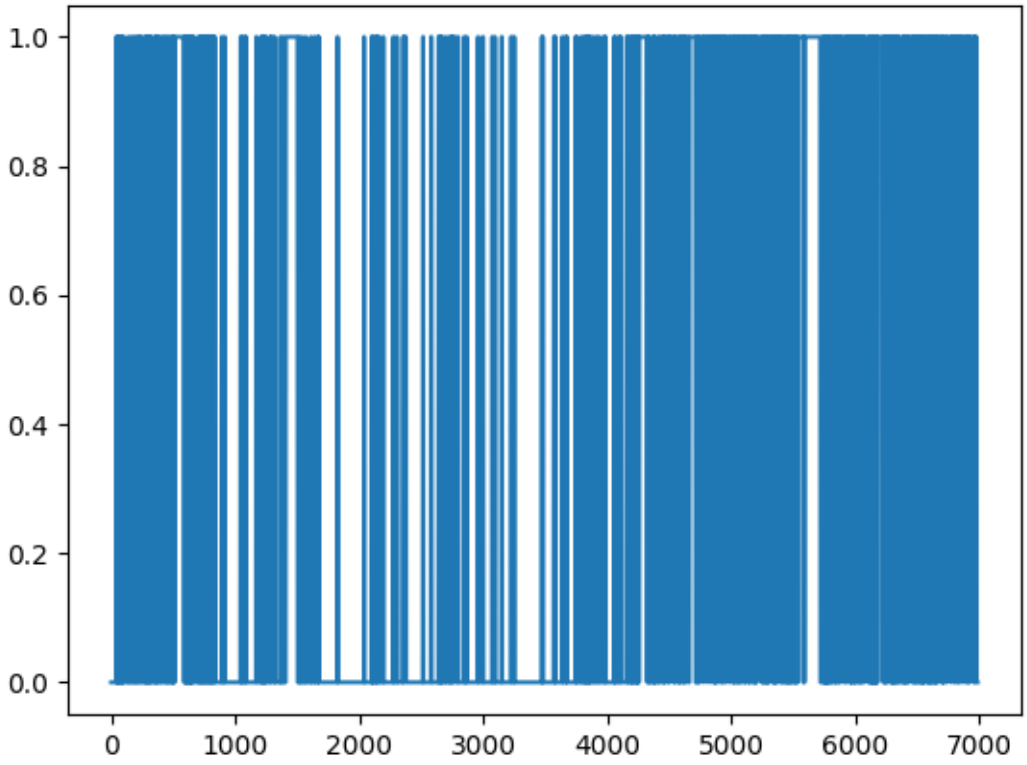

Figura 22. Visualizador de la funcion *activity*. Gráfica del eje del tiempo y su valor en cada instante (1 o 0) al correr *test2.py*. Gráfica generada mediante la biblioteca *matplotlib*.

El visualizador de la funcion *intervals*, también indicado en la Figura 20 muestra la forma de indice en consola y una gráfica por cada espectrograma mostrando el valor en el eje del tiempo que puede ser 1 o -1, donde 1 indica el inicio de un intervalo de actividad y -1 el fin de uno de estos intervalos. Esto se muestra en la Figura 23. Si el valor es 0 entonces es un segmento vacío y no es tomado en cuenta para el resultado final.

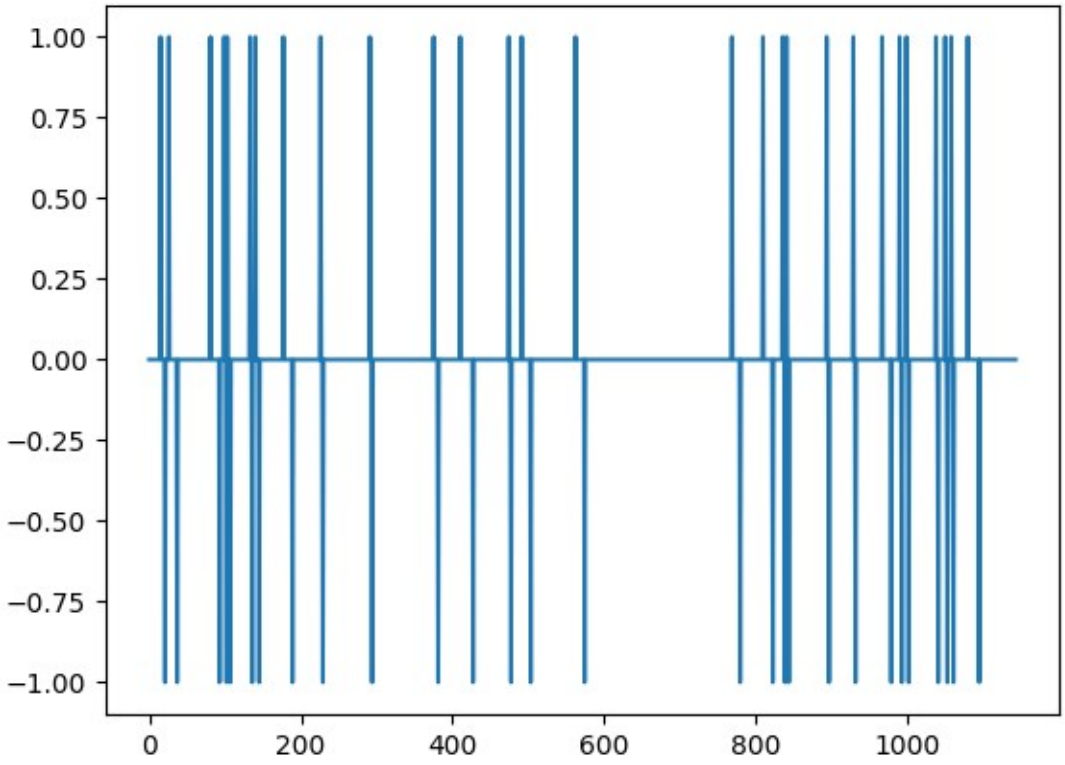

Figura 23. Visualizador de la funcion i*ntervals*. Gráfica del eje del tiempo y su valor en cada instante (1, -1 o 0) al correr *test2.py*. Gráfica generada mediante la biblioteca *matplotlib*.

Finalmente, la Figura 20 también muestra cómo se ve la consola al reproducir audio de uno de los archivos wav. En las Figuras 24 y 25 se muestran los archivos de audio generados al correr el programa.

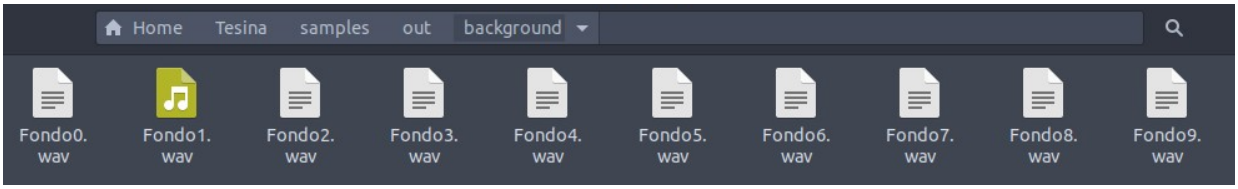

Figura 24. Archivos wav producidos para ruido de fondo al correr el programa.

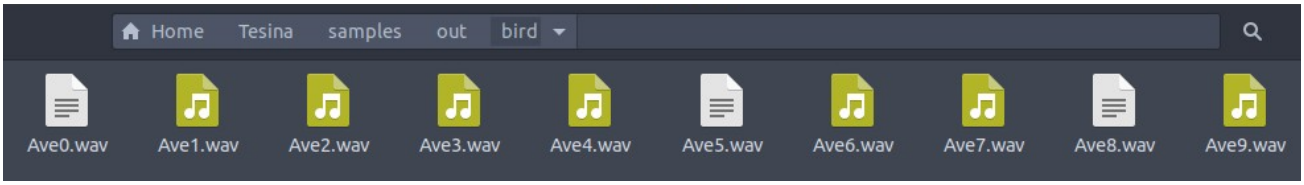

Figura 25. Archivos wav producidos para canto de ave al correr el programa.

Como consecuencia de un umbral más alto existieron archivos en los que no se clasificó canto de ave y se generaron archivos vacíos. También es notable que existen más archivos de fondo vacío, esto puede deberse a que en las operaciones de morfología se erosionaron más regiones o también a que los segmentos de ruido de fondo no fueron los suficientes para generar un archivo de audio.

Demostración 3

En esta prueba se procesan 20 archivos de audio y se usan las últimas funciones de flujo opcional a mostrar, descritas en la Figura 10: *blackandwhite* y *plotterbw*. Se muestra únicamente la salida de estas funciones. El código para llevar a cabo esto se guarda en un archivo llamdo *test3.py* y es el mostrado en la Figura 12 más el que se muestra en la Figura 26. Adicionalmente se utilizará un umbral más bajo con valor de 0.2 para clasificar actividad mayor al 20% del valor máximo del espectrograma.

```
specs_names= spec.reader(args.DIR,args.num_wavs)
samples_rates,samples=spec.sampler(specs_names)
frequencies,times,specs=spec.get_spec(samples,samples_rates)

specprueba=specs[:]
specsbw=specs[:]
specsbw=spec.blackandwhite(specsbw)
print("\n Salida de blackandwhite")
print(specsbw)
display.plotterbw(frequencies,times,specsbw)
specprueba=process.adjustment(specprueba)

specprueba=process.activity(specprueba,args.percentage,show=False)
indicesAve,indicesFondo=process.intervals(specprueba,show=False)

indicesAve=process.index2samples(indicesAve,samples,specs)
ave=process.bird(indicesAve,samples)

indicesFondo=process.index2samples(indicesFondo,samples,specs)
fondo=process.bird(indicesFondo,samples)

wavsAve=result.outwavAves(ave,'/home/royfocker/samples/out/bird/')
wavsFondo=result.outwavFondos(fondo,'/home/royfocker/samples/out/background/')
```

Figura 26. Código del archivo *test3.py*. Captura de pantalla de Sublime Text.

Los resultados de correr este código se muestran en la Figura 27 a continuación. En esta podemos observar que los valores de 1 y 0 son tomados como *True* o *False* por *numpy*.

```
royfocker@royfocker:~$ python3 test3.py /home/royfocker/samples/ --num_wavs 20 --percentage 0.2
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34257.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34246.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34259.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34229.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34242.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34219.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34254.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34268.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34264.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34252.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34210.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34237.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34263.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34241.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34227.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34223.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34234.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34226.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34209.wav...
Processing /home/royfocker/samples/LIFECLEF2017_BIRD_XC_WAV_RN34203.wav...

 Salida de blackandwhite
[array([[False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]]), array([[False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]]), array([[ True,  True,  True, ..., False, False, False],
       [ True,  True,  True, ..., False, False, False],
       [ True,  True,  True, ..., False, False, False],
```

Figura 27. Salida de *test3.py*.

En la Figura 28 se muestra una de las gráficas de la función *plotterbw*. Se sigue el método que con la función *plotter* pero el mapa de colores es de tipo escala de grises y como solo existen dos valores posibles solo se observa el blanco y el negro en la gráfica, donde el negro es la actividad en el espectrograma y el blanco la no actividad.

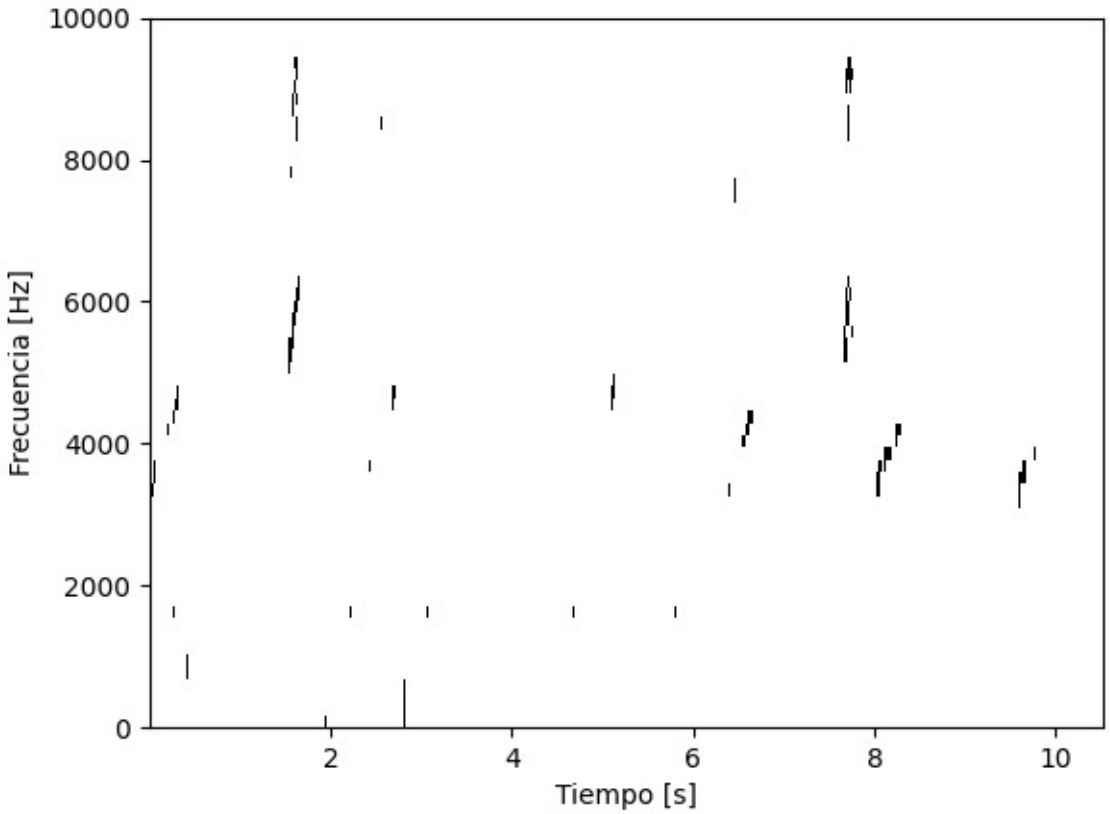

Figura 28. Gráfica de un espectrograma de la función *blackandwhite*, visualizada gracias a la función *plotterbw* al correr *test3.py*. Gráfica generada mediante la biblioteca *matplotlib*.

Finalmente, en las Figuras 29 y 30 se muestran los archivos de audio generados al correr el programa.

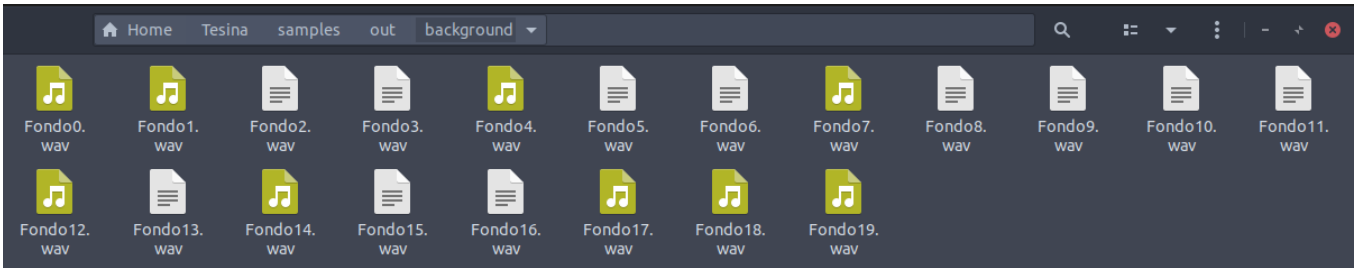

Figura 29. Archivos wav producidos para ruido de fondo al correr el programa.

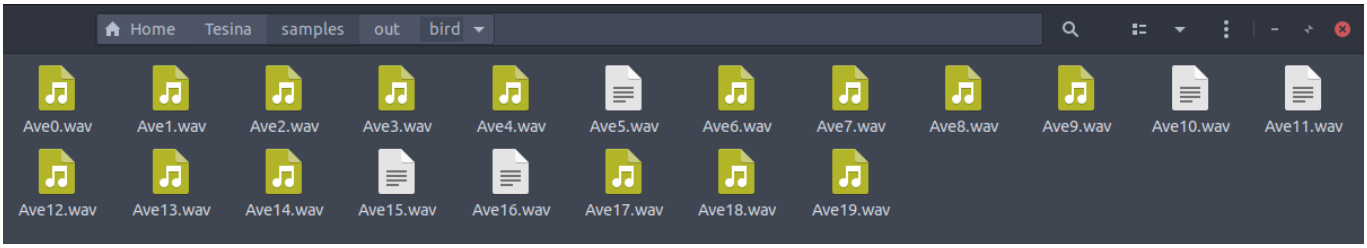

Figura 30. Archivos wav producidos para canto de ave al correr el programa.

Como resultado de un umbral menor se observa que hay pocos archivos de audio de canto de ave vacíos y varios archivos de ruido de fondo que sí están vacíos, esto se debe a que casi toda la actividad dentro del archivo de audio pasa a considerarse como canto de ave, dejando menor margen para construir archivos de ruido de fondo.

Conclusiones

Al llegar al capítulo final de este texto se asegura que los objetivos planteados en la introducción se han cumplido satisfactoriamente. El objetivo general, Desarrollar mediante técnicas de Inteligencia Artificial una CNN que sea capaz de identificar cantos de aves, está cumplido respecto a todo lo que concierne al trabajo de preprocesamiento de audio en el Departamento de Ciencias de la computación del IIMAS, esto porque el código desarrollado en este proyecto quedó a disposición del departamento.

Así mismo, el objetivo específico de desarrollar una biblioteca para Python, de código abierto, para el pre-procesamiento de audio que sirva como herramienta para proyectos en el área de la bioacústica ha quedado cumplido puesto que esta biblioteca se encuentra actualmente pública gracias a PyPi, donde cualquier usuario de Python puede instalarla, además el código está público y a disposición de cualquier persona para hacer con él lo que crea más conveniente.

Como la mayoría de los proyectos de software, se puede actualizar y mejorar la biblioteca preBird si se llegaran a encontrar *bugs* o fallas en el código. Existen varias ideas que pueden implementarse a esta biblioteca como alguna función que pueda exportar los sets de datos en formato *pickle* para moverlos de manera ligera y en cantidades grandes. También existe espacio para funciones que guarden las imágenes de los espectrogramas en formato png de manera automática. Se entiente que conforme el proyecto avance también este código puede avanzar e involucrarse aún más con otras áreas.

Existe espacio para la mejora de escritura de archivos de audio actualmente en la librería, esto podría lograrse utilizando otras bibliotecas de python especializadas en archivos de audio de las cuales no existe conocimiento actualmente o no han sido programadas. Si este proyecto continúa, podría ser implementado en computadoras portátiles.

Finalmente, es importante decir que las pruebas realizadas en este texto y fuera de él son consistentes, por lo que se afirma que esta biblioteca cumple su propósito y es totalmente funcional, entregando audios limpios con ruido de fondo y audios limpios con cantos de ave, que están listos para pasar a ser espectrogramas y más adelante podrían ser procesados por una CNN especializada en reconocer cantos de aves.

Referencias bibliográficas y electrónicas

Barrowclough GF, Cracraft J, Klicka J, Zink RM (2016) *How Many Kinds of Birds Are There and Why Does It Matter?* PLoS ONE 11(11): e0166307. https://doi.org/10.1371/journal.pone.0166307

Berlanga, H., H. Gómez de Silva, V. M. Vargas-Canales, V. Rodríguez-Contreras, L. A. Sánchez-González, R. Ortega-Álvarez y R. Calderón-Parra. (2015). *Aves de México: Lista actualizada de especies y nombres comunes.* CONABIO, México D.F. https://www.biodiversidad.gob.mx/media/1/ciencia-ciudadana/documentos/checklist_aves_mexico_2015x.pdf

BirdLife International. (2017). *We have lost over 150 bird species since 1500.* Descargado de http://www.birdlife.org on 21/09/2020

BirdLife International. (2018). *State of the world's birds: taking the pulse of the planet.* Cambridge, UK: BirdLife International. https://www.birdlife.org/sites/default/files/attachments/BL_ReportENG_V11_spreads.pdf

Carbonnelle, P. (2020). *PYPL PopularitY of Programming Language.* http://pypl.github.io/PYPL.html

Gordillo Martínez, A., Ortíz Ramírez, M. F., Navarro Sigüenza, A. G. (2013). *Estructura y evolución de las vocalizaciones de las aves.* Universidad Nacional Autónoma de México: *Revista ciencias, volumen 109-110*, 32-40.

IUCN. (2013). *The IUCN Red List of Threatened Species.* Version 2013.2. International Union for Conservation of Nature. Disponible en línea: http://www.iucnredlist.org

Minsky, M., & Papert, S. (1969). Perceptrons. Cambridge, MA: MIT Press.

Ralph, C. J., Geupel, G. R., Pyle, P., Martin, T. E., DeSante, D. F., Milá, B. (1996). *Manual de métodos de campo para el monitoreo de aves terrestres.* Gen. Tech. Rep. PSW-GTR159. Albany,CA: Pacific Southwest Research Station, Forest Service, U.S. Department of Agriculture, 46 p.

Richards, D. G. (1981). *Enviromental acoustics and censues of singing birds*. Studies in Avian Biology No. 6:297-300. https://pdfs.semanticscholar.org/33a0/d37e127a17b1ba5a4da60deb6798cb3f3188.pdf

Schmidhuber, J. (2015). *Deep learning in neural networks: An overview.* Neural Networks, 61, 85–117. doi:10.1016/j.neunet.2014.09.003

SEO BirdLife. (2016). *El mundo podría tener unas 18.000 especies de aves.* https://www.seo.org/2016/12/28/mundo-podria-unas-18-000-especies-aves/

Suarez, A., Jimenez, A., Castro Franco, M., Roa, A. (2016). *Clasificación automática de coberturas del suelo en imágenes satelitales utilizando redes neuronales convolucionales: Un caso aplicado en Parques Nacionales Naturales de Colombia.* Recuperado de: https://www.researchgate.net/publication/317226045_Clasificacion_automatica_de_coberturas_del_suelo_en_imagenes_satelitales_utilizando_redes_neuronales_convolucionales_Un_caso_aplicado_en_Parques_Nacionales_Naturales_de_Colombia

Tubaro, P. L. (1999). *Bioacústica aplicada a la sistemática, conservación y manejo de poblaciones naturales de aves.* Laboratorio de Biología del Comportamiento, Instituto de Biología y Medicina Experimental. https://pdfs.semanticscholar.org/7626/6ebcbcafdf6b4298acc46ef8dadc87cacc30.pdf

Tung, L. (2019). *Programming languages: Python developers now outnumber Java ones.* CBS Interactive. https://www.zdnet.com/article/programming-languages-python-developers-now-outnumber-java-ones/

Valueva, M. V., Nagornov, N. N., Lyakhov, P. A., Valuev, G. V., & Chervyakov, N. I. (2020). *Application of the residue number system to reduce hardware costs of the convolutional neural network*

implementation. Mathematics and Computers in Simulation, 177, 232–243. doi:10.1016/j.matcom.2020.04.031

Yani, M., Irawan, B., Setiningsih, C. (2019). *Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail*. Journal of Physics: Conference Series. 1201. 012052. doi: 10.1088/1742-6596/1201/1/012052.