Universidad Nacional de Ingeniería

Área de Conocimiento de Tecnología de la Información y Comunicación

Método de Procesamiento de Sonidos del Corazón para la Detección de Anomalías Cardiacas Basado en Machine Learning

Trabajo Monográfico para optar al título de Ingeniero Electrónico

Elaborado por:		Tutor:	Asesor:
Br. Pablo Josué Vásquez Guevara Carnet: 2019-0036U	Br. Rommel Antonio López Mercado Carnet: 2019-0120U	Msc. Fernando Antonio Flores Guido	TeknL. Marco Antonio Munguía Mena

02 de junio de 2025
Managua, Nicaragua

A Dios

Por darme la inspiración, la inteligencia y las fuerzas necesarias para llevar a cabo este proyecto. Su guía me ha sostenido en cada etapa, permitiéndome concluir con éxito este trabajo.

A mi padre

por tu dedicación, esfuerzo y apoyo constante. Tus conocimientos y tu experiencia fueron fundamentales para el desarrollo de este proyecto.

A mi madre

por tu amor incondicional, tus sacrificios y por ser siempre mi mayor motivación. Tus palabras y tus oraciones me han impulsado a seguir adelante, incluso en los momentos más difíciles.

A nuestro asesor

TeknL. Marco Antonio Munguía Mena, por su orientación, compromiso y disposición durante todo este proceso. Su guía fue clave para lograr los objetivos planteados.

A todos los que, de una u otra forma, aportaron a la culminación de este trabajo, mi más sincero agradecimiento.

Pablo Josué Vásquez Guevara

AGRADECIMIENTO

A Dios

Agradezco primeramente a Dios, por ser mi guía constante, por darme la fortaleza, la sabiduría y la perseverancia para superar cada obstáculo a lo largo de mi formación académica. Su presencia me ha acompañado en cada etapa, dándome la paciencia y el ánimo necesarios para alcanzar esta meta.

De manera muy especial, expreso mi profundo agradecimiento a mi madre, cuyo amor incondicional, apoyo y sacrificio han sido fundamentales en mi vida. A mis abuelos y tíos, gracias por sus palabras de aliento, por creer en mí y por estar presentes en cada momento importante de este proceso. Sin el respaldo de mi familia, este logro no habría sido posible.

A nuestro asesor

Expresamos nuestro más sincero agradecimiento al TeknL. Marco Antonio Munguía Mena, quien, como asesor de nuestra monografía, desempeño un papel fundamental en el desarrollo de esta monografía titulada “Método de procesamiento de sonidos del corazón para la detección de Anomalía Cardiaca Basado en Machine Learning”.

Su dedicación, paciencia y valiosa enseñanza nos guiaron a lo largo de este proceso, brindándonos el conocimiento y la orientación necesario para afrontar cada desafío con determinación. Su apoyo incondicional, sus consejos y compromiso con nuestra formación fueron clave para la culminación de este trabajo.

Gracias, estimado asesor, por su tiempo, por compartir su experiencia y por motivarnos a alcanzar nuestras metas con excelencia y pasión por la ingeniería.

Rommel Antonio Lopez Mercado

RESUMEN

Las enfermedades cardiovasculares representan una de las principales causas de mortalidad a nivel mundial, lo que resalta la importancia de desarrollar herramientas tecnológicas para su detección temprana. Este trabajo aborda el desarrollo de un método para el procesamiento y análisis de sonidos cardíacos con el objetivo de detectar anomalías mediante técnicas de *machine learning*. La monografía destaca el papel de la inteligencia artificial en el ámbito de la salud, facilitando el análisis rápido y preciso de señales acústicas del corazón para identificar patrones anómalos que podrían pasar desapercibidos en una evaluación auditiva convencional.

Se propone un sistema automatizado orientado al procesamiento de sonidos cardíacos para la detección de anomalías mediante técnicas de inteligencia artificial. El proyecto combina bases de datos especializadas en auscultación, como las proporcionadas por PhysioNet, con herramientas de programación como Scikit-learn en Python. El enfoque principal se centra en la implementación de un método que incluye el preprocesamiento de señales, la extracción de características relevantes utilizando coeficientes cepstrales en la frecuencia Mel (MFCC) y la posterior clasificación de los sonidos mediante un modelo de Máquinas de Soporte Vectorial (SVM). Como parte complementaria, se desarrolló una aplicación móvil en Android Studio que permite la adquisición de sonidos cardíacos a través del micrófono del dispositivo, los cuales son utilizados como entrada para el sistema de análisis.

Los objetivos específicos incluyen la implementación de un algoritmo eficiente para la detección de patrones acústicos diferenciadores, la validación del sistema con bases de datos previamente etiquetadas y la evaluación de su desempeño mediante métricas de clasificación. Los resultados obtenidos evidencian la viabilidad del método propuesto y su potencial como una herramienta de apoyo para la detección temprana de afecciones como soplos y arritmias.

Este enfoque contribuye a una atención médica más eficiente y accesible, permitiendo un monitoreo preventivo sin necesidad de equipos médicos especializados. Finalmente, el trabajo subraya la importancia de la integración de tecnologías avanzadas en el diagnóstico y tratamiento de enfermedades cardiovasculares,

promoviendo innovaciones en el campo de la telemedicina y el análisis de bioseñales. Así mismo, se plantean futuras mejoras, como la optimización del algoritmo de clasificación y la integración con plataformas de salud digital, para ampliar su alcance y precisión en entornos clínicos y no clínicos.

ABSTRACT

Cardiovascular diseases represent one of the leading causes of mortality worldwide, highlighting the importance of developing technological tools for early detection. This work addresses the development of a method for the processing and analysis of heart sounds with the aim of detecting anomalies through machine learning techniques. The monograph emphasizes the role of artificial intelligence in the healthcare field, facilitating the rapid and accurate analysis of heart sound signals to identify abnormal patterns that might go unnoticed during conventional auditory evaluations.

An automated system is proposed, focused on the processing of heart sounds for anomaly detection using artificial intelligence techniques. The project combines specialized auscultation databases, such as those provided by PhysioNet, with programming tools like Scikit-learn in Python. The main approach focuses on implementing a method that includes signal preprocessing, the extraction of relevant features using Mel Frequency Cepstral Coefficients (MFCC), and the subsequent classification of sounds using a Support Vector Machine (SVM) model. As a complementary component, a mobile application was developed in Android Studio to acquire heart sounds through the device's microphone, which are then used as input for the analysis system.

The specific objectives include the implementation of an efficient algorithm for detecting distinctive acoustic patterns, validating the system with previously labeled databases, and evaluating its performance using classification metrics. The results obtained demonstrate the feasibility of the proposed method and its potential as a support tool for the early detection of conditions such as murmurs and arrhythmias.

This approach contributes to more efficient and accessible medical care, allowing preventive monitoring without the need for specialized medical equipment. Finally, the work highlights the importance of integrating advanced technologies in the diagnosis and treatment of cardiovascular diseases, promoting innovations in the field of telemedicine and biosignal analysis. Future improvements are also proposed, such as optimizing the classification algorithm and integrating the system with digital health

platforms to expand its scope and accuracy in both clinical and non-clinical environments.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE ECUACIONES

1. INTRODUCCIÓN

La detección temprana de anomalías en los sonidos cardiacos es un aspecto crucial en la atención médica, ya que puede marcar la diferencia en la vida y el bienestar de los pacientes. tradicionalmente, la auscultación cardiaca, que implica escuchar los sonidos del corazón con un estetoscopio, ha sido una herramienta fundamental para los médicos en la evaluación de la salud cardiovascular de los pacientes. Sin embargo, esta técnica depende en gran medida de la experiencia y habilidad del profesional lo que a veces puede llevar a diagnóstico erróneo o a la detección tardía de problemas cardiacos.

En los últimos años, la inteligencia artificial (IA) ha surgido como una poderosa aliada en la detección de anomalías cardiacas a través de la auscultación. Los algoritmos de IA pueden analizar de manera rápida y precisa los sonidos cardiacos, identificando patrones sutiles y anormales que pueden pasar desapercibidos para el oído humano. Estos algoritmos son capaces de procesar grandes cantidades de datos de auscultación de manera consistente y objetiva, lo que hace posible una detección más temprana y precisa de problemas cardiacos, como soplos, arritmias, valvulopatías y otros trastornos.

Es por ello que se ha optado por el desarrollo de un sistema automatizado de detección, que permita discernir si existe alguna anomalía en el corazón de los pacientes, empleando técnicas de inteligencia artificial además de otras herramientas como el sitio PhysioNet y la librería Scikit-learn del lenguaje de programación Python. Para poder elaborar dicho sistema, se empezó entrenando un algoritmo utilizando las bases de datos de sonidos de latidos del corazón que fueron proporcionados por la página antes mencionadas, las cuales nos dan acceso a sonidos de distintos estados de salud del corazón.

La inteligencia artificial progresivamente está llegando a ser parte fundamental en el ámbito de la medicina, debido a tal motivo, el desarrollo de proyectos de esta índole permitirá reducir la frecuencia de errores médicos además de mejorar la precisión diagnostica a través de la integración, el análisis y la interpretación de información por algoritmos y software. La automatización de actividades repetitivas

liberará tiempo al personal de salud, además, permitirá disminuir la carga administrativa y el agotamiento profesional.

2. ANTECEDENTES

La investigación de este estudio es realizar procesos no invasivos para monitorear signos vitales, con el propósito de implementar procesos eficientes y reducir el tiempo que emplea el personal de salud al adquirir los resultados de los signos vitales del paciente.

La ingeniería electrónica siempre ha colaborado con una gran facilidad a la medicina para poder mantener y mejorar el nivel de la salud de la sociedad nicaragüenses. Generalmente este apoyo se da en forma de equipos robustos electrónicos robustos, de tal manera puedan asistir la facilidad a los médicos o pacientes en momentos de necesidad.

La clasificación de electrocardiograma (ECG) ha experimentado un notable impulso gracias a la creciente popularidad de los modelos de machine learning. Los investigadores han adoptado diversas técnicas dentro de este campo, sin embargo, la falta de estandarización en los modelos de aprendizaje automático ha surgido como un desafío significativo, la carencia de un marco normativo solido ha llevado a la conclusión de que existe un potencial considerable para mejorar los resultados en comparación con enfoques previos (Rojas, Clasificacion de Señales ECG para la Deteccion de Enfermedades Cardiacas: Un estudio comparativo).

La sangre es un elemento vital para la subsistencia de las células, según Arthur Guyton (2016, p.3). Es necesaria su circulación en el cuerpo a través de los vasos sanguíneos. El corazón sirve como sistema de bombeo de la sangre para su flujo a través de los vasos sanguíneos, y es debido a este flujo y el cerrado y apertura de válvulas en el corazón por lo que se generan los sonidos cardiacos (SC).

Según Ortigosa et al. (2018) en su estudio titulado" Variabilidad De La Frecuencia Cardíaca: Investigación Y Aplicaciones Prácticas Para El Control De Los Procesos Adaptativos En El Deporte" (p.122) encontraron que, la frecuencia cardiaca es un objetivo importante de la investigación para la medicina tradicional y especialmente para los fisiólogos deportivo. El registro cuantitativo de la frecuencia cardiaca comienza hasta mediado del siglo XVII, cuando John Floyer, un médico ingles crea uno de los primeros dispositivos que permite medir la frecuencia cardiaca y la

variabilidad de la respiración (Billman, 2011). Esto no es hasta mediado del sigo XX, cuando apareció el galvanómetro cordón desarrollado por el fisiólogo alemán Wilhelm Einthoven, cuando la ciencia empieza a obtener registros continuos de la actividad bioeléctrica del corazón. Estos registros denominados electrocardiogramas (ECG), se componen de tres secciones: onda P, complejo QRS y onda T, que representa la despolarización y repolarización de las aurículas y los ventrículos. Estos avances permitieron a la definición de las características electrocardiográfica de numerosas enfermedades y procesos cardiovasculares.

En un estudio titulado “Diseño de un canal de instrumentación para un sistema de electrocardiograma y un pulsioxímetro.” (Fernández, 2014, pág. 9). En este trabajo se manejaron teorías de electrocardiografía forma característica de la onda cardiaca, derivaciones bipolares y unipolares. En este trabajo se relaciona con la investigación en curso ya que propone la construcción de un simulador de pulsos del corazón las diferentes formas de ondas del corazón entre otros.

Según Paz y Herrera (2017) en su estudio titulado “Diseño y construcción de un sistema que simule las señales eléctricas del corazón.” Realizado en Managua, Nicaragua; desarrollan e implementan un simulador de ECG versátil y de bajo costo que reproduzca un ECG bajo condiciones de registro mono canal capaz de alimentar la entrada de un electrocardiógrafo.

Según Rojas, Romero, Romero (2013). En su investigación titulada” MODELO DE PROCESAMIENTO DIGITAL DE SEÑALES CARDIACAS DESARROLLADO EN MATLAB” realizada en la Universidad Dr. RAFAEL BELLOSO CHACIN de Colombia (p.21). Tuvo como principal objetivo el diseño de un modelo de procesamiento digital de señales cardiacas , aprovechando las potencialidades del software de Matlab, la investigación fue de carácter documental y pretendía lograr una representación confiable de las señales eléctricas del corazón con el mínimo de ruido posible , para su correcto análisis, además , el diseño no fue experimental puesto que no manipula la información cardiaca de los pacientes, sino solamente, las características de la variable de estudio Como resultado, se obtuvo el modelo de procesamiento digital de señales cardíacas por medio de la herramienta Matlab, corroborando su aplicabilidad.

Según Nieto y Vega (2017) en su estudio titulado “Diseño de un prototipo de medición de señales fisiológica utilizadas en Biofeedback” llegaron a una conclusión que, particularmente, el módulo de frecuencia cardiaca presenta una alta sensibilidad a los movimientos del paciente, siendo este uno de los principales aspectos a mejorar a futuro. A su vez, es deseable el diseño de una estructura reproducible para la etapa de medición.

La organización mundial de la salud (OMS) nos menciona que las enfermedades cardiovasculares (ECV) son la principal causa de muerte en todo el mundo. Cada año mueren personas a causas de enfermedades cardiovasculares que por cualquier otra causa. Las muertes de ECV afectan por igual a ambos sexos, y más del 80% se produce en países de ingresos bajos y medios. 17.3 millones de personas murieron por ECV en el 2008, según datos de la OMS.

Según informes de la (Organizacion Mundial de la Salud, 2021). una de cada tres personas sufre de presión arterial ,causante del cincuenta por ciento de las muertes provocadas por infartos y enfermedades relacionadas al corazón, este tipo de patologías puede dar paso a otro tipo de trastornos de la salud cardiovasculares , cerebrovasculares y renales, Los factores que favorecen el desarrollo de dicha enfermedad son malos hábitos alimenticios (sobrepeso), controles médicos periódicamente incumplidos ,llevar una vida sedentaria ,y sobre todo el consumo de tabaco y alcohol.

3. JUSTIFICACIÓN

Según Facila, et al. (2024). Las enfermedades cardiovasculares constituyen una de las principales causas de morbilidad y mortalidad en todo el mundo, su detección temprana desempeña un papel crucial en la prevención y el tratamiento eficaz. A pesar de los avances en la tecnología médica, los métodos de detección actuales, como el electrocardiograma (ECG) y el monitoreo clínico periódico, a menudo presentan limitaciones en términos de sensibilidad y precisión debido al factor humano. Esto resulta en la pérdida de oportunidades para la intervención temprana y la atención médica, lo que puede tener graves consecuencias para la salud de los pacientes.

Los avances en el campo de la inteligencia artificial, particularmente en el aprendizaje profundo, han demostrado un gran potencial para mejorar la detección de anomalías cardiacas. Los algoritmos de inteligencia artificial tienen la capacidad de procesar grandes volúmenes de datos de manera eficiente e identificar patrones sutiles en señales biométricas, como la obtenidas a partir de ECG y otros dispositivos de monitoreo. Esta capacidad combinada con la disponibilidad de registro de ECG en línea y la proliferación de dispositivos portátiles, brinda una oportunidad única para desarrollar un sistema de detección de anomalías cardiaca más preciso y oportuno

La presente investigación se justifica en la medida en que aborda preguntas de investigación críticas, como la selección y optimización de algoritmos de inteligencia artificial, la recopilación y procesamientos de señales cardiacas y validación propias del método. Además, esta investigación no solo tiene el potencial de tener un impacto inmediato en la calidad de la atención médica, sino que también sienta la bases para futuros desarrollos en la detención de enfermedades cardiovasculares y puede extenderse a la detección de otras condiciones médicas. El impacto a largo plazo de este método podría reducir costos en atención médica y salvar vidas al identificar problemas cardiacos en sus etapas más tempranas y tratables.

4. OBJETIVOS

4.1. Objetivo general

- Implementar un método de procesamiento de sonidos del corazón para la detección de anomalías cardiacas utilizando machine learning.

4.2. Objetivos específicos

- Desarrollar una fase de preprocesamiento que incluya el remuestreo y segmentación de los sonidos cardíacos, preparando los datos para su análisis posterior.
- Extraer características con capacidad discriminativa de los sonidos cardíacos que incrementen la precisión en su clasificación.
- Aplicar técnicas de clasificación sobre los sonidos cardíacos, utilizando las características previamente extraídas.
- Desarrollar una aplicación móvil capaz de adquirir sonidos cardíacos a través del micrófono del dispositivo móvil.
- Evaluar la efectividad del método propuesto mediante pruebas con sonidos cardíacos recolectados tanto desde la aplicación desarrollada como desde repositorios médicos especializados.

5. MARCO TEÓRICO

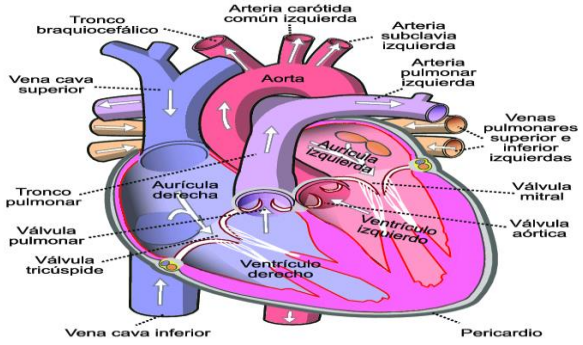
5.1. Anatomía y Fisiología del Corazón

El corazón humano está formado por tejido muscular y es una parte esencial del proceso circulatorio el peso de este órgano varía según la edad, el tamaño y el peso de la persona, pero podemos decir que nuestro corazón pesa alrededor de un 0.40-0.45% de nuestro peso corporal además el corazón está dividido en tres capas: pericardio (por fuera), miocardio (musculo) y endocardio (por dentro), (Ballesteros P. A., 2009).

El corazón es un órgano muscular hueco que se encuentra en el tórax, entre los pulmones. Este órgano muscular hueco es muy esencial para mantenernos vivos, ya que es el encargado de bombear la sangre oxigenada a todos nuestros tejidos mediante los vasos sanguíneos del sistema circulatorio. El corazón también tiene cuatros válvulas que permiten que la sangre fluya en una sola dirección las cuales son: la válvula mitral, la válvula tricúspide, la válvula aortica y la válvula pulmonar (Ballesteros P. A., 2009).

Figura 1:

Diagrama del Corazón Humano.



Nota. La figura muestra cómo se compone la anatomía del corazón humana. Fuente: Wikipedia Contributors (2010).

La arteria aorta es el vaso sanguíneo más grande que sale del corazón y distribuye sangre rica en oxígeno a todo el cuerpo. Las venas cava superior e inferior son las que llevan la sangre en oxigeno de vuelta al corazón.

5.2. Fisiología de corazón

El corazón se compone de dos aurículas y dos ventrículos. La sangre llega al corazón por las aurículas y sale impulsada por los ventrículos. El corazón y los vasos sanguíneos (venas y arterias) tienen la misión común de llevar la sangre a todas las células del organismo para que obtengan el oxígeno, los nutrientes y otras sustancias necesarias. Constituyen un sistema perfecto de riego con sangre rica en oxígeno y recolección de la que es pobre en oxígeno y está cargada de detritus. Mientras que los vasos sanguíneos actúan como las tuberías conductoras de la sangre, el corazón es la bomba que da el impulso para que esa sangre recorra su camino. Con cada latido el corazón impulsa una cantidad (habitualmente, 60-90 ml) de esa sangre hacia los vasos sanguíneos (García, 2009, pag.41).

Son fundamentalmente los ventrículos los que se encargan del trabajo de impulsar la sangre. Las aurículas, en cambio, contribuyen al relleno óptimo de los ventrículos en cada latido. El movimiento de aurículas y ventrículos se hace de forma ordenada y coordinada, en un ciclo que se repite (ciclo cardíaco) con cada latido, en el cual lo más importante, en primer lugar, es el llenado de los ventrículos; posteriormente, tiene lugar su vaciamiento mediante la eyección de esa sangre al torrente circulatorio. El ciclo cardíaco presenta dos fases: diástole y sístole. La diástole es el período del ciclo en el cual los ventrículos están relajados y se están llenando de la sangre que luego tendrán que impulsar. Para que puedan llenarse, las válvulas de entrada a los ventrículos (mitral y tricúspide) tienen que estar abiertas. Y para que la sangre no se escape aún, las válvulas de salida de los ventrículos (aórtica y pulmonar) deben estar cerradas, (García, 2009, pag.42).

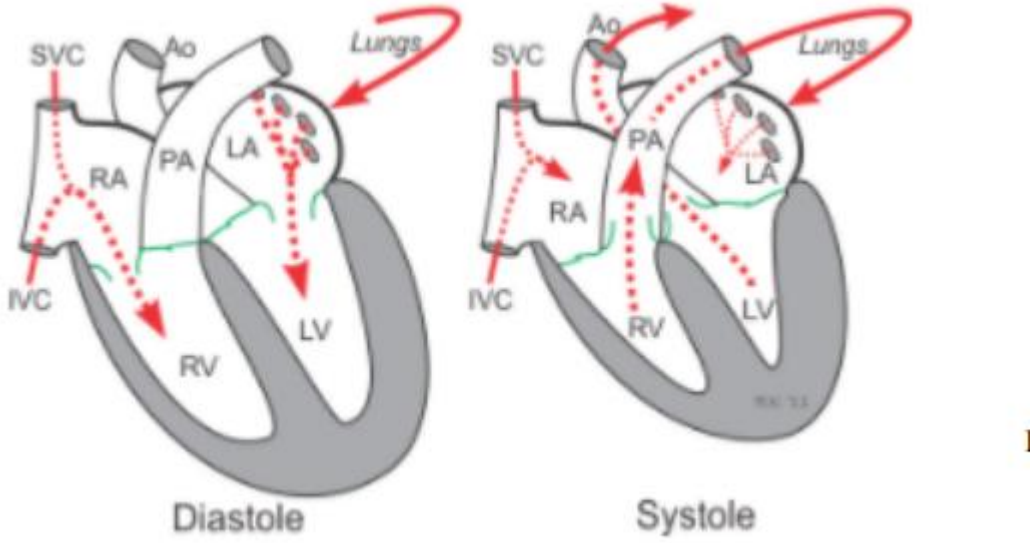
El ciclo cardiaco comprende los procesos fisiológicos que involucran un latido del corazón. Este compuesto de dos partes principales: sístole y diástole. La diástole representa el momento en que los ventrículos se encuentran relajados. Como se observa en la figura (), la sangre fluye lentamente desde las cámaras superiores del corazón (aurículas izquierda y derecha), hacia las cámaras inferiores (ventrículo izquierdo y derecho). La sangre de la aurícula derecha llega a través de las venas cava

inferior y superior. Por su parte, la aurícula izquierda recibe sangre oxigenada de los pulmones (Rojas, 2020).

Por otro lado, la sístole representa el proceso en el que tiene lugar la contracción de los ventrículos. Las válvulas aortica y pulmonar se abren con el fin de permitir el paso de la sangre que estos han eyectado hacia las aurículas respectivas.

Figura 2

Parte del Ciclo Cardiaco



Nota: La figura muestra el funcionamiento del Ciclo Cardiaco. Fuente: Mosquera (2020).

Un latido cardiaco se completa en dos fases:

- **Sístole:** la sístole se produce en un ciclo cardiaco que se inicia la contracción de las aurículas, seguida por la contracción de los ventrículos y finaliza con la relajación completa del corazón. La sístole ventricular es particularmente importante, ya que es responsable de enviar la sangre oxigenada y rica en nutrientes a todo el cuerpo.

La duración de la sístole puede variar dependiendo de factores como la frecuencia cardiaca, la edad y la salud general del individuó, en un corazón sano, la sístole generalmente dura alrededor de 0.3 segundo, mientras que, en una persona con problemas cardiaco, la duración de la sístole puede ser más larga o corta.

- **Diástole:** es una fase muy importante del ciclo cardiaco, ya que permite que el corazón se llene de sangre y se prepare para la siguiente contracción. La duración de la diástole también puede variar en función de la edad, la salud y otros factores, pero generalmente es más larga que la sístole.

  El tiempo de llenado diastólico final (TLDF) es una unidad que se utiliza para evaluar la función diastólica del corazón. Esta medida se refiere al tiempo que tarda la sangre en llenar los ventrículos durante la diástole, y se utiliza para evaluar si el corazón está funcionando adecuadamente para llenarse de sangre antes de la siguiente contracción.

5.3. Enfermedades cardiovasculares

Las enfermedades cardiovasculares son un grupo de trastornos que afectan el corazón y los vasos sanguíneos (OMS, 2021). Estas enfermedades pueden incluir la enfermedad coronaria, la insuficiencia cardiaca, las arritmias, las valvulopatías, las miocardiopatías, las enfermedades inflamatorias del corazón y las enfermedades congénitas del corazón.

Las enfermedades cardiovasculares pueden ser causadas por una variedad de factores, incluyendo la hipertensión arterial, el colesterol alto, el tabaquismo, diabetes y la obesidad. Este tipo de enfermedades son la causa principal de mortalidad en todos los grupos étnicos y raciales, según la Federación Mundial del Corazón (Mark Miller, 2024, pág. 3), las enfermedades cardiovasculares causan 17.5 millones de muertes al año, tantas como el resultado de sumar la provocadas por SIDA, tuberculosis, malaria, diabetes, cáncer y patología respiratoria crónica.

5.4. Tipos de enfermedades cardiovasculares

- **La cardiopatía coronaria:** la enfermedad coronaria (CHD) es el tipo más común de patología cardiaca y ocurre cuando la placa se acumula en la arteria conectada al corazón. También se llama arteria coronaria (CAD).
- **La insuficiencia cardiaca:** ocurre cuando el miocardio se vuelve rígido o débil. Esta enfermedad solo puede afectar el lado derecho o izquierdo del corazón. La presión arterial y CAD altos son causas comunes de insuficiencia cardiaca.
- **Las arritmias:** estos son problemas con la frecuencia cardiaca (pulsos), esto sucede cuando el sistema eléctrico del corazón no funciona correctamente. El corazón puede palpitar demasiado rápido, demasiado lento o en forma irregular.
- **Las enfermedades de las válvulas cardiaca:** ocurre cuando una de las cuatro válvulas en el corazón no funciona correctamente. Algunos problemas cardiacos, como ataques cardiacos, enfermedad cardiaco o infección, pueden causar enfermedades de la válvula cardiaca, algunas personas nacen con problemas de válvula cardiaca
- **Presión arterial alta (hipertensión):** es una de las enfermedades cardiovasculares que puede causar otros problemas, como ataques cardiacos, insuficiencia cardiaca y accidente cerebrovascular. El accidente cerebrovascular es causado por la falta de flujo sanguíneo al cerebro. Esto tienen muchos factores de riesgo que son los mismo que la enfermedad cardiaca.
- **La enfermedad cardiaca congénita:** es un problema con la estructura y la función del corazón que existe al nacer. Este término puede describir muchos problemas diferentes que afectan al corazón.

Factores de Riesgo:

- **Edad:** El envejecimiento aumenta el riesgo de que las arterias se dañen y se estrechen, y de que el músculo cardíaco se debilite o engrose.

- **Sexo:** En general, los hombres corren mayor riesgo de padecer una enfermedad cardíaca. El riesgo en las mujeres aumenta después de la menopausia.

- **Fumar:** Las sustancias presentes en el humo de tabaco dañan las arterias. Los ataques cardíacos son más comunes en fumadores que en no fumadores.

- **Dieta poco saludable**: Las dietas con alto contenido de grasas, sal, azúcar y colesterol están asociadas con una enfermedad cardíaca.

- **Diabetes:** La diabetes aumenta el riesgo de presentar una enfermedad cardíaca. La obesidad y la presión arterial alta aumentan el riesgo de tener diabetes y enfermedad cardíaca.

- **Obesidad:** El exceso de peso normalmente empeora otros factores de riesgo de la enfermedad cardíaca.

- **Mala higiene dental:** Es importante cepillarse los dientes y las encías, y usar hilo dental con frecuencia. Además, también es necesario hacerse revisiones dentales con regularidad. Los problemas en los dientes y las encías facilitan que los gérmenes entren en el torrente sanguíneo y lleguen al corazón. Esto puede causar endocarditis.

5.5. Auscultación cardiaca

La auscultación del corazón es una técnica crítica que requiere una audición excepcional y la capacidad de discernir diferencias sutiles en el tono y la duración de los sonidos cardíacos. Los médicos con dificultades auditivas pueden mejorar su capacidad de escuchar utilizando estetoscopios con amplificación, lo que les permite percibir con mayor claridad. Es importante recordar que los sonidos agudos se detectan mejor con el diafragma del estetoscopio, mientras que los sonidos más graves se aprecian con mayor nitidez utilizando la campana. Es fundamental evitar aplicar una presión excesiva al usar la campana, ya que esto podría convertir la piel debajo en un diafragma, lo que impediría la detección de sonidos de muy bajo tono.

El examen del precordio debe realizarse de manera sistemática. Comienza con el paciente en posición de decúbito lateral izquierdo, auscultando desde el lugar donde se percibe el choque de la punta del corazón. Luego, el paciente cambia al decúbito supino, y la auscultación continúa en el borde esternal inferior izquierdo, avanzando en dirección cefálica para explorar cada espacio intercostal. Después, se procede a auscultar en dirección caudal, desde el borde esternal derecho superior. Además, el médico debe prestar atención a las áreas sobre la axila izquierda y por encima de las clavículas. A continuación, el paciente se sienta erguido para evaluar la parte posterior, inclinándose hacia adelante si es necesario para auscultar los soplos diastólicos aórticos y pulmonares, así como los roces pericárdicos.

Los principales hallazgos en la auscultación cardíaca se dividen en tres categorías:

- **Sonidos cardíacos:** Estos sonidos breves y temporales que se producen durante la apertura y cierre de las válvulas cardíacas. Se pueden clasificar como sonidos sistólicos o diastólicos.
- **Soplos:** Los soplos resultan de la turbulencia en el flujo sanguíneo y son de duración más prolongada que los sonidos cardíacos. Pueden ser sistólicos, diastólicos o continuos y se evalúan según su intensidad, ubicación y relación con el ciclo cardíaco.
- **Roces:** Los roces son sonidos agudos y ásperos que a menudo consisten en 2 o 3 componentes separados. Pueden variar en relación con la posición del cuerpo y pueden ser casi continuos en pacientes con taquicardia.

5.6. Procesamiento de señales

En el análisis de sonidos cardiacos, se empelan diversas técnicas de procesamiento de señales con el objetivo de extraer características relevantes a partir de grabaciones de audio. Estas características permiten representar de forma cuantitativa y significativa la información contenida en los sonidos del corazón, facilitando así la detección automática de posibles anomalías cardiacas mediante algoritmos de aprendizaje supervisado.

El procesamiento de señales en este contexto incluye etapas como el procesamiento (filtrado y eliminación de ruido), la segmentación de los ciclos cardiacos, y la extracción de características específicas, tales como los coeficientes cepstrales en la frecuencia Mel (MFCC), la energía de la señal, la frecuencia fundamental, entre otros parámetros. Estas características se utilizan posteriormente como entradas para clasificadores o modelos predictivos que permiten distinguir entre sonidos cardiacos normales y anormales.

A continuación, se detallan las técnicas más utilizadas en el procesamiento de señales cardiacas y su aplicación en el diagnostico computacional de enfermedades del corazón.

5.7. Transformada Rápida de Fourier

La transformada rápida de Fourier (FFT) es un algoritmo eficiente para calcular la transformada discreta de Fourier (DFT) y su inversa, que se ha utilizado ampliamente en diversos campos desde mediados del siglo XX, especialmente en ingeniería biomédica. La principal ventaja de la FFT radica en su baja complejidad computacional (O (N log N)), lo que la hace ideal para analizar una variedad de señales biomédicas. En ingeniería biomédica, las FFT se utilizan para analizar electrocardiogramas (ECG), electroencefalogramas (EEG) y otras señales biomédicas para el diagnóstico y estudio de trastorno como arritmias, isquemia miocárdica y epilepsia, así como para una variedad de aplicaciones como el análisis de la calidad del sueño (Fernando, 2024a, pag.155).

La transformada de Fourier describe la intensidad y la fase de una señal en cada frecuencia. Nos proporciona una manera de comprender y analizar las señales de otra perspectiva. El teorema de muestreo es la base del procesamiento digital de señales. Establece que, para recuperar una señal discreta de una señal continua sin distorsión, la frecuencia de muestreo debe ser al menos el doble de la frecuencia más alta de la señal, este principio es importante para evitar el aliasing durante la digitalización. Estos conceptos y métodos tienen una amplia gama de aplicaciones en muchas areas del mundo real, como el procesamiento de audio y video, los sistemas de comunicación, el procesamiento de las señales de radar, etc. Al realizar el análisis

de Fourier de las señales, podemos comprender mejor sus propiedades, procesarlas y analizarla eficazmente. Además del campo matemático, la FFT tiene una amplia gama de aplicaciones en la ingeniería biomédica de cuellos, mediante el desarrollo de nuevos algoritmos y la integración de aprendizaje automático para optimizar el procesamiento de señales, etc. (Fernando, 2024b, pag.155).

5.8. Aplicación de la FFT en señales Biomédicas

En la ingeniería biomédica, el estudio y tratamiento de señales como ECG, EEG y EMG es esencial, ya que estas contienen información clave para el diagnóstico de enfermedades. La Transformada Rápida de Fourier (FFT) se ha convertido en una herramienta crucial dentro de este campo, debido a su capacidad para transformar señales complejas en componentes de frecuencia, facilitando su análisis. Esto permite no solo una mejor visualización de la información útil, sino también la eliminación de ruidos no deseados, lo que mejora la calidad del procesamiento y la interpretación clínica (Fernando, 2024c, pag.155).

5.9. Uso en electrocardiogramas (ECG)

La Transformada Rápida de Fourier (FFT) es una técnica que permite convertir una señal de ECG del dominio del tiempo al dominio de la frecuencia. Su uso en electrocardiogramas ayuda a identificar componentes de frecuencia, detectar anomalías, eliminar ruidos y extraer características útiles para análisis automáticos o con inteligencia artificial. Es especialmente útil para filtrar la señal y analizar el comportamiento del corazón desde una perspectiva espectral. (Fernando, 2024d, pag.155).

5.9.1. Filtrado de ruido con FFT

Uno de los inconvenientes frecuentes al registrar señales de ECG es la presencia de ruido a 50 o 60 Hz, generado por la red eléctrica. Esta interferencia puede deteriorar la calidad de la señal y dificultar su correcta interpretación médica. A través de la transformada rápida de Fourier (FFT), es posible detectar y aislar esta frecuencia específica. Posteriormente, se aplica un filtro de banda eliminada (band-stop) para suprimir dichas frecuencias, lo que permite limpiar la señal del ECG del ruido eléctrico. Este proceso de eliminación de ruido mejora considerablemente la calidad de la señal

y hace que los análisis posteriores, como la evaluación del ritmo cardiaco o la detección de arritmias como la taquicardia sean mas preciso.

Además, la transformada de Fourier de Tiempo Corto (STFT) se emplea para analizar como varia la energía de la señal en el tiempo. Esta técnica permite calcular la intensidad de las frecuencias en distintos momentos, y a partir de esa información se extraen característica que luego pueden ser utilizadas por algoritmos de clasificación para identificar patrones cardiacos o anomalías.

5.10. Transformada de wavelet discreta (DWT)

La transformada de wavelet discreta (DWT) es una herramienta matemática utilizada para descomponer datos de manera jerárquica, desde un nivel general hasta los detalles más específicos. Esta técnica representa una función en términos de una aproximación general y una serie de detalles que capturan las variaciones más finas de la señal. Independientemente del tipo de datos (señales, imágenes, etc.), la DWT ofrece una metodología eficiente y precisa para analizar y representar la cantidad de detalles presente en los datos (yassen et al. 2018, pag.6).

Las wavelets permiten realizar un análisis basado en escala, lo que hace especialmente útiles en aplicaciones como el procesamiento de señales, las matemáticas y el análisis numérico. Debido a su capacidad para proporcionar una representación simultánea en el dominio del tiempo y la frecuencia, la DWT se considera una alternativa poderosa a la transformada rápida de Fourier (FFT). Esta característica es particularmente ventajosa cuando se trabaja con señales no estacionarias, es decir aquellas cuyas propiedades cambian con el tiempo.

En el campo del procesamiento de señales de voz, la DWT ha demostrado un alto rendimiento, especialmente cuando se combina con técnica como los coeficientes cepstrales en la frecuencia Mel (MFCC). Esta combinación mejora aún más la precisión y eficiencia del análisis.

5.10.1. Característica de la Wavelet

La DWT puede describirse como ondas de corta duración que concentran su energía e información en un intervalo de tiempo limitado. Según (Yassen et al. 2018,

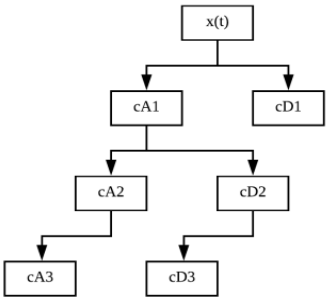
pag.7) afirma que estas ondas tienen un valor promedio cercano a cero y están localizadas tanto en el tiempo como en la frecuencia. Una señal puede descomponerse en dos componentes principales.

- Componente de baja frecuencia: contiene la mayor parte de la información relevante de la señal.
- Componente de alta frecuencia: transmiten detalles más finos y variaciones rápidas.

Los componentes de baja frecuencia se denominan aproximaciones, mientras que los de alta frecuencia se conocen como detalles. Por ejemplo, se muestra como una señal puede descomponerse en dos niveles: en el primer nivel, se obtienen los componentes aproximados (cA1) y detallados (cD1), mientras que, en el segundo nivel, se obtienen cA2 y cD2. Este proceso de descomposición incluye un muestreo descendente, que elimina la redundancia en los datos y garantiza una representación eficiente.

Figura 3

Árbol de descomposición de wavelet



Nota: La figura muestra el árbol de descomposición de wavelet, es una estructura de datos que representa la descomposición de una señal o datos, yaseen et al. (2018).

Matemáticamente la DWT se puede expresar en la siguiente expresión como:

$$DWT(x,y) = \int_{-\infty}^{\infty} Z(t) \frac{1}{\sqrt{|2^x|}} \varphi\left(\frac{t-2^x y}{2^x}\right) dt$$

Ecuación 1: Transformada de Wavelet Discreta (Tomado de Yaseen, 2018).

Esta ecuación se puede utilizar para extraer datos más útiles de la señal de audio utilizando DWT.

5.11. Mel-Frequency cepstral Coefficients (MFCC) y su aplicación en el análisis de señales cardiacas.

Los MFCC son una técnica ampliamente utilizada en el procesamiento de señales de audio, particularmente en reconocimiento de voz y análisis biomédico. Según (uday, 2025) se basan en la percepción humana del sonido y modelan como el oído humano interpreta diferentes frecuencias.

5.12. ¿Cómo se calculan los MFCC?

El cálculo de los coeficientes cepstrales en la frecuencia Mel (MFCC) es una técnica ampliamente utilizada en el procesamiento de señales, especialmente en el análisis de señales biomédicas y de audio. El proceso puede dividirse en varias etapas fundamentales:

5.12.1. Procesamiento de señal

- Normalización y remuestreo: la señal de entrada se normaliza para reducir variaciones en la amplitud y se remuestrea a una frecuencia estándar, por ejemplo, 8 kHz, con el fin de asegurar consistencia en el análisis.
- Aplicación de ventana de Hamming: se segmenta la señal en tramas (Frames) y se aplica una ventana de Hamming para reducir los efectos de discontinuidad en los bordes de cada segmento.

5.13. Transformación en el dominio de la frecuencia

Se aplica la transformada rápida de Fourier (FFT) para obtener el espectro de frecuencia de cada segmento de la señal.

5.14. Bancos de filtros Mel

Se utiliza un conjunto de filtros en escala Mel, diseñados para simular la percepción auditiva humana. Esta escala es más sensible a variaciones en bajas frecuencias, lo que permite una mejor representación perceptual del contenido sonoro.

5.15. Cálculo de los coeficientes cepstrales

Se toma el logaritmo de las energías obtenidas tras el paso por los filtros Mel, y posteriormente se aplica la Transformada Discreta del Coseno (DCT). Esto permite obtener un conjunto de coeficientes que representan las características más relevantes de la señal en el dominio cepstrales: los MFCC.

5.16. Aplicaciones de los MFCC en el análisis de señales cardiacas

Los MFCC, originalmente utilizados en el reconocimiento de Voz, han demostrado ser eficaces en el análisis de sonidos cardiacos gracias a su capacidad para capturar las características espectrales de una señal de manera compacta. Entre sus aplicaciones más relevantes se encuentran:

- **Detección de anomalías cardiacas:** se utilizan para identificar patologías como soplos cardiacos, arritmia y valvulopatías
- **Reducción de ruido:** su aplicación en combinación con técnicas de filtrados mejora la calidad del análisis, eliminando inferencias en la señal cardiaca.

5.17. Filtros digitales

- Los filtros digitales se utilizan para eliminar ruido no deseado de las grabaciones de sonidos cardiacos. Esto puede incluir ruido ambiente artefactos de grabaciones o ruido eléctrico.
- La aplicación de filtros digitales específicos puede mejorar la calidad de la señal y hacer que los sonidos cardiacos sean más distinguibles.

5.18. Machine Learning

5.18.1. ¿Qué es Machine Learning?

El aprendizaje automático se enfoca en la construcción de modelos capaces de identificar patrones y estructuras a partir de los datos proporcionados. De tal forma que podamos usar dichos patrones en datos que no han sido observados previamente. Existen dos tipos principales de modelos de aprendizaje automático, a saber, los modelos de aprendizaje supervisado y los modelos de aprendizaje no supervisado (Hastie et al., 2009).

En el aprendizaje supervisado asumimos conocido el llamado conjunto de entrenamiento (training sample). Cada dato de dicho conjunto está formado por las

variables explicativas o conjunto de características (features) y la variable respuesta. El objetivo del aprendizaje supervisado (Hastie et al., 2009) es construir, a partir de los datos de entrenamiento, un modelo de predicción. Usando este modelo, podremos predecir la variable respuesta de un nuevo conjunto de datos no vistos (test sample) únicamente conociendo sus variables explicativas.

Por otro lado, en el aprendizaje no supervisado (Hastie et al., 2009) no existe variable respuesta. Es decir, el conjunto de entrenamiento solo está formado por variables explicativas. Debido a esto, el objetivo del aprendizaje no supervisado es encontrar modelos que permitan describir como se organizan y agrupan los datos. Estos modelos se utilizarán para explicar los patrones existentes en los nuevos datos no observados.

5.18.2. ¿Cómo funciona el Machine learning?

Existen cuatro etapas para el desarrollo de un proceso en Machine learning, las cuales se enumeran a continuación

1. **Seleccionar y preparar datos de entrenamiento:** Los datos se utilizarán para alimentar el modelo de machine learning con el objetivo de que este aprenda a resolver el problema para el cual ha sido diseñado, inicialmente los datos deben ser etiquetados para indicarle al modelo las características que debe identificar y una vez que ha sido entrenado deberá extraer y detectar las características por su cuenta.

   Cabe recalcar que los datos deben prepararse, organizarse y limpiarse cuidadosamente. De lo contrario, el entrenamiento del modelo de machine learning puede estar sesgado por lo que los resultados de sus predicciones futuras se verán afectados directamente.
2. **Seleccionar un algoritmo:** El siguiente paso es seleccionar un algoritmo para que sea ejecutado sobre el conjunto de datos de entrenamiento. El tipo de algoritmo que se deberá de emplear estará en dependencia del tipo y volumen de los datos de entrenamiento, así como del tipo de problema que hay que resolver.

3. **Entrenar al algoritmo:** Este es un proceso de repetición. Las variables se ejecutan a través del algoritmo y los resultados se comparan con los que debería haber producido. Los pesos y el sesgo se pueden ajustar para aumentar la precisión del resultado.

   Después las variables se vuelven a ejecutar hasta que el algoritmo produzca el resultado correcto en la mayoría de los casos. El algoritmo entrenado es el modelo de Machine learning.
4. **Uso y mejora del modelo:** esta fase consiste en la utilización del modelo sobre nuevos datos los cuales estarán relacionados con el problema a resolver.

5.19. Tipos de Machine learning

A medida que la inteligencia artificial ha evolucionado y machine learning ha proliferado, han surgido diferentes vertientes de esta tecnología, así como distintos tipos de algoritmos de machine learning.

El machine learning se distribuye en dos grandes modelos: **Machine learning supervisado y machine learning no supervisado,** para la realización de este trabajo nos enfocamos en el primer tipo

Machine learning supervisado: Se define por el uso de conjuntos de datos etiquetados para entrenar algoritmos que clasifiquen datos o predigan resultados de forma precisa. Esta modalidad de Machine learning es dependiente de la intervención humana para etiquetar clasificar e introducir datos en el algoritmo y así generar los datos de salida esperados.

Existen dos tipos de datos que pueden ser introducidos en el algoritmo:

- **Clasificación:** Es un método de machine learning supervisado donde el modelo intenta predecir la etiqueta correcta de ciertos datos de entrada. En clasificación, el método es completamente entrenado usando datos de entrenamiento, y luego es evaluado usando datos de prueba antes de ser usado para desarrollar predicciones de datos nuevos. Por ejemplo, para determinar si un paciente está enfermo o no como es el caso de este trabajo, o bien si un correo electrónico es spam.

- **Regresión**: La tarea de predicción es una regresión cuando la variable objetivo es de tipo continuo, un ejemplo puede ser la predicción del salario de una persona dado según su grado de educación, experiencia laboral previa, ubicación geográfica y nivel de antigüedad.

Entre algunas de las aplicaciones prácticas de este tipo de Machine learning encontramos

- La predicción de coste de un siniestro en el caso de las compañías de seguros
- La detección de fraude bancario por parte de entidades financieras.
- La previsión de avería en la máquina de una compañía

5.20. Machine learning no supervisado

El aprendizaje no supervisado, también conocido como aprendizaje automático no supervisado, utiliza algoritmos de aprendizaje automático para analizar y agrupar conjuntos de datos no etiquetados. Estos algoritmos descubren patrones ocultos o agrupaciones de datos sin la intervención humana.

Existen dos tipos de algoritmos para Machine learning no supervisado

- **Clustering**: Clasifica en grupos los datos de salida. Es el caso de las segmentaciones de clientes según que hayan comprado.
- **Asociación:** Descubre reglas dentro del conjunto de datos. Por ejemplo, aquellos clientes que compran un coche también contratan un seguro, por lo que el algoritmo detecta esta regla.

Entre algunos casos prácticos en los que utiliza este tipo de Machine learning esta:

- La segmentación del tipo de clientes en un banco
- La clasificación del tipo de pacientes en un hospital
- El sistema de recomendaciones de contenido según el consumo del usuario en plataformas de streaming de video.

En el marco de la presente monografía, se explorará el desarrollo de un método innovador para el procesamiento de sonidos cardiacos con el objetivo primordial de detectar posibles anomalías cardiacas. En este estudio, nos centraremos

exclusivamente en algoritmo de machine learning supervisado, el algoritmo se seleccionará entre varios modelos con base en el desempeño.

5.21. Detección de anomalias cardiacas mediante aprendizaje supervisado

Para entrenar el algoritmo de machine learning supervisado, nos apoyaremos de datos previamente etiquetados, los cuales obtendremos de repositorios de datos médicos disponible de internet. La utilización de estos datos etiquetados nos permitirá identificar patrones sonoros vinculados a diversas condiciones cardiacas. En este proceso, nos enfocaremos en la detección temprana de anomalías cardiaca mediante la aplicación de técnicas de clasificación.

La utilización exclusiva de machine learning supervisado simplificara el proceso, concentrándonos en la selección de datos médicos para el entrenamiento del algoritmo y la generación de predicciones precisas sobre condiciones cardiacas específica. Este enfoque se alinea con el propósito de obtener una evaluación precisa de los datos sonoros del corazón, facilitando la detección y predicción de posibles problemas cardiacos de manera eficiente.

6. DISEÑO METODOLÓGICO

6.1. Tipo de investigación

La naturaleza de nuestra investigación es de tipo aplicada, puesto que implica una propuesta de solución a un problema practico, en nuestro caso, específicamente desarrollaremos un algoritmo que permita sensar los sonidos del ritmo de los latidos cardiacos, en los pacientes propensos a sufrir enfermedades del corazón o que cuentan con un historial afectado por las mismas, con el fin de contribuir a una temprana detección de cualquier posible anomalía que pueda afectar negativamente al vida de los pacientes.

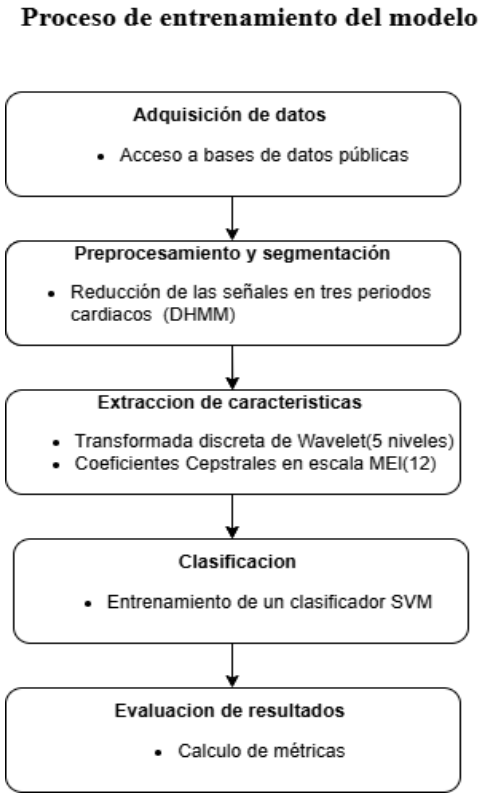
Por otra parte, nuestra investigación también presenta un carácter cuantitativo, debido a que la eficiencia de nuestro modelo deberá ser verificada por medio de pruebas en diversos pacientes, esto nos permitirá probar la precisión que tiene nuestro dispositivo al momento de realizar las medidas de los signos vitales, de este modo contamos con un método cuantificable que nos permite medir la eficacia de nuestro modelo.

Los sonidos cardiacos albergan información determinante acerca del estado y funcionamiento del corazón, motivo por el cual su análisis reviste una importancia capital en la detección precoz de potenciales anomalias. El procesamiento de estas señales demanda la aplicación de diversas técnicas orientadas a la transformación de los datos brutos en información de utilidad para un modelo de clasificación. En el presente estudio, se implementó un enfoque que comprende las distintas etapas del proceso de entrenamiento del modelo.

El siguiente esquema representa el proceso de entrenamiento del modelo, desde la adquisición de datos hasta la validación de los resultados. Esta fase permite al algoritmo aprender a diferenciar entre sonidos cardiacos normales y anómalos a partir de un conjunto etiquetado de datos.

Figura 4

Proceso de Entrenamiento del Modelo.



Nota: Fuente propia.

6.2. Adquisición de datos

Normalmente en esta etapa se utilizan distintos equipos o sensores para la recolección de muestras (Como estetoscopios electrónicos), sin embargo, en este trabajo, empleamos dos bases de datos en el proceso de entrenamiento de nuestro algoritmo. La primera de ellas fue obtenida del sitio web PhysioNet.org, el cual ofrece acceso gratuito a datos biomédicos y software de investigación. Esta base de datos fue diseñada originalmente para entrenar algoritmos de clasificación en concursos dirigidos a investigadores. No obstante, debido a su fácil acceso, decidimos utilizarla en nuestro proyecto. La base de datos está compuesta por cinco conjuntos (enumerados de la A a la E) y contiene un total de 3,126 grabaciones de sonidos cardíacos, con duraciones variables que van desde cinco segundos hasta dos minutos, todas fueron re muestreadas a 2000Hz y se encuentra en formato “.wav”. Estas

grabaciones se dividen en dos categorías: normales y anormales (compuesta por grabaciones de personas con distintas afecciones cardiacas).

Figura 5

Categoría de Base de Datos 1.

Database	Diagnosis	Meaning
training-a	Normal	Normal control group
training-a	MVP	Mitral valve prolapse
training-a	Benign	Innocent or benign murmurs
training-a	AD	Aortic disease
training-a	MPC	Miscellaneous pathological conditions
training-b	Normal	Normal control group
training-b	CAD	Coronary artery disease
training-c	Normal	Normal control group
training-c	MR	Mitral regurgitation
training-c	AS	Aortic stenosis
training-d	Normal: NHC	Recordings collected from 19 normal subjects, aged from 18 to 40 years
training-d	Normal: MARS500	Recordings collected from 6 volunteers (astronauts), a part of the MARS500 project promoted by European Spatial Agency
training-d	Pathologic	pathologic recordings
training-e	Normal	Normal control group
training-e	CAD	Coronary artery disease
training-f	Normal	Normal control group
training-f	Pathologic	pathologic recordings

Nota: Fuente propia.

Las grabaciones catalogadas como "normales" corresponden a sujetos cuyo corazón se encontraba en condiciones óptimas al momento de la grabación, mientras que las grabaciones del grupo "anormal" provienen de personas con problemas cardíacos diagnosticados. Originalmente, la base de datos se dividió de la siguiente manera para los fines del concurso: conjuntos de entrenamiento (de A a F) con 3,153 grabaciones y conjuntos de prueba (de B a E) con 1,277 grabaciones. Sin embargo, esta división no fue empleada, debido a motivos prácticos de carga computacional en su lugar usamos un subconjunto de la misma, tomamos solamente el set “A” y lo dividimos en dos conjuntos a razón de 80% de prueba y 20% de entrenamiento.

La segunda base de datos consta de dos conjuntos de datos: normal y anormal, pero a diferencia de la anterior todos los datos (normal y anormal) pertenecen a una de

cinco categorías, Una categoría normal (N) y cuatro categorías anormales. estenosis aórtica (EA) estenosis mitral (EM) regurgitación mitral (RM) prolapso de la válvula mitral (PVM), El número total de archivos de audio fue de 1.000 para las categorías normal y anormal (200 archivos de audio por categoría), los archivos están en formato .wav. Esta base de datos fue obtenida de internet, y todos sus archivos estaban muestreados a una frecuencia de 8000Hz.

6.3. Pre-precesamiento

6.3.1. Segmentación

Para esta etapa, el objetivo consistió en identificar los componentes principales de los sonidos cardíacos, es decir, los sonidos S1 y S2 en cada grabación para facilitar su procesamiento posterior, sin embargo, debido a la presencia de ruidos como murmullos, sonidos respiratorios y otras interferencias, implementamos el algoritmo propuesto por Schmidt et al. (2010), basado en Modelos Ocultos de Markov dependientes del tiempo (DHMM), el cual ha demostrado alta precisión en tareas similares.

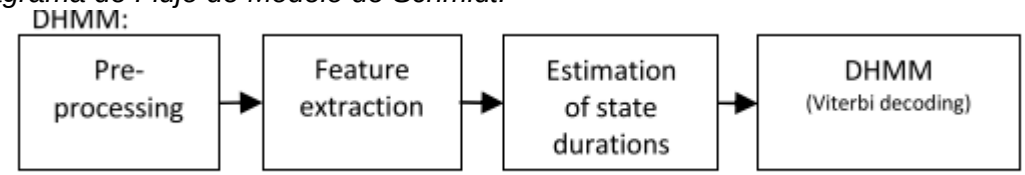
Se implementó una adaptación a Python del código original en MATLAB, el cual se encontraba disponible en el repositorio del autor. Dicha adaptación incluyó una etapa de preprocesamiento que aplicó un filtro pasa banda Butterworth de cuarto orden con el fin de atenuar las componentes de frecuencia que se encontraban fuera del rango típico de los sonidos cardíacos (25Hz – 400Hz). Adicionalmente, se realizó una supresión de picos ruidosos mediante la identificación basada en umbrales y una etapa de extracción de características, que consistió en el cálculo de una envolvente homomórfica, la cual se empleó posteriormente como dato de observación para los modelos de Markov.

Finalmente se realizó un reentrenamiento del algoritmo de Schmidt utilizando cada archivo del conjunto de datos A, que pertenece a la base de datos 1, una vez finalizado el proceso, aplicamos el modelo de segmentación las señales de la base de datos 2 procesándolas individualmente.

En la siguiente figura se presenta un diagrama tomado el trabajo de Schmidt, que resume el flujo de entrenamiento de su modelo original.

Figura 6

Diagrama de Flujo de Modelo de Schmidt.

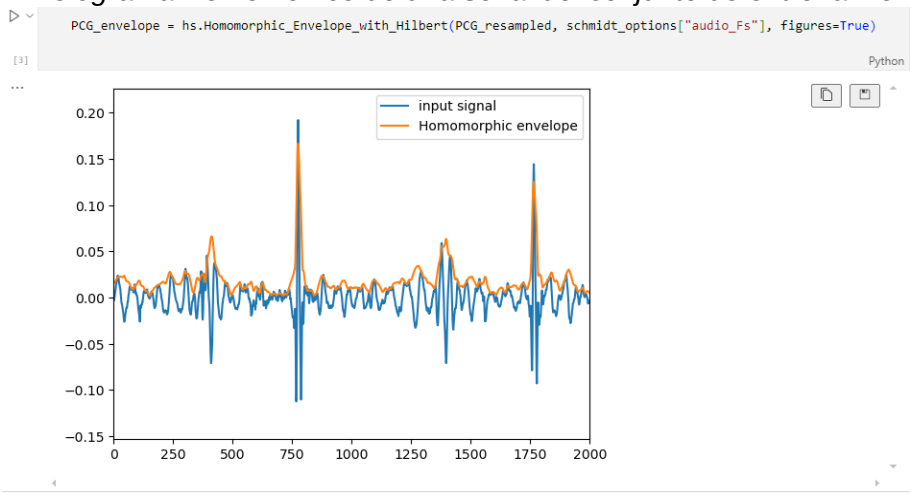


Nota: El diagrama ilustra el diseño del sistema basado en el DHMM. Fuente: Schmidt et al. (2009).

En el siguiente fragmento de código se muestra algunas de las etapas del preprocesamiento que se realizó para lograr la segmentación en Python:

Figura 7

Cálculo del Envelograma Homomorfico de una señal del conjunto de entrenamiento.



Nota: Fuente propia.

Figura 8

Atenuación de picos en la señal

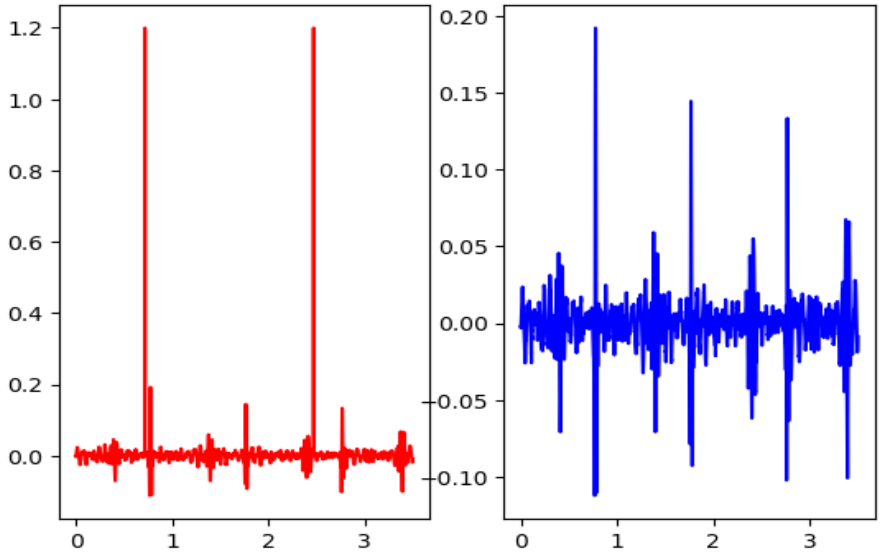
```
# Spikes removal
PCG_spikes = PCG_resampled[0:3500]
random_indices = np.random.randint(0, len(PCG_spikes), 2)
PCG_spikes[random_indices] = 1.2

fig = plt.figure()
ax1 = fig.add_subplot(1, 2, 1)
t = np.linspace(0, len(PCG_spikes), len(PCG_spikes)) / Fs
ax1.plot(t, PCG_spikes, color='red')

ax2 = fig.add_subplot(1, 2, 2)
s = hs.schmidt_spike_removal(PCG_spikes, Fs)
ax2.plot(t, s, color='blue')
```

[4]

Nota: Fuente propia.

Figura 9

Supresión de Picos

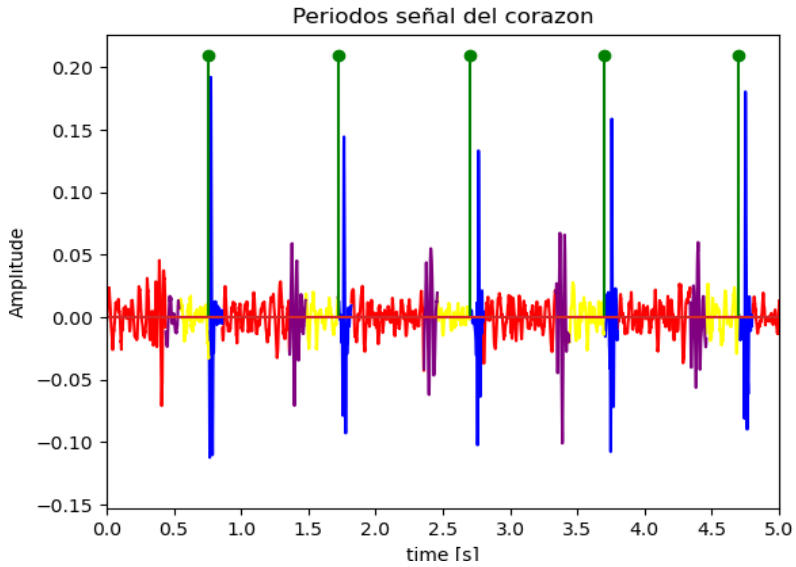


Nota: Fuente propia.

Los resultados de esta etapa fueron grabaciones divididas en tres ciclos cardíacos completos. Esta división se utilizó posteriormente en la extracción de características. La siguiente imagen muestra la representación gráfica de una de las señales luego del proceso de segmentación.

Figura 10

Análisis *por Períodos de la Señal Cardíaca*



Nota: Fuente propia.

los distintos componentes del ciclo cardiaco han sido representados en diferentes colores, y se ha marcado el inicio de cada ciclo cardiaco.

6.3.2. Extracción de Características

La extracción de características es un paso fundamental en el reconocimiento de patrones y el aprendizaje automático. Su objetivo es transformar las señales de audio originales en un conjunto de características relevantes para su posterior análisis o procesamiento. Este proceso reduce la complejidad de los datos al conservar la información más importante, lo que optimiza el rendimiento de los modelos de aprendizaje automático. En este trabajo, se emplearon dos tipos de características comunes en el análisis de sonidos cardíacos: los Coeficientes Cepstrales en la frecuencia de Mel (MFCC) y la Transformada Discreta de Wavelets (DWT).

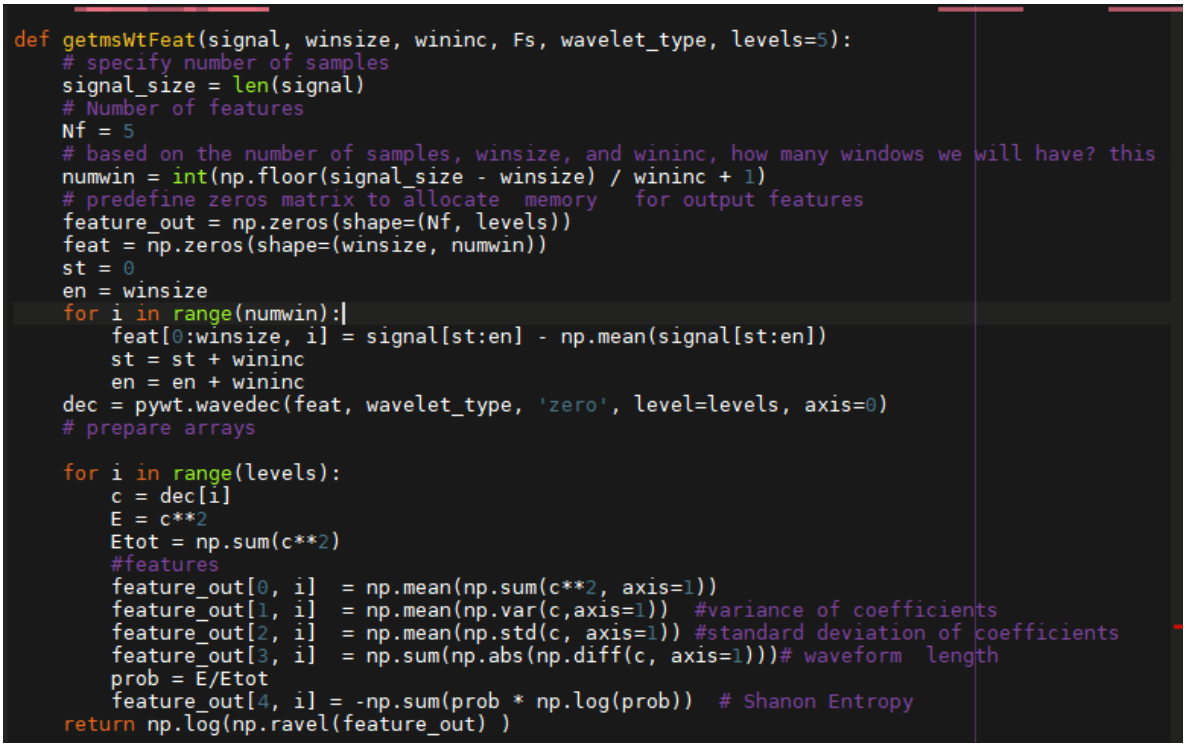
Estas características se extrajeron para cada uno de los tres periodos segmentados por señal Luego combinamos ambas características en un solo vector, el cual usamos como entrada para nuestro algoritmo de clasificación de **“Maquinas de soporte vectorial”,** decidimos hacerlo de esta forma, debido a que estudios anteriores a este trabajo, como (Yaseen G.-Y. S., 2018) han demostrado que la combinación de estas características proporciona un mayor grado de precisión a la hora de clasificar los datos en conjunto con un clasificador como el antes mencionado .

Para cada segmento de señal, calculamos doce coeficientes cepstrales. De esta manera, cada segmento de señal independiente tiene doce valores de coeficientes cepstrales. Los parámetros utilizados para extraer estas características fueron: frecuencia de muestreo de 8,000 Hz, duración de la ventana de 60 ms, con un salto de 20 ms entre cada una. Además, realizamos cinco niveles de descomposición temporal para cada segmento. En cada nivel de descomposición, obtuvimos cinco características: media, varianza, desviación estándar, longitud de onda y entropía de Shannon. En total, para cada muestra de audio, se extrajeron 37 características. Finalmente se promedió el valor de cada uno de los coeficientes que conforman el vector de características a lo largo de los tres segmentos de cada señal.

Los siguientes fragmentos de código ilustran las funciones empleadas para la extracción de características cepstrales (MFCC) y de la transformada discreta de Wavelet. Asimismo, muestran cómo estas características se combinan en un arreglo unificado que sirve como entrada para el algoritmo de clasificación.

Figura 11

Extracción de Transformada de Wavelet.

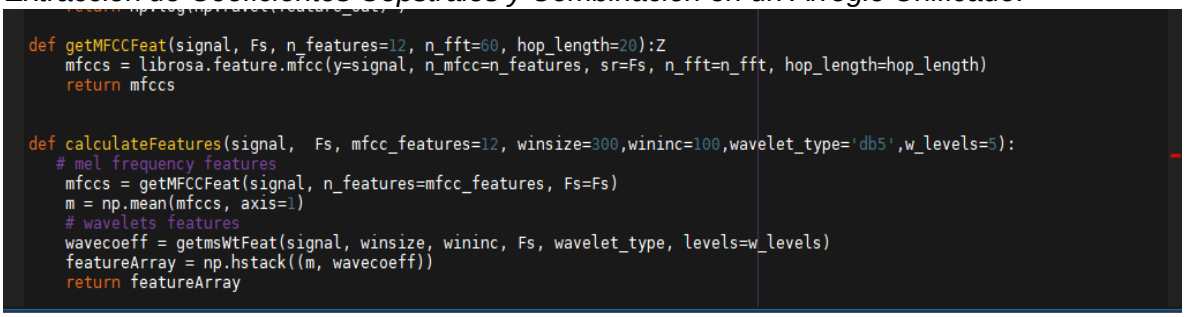


```
def getmsWtFeat(signal, winsize, wininc, Fs, wavelet_type, levels=5):
    # specify number of samples
    signal_size = len(signal)
    # Number of features
    Nf = 5
    # based on the number of samples, winsize, and wininc, how many windows we will have? this
    numwin = int(np.floor(signal_size - winsize) / wininc + 1)
    # predefine zeros matrix to allocate  memory   for output features
    feature_out = np.zeros(shape=(Nf, levels))
    feat = np.zeros(shape=(winsize, numwin))
    st = 0
    en = winsize
    for i in range(numwin):
        feat[0:winsize, i] = signal[st:en] - np.mean(signal[st:en])
        st = st + wininc
        en = en + wininc
    dec = pywt.wavedec(feat, wavelet_type, 'zero', level=levels, axis=0)
    # prepare arrays

    for i in range(levels):
        c = dec[i]
        E = c**2
        Etot = np.sum(c**2)
        #features
        feature_out[0, i]  = np.mean(np.sum(c**2, axis=1))
        feature_out[1, i]  = np.mean(np.var(c,axis=1))  #variance of coefficients
        feature_out[2, i]  = np.mean(np.std(c, axis=1)) #standard deviation of coefficients
        feature_out[3, i]  = np.sum(np.abs(np.diff(c, axis=1)))# waveform  length
        prob = E/Etot
        feature_out[4, i] = -np.sum(prob * np.log(prob))  # Shanon Entropy
    return np.log(np.ravel(feature_out) )
```

Nota: Fuente propia.

Figura 12

Extracción *de Coeficientes Cepstrales y Combinación en un Arreglo Unificado.*



```
def getMFCCFeat(signal, Fs, n_features=12, n_fft=60, hop_length=20):Z
    mfccs = librosa.feature.mfcc(y=signal, n_mfcc=n_features, sr=Fs, n_fft=n_fft, hop_length=hop_length)
    return mfccs


def calculateFeatures(signal,  Fs, mfcc_features=12, winsize=300,wininc=100,wavelet_type='db5',w_levels=5):
   # mel frequency features
    mfccs = getMFCCFeat(signal, n_features=mfcc_features, Fs=Fs)
    m = np.mean(mfccs, axis=1)
    # wavelets features
    wavecoeff = getmsWtFeat(signal, winsize, wininc, Fs, wavelet_type, levels=w_levels)
    featureArray = np.hstack((m, wavecoeff))
    return featureArray
```

Nota: Fuente propia.

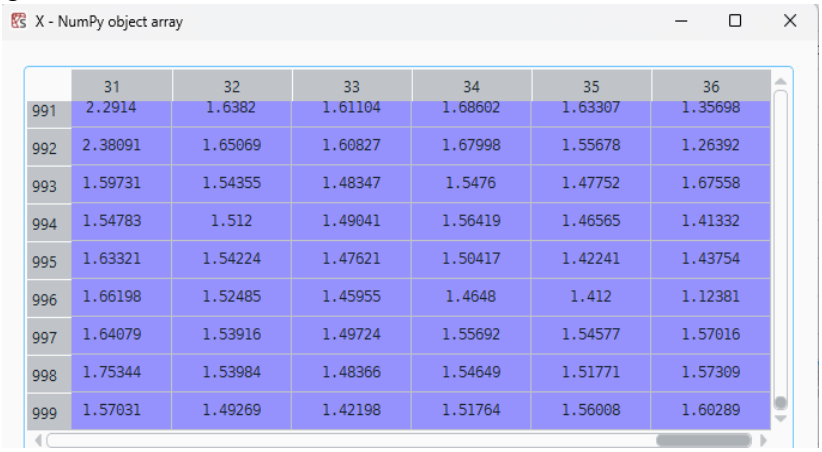
Las siguientes figuras mostradas realizan las siguientes tareas:

getMFCCFeat: Calcula los 12 coeficientes cepstrales a partir de cada segmento de señal.

getmsWtFeat: Realiza la descomposición en cinco niveles y extrae cinco características por cada nivel que son: media, varianza, desviación estándar, longitud de onda y entropía de Shannon.

Calculate Features: Combina ambos vectores de características en una sola estructura, lista para alimentar un clasificador.

Figura 13: *Arreglo de Características.*



	31	32	33	34	35	36
991	2.2914	1.6382	1.61104	1.68602	1.63307	1.35698
992	2.38091	1.65069	1.60827	1.67998	1.55678	1.26392
993	1.59731	1.54355	1.48347	1.5476	1.47752	1.67558
994	1.54783	1.512	1.49041	1.56419	1.46565	1.41332
995	1.63321	1.54224	1.47621	1.50417	1.42241	1.43754
996	1.66198	1.52485	1.45955	1.4648	1.412	1.12381
997	1.64079	1.53916	1.49724	1.55692	1.54577	1.57016
998	1.75344	1.53984	1.48366	1.54649	1.51771	1.57309
999	1.57031	1.49269	1.42198	1.51764	1.56008	1.60289

Nota: Fuente propia.

La figura (13) ilustra las dimensiones del arreglo de características utilizado para entrenar nuestro algoritmo de clasificación de máquinas de soporte vectorial. Las filas de este arreglo corresponden al número de señales empleadas, mientras que las columnas representan la totalidad de las características extraídas por cada señal.

6.3.3. Clasificación

La última fase del análisis de los sonidos cardíacos es la clasificación, en la cual las características extraídas en el bloque anterior son utilizadas como entrada para un algoritmo que permita distinguir entre señales normales y patológicas. En este trabajo se empleó una Máquina de Soporte Vectorial (Support Vector Machine, SVM, por sus siglas en inglés).

Las SVM son algoritmos de aprendizaje supervisado ampliamente utilizados en tareas de clasificación, y han demostrado ser eficaces en la identificación de patrones anómalos en señales biomédicas como los sonidos cardíacos (Al-Shannaq et al., 2025). Su funcionamiento se basa en encontrar el hiperplano óptimo que separa las clases de datos con el mayor margen posible. Para ello, las SVM utilizan un conjunto de datos conocidos como vectores de soporte, que son los puntos más cercanos al límite de decisión y que definen su orientación. Este hiperplano permite clasificar nuevas muestras según su posición relativa a dicho límite.

Para manejar problemas de clasificación no lineal, las SVM implementan funciones denominadas kernels, que permiten proyectar los datos a espacios de mayor dimensión, donde es más factible encontrar una separación lineal entre clases. Entre las funciones kernel más comunes se encuentran: lineal, polinómica, sigmoide, y la función de base radial gaussiana (Radial Basis Function, RBF), entre otras (Han et al., 2011). La selección del kernel adecuado es un aspecto crucial, ya que afecta directamente el rendimiento del modelo

En nuestro caso, se optó por emplear el kernel RBF (Radial Basis Function), ya que, tras múltiples pruebas empíricas, fue el que ofreció los mejores resultados de clasificación para los sonidos cardíacos procesado.

Figura 14

Fase de preparación previa a la ejecución del algoritmo de clasificación.

```
from sklearn import svm
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

X_train = np.nan_to_num(X_train, nan=0.0)
X_test  = np.nan_to_num(X_test, nan=0.0)

scaler= StandardScaler()
# SVC needs proper scaling of the input data
scaler.fit(X_train)
X_train = scaler.transform(X_train)

# Training the SVC
clf = svm.SVC(kernel="rbf")
clf.fit(X_train,y_train)

# Predict
# Scale also test data
scaler.fit(X_test)
X_test = scaler.transform(X_test)

 # make predictions
yp = clf.predict(X_test)

#============= Classification results===============
# confusion matrix
cf=confusion_matrix(y_test, yp)
sns.heatmap(cf, annot=True, fmt='3.2f')
plt.show()
accuracy_score(y_test,yp)
```

Nota: Fuente propia.

En la figura (14) se ilustra la fase de preparación previa a la ejecución del algoritmo de clasificación. Inicialmente, se gestionan los valores faltantes sustituyéndolos por ceros, seguido del escalado de las características en los conjuntos de entrenamiento y prueba mediante StandardScaler. A continuación, se entrena un modelo SVM utilizando los datos de entrenamiento ya escalados, el cual se emplea posteriormente para generar predicciones sobre el conjunto de prueba también escalado. Finalmente, se evalúa el rendimiento del modelo a través del cálculo de la matriz de confusión y las métricas resultantes, cuyo análisis detallado se presentará en la sección de “resultados”.

6.4. Diseño de la aplicación móvil para la adquisición de sonidos cardiaco

El análisis de sonidos cardiacos es una herramienta esencial en la detección de enfermedades cardiovasculares, ya que permite identificar anomalías en el funcionamiento del corazón de forma no invasiva. Con el avance de la tecnología, los dispositivos móviles se han convertido en herramientas poderosas capaces de realizar tareas complejas, como la adquisición y procesamiento de datos de audios. Esta monografía tiene como objetivo implementar un método de preprocesamiento de sonidos del corazón para la detección de anomalías cardiaca, unos de los propósitos para cumplir dicho objetivo es desarrollar una aplicación móvil que permita la adquisición de sonidos a través del micrófono celular, facilitando así el monitoreo inicial de la salud cardiaca.

La aplicación se centra en la adquisición y almacenamiento de estos sonidos para su posterior análisis mediante técnicas de algoritmo de machine learning. A través de este informe se documenta el proceso de desarrollo de la aplicación, desde la configuración inicial de Android studio. La finalidad es crear una herramienta accesible que permita a los usuarios y profesionales de la salud registrar sonidos cardiacos de manera sencilla, contribuyendo a la detección temprana de anomalía y a la mejora de la calidad de vida de los pacientes.

De esta manera la aplicación busca acercar las herramientas de diagnóstico a más personas, aprovechando las ventajas de los dispositivos móviles. Además, este proyecto abre puertas a futuras investigaciones que puedan perfeccionar el analis de estos sonidos.

Para el desarrollo de la aplicación móvil que permite la adquisición de sonidos cardiacos, se siguió una metodología estructurada en varias etapas, cada una con el objetivo de garantizar una implementación funcional y precisa. A continuación, se describen los pasos realizados durante el desarrollo del proyecto en Android studio.

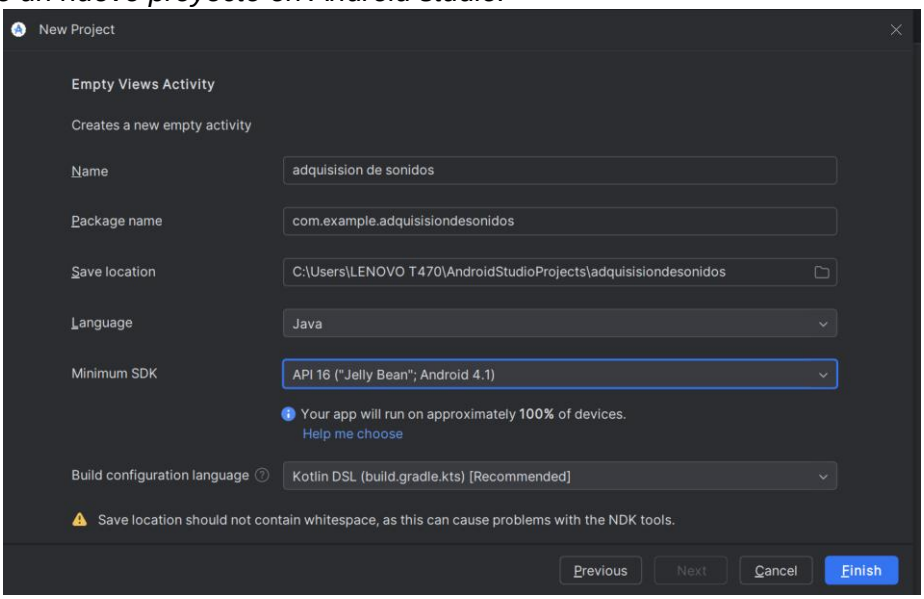
6.4.1. Configuración del entorno de desarrollo Android studio:

Se utilizo Android studio como plataforma de desarrollo debido a su compatibilidad con el sistema operativo Android y sus amplias herramientas de diseño y depuración.

El primer paso fue crear un nuevo proyecto en Android studio, configurando las versiones mínimas, en este caso se eligió la versión 4.1 para asegurar un 95% de compatibilidad con la mayoría de los dispositivos móviles. Posteriormente se eligió el lenguaje java en vez de kotlin debido a que la mayoría de información para el desarrollo de la aplicación está orientada para personas que dominan java en lugar kotlin, una vez que seleccionamos el lenguaje a trabajar para el desarrollo de la aplicación le dimos en finish.

Figura 15

Creación de un nuevo proyecto en Android studio.

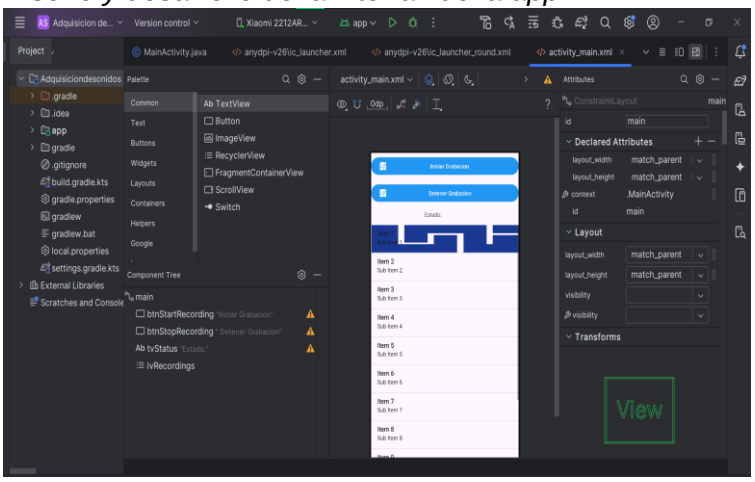


Nota: Fuente propia.

Una vez que se le dio en el botón de finish se nos carga una interfaz donde se procedió a diseñar la interfaz de usuario sencilla e intuitiva para facilitar al usuario la interacción con la aplicación. La pantalla principal incluye botones para iniciar y detener la grabación de sonidos, así como una lista de selección donde nos permite visualizar los datos de audios almacenado. Se utilizaron componentes como Button y TextView para los elementos interactivos, y se diseñó un layout que garantizara la practicidad de la aplicación en diferentes tamaños de pantalla .

Figura *16*

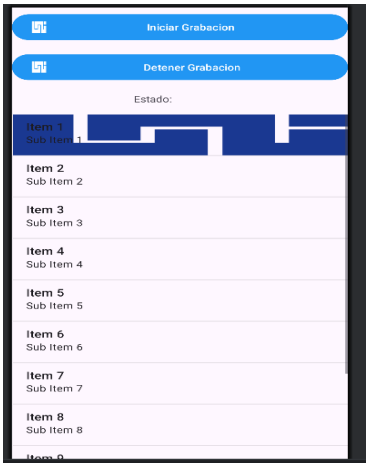
Diseño y desarrollo de la interfaz de la app



Nota: Fuente propia.

Figura 17

Interfaz de la app



Nota: Fuente propia.

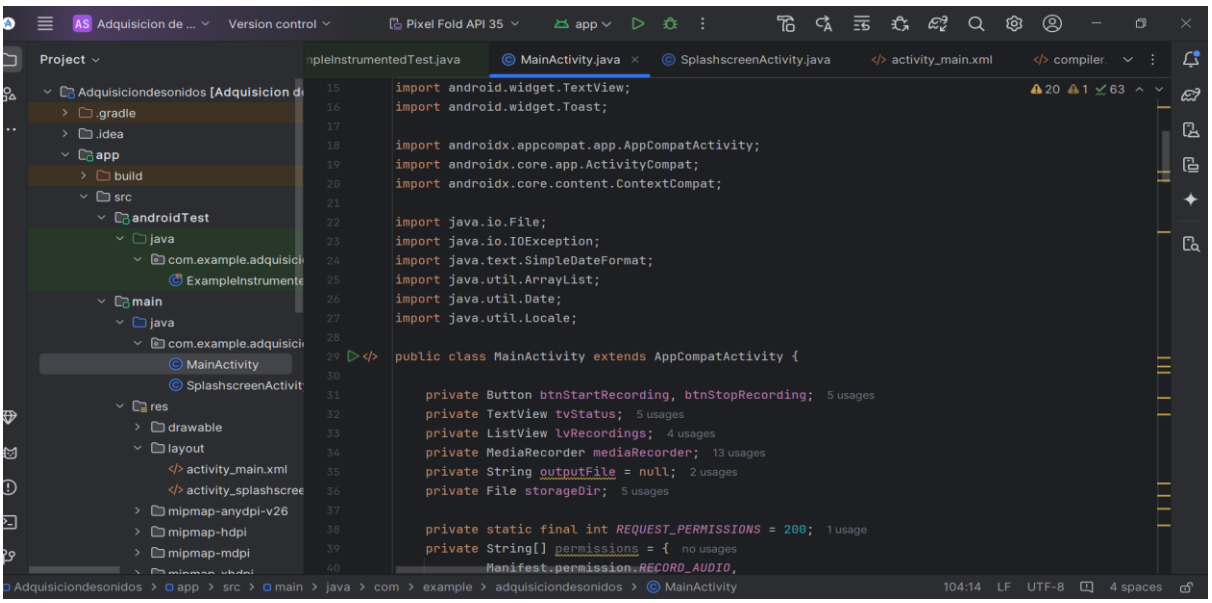
Luego que se diseñó la interfaz de la aplicación se añadieron los permisos necesarios en el archivo AndroidManifest.xml, principalmente el permiso para acceder al micrófono del dispositivo (RECORD_AUDIO), permitiendo que la aplicación pueda capturar sonidos.

6.4.2. Almacenamiento y procesamiento del audio

Para implementar la funcionalidad de captura y almacenamiento de sonidos cardiacos, se desarrolló un Código en java dentro de la clase mainActivity que permite iniciar, detener y guardar grabaciones de audio utilizando el micrófono del dispositivo Android. A continuación, se detalla cada aspecto del proceso:

Figura 18

Código fuente de la aplicación.



Nota: Fuente propia

6.4.3. Configuración de permiso

Para poder grabar y almacenar los sonidos, se solicitó al usuario el permiso necesario en tiempo de ejecución, específicamente para acceder al micrófono (RECORD_AUDIO) y al almacenamiento (WRITE_EXTERNAL_STORAGE y READ_EXTERNAL_STORAGE). Estos permisos fueron manejados a través de una verificación en el método onRequestPermissionsResult.

Figura 19

Configuración de permisos de usuario.

```
private static final int REQUEST_PERMISSIONS = 200;  1 usage
private String[] permissions = {  no usages
        Manifest.permission.RECORD_AUDIO,
        Manifest.permission.WRITE_EXTERNAL_STORAGE,
        Manifest.permission.READ_EXTERNAL_STORAGE
};
```

Nota: Fuente propia.

Figura 20

Manejo y respuesta de la solicitud de permiso de usuario

```
@Override  14 usages
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults)
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == REQUEST_PERMISSIONS) {
        boolean allGranted = true;
        for (int result : grantResults) {
            if (result != PackageManager.PERMISSION_GRANTED) {
                allGranted = false;
                break;
            }
```

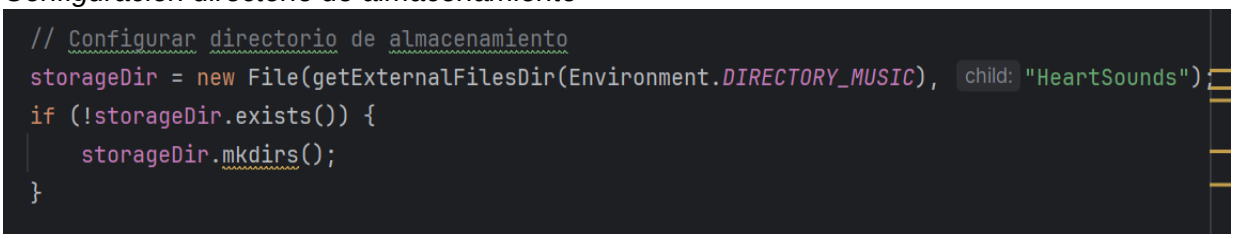
Nota: Fuente propia.

6.4.4. Configuración del directorio de almacenamiento:

Se creo un directorio específico para guardar las grabaciones de audio, llamado HeartSounds. Este directorio se genera en la carpeta de música del dispositivo (Environment.DIRECTORY_MUSIC), utilizando getExternalFilesDir para garantizar que las grabaciones sean accesibles y organizadas en un solo lugar.

Figura 21

Configuración directorio de almacenamiento



```
// Configurar directorio de almacenamiento
storageDir = new File(getExternalFilesDir(Environment.DIRECTORY_MUSIC),  child: "HeartSounds");
if (!storageDir.exists()) {
    storageDir.mkdirs();
}
```

Nota: Fuente propia

6.4.5. Inicio de la grabación:

Para iniciar la adquisición de sonidos cardiacos, se implementó el método startRecording, el cual configura el objeto MediaRecorder para grabar el audio.

En este método se generó un nombre único para cada archivo de audio basado en la fecha y hora actual, permitiendo que cada grabación tenga un nombre único.

Figura 22

Inicialización de la grabación de los sonidos capturados

```
private void startRecording() {  1 usage
    String timeStamp = new SimpleDateFormat( pattern: "yyyyMMdd_HHmmss", Locale.getDefault()).format
    String fileName = "heart_sound_" + timeStamp + ".m4a";

    File audioFile = new File(storageDir, fileName);
    outputFile = audioFile.getAbsolutePath();

    mediaRecorder = new MediaRecorder();
    mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
    mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
    mediaRecorder.setAudioChannels(1);
    mediaRecorder.setAudioSamplingRate(44100);
    mediaRecorder.setAudioEncodingBitRate(192000);
    mediaRecorder.setOutputFile(outputFile);
```

Nota: Fuente propia.

6.4.6. La configuración de MediaRecorder incluyo los siguientes parámetros:

- **Fuente de audio:** Se uso Mic para capturar sonido desde el micrófono del dispositivo.
- **Formato de salida:** Se selecciono el formato MPEG_4 para almacenar el audio en un archivo .m4a.
- **Configuración de audio:** Se utilizó el codificador AAC, con una taza de muestreo de 44,100 Hz y una tasa de bits de 192 kbps asegurando una calidad de audio adecuada para el análisis posterior.

Figura 23

Configuración y preparación del MediaRecorder para grabación de audio.

```
mediaRecorder = new MediaRecorder();
mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
mediaRecorder.setAudioChannels(1);
mediaRecorder.setAudioSamplingRate(44100);
mediaRecorder.setAudioEncodingBitRate(192000);
mediaRecorder.setOutputFile(outputFile);
```

Nota: Fuente propia.

6.4.7. Finalización de la grabación:

Una vez que el usuario finaliza la grabación, el método stopRecording se encarga de detener y liberar los recursos del MediaRecorder.

Despues de detener la grabación, el estado de la interfaz de usuario cambia para permitir nuevas grabaciones, y se actualiza la lista de grabaciones disponibles.

Figura 24

Ajuste de finalización de grabación de audio.

```
private void stopRecording() {  1 usage
    try {
        mediaRecorder.stop();
        mediaRecorder.release();
        mediaRecorder = null;

        tvStatus.setText("Estado: Grabación detenida.");
        btnStartRecording.setVisibility(View.VISIBLE);
        btnStopRecording.setVisibility(View.GONE);

        // Actualizar lista de grabaciones
        loadRecordings();
    } catch (RuntimeException stopException) {
        // Manejar excepciones si se detiene la grabación antes de iniciarla correctamente
        stopException.printStackTrace();
        Toast.makeText( context: this,  text: "Error al detener la grabación.", Toast.LENGTH_SHORT).sh
    }
}
```

Nota: Fuente propia.

6.4.8. Carga de grabaciones existentes:

Para facilitar a las grabaciones, el método loadRecordings busca todos los archivos en el directorio HeartSounds y lo muestra en una lista (ListView). Si se encuentran archivos de grabación, se listan mediante un adaptador (ArrayAdapter<File>). En caso contrario, se indica al usuario que no hay grabaciones disponibles.

Figura 25

Carga de las grabaciones de audios almacenados

```
private void loadRecordings() {  2 usages
    File[] files = storageDir.listFiles();
    if (files != null && files.length > 0) {
        ArrayList<File> fileList = new ArrayList<>();
        for (File file : files) {
            if (file.isFile()) {
                fileList.add(file);
            }
        }
        ArrayAdapter<File> adapter = new ArrayAdapter<>( context: this, android.R.layout.simple_list
        lvRecordings.setAdapter(adapter);
    } else {
        ArrayList<String> noFiles = new ArrayList<>();
        noFiles.add("No hay grabaciones");
        ArrayAdapter<String> adapter = new ArrayAdapter<>( context: this, android.R.layout.simple_li
        lvRecordings.setAdapter(adapter);
    }
}
```

Nota: Fuente propia.

6.4.9. Reproducción de grabaciones:

La funcionalidad de reproducción de grabaciones se implementó en el método playAudio, que utiliza la clase MediaPlayer para reproducir el archivo de audio seleccionado.

Al seleccionar un archivo dentro de la lista de grabaciones, se indica la reproducción y se muestra un mensaje en la interfaz indicando que el audio se está reproduciendo, liberando el Mediaplayer y actualizando el estado de la aplicación.

Este proceso de almacenamiento y gestión de archivos permite capturar y organizar los sonidos cardiacos de manera eficiente, facilitando su uso en análisis futuros. Además, la funcionalidad de reproducción es fundamental para verificar la calidad de las grabaciones antes de su procesamiento y validación dentro del algoritmo desarrollado.

Figura 26

Ajuste de la reproducción de las grabaciones de audios.

```
private void playAudio(File audioFile) {  1 usage
    MediaPlayer mediaPlayer = new MediaPlayer();
    try {
        mediaPlayer.setDataSource(audioFile.getAbsolutePath());
        mediaPlayer.prepare();
        mediaPlayer.start();
        tvStatus.setText("Reproduciendo: " + audioFile.getName());

        // Actualizar el estado al finalizar la reproducción
        mediaPlayer.setOnCompletionListener(new MediaPlayer.OnCompletionListener() {
            @Override
            public void onCompletion(MediaPlayer mp) {
                tvStatus.setText("Reproducción finalizada.");
                mp.release();
            }
        });

    } catch (IOException e) {
        e.printStackTrace();
        Toast.makeText( context: this,  text: "Error al reproducir el audio.", Toast.LENGTH_SHORT).sho
    }
```

Nota: Fuente propia.

Gracias a este sistema de gestión de archivos, los sonidos cardiacos pueden ser organizados de manera eficiente, permitiendo su uso en estudios médicos, investigaciones o el entrenamiento del algoritmo de detección de anomalías cardiacas.

6.5. Evaluación del rendimiento del algoritmo

La matriz de confusión es una herramienta fundamental para evaluar el rendimiento de un algoritmo, especialmente en el contexto del "aprendizaje supervisado" (como el que estamos utilizando). Ofrece una visión detallada sobre el desempeño del modelo, permitiendo identificar tanto sus aciertos como sus errores y debilidades. En la matriz, cada fila representa las instancias de una clase real, mientras que cada columna refleja las instancias de una clase predicha, o viceversa. Así, la diagonal de la matriz muestra las instancias correctamente clasificadas. El término "matriz de confusión" se debe a su capacidad para revelar fácilmente si el sistema confunde dos clases, es decir, si tiende a etiquetar erróneamente una clase como otra, Ahmed (2025).

Figura 27

Matriz de confusión.

		Predicción	
		Positivos	**Negativos**
Observación	**Positivos**	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

Nota: Representación gráfica que muestra los aciertos y errores de un modelo de clasificación al comparar predicciones con valores reales (Zelada, 2017).

Dentro de la matriz de confusión encontramos cuatro categorías que son:

- **Verdadero positivo (TP)**: Esta categoría corresponde a todos los casos en que el modelo predijo correctamente la clase positiva. En el contexto de nuestro trabajo sobre el procesamiento de sonidos cardíacos, un Verdadero Positivo (TP) ocurre cuando una grabación de sonido cardíaco que realmente pertenece a un corazón sano fue correctamente identificada como tal por nuestro modelo**.** En otras palabras, el modelo predijo que el sonido cardíaco correspondía a un corazón sano, y esta predicción coincidió con la realidad de que el corazón grabado era efectivamente sano.

- **Verdadero negativo (TN**): El modelo predijo correctamente la clase negativa. Un Verdadero Negativo, en el marco de nuestro trabajo, se da cuando una grabación de sonido cardíaco que realmente pertenece a un corazón enfermo fue correctamente identificada como tal por el modelo. Esto significa que el modelo predijo que el sonido cardíaco correspondía a un corazón enfermo, y esta predicción coincidió con la realidad de que el corazón grabado presentaba una condición patológica.
- **Falso positivo (FP):** El modelo predijo incorrectamente la clase positiva. ocurre cuando el modelo de procesamiento de sonidos cardíacos identifica una grabación que realmente pertenece a un corazón enfermo como si fuera de un corazón sano**.** En otras palabras, el modelo predijo incorrectamente que el sonido cardíaco correspondía a un corazón sano, cuando en realidad el corazón grabado presentaba una condición patológica.
- **Falso negativo (FN):** El modelo predijo incorrectamente la clase negativa. se da cuando el modelo de procesamiento de sonidos cardíacos identifica una grabación que realmente pertenece a un corazón sano como si fuera de un corazón enfermo**.** Esto significa que el modelo predijo incorrectamente que el sonido cardíaco correspondía a un corazón enfermo, cuando en realidad el corazón grabado era sano.

La matriz de confusión aplicada al caso específico de este trabajo queda de la siguiente manera.

Figura 28

Banco de datos.

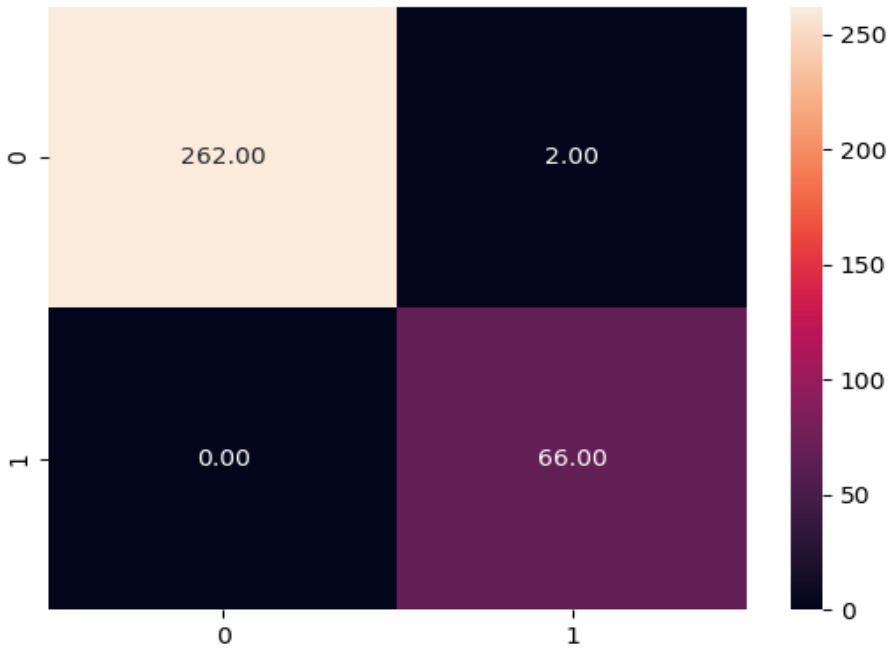
Matriz de Confusión	Predicho: Corazón Sano	Predicho: Corazón Enfermo
Real: Corazón Sano	Verdadero Positivo (TP)	Falso Negativo (FN)
Real: Corazón Enfermo	Falso Positivo (FP)	Verdadero Negativo (TN)

Nota: Fuente propia.

Luego de haber procesado las señales de nuestro banco de datos, utilizando las especificaciones antes mencionadas, logramos obtener un rendimiento bastante satisfactorio, que muestra un porcentaje de clasificación bastante acertado, la imagen de abajo muestra la matriz de confusión que obtuvimos al final de nuestro experimento, así como la precisión del modelo, como se puede apreciar, el porcentaje de falsos verdaderos y falsos negativos es extremadamente bajo.

Figura 29

Resultado Final: Matriz de Confusión del Algoritmo.



Nota: La figura muestra la estructura básica de una matriz de confusión.

Para comprender en profundidad la matriz de confusión, es esencial de igual forma, entender las métricas importantes que se utilizan para medir el rendimiento de un modelo.

6.5.1. Precisión

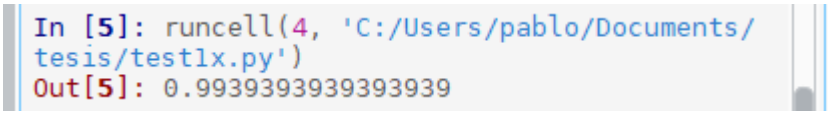
La precisión mide el número total de clasificaciones correctas dividido por el número total de casos.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Ecuación 2: Cálculo de la Precisión Global “Accuracy”, Ahmed (2025).

Figura 30

Valor de Precisión del Modelo.



Nota: Fuente propia.

6.5.2. Sensibilidad

mide el número total de verdaderos positivos identificados por el modelo dividido por el número total de positivos reales.

$$Recall = \frac{TP}{TP + FN} \; or \frac{True\ Positive}{Actual\ Results}$$

Ecuación 3: Sensibilidad Recall (Tomado de Ahmed, 2025).

6.5.3. Especificidad

La especificidad mide el número total de verdaderos negativos dividido por el número total de negativos reales.

$$Specificity = \frac{TN}{TN + FP}$$

Ecuación 4: Especificidad “Specificity” (Tomado de Ahmed, 2025).

Existen otras métricas importantes, sin embargo, estas métricas Juntas, proporcionan una visión integral del rendimiento del modelo, permitiendo una toma de decisiones más informada en contextos críticos como el diagnóstico médico.

6.5.4. Calculando la sensibilidad y especificidad del modelo

Tomando en cuenta los puntos antes mencionados el cálculo de dichas matrices para nuestro modelo seria de la siguiente manera.

6.5.5. Sensibilidad

$$\frac{262}{262 + 0} = 1$$

Ecuación 5

Cálculo de sensibilidad (Tomado de Ahmed, 2025).

Este valor indica que el modelo identificó correctamente todos los casos negativos (sin falsos positivos). Es decir, no clasificó erróneamente ningún caso sano como enfermo.

6.5.6. Especificidad

$$\frac{66}{66+2} = 0.97$$

Ecuación 6: Caculo de especificidad (Tomado de Ahmed, 2025).

La especificidad mide el número total de verdaderos negativos dividido por el número total de verdaderos negativos.

6.6. Pruebas y resultado de la aplicación

Para evaluar el correcto funcionamiento de la aplicación de adquisición de adquisición de sonidos cardiacos, se realizaron diversas pruebas centradas en la funcionalidad, calidad del audio, rendimiento y experiencia del usuario. A continuación, se detalla los resultados obtenidos

6.6.1. Pruebas de funcionalidad

Para evaluar la funcionalidad de la aplicación se llevaron a cabo diferentes métodos donde una de ellas es comprobar que cada una de las características implementadas en la aplicación operara sin errores. Se realizaron en distintos dispositivos Android para garantizar la compatibilidad y estabilidad

6.6.2. Instalación de la aplicación

6.6.2.1. Activar el modo desarrollador en el telefono

Primero, ingreso a “Configuración” del teléfono y voy a la sección “Acerca del teléfono”. Buscamos la opción “número de compilación” y la presionamos siete veces consecutivas hasta que aparezca un mensaje indicando que el “Modo Desarrollador ha sido activado”. Luego, regresamos al menú principal de “Configuración”, entramos en las “Opciones de desarrollador” y activo la opción “Depuración por USB” a continuación las siguientes imágenes.

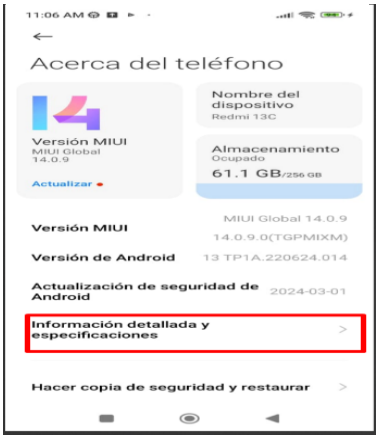
Figura 31

Ajuste del dispositivo

Nota: Fuente propia.

Figura 32

Acerca del telefono



Nota: Fuente propia.

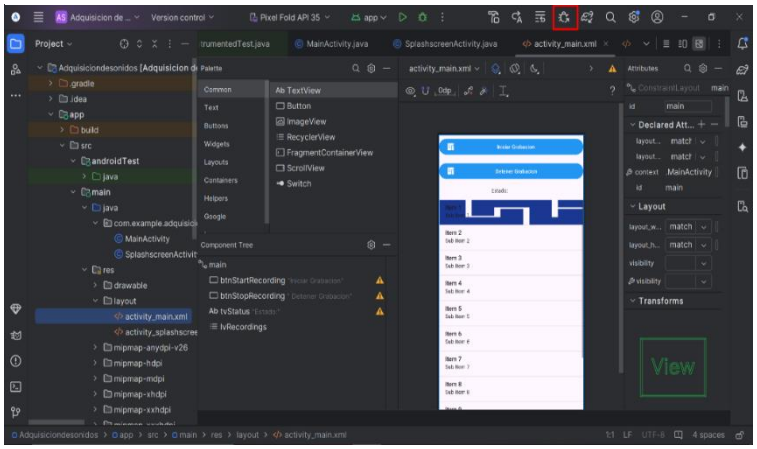
Figura 33

Numero de compilación

Nota: Fuente propia.

Figura 34

Compilación de la app al dispositivo

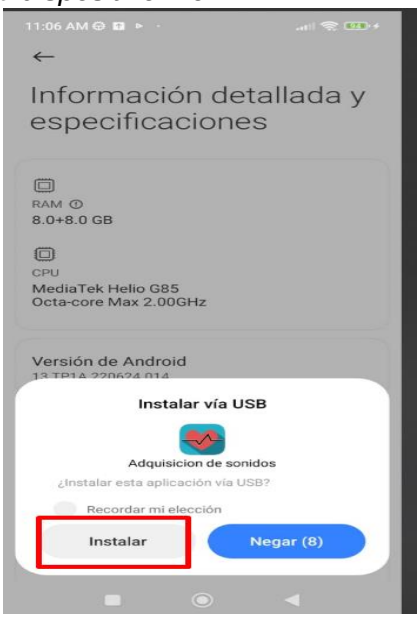


Nota: Fuente propia.

Una vez que el telefono este en modo desarrollador, abrimos la IDE de Android Studio con nuestra aplicación y ejecutamos el código. Esto iniciara el proceso de instalación de la aplicación, tras lo cual se mostrarán las siguientes imágenes.

Figura 35

Permisos para instalar la app al dispositivo móvil



Nota: Fuente propia.

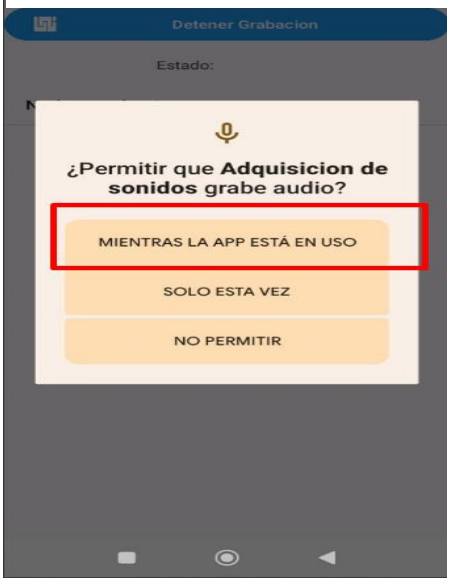
Seleccionamos la opción “instalar” y, al confirmar la, comenzara el proceso de instalación en el telefono. A continuación, se muestran las siguientes imágenes.

Figura 36

Página de presentación de la app

Figura 37

Solicitud de permiso de grabación de audio



Nota: Fuente propia.

Nota: Fuente propia.

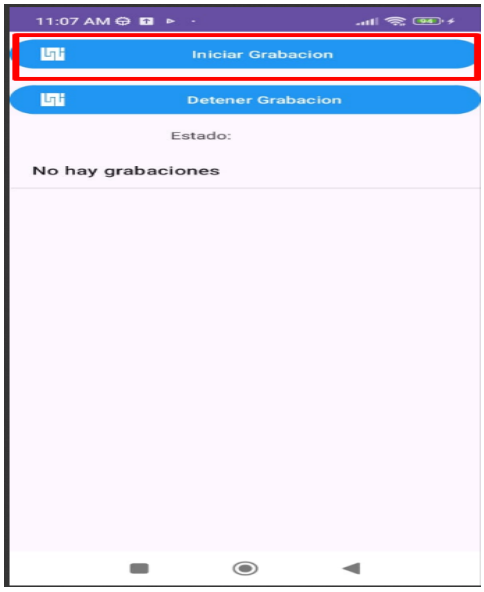
Una vez finalizada la instalación, aparecerá la página de inicio ver figura (24). Luego, se mostrará un cuadro de notificaciones ver figura (25), donde seleccionamos la opción “Mientras la app está en uso” para otorgar los permisos necesarios.

Con los permisos concedidos, la aplicación estará lista para capturar sonidos cardiacos. A continuación, seleccionamos un pequeño grupo de personas y grabamos sus sonidos.

En la siguiente imagen se muestra paso a paso y en tiempo real, el proceso de captura de sonidos. Para iniciar, seleccionamos la opción “iniciar grabación” y la aplicación comenzara automáticamente a capturar los sonidos”

Figura 38

Inicialización de la captura de sonidos cardiacos.

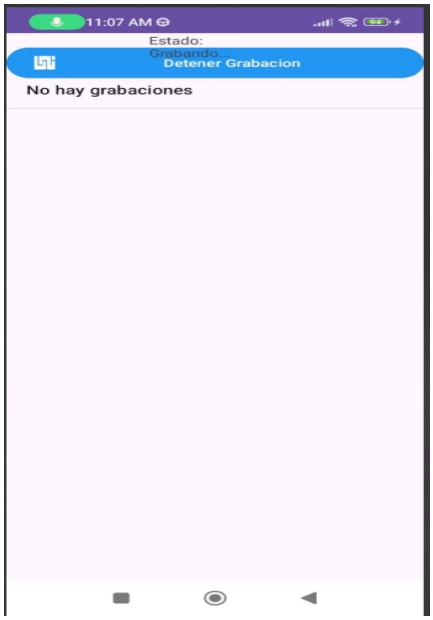


Nota: Fuente propia.

Se procede a iniciar la grabación a continuación.

Figura 39

Grabando sonidos

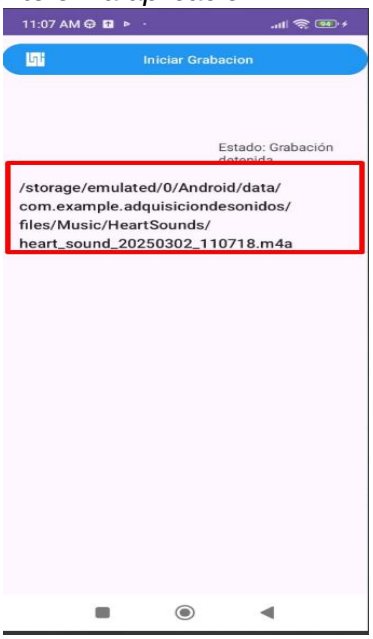


Nota: Fuente propia.

Despues de iniciar la grabación con cualquier paciente, seleccionamos la opción “finalizar grabación” para registrar el sonido en la aplicación. Posteriormente, este podrá ser extraído para la validación del algoritmo desarrollado, a continuación.

Figura 40

Grabación registrada correctamente en la aplicación.



Nota: Fuente propia.

Como se observa en la imagen, la grabación ha sido registrada correctamente en la aplicación, permitiendo almacenar y gestionar los datos de audios capturados. Esta información podrá ser utilizada posteriormente para su análisis y validación dentro del algoritmo desarrollado.

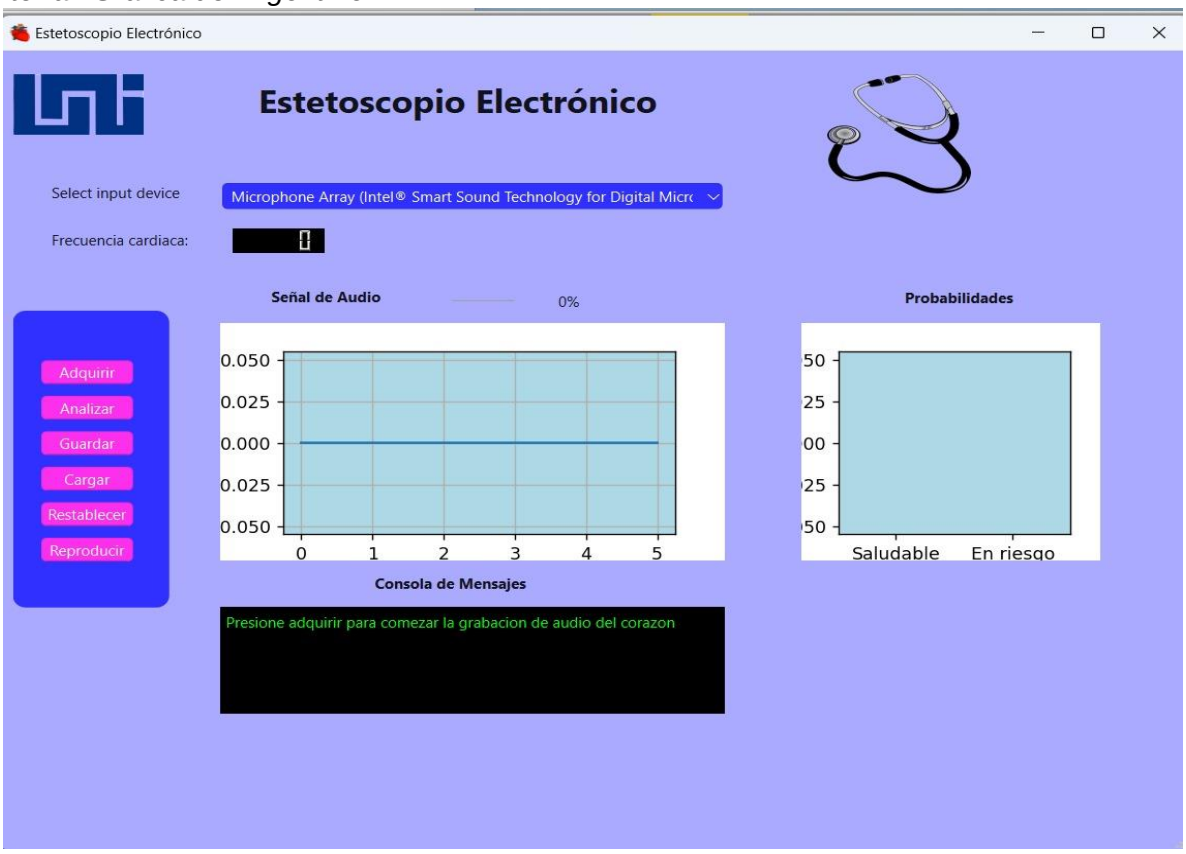
6.7. Resultados finales

Finalmente probamos en conjunto el funcionamiento de nuestra aplicación móvil con nuestro algoritmo entrenado de Machine learning, para ello, conectamos a través de bluetooth o USB el dispositivo móvil en el que este instalada nuestra app con la computadora portátil donde esté trabajando el algoritmo, con el fin de enviarle las muestras recopiladas y que las procese. Una vez recibidas, este último aplica todos los pasos del procesamiento de señales antes descritos en la sección de Metodología, y realiza la clasificación.

Para facilitar la visualización del proceso de clasificación, hemos desarrollado una interfaz gráfica que mejora la comprensión del flujo de trabajo.

Figura 41

Interfaz Gráfica del Algoritmo.



Nota: Fuente propia.

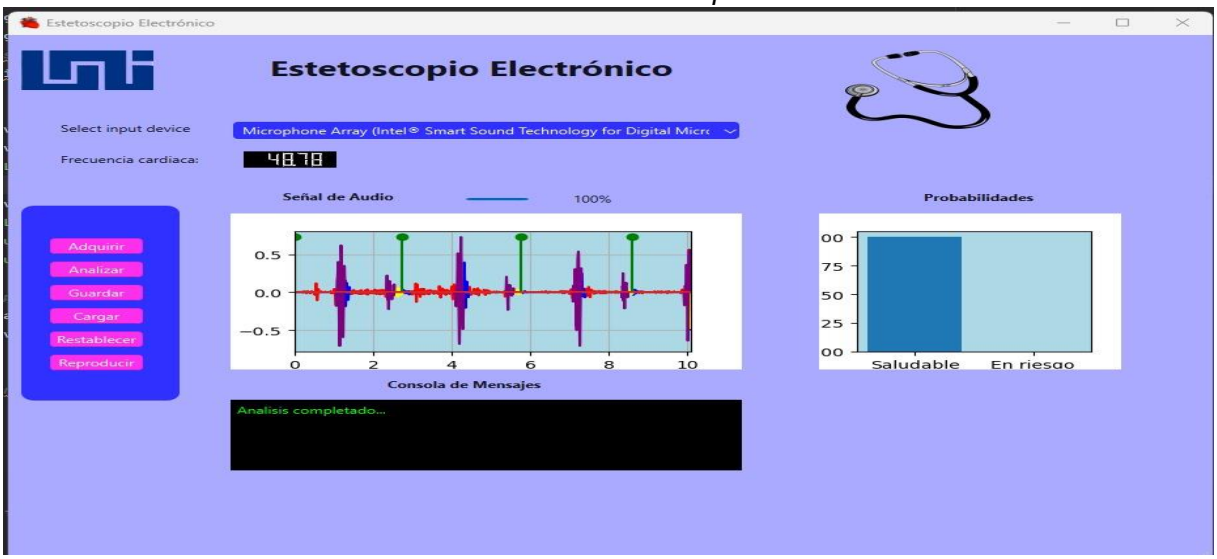
Esta interfaz permite la adquisición de datos provenientes de diversas fuentes, como archivos de audio descargados de Internet, grabaciones enviadas desde la aplicación móvil y registros de sonido capturados directamente por el micrófono de la computadora portátil utilizada en el proceso. Gracias a esta herramienta, se optimiza la interacción con el sistema y se proporciona una visión más clara de cómo se procesan y clasifican los datos acústicos.

Para procesar los audios capturados mediante el micrófono de la computadora portátil, hemos integrado un estetoscopio electrónico. Este dispositivo permite amplificar significativamente el volumen de las grabaciones, mejorando la calidad del sonido y garantizando que las señales de interés sean más claras y precisas. Al utilizar el estetoscopio, conseguimos una mejor captación de los sonidos, lo cual es crucial para asegurar la precisión en los análisis posteriores y la clasificación adecuada de los datos.

Primeramente, realizamos una prueba con algunas muestras de audio grabadas desde la aplicación móvil y la enviamos a través de bluetooth al computador portátil y este fue el resultado.

Figura 42

Resultados Obtenidos de los Audios Grabados de la Aplicación.



Nota: Fuente propia.

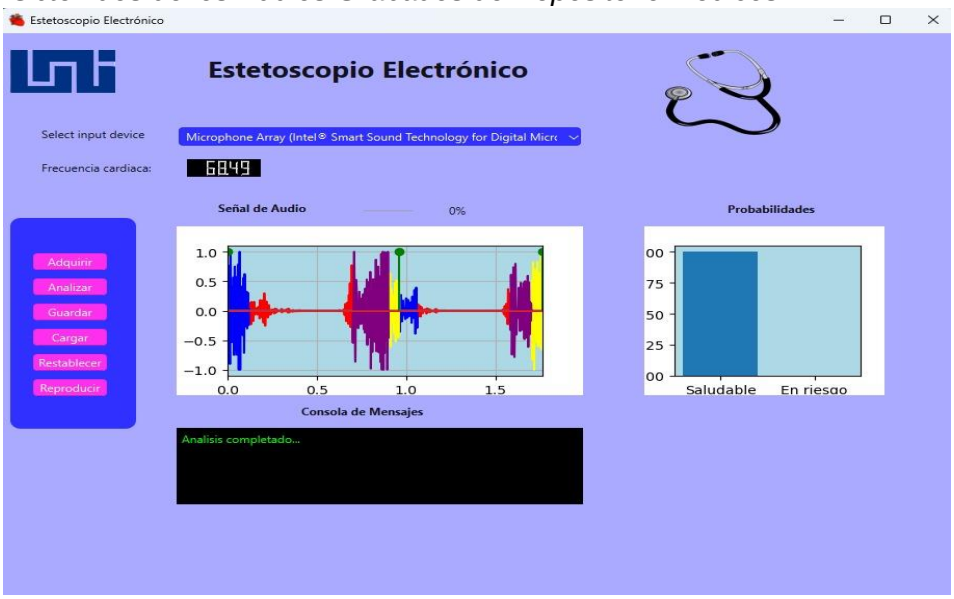
Una vez que las señales son recibidas en el computador de trabajo, iniciamos su procesamiento haciendo clic en el botón de 'Cargar', lo que nos permite abrir uno de los archivos almacenados en el disco. A continuación, procedemos a presionar el botón 'Analizar', y, tras unos minutos, obtenemos la clasificación de la señal. Como se puede observar en la imagen anterior, la muestra de audio procesada se presenta dividida en sus distintos componentes del ciclo cardiaco, incluyendo su frecuencia cardíaca, así como una matriz de probabilidades. En este caso, la matriz indica con alta certeza que la grabación corresponde al grupo de señales de corazones sanos. (La parte azul oscura resalta la tendencia de la probabilidad hacia este grupo).

Para este apartado, fue necesario realizar algunos pasos previos. En primer lugar, tuvimos que convertir el formato de las señales, ya que originalmente son obtenidas de la aplicación en formato *m4a*. Sin embargo, el algoritmo requiere muestras de audio en formato *wav*. Además, fue necesario realizar un re-muestreo a 8000 Hz, ya que las señales captadas por la aplicación tienen una frecuencia de 44,000 Hz, lo que las hace incompatibles para el procesamiento en el sistema.

También se realizaron pruebas con señales provenientes de la base de datos utilizada para entrenar el algoritmo. El proceso es el mismo que en el caso anterior: primero, hacemos clic en el botón 'Cargar' y luego en 'Analizar'. A continuación, el algoritmo procesa la muestra y determina su tendencia. En este caso, el resultado fue el siguiente:

Figura 43

Resultado Obtenidos de los Audios Grabados de Repositorio Médicos



Nota: Fuente propia.

En la imagen anterior, se puede observar que la muestra fue clasificada correctamente dentro del grupo de señales sanas. Para verificar esto, notamos que la señal procesada lleva el nombre 'New_N_001.wav', el cual corresponde a la categoría de señales sanas en la base de datos. En esta base de datos, todos los archivos de señales sanas comienzan con la letra 'N

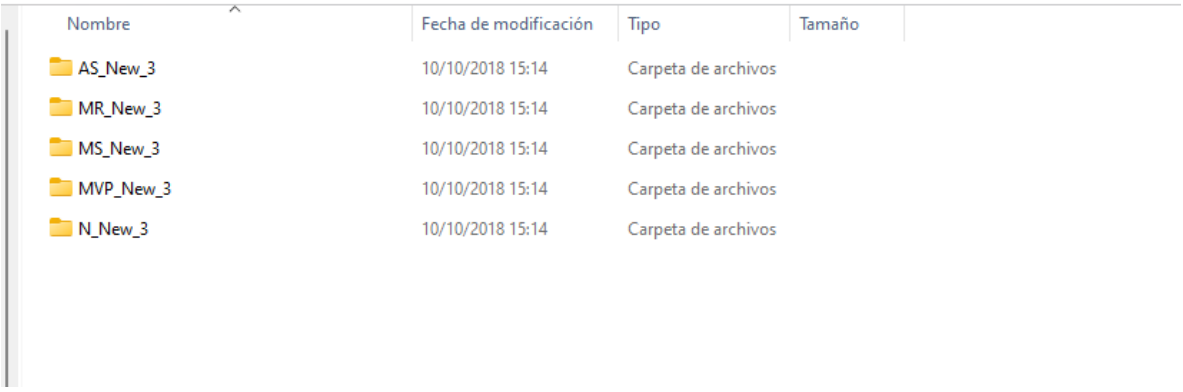
Figura 44

Base de Datos de las Grabaciones Capturadas de la Aplicación.

Nota: Fuente propia.

Figura 45

Categoría de Base de Datos 1.



Nota: Fuente propia.

El algoritmo procesa los sonidos de audios grabados y lo compara con las muestras existente en la base de datos. En este caso, la señal analizada fue clasificada como “sana”, ya que su etiqueta coincide con el formato para almacenar señales cardíacas normales (“N_New_001.wav”).

Se observa la estructura de la base de datos, donde las carpetas organizan las señales según su clasificación. La carpeta “N_New_3” almacena los sonidos cardiacos saludables, lo que confirma que el archivo procesado fue correctamente identificado.

Este resultado demuestra que el algoritmo es capaz de clasificar adecuadamente las señales cardiacas según su categoría, facilitando un diagnóstico preliminar basado en la comparación con una base de datos previamente establecida.

7. CONCLUSIÓN

Se logro implementar un método de procesamiento de sonidos del corazón basado en técnicas de machine learning, el cual permitió identificar anomalías cardiacas con una precisión considerable. Este enfoque demostró ser una herramienta eficaz para apoyar la evaluación de la salud cardiaca a través del análisis automatizado de las señales acústicas del corazón.

Se desarrollo exitosamente una fase de preprocesamiento que incluyo el re-muestreo y la segmentación de los sonidos cardiacos, mejorando la calidad de las señales para su posterior análisis. Esta etapa fue fundamental para estandarizar los datos y facilitar una extracción de característica más eficiente y precisa.

Se lograron extraer características relevantes, como los coeficientes cepstrales en la frecuencia de Mel (MFCC), que ofrecieron una alta capacidad discriminativa entre sonidos anormales y anómalos. Esta extracción fue clave para mejorar el rendimiento de los clasificadores utilizados.

La aplicación de técnicas de clasificación permitió categorizar los sonidos cardíacos con un nivel satisfactorio de exactitud. Los modelos entrenados con los datos extraídos mostraron una buena capacidad para identificar señales con anomalías, validando la eficacia del enfoque supervisado propuesto.

Se desarrollo una aplicación móvil funcional, capaz de adquirir sonidos cardiacos mediante el micrófono del dispositivo. Esta herramienta permitió facilitar la captura directa de datos, ampliando las posibilidades de uso del método propuesto e entornos reales o de bajo costo.

La efectividad del método fue evaluada exitosamente mediante pruebas realizadas con grabaciones provenientes tanto de la aplicación desarrollada como de repositorios médicos. Los resultados confirmaron la viabilidad del enfoque para detectar anomalías cardiacas, demostrando su potencial como herramienta de apoyo en el diagnóstico temprano.

8. RECOMENDACIONES

Actualmente, la aplicación móvil desarrollada en este trabajo permite la adquisición de sonidos cardíacos utilizando el micrófono del dispositivo; sin embargo, el análisis y diagnóstico de dichos sonidos se realiza por separado, ejecutando el algoritmo de procesamiento en un entorno externo programado en Python. Esta división funcional limita la eficiencia del sistema, ya que requiere transferir los archivos manualmente para su procesamiento.

Para futuras versiones del proyecto, se recomienda unificar el proceso de captura y diagnóstico en una sola plataforma móvil, integrando el algoritmo Python directamente con la aplicación Java. Esto permitiría que el usuario obtenga el análisis de su sonido cardíaco inmediatamente después de grabarlo, sin depender de computadoras externas ni pasos intermedios.

8.1. Implementación sugerida:

- Uso de bibliotecas como Chaquopy o Kivy: Permiten integrar código Python dentro de una aplicación Android construida en Java, facilitando la ejecución del modelo de machine learning directamente desde el entorno móvil.
- Conversión del modelo a TensorFlow Lite: Si el algoritmo se puede representar como un modelo entrenado, puede convertirse y cargarse como un archivo. tflite para su uso eficiente y rápido en dispositivos móviles.

8.2. Beneficios esperados:

1. **Procesamiento y diagnóstico en tiempo real**

- El usuario podrá recibir un diagnóstico inmediato justo después de la grabación, sin necesidad de intervención externa.

2. **Mejora en la portabilidad y autonomía**

- La aplicación funcionará de forma independiente, ideal para zonas rurales o sin acceso constante a computadoras o internet.

8.3. Mayor seguridad y privacidad

- Al evitar transferencias de archivos entre dispositivos, se protege mejor la información biomédica del paciente.

3. **Simplificación de la experiencia del usuario**

- Todo el proceso, desde la captura hasta el diagnóstico, ocurrirá en una sola interfaz, lo que facilita su uso incluso por personas sin conocimientos técnicos.

9. BIBLIOGRAFÍA

Ahmed, N. A. (10 de noviembre de 2024). *datacamp*. Obtenido de https://www.datacamp.com/tutorial/what-is-a-confusion-matrix-in-machine-learning

Anonimo. (09 de septiembre de 2010). *Wikipedia (Fotografia)*. Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Coraz%C3%B3n_humano#/media/Archivo:Diagram_of_the_human_heart_(cropped)_es.svg

anonimo. (s.f.). *datacamp*. Obtenido de https://www.datacamp.com/blog/classification-machine-learning

anonimo. (s.f.). *Machine Learning: definición, funcionamiento, usos*. Obtenido de DataScientest: https://datascientest.com/es/machine-learning-definicion-funcionamiento-usos

Ballesteros, P. A. (2009). Anatomía del Corazón. *libro de la salud cardiovascular*, 35.

C., G. A. (s.f.). *TRATADO DE FISIOLOGIA MEDICA.* SEPTIMA EDICION .

Enfermedades Cardiovasculares . (11 de junio de 2021). Obtenido de Organizacion Mundial de la Salud: https://www.who.int/es/news-room/fact-sheets/detail/cardiovascular-diseases-(cvds)

Fernández, A. G. (2014). *DISEÑO DE UN CANAL DE INSTRUMENTACIÓN PARA UN SISTEMA ELECTROCARDIOGRAMA Y UN PULSIOXÍMETRO.*

Fernando, A. (2024). Utilizing fast fourier transform in the processing of biomedical signals: An analytical approach. En A. Fernando, *Actas de la 2a Conferencia Internacional sobre Física Matemática y Simulación Computacional* (pág. 155).

Gonzalo Esteban MosqueraRojas . (2020). Clasificacion de Señales de ECG para la deteccion de enfermedades cardiacas: un estudio comparativo. Bogota D.C, Colombia.

Guyton, A. (1955). EFFECT OF MEAN CIRCULATORY FILLING PRESSURE AND OTHER PERIPHERAL CIRCULATORY FACTORS ON CARDIAC OUTPUT. *AMERICAN JOURNAL OF PHYSIOLOGY*, 3.

Kevin Rojas, C. R. (2013). MODELO DE PROCESAMIENTO DIGITAL DE SEÑALES CARDIACAS DESARROLLADO EN MATLAB. *Telematique* , 21.

Lorenzo Fácila Rubio, C. L.-P. (2024). Nuevas tecnologías para el diagnóstico, tratamiento y seguimiento de las enfermedades cardiovasculares. *Revista Española de Cardiologia* , 88.

Losada, N. (s.f.). *Geo innova*. Obtenido de https://geoinnova.org/blog-territorio/iot-el-internet-de-las-cosas/?gad=1&gclid=CjwKCAjwge2iBhBBEiwAfXDBR1JSEHP50kYylNjvIs0-HXlibCKEb44hGN9UUZwvXwOT1LmyhMV_ChoCxtkQAvD_BwE

M.A.Anusuya, S. (2011). Comparison of Different Speech Feature Extraction Techniques with and without Wavelet Transform to Kannada Speech Recognition. *International Journal of Computer Applications* .

M.latarjet, & Liard, A. R. (s.f.). *Anatomia Humana.* Tercera edicion Volumen II .

Mark Miller, M. D. (2024). World Heart Federation. *World Heart Report 2024*, pág. 3. Obtenido de World Heart Federation.

Nieto, N., & Laura, V. M. (2017). Diseño de un prototipo de medición de medicion de señales fisiologica utilizadas en Biofeedback. 139.

Ortigosa, J., E. Reigal, R., Carranque, G., & Hernández-Mendo, A. (2018). VARIABILIDAD DE LA FRECUENCIA CARDÍACA: INVESTIGACIÓN Y APLICACIONES PRACTICAS PARA EL CONTROL DE LOS PROCESOS ADAPTIVOS EN EL DEPORTE. *Revista Iberoamericana de Psicología del*, 122.

Paz Campos, L. F., & Herrera Velasquez, L. I. (2017). Diseño y construcción de un sistema que simule las señales. 2.

Rojas, G. E. (2020). *clasificacion de señales ecg para la deteccion de enfermedades cardiacas: un estudio comparativo.* Bogota D.C, Colombia.

S E Schmidt, C. H.-H. (2010). Segmentation of heart sound recordings by a duration-dependent hidden Markov model. *IOPSCience*.

Salud, O. M. (11 de junio de 2021). *Organizacion Mundial de la Salud*. Obtenido de Organizacion Mundial de la Salud: https://www.who.int/es/news-room/fact-sheets/detail/cardiovascular-diseases-(cvds)

Samit Kumar Ghosh, P. R. (2020). Heart Sound Data Acquisition and Preprocessing Techniques. *ResearchGate*.

Trevor Hastie, R. T. (2008). *The Elements of Statistical Learning Data Mining, Inference, and Prediction.* California: Springer.

uday. (1 de mayo de 2025). *MFCC Technique for Speech Recognition*. Obtenido de Analytics: https://www.analyticsvidhya.com/blog/2021/06/mfcc-technique-for-speech-recognition/

Uribe, J. I. (2019). *Clasificación del audio cardiaco mediante representacion escasa de señales y aprendizaje automatico.* Mexico.

Yaseen, G.-y. s. (2018). Classification of Heart Sound Signal Using Multiple Features. *applied Sciences*, 6.

Zelada, C. (10 de mayo de 2017). *RPubs by RStudio (Fotografia)*. Obtenido de Evaluacion de modelos de clasificacion: https://rpubs.com/chzelada/275494

10. ANEXOS

10.1. Código fuente de la aplicación

```
package com.example.adquisiciondesonidos;

import android.Manifest;
import android.content.Context;
import android.content.pm.PackageManager;
import android.media.MediaPlayer;
import android.media.MediaRecorder;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.Toast;

import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import androidx.core.content.ContextCompat;

import java.io.File;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Locale;
```

```
public class MainActivity extends AppCompatActivity {

    private Button btnStartRecording, btnStopRecording;
    private TextView tvStatus;
    private ListView lvRecordings;
    private MediaRecorder mediaRecorder;
    private String outputFile = null;
    private File storageDir;

    private static final int REQUEST_PERMISSIONS = 200;
    private String[] permissions = {
            Manifest.permission.RECORD_AUDIO,
            Manifest.permission.WRITE_EXTERNAL_STORAGE,
            Manifest.permission.READ_EXTERNAL_STORAGE
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Inicializar vistas
        btnStartRecording = findViewById(R.id.btnStartRecording);
        btnStopRecording = findViewById(R.id.btnStopRecording);
        tvStatus = findViewById(R.id.tvStatus);
        lvRecordings = findViewById(R.id.lvRecordings);

        // Verificar y solicitar permisos
        if (ContextCompat.checkSelfPermission(getApplicationContext(),
Manifest.permission.WRITE_EXTERNAL_STORAGE) !=
PackageManager.PERMISSION_GRANTED &&
```

```
ActivityCompat.checkSelfPermission(getApplicationContext(),
Manifest.permission.RECORD_AUDIO) !=
PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(MainActivity.this, new
String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE,
Manifest.permission.RECORD_AUDIO}, 1000);
    }

    // Configurar directorio de almacenamiento
    storageDir = new File(getExternalFilesDir(Environment.DIRECTORY_MUSIC),
"HeartSounds");
    if (!storageDir.exists()) {
        storageDir.mkdirs();
    }

    // Configurar listeners para los botones
    btnStartRecording.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            startRecording();
        }
    });

    btnStopRecording.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            stopRecording();
        }
    });

    // Cargar grabaciones existentes
```

```
        loadRecordings();

        // Manejar clics en la lista para reproducir audio
        lvRecordings.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view, int position, long
id) {
                Object item = parent.getItemAtPosition(position);
                if (item instanceof File) {
                    File selectedFile = (File) item;
                    playAudio(selectedFile);
                }
            }
        });
    }

    //Verifica si la aplicación tiene los permisos necesarios

    /*private boolean hasPermissions(Context context, String... permissions) {
        for (String permission : permissions) {
            if (ContextCompat.checkSelfPermission(context, permission) !=
PackageManager.PERMISSION_GRANTED) {
                return false;
            }
        }
        return true;
    }*/

    private void startRecording() {
        String timeStamp = new SimpleDateFormat("yyyyMMdd_HHmmss",
Locale.getDefault()).format(new Date());
```

```
        String fileName = "heart_sound_" + timeStamp + ".m4a";

        File audioFile = new File(storageDir, fileName);
        outputFile = audioFile.getAbsolutePath();

        mediaRecorder = new MediaRecorder();
        mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
        mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
        mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
        mediaRecorder.setAudioChannels(1);
        mediaRecorder.setAudioSamplingRate(44100);
        mediaRecorder.setAudioEncodingBitRate(192000);
        mediaRecorder.setOutputFile(outputFile);

        try {
            mediaRecorder.prepare();
            mediaRecorder.start();
            tvStatus.setText("Estado: Grabando...");
            btnStartRecording.setVisibility(View.GONE);
            btnStopRecording.setVisibility(View.VISIBLE);
        } catch (IOException e) {
            e.printStackTrace();
            Toast.makeText(this, "Error al iniciar la grabación.",
Toast.LENGTH_SHORT).show();
        } catch (IllegalStateException e) {
            e.printStackTrace();
            Toast.makeText(this, "Error en el estado de la grabación.",
Toast.LENGTH_SHORT).show();
        }
    }
```

```
    private void stopRecording() {
        try {
            mediaRecorder.stop();
            mediaRecorder.release();
            mediaRecorder = null;

            tvStatus.setText("Estado: Grabación detenida.");
            btnStartRecording.setVisibility(View.VISIBLE);
            btnStopRecording.setVisibility(View.GONE);

            // Actualizar lista de grabaciones
            loadRecordings();
        } catch (RuntimeException stopException) {
            // Manejar excepciones si se detiene la grabación antes de iniciarla
correctamente
            stopException.printStackTrace();
            Toast.makeText(this, "Error al detener la grabación.",
Toast.LENGTH_SHORT).show();
        }
    }

    //Carga las grabaciones almacenadas y las muestra en la lista.

    private void loadRecordings() {
        File[] files = storageDir.listFiles();
        if (files != null && files.length > 0) {
            ArrayList<File> fileList = new ArrayList<>();
            for (File file : files) {
                if (file.isFile()) {
                    fileList.add(file);
```

```
            }
        }
        ArrayAdapter<File> adapter = new ArrayAdapter<>(this,
android.R.layout.simple_list_item_1, fileList);
        lvRecordings.setAdapter(adapter);
    } else {
        ArrayList<String> noFiles = new ArrayList<>();
        noFiles.add("No hay grabaciones");
        ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
android.R.layout.simple_list_item_1, noFiles);
        lvRecordings.setAdapter(adapter);
    }
}
//Reproduce el audio seleccionado.

private void playAudio(File audioFile) {
    MediaPlayer mediaPlayer = new MediaPlayer();
    try {
        mediaPlayer.setDataSource(audioFile.getAbsolutePath());
        mediaPlayer.prepare();
        mediaPlayer.start();
        tvStatus.setText("Reproduciendo: " + audioFile.getName());

        // Actualizar el estado al finalizar la reproducción
        mediaPlayer.setOnCompletionListener(new
MediaPlayer.OnCompletionListener() {
            @Override
            public void onCompletion(MediaPlayer mp) {
                tvStatus.setText("Reproducción finalizada.");
                mp.release();
            }
```

```
            });

        } catch (IOException e) {
            e.printStackTrace();
            Toast.makeText(this, "Error al reproducir el audio.",
Toast.LENGTH_SHORT).show();
        }
    }

    //Maneja la respuesta de la solicitud de permisos.

    @Override
    public void onRequestPermissionsResult(int requestCode, String[] permissions,
int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (requestCode == REQUEST_PERMISSIONS) {
            boolean allGranted = true;
            for (int result : grantResults) {
                if (result != PackageManager.PERMISSION_GRANTED) {
                    allGranted = false;
                    break;
                }
            }
            if (!allGranted) {
                Toast.makeText(this, "Permisos necesarios no otorgados.",
Toast.LENGTH_SHORT).show();
                btnStartRecording.setEnabled(false);
            }
        }
    }
}
```

10.2. codigo de solicitud de permisos del micrófono y ajuste página de inicio de la aplicación

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.RECORD_AUDIO" />
    <uses-permission
        android:name="android.permission.WRITE_EXTERNAL_STORAGE"
        android:maxSdkVersion="28" />
    <uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE" />

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.AdquisicionDeSonidos"
        tools:targetApi="31">
        <activity
            android:name=".SplashscreenActivity"
            android:configChanges="orientation|keyboardHidden|screenSize"
            android:exported="true"
            android:label="@string/title_activity_splashscreen"
            android:theme="@style/Theme.AdquisicionDeSonidos.Fullscreen">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
```

```
            </intent-filter>
        </activity>
    </application>

</manifest>
```

10.3. Codigo de splash screen activity java

```
package com.example.adquisiciondesonidos;

import android.annotation.SuppressLint;

import androidx.appcompat.app.ActionBar;
import androidx.appcompat.app.AppCompatActivity;

import android.content.Intent;
import android.os.Build;
import android.os.Bundle;
import android.os.Handler;
import android.os.Looper;
import android.view.MotionEvent;
import android.view.View;
import android.view.WindowInsets;

import com.example.adquisiciondesonidos.databinding.ActivitySplashscreenBinding;

/**
 * An example full-screen activity that shows and hides the system UI (i.e.
 * status bar and navigation/system bar) with user interaction.
 */
public class SplashscreenActivity extends AppCompatActivity {
    /**
     * Whether or not the system UI should be auto-hidden after
     * {@link #AUTO_HIDE_DELAY_MILLIS} milliseconds.
     */
    private static final boolean AUTO_HIDE = true;

    /**
     * If {@link #AUTO_HIDE} is set, the number of milliseconds to wait after
     * user interaction before hiding the system UI.
     */
    private static final int AUTO_HIDE_DELAY_MILLIS = 3000;

    /**
```

```
    * Some older devices needs a small delay between UI widget updates
    * and a change of the status and navigation bar.
    */
   private static final int UI_ANIMATION_DELAY = 300;
   private final Handler mHideHandler = new Handler(Looper.myLooper());
   private View mContentView;
   private final Runnable mHidePart2Runnable = new Runnable() {
      @SuppressLint("InlinedApi")
      @Override
      public void run() {
         /* Delayed removal of status and navigation bar
         if (Build.VERSION.SDK_INT >= 30) {
            mContentView.getWindowInsetsController().hide(
                  WindowInsets.Type.statusBars() |
WindowInsets.Type.navigationBars());
         } else {
            // Note that some of these constants are new as of API 16 (Jelly Bean)
            // and API 19 (KitKat). It is safe to use them, as they are inlined
            // at compile-time and do nothing on earlier devices.

mContentView.setSystemUiVisibility(View.SYSTEM_UI_FLAG_LOW_PROFILE
                  | View.SYSTEM_UI_FLAG_FULLSCREEN
                  | View.SYSTEM_UI_FLAG_LAYOUT_STABLE
                  | View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY
                  | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION
                  | View.SYSTEM_UI_FLAG_HIDE_NAVIGATION); */

      }
   };
   private View mControlsView;
   private final Runnable mShowPart2Runnable = new Runnable() {
      @Override
      public void run() {
         /* Delayed display of UI elements
         ActionBar actionBar = getSupportActionBar();
         if (actionBar != null) {
            actionBar.show();
         }
         mControlsView.setVisibility(View.VISIBLE);*/
      }
   };
   private boolean mVisible;
   private final Runnable mHideRunnable = new Runnable() {
```

```
        @Override
        public void run() {
            hide();
        }
    };
    /**
     * Touch listener to use for in-layout UI controls to delay hiding the
     * system UI. This is to prevent the jarring behavior of controls going away
     * while interacting with activity UI.
     */
    private final View.OnTouchListener mDelayHideTouchListener = new
View.OnTouchListener() {
        @Override
        public boolean onTouch(View view, MotionEvent motionEvent) {
            switch (motionEvent.getAction()) {
                case MotionEvent.ACTION_DOWN:
                    if (AUTO_HIDE) {
                        delayedHide(AUTO_HIDE_DELAY_MILLIS);
                    }
                    break;
                case MotionEvent.ACTION_UP:
                    view.performClick();
                    break;
                default:
                    break;
            }
            return false;
        }
    };
    private ActivitySplashscreenBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        binding = ActivitySplashscreenBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());

        Thread myThread = new Thread (){
            @Override
            public void run (){
                try{
                    sleep(2000);
```

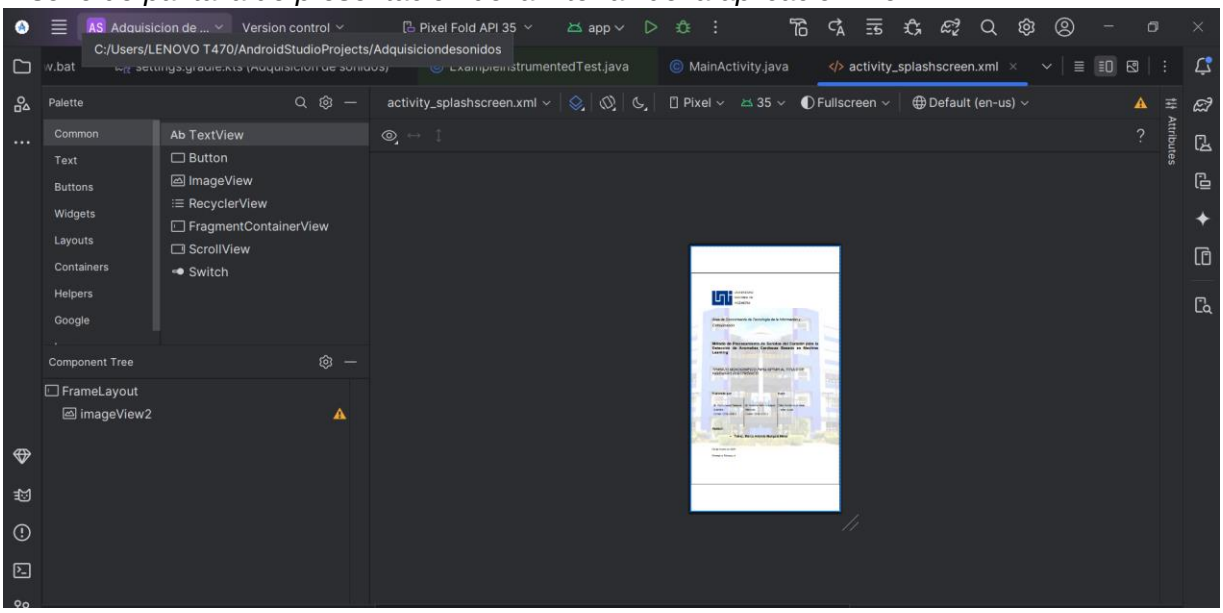
```
            Intent Intent = new Intent(getApplicationContext(), MainActivity.class);
            startActivity(Intent);
        }catch (InterruptedException e){
            e.printStackTrace();
        }
    }

};;

myThread.start();

 /* mVisible = true;
  mControlsView = binding.fullscreenContentControls;
  mContentView = binding.fullscreenContent;

  // Set up the user interaction to manually show or hide the system UI.
  mContentView.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View view) {
          toggle();
      }
  });

  // Upon interacting with UI controls, delay any scheduled hide()
  // operations to prevent the jarring behavior of controls going away
  // while interacting with the UI.
  binding.dummyButton.setOnTouchListener(mDelayHideTouchListener); */
}

@Override
protected void onPostCreate(Bundle savedInstanceState) {
  super.onPostCreate(savedInstanceState);

  // Trigger the initial hide() shortly after the activity has been
  // created, to briefly hint to the user that UI controls
  // are available.
  delayedHide(100);
}

private void toggle() {
  if (mVisible) {
      hide();
  } else {
```

```
            show();
        }
    }

    private void hide() {
        // Hide UI first
        ActionBar actionBar = getSupportActionBar();
        if (actionBar != null) {
            actionBar.hide();
        }
       // mControlsView.setVisibility(View.GONE);
        mVisible = false;

        // Schedule a runnable to remove the status and navigation bar after a delay
        mHideHandler.removeCallbacks(mShowPart2Runnable);
        mHideHandler.postDelayed(mHidePart2Runnable, UI_ANIMATION_DELAY);
    }

    private void show() {
        // Show the system bar
        if (Build.VERSION.SDK_INT >= 30) {
            mContentView.getWindowInsetsController().show(
                    WindowInsets.Type.statusBars() | WindowInsets.Type.navigationBars());
        } else {

mContentView.setSystemUiVisibility(View.SYSTEM_UI_FLAG_LAYOUT_FULLSCRE
EN
                    | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION);
        }
        mVisible = true;

        // Schedule a runnable to display UI elements after a delay
        mHideHandler.removeCallbacks(mHidePart2Runnable);
        mHideHandler.postDelayed(mShowPart2Runnable, UI_ANIMATION_DELAY);
    }

    /**
     * Schedules a call to hide() in delay milliseconds, canceling any
     * previously scheduled calls.
     */
    private void delayedHide(int delayMillis) {
        mHideHandler.removeCallbacks(mHideRunnable);
        mHideHandler.postDelayed(mHideRunnable, delayMillis);
```

```
    }
}
```

Figura 46

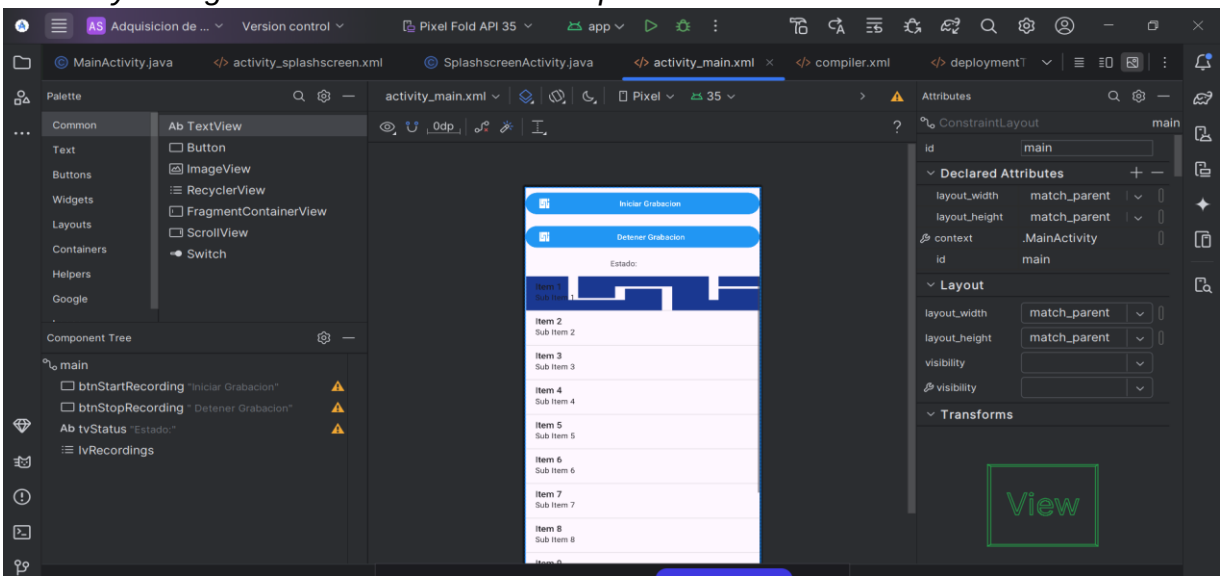
Diseñó de pantalla de presentación de la interfaz de la aplicación móvil



Nota: Fuente propia.

Figura 47

Diseño y configuración de botones de la aplicación móvil.



Nota: Fuente propia.

10.4. Código del algoritmo

10.4.1. Procesamiento y clasificación

```
# calculando MFCC
import numpy as np
import librosa
import librosa.display
import hmm_segmentation as hs
import pywt
def getmsWtFeat(signal, winsize, wininc, Fs, wavelet_type, levels=5):
    # specify number of samples
    signal_size = len(signal)
    # Number of features
    Nf = 5
    # based on the number of samples, winsize, and wininc, how many windows we will have? this is "numwin"
    numwin = int(np.floor(signal_size - winsize) / wininc + 1)
    # predefine zeros matrix to allocate  memory   for output features
    feature_out = np.zeros(shape=(Nf, levels))
    feat = np.zeros(shape=(winsize, numwin))
    st = 0
    en = winsize
    for i in range(numwin):
        feat[0:winsize, i] = signal[st:en] - np.mean(signal[st:en])
        st = st + wininc
        en = en + wininc
    dec = pywt.wavedec(feat, wavelet_type, 'zero', level=levels, axis=0)
    # prepare arrays
    for i in range(levels):
        c = dec[i]
```

```
        E = c**2
        Etot = np.sum(c**2)
        #features
        feature_out[0, i]  = np.mean(np.sum(c**2, axis=1))
        feature_out[1, i]  = np.mean(np.var(c,axis=1))  #variance of coefficients
        feature_out[2, i]  = np.mean(np.std(c, axis=1)) #standard deviation of coefficients
        feature_out[3, i]  = np.sum(np.abs(np.diff(c, axis=1)))# waveform  length
        prob = E/Etot
        feature_out[4, i] = -np.sum(prob * np.log(prob))  # Shanon Entropy
    return np.log(np.ravel(feature_out) )


def getMFCCFeat(signal, Fs, n_features=12, n_fft=60, hop_length=20):
    mfccs = librosa.feature.mfcc(y=signal, n_mfcc=n_features, sr=Fs, n_fft=n_fft,
hop_length=hop_length)
    return mfccs
def calculateFeatures(signal,  Fs, mfcc_features=12,
winsize=300,wininc=100,wavelet_type='db5',w_levels=5):
   # mel frequency features
    mfccs = getMFCCFeat(signal, n_features=mfcc_features, Fs=Fs)
    m = np.mean(mfccs, axis=1)
    # wavelets features
    wavecoeff = getmsWtFeat(signal, winsize, wininc, Fs, wavelet_type,
levels=w_levels)
    featureArray = np.hstack((m, wavecoeff))
    return featureArray
def calcularPeriodos(states):
    puntos_de_cambio= np.where(np.diff(states)!=0)
    periodos= list()
    for i in range(len(puntos_de_cambio[0])):
```

```
        xi= puntos_de_cambio[0][i]
        if states[xi]==2.0:
            periodos.append(xi)
    return periodos

import numpy as np
import wave
import librosa as li
import scipy as sp
import matplotlib.pyplot as plt
from scipy.io import wavfile
from scipy.stats import multivariate_normal
import ctypes

def normalise_signal(signal):
    """
    This function subtracts the mean and divides by the standard deviation of
    a (1D) signal in order to normalise it for machine learning applications.

    Inputs:
    signal: the original signal

    Outputs:
    normalised_signal: the original signal, minus the mean and divided by
    the standard deviation.

    Developed by David Springer for the paper:
    D. Springer et al., ?Logistic Regression-HSMM-based Heart Sound
```

```
    Segmentation,? IEEE Trans. Biomed. Eng., In Press, 2015.


    ported to Python by Pablo Vásquez
    """
    mean_of_signal = np.mean(signal)
    standard_deviation = np.std(signal)
    normalised_signal = (signal - mean_of_signal) / standard_deviation
    return normalised_signal


def Homomorphic_Envelope_with_Hilbert(input_signal, sampling_frequency,
lpf_frequency=8, figures=False):
    """
    This function finds the homomorphic envelope of a signal, using the method
described in the  following publications:


    S. E. Schmidt et al., ?Segmentation of heart sound recordings by a  duration-
dependent hidden Markov model.,? Physiol. Meas., vol. 31, no. 4,
    pp. 513?29, Apr. 2010.


    C. Gupta et al., ?Neural network classification of homomorphic segmented  heart
sounds,? Appl. Soft Comput., vol. 7, no. 1, pp. 286?297, Jan. 2007.


    D. Gill et al., ?Detection and identification of heart sounds using  homomorphic
envelogram and self-organizing probabilistic model,? in  Computers in Cardiology,
2005, pp. 957?960.
    (However, these researchers found the homomorphic envelope of shannon
energy.)


    In I. Rezek and S. Roberts, ?Envelope Extraction via Complex Homomorphic
Filtering. Technical Report TR-98-9,? London, 1998, the researchers state  that the
singularity at 0 when using
```

the natural logarithm (resulting in values of -inf) can be fixed by using a complex valued signal. They motivate the use of the Hilbert transform to find the analytic signal, which is a

conversion of a real-valued signal to a complex-valued signal, which is unaffected by the singularity.

A zero-phase low-pass Butterworth filter is used to extract the envelope.

Inputs:

input_signal: the original signal (1D) signal

samplingFrequency: the signal's sampling frequency (Hz)

lpf_frequency: the frequency cut-off of the low-pass filter to be used in the envelope extraction (Default = 8 Hz as in Schmidt's publication).

figures: (optional) boolean variable dictating the display of a figure of both the original signal and the extracted envelope:

Outputs:

homomorphic_envelope: The homomorphic envelope of the original signal (not normalised).

This code was developed by David Springer for comparison purposes in the paper: D. Springer et al., ?Logistic Regression-HSMM-based Heart Sound

Segmentation,? IEEE Trans. Biomed. Eng., In Press, 2015.

Ported to Python by Pablo Vásquez

"""

```
# 8Hz 1st order Butterworth lpf_frequency
hilbert_signal = np.abs(sp.signal.hilbert(input_signal))
b, a = sp.signal.iirfilter(1, 2 * np.pi * lpf_frequency, btype="low", ftype="butter", output='ba', fs=sampling_frequency)
homomorphic_envelope = np.exp(sp.signal.filtfilt(b, a, np.log(hilbert_signal)))
```

```
    # Remove spurious spikes in first sample
    homomorphic_envelope[0] = homomorphic_envelope[1]
    if figures:
        plt.figure()
        plt.plot(input_signal, label="input signal")
        plt.plot(homomorphic_envelope, label="Homomorphic envelope")
        plt.legend(loc='best')
        plt.xlim([0, 2000])
        plt.show()
    return homomorphic_envelope
def default_Schmidt_HSMM_options():
    schmidt_options = dict()  # The sampling frequency at which to extract signal features:
    schmidt_options["audio_Fs"] = 1000
    # The downsampled frequency.Set to 50 in Schmidt paper
    schmidt_options["audio_segmentation_Fs"] = 50
    # Tolerance for S1 and S2 localization
    schmidt_options["segmentation_tolerance"] = 0.1  # seconds
    #  Whether to use the mex code or not:
    schmidt_options["use_mex"] = False
    return schmidt_options


def wavread(signal):
    PCG, Fs = li.load(signal, sr=None)
    w = wave.open(signal, 'rb')
    nbits = w.getsampwidth()
    return [PCG, Fs, nbits]
def resample(PCG, Fs, Fs1):
```

```
    # Resample data
    pcgr = li.resample(PCG, orig_sr=Fs1, target_sr=Fs)
    return pcgr
def getSchmidtPCGFeatures(audio_data, Fs, figures=False):
    """
    Get the features used in the Schmidt segmentation algorithm. .
    INPUTS:
    audio_data: array of data from which to extract features
    Fs: the sampling frequency of the audio data
    figures (optional): boolean variable dictating the display of figures
    OUTPUTS:
    PCG_Features: array of derived features
    featuresFs: the sampling frequency of the derived features. This is set
    in default_Schmidt_HSMM_options.m

    This code is derived from the paper:
    S. E. Schmidt et al., "Segmentation of heart sound recordings by a
    duration-dependent hidden Markov model," Physiol. Meas., vol. 31,
    no. 4, pp. 513-29, Apr. 2010.
    Developed by David Springer for comparison purposes in the paper:

    D. Springer et al., ?Logistic Regression-HSMM-based Heart Sound
    Segmentation,? IEEE Trans. Biomed. Eng., In Press, 2015.

    Ported to Python by Pablo Vásquez
    """
    schmidt_options = default_Schmidt_HSMM_options()
    # Downsampled feature sampling frequency
```

```
    featuresFs = schmidt_options["audio_segmentation_Fs"]

    # 25-400Hz 4th order Butterworth band pass
    audio_data = butterworth_low_pass_filter(audio_data, 2, 400, Fs, False)
    audio_data = butterworth_high_pass_filter(audio_data, 2, 25, Fs, False)

    # Spike removal from the original paper:
    audio_data = schmidt_spike_removal(audio_data, Fs)

    # Find the homomorphic envelope
    homomorphic_envelope = Homomorphic_Envelope_with_Hilbert(audio_data, Fs)

    # Down sample the envelope:
    downsampled_homomorphic_envelope = resample(homomorphic_envelope,
featuresFs, Fs)

    # normalise the envelope:
    PCG_Features = normalise_signal(downsampled_homomorphic_envelope)

    # Plotting figures
    if figures:
        fig = plt.figure()
        ax1 = fig.add_subplot(1, 2, 1)
        fig.suptitle('PCG Features')
        t1 = np.linspace(1, len(audio_data), len(audio_data)) / Fs
        ax1.plot(t1, audio_data, color="black")
        ax1.set_title("Original signal")
        ax2 = fig.add_subplot(1, 2, 2)
        ax2.set_title("Derived features")
```

```
        t2 = np.linspace(1, len(PCG_Features), len(PCG_Features)) / featuresFs
        ax2.plot(t2, PCG_Features, color="red", label="Derived features")

    return PCG_Features, featuresFs
def expand_qt(original_qt, old_fs, new_fs, new_length):
    """
    Function to expand the derived HMM states to a higher sampling frequency.
    Developed by David Springer for comparison purposes in the paper:
    D. Springer et al., "Logistic Regression-HSMM-based Heart Sound
    Segmentation," IEEE Trans. Biomed. Eng., In Press, 2015.

    INPUTS:
    original_qt: the original derived states from the HMM
    old_fs: the old sampling frequency of the original_qt
    new_fs: the desired sampling frequency
    new_length: the desired length of the qt signal

    Outputs:
    expanded_qt: the expanded qt, to the new FS and length
    Copyright (C) 2016  David Springer
    dave.springer@gmail.com
    Ported to Python by Pablo Vásquez
    """

    original_qt = original_qt.flatten()
    expanded_qt = np.zeros(shape=(new_length))
    diff = np.where(np.diff(original_qt))
    indeces_of_changes = np.array(diff)
```

```
    indeces_of_changes= np.reshape(indeces_of_changes,
(indeces_of_changes.shape[1], ))
    indeces_of_changes = np.hstack((indeces_of_changes, len(original_qt)))
    start_index = 0
    for i in range(len(indeces_of_changes)):
        end_index = indeces_of_changes[i]
        mid_point = int(np.round((end_index - start_index) / 2) + start_index)
        value_at_mid_point = original_qt[mid_point]
        expanded_start_index = int(np.round((start_index / old_fs) * new_fs))
        expanded_end_index = int(np.round((end_index / old_fs) * new_fs))
        if expanded_end_index > new_length:
            expanded_end_index = new_length
        expanded_qt[expanded_start_index:expanded_end_index] =
value_at_mid_point
        start_index = end_index
    return expanded_qt
def butterworth_low_pass_filter(original_signal, order, cutoff, sampling_frequency,
figures=False):
    """
    Low-pass filter a given signal using a forward-backward, zero-phase
    butterworth low-pass filter.

    INPUTS:
    original_signal: The 1D signal to be filtered
    order: The order of the filter (1,2,3,4 etc). NOTE: This order is
    effectively doubled as this function uses a forward-backward filter that
    ensures zero phase distortion
    cutoff: The frequency cutoff for the low-pass filter (in Hz)
    sampling_frequency: The sampling frequency of the signal being filtered
```

```
    (in Hz).
    figures (optional): boolean variable dictating the display of figures

    OUTPUTS:
    low_pass_filtered_signal: the low-pass filtered signal.

    This code is derived from the paper:
    S. E. Schmidt et al., "Segmentation of heart sound recordings by a
    duration-dependent hidden Markov model," Physiol. Meas., vol. 31,
    no. 4, pp. 513-29, Apr. 2010.

    Developed by David Springer for comparison purposes in the paper:
    D. Springer et al., ?Logistic Regression-HSMM-based Heart Sound
    Segmentation,? IEEE Trans. Biomed. Eng., In Press, 2015.
    Ported to Python by Pablo Vásquez
    """
    # Get the butterworth filter coefficients
    [B_low, A_low] = sp.signal.iirfilter(order, cutoff, btype="low", ftype="butter",
output='ba', fs=sampling_frequency)
    if figures:
        w, h = sp.signal.freqs(b, a, 4000)
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        ax.semilogx(w / (2 * np.pi), 20 * np.log10(np.maximum(abs(h), 1e-5)))
        ax.set_title('Low-pass filter frequency response')
        ax.set_xlabel('Frequency [Hz]')
        ax.set_ylabel('Amplitude [dB]')
        ax.axis((10, 1000, -100, 10))
        ax.grid(which='both', axis='both')
```

```
        plt.show()
        # Forward-backward filter the original signal using the butterworth
    # coefficients, ensuring zero phase distortion
    low_pass_filtered_signal = sp.signal.filtfilt(B_low, A_low, original_signal)
    if figures:
        plt.figure()
        plt.title('Original vs. low-pass filtered signal')
        plt.plot(original_signal, color='blue', label='Original Signal')
        plt.plot(low_pass_filtered_signal, color='red', label='Low-pass filtered signal')
        plt.legend(loc='best')
    return low_pass_filtered_signal
def butterworth_high_pass_filter(original_signal, order, cutoff, sampling_frequency,
figures=False):
    """
    High-pass filter a given signal using a forward-backward, zero-phase
    butterworth filter.

    INPUTS:
    original_signal: The 1D signal to be filtered
    order: The order of the filter (1,2,3,4 etc). NOTE: This order is
    effectively doubled as this function uses a forward-backward filter that
    ensures zero phase distortion
    cutoff: The frequency cutoff for the high-pass filter (in Hz)
    sampling_frequency: The sampling frequency of the signal being filtered
    (in Hz).
    figures (optional): boolean variable dictating the display of figures
    OUTPUTS:
    high_pass_filtered_signal: the high-pass filtered signal.
```

```
    This code is derived from the paper:
    S. E. Schmidt et al., "Segmentation of heart sound recordings by a
    duration-dependent hidden Markov model," Physiol. Meas., vol. 31,
    no. 4, pp. 513-29, Apr. 2010.

    Developed by David Springer for comparison purposes in the paper:
    D. Springer et al., ?Logistic Regression-HSMM-based Heart Sound
    Segmentation,? IEEE Trans. Biomed. Eng., In Press, 2015.

    Ported to Python by Pablo Vásquez
    """
    # Get the butterworth filter coefficients

    [B_high, A_high] = sp.signal.iirfilter(order, cutoff, btype="high", ftype="butter",
output='ba',
                                           fs=sampling_frequency)
    # Forward-backward filter the original signal using the butterworth coefficients,
ensuring zero phase distortion
    high_pass_filtered_signal = sp.signal.filtfilt(B_high, A_high, original_signal)
    if figures:
        w, h = sp.signal.freqs(b, a, 4000)
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        ax.semilogx(w / (2 * np.pi), 20 * np.log10(np.maximum(abs(h), 1e-5)))
        ax.set_title('High-pass filter frequency response')
        ax.set_xlabel('Frequency [Hz]')
        ax.set_ylabel('Amplitude [dB]')
        ax.axis((10, 1000, -100, 10))
        ax.grid(which='both', axis='both')
```

```
        plt.show()
        plt.figure()
        plt.title('Original vs. high-pass filtered signal')
        plt.plot(original_signal, label='Original Signal')
        plt.plot(high_pass_filtered_signal, color='red', label='High-pass filtered signal')
        plt.legend(loc='best');
    return high_pass_filtered_signal
def schmidt_spike_removal(original_signal, fs):
    """
    This function removes the spikes in a signal as done by Schmidt et al in
    the paper:
    Schmidt, S. E., Holst-Hansen, C., Graff, C., Toft, E., & Struijk, J. J.
    (2010). Segmentation of heart sound recordings by a duration-dependent
    hidden Markov model. Physiological Measurement, 31(4), 513-29.

    The spike removal process works as follows:
    (1) The recording is divided into 500 ms windows.
    (2) The maximum absolute amplitude (MAA) in each window is found.
    (3) If at least one MAA exceeds three times the median value of the MAA's,
    the following steps were carried out. If not continue to point 4.
        (a) The window with the highest MAA was chosen.
        (b) In the chosen window, the location of the MAA point was identified as the top
of the noise spike.
        (c) The beginning of the noise spike was defined as the last zero-crossing point
before theMAA point.
        (d) The end of the spike was defined as the first zero-crossing point after the
maximum point.
        (e) The defined noise spike was replaced by zeroes.
        (f) Resume at step 2.
```

```
    (4) Procedure completed.

    Inputs:
    original_signal: The original (1D) audio signal array
    fs: the sampling frequency (Hz)
    Outputs:
    despiked_signal: the audio signal with any spikes removed.

    This code is derived from the paper:     S. E. Schmidt et al., "Segmentation of heart sound recordings by a   duration-dependent hidden Markov model," Physiol. Meas., vol. 31,
    no. 4, pp. 513-29, Apr. 2010.

    Developed by David Springer for comparison purposes in the paper:  D. Springer et al., ?Logistic Regression-HSMM-based Heart Sound  Segmentation,? IEEE Trans. Biomed. Eng., In Press, 2015.

    Ported to Python by Pablo Vásquez
    """
    # Find the window size  (500 ms)
    windowsize = int(np.round(fs / 2))

    # Find any samples outside of a integer number of windows:
    trailingsamples = np.mod(len(original_signal), windowsize)

    # Reshape the signal into a number of windows:
    trimmed_signal = original_signal[0:len(original_signal) - trailingsamples]
    sampleframes = np.reshape(trimmed_signal, (windowsize, int(trimmed_signal.size / windowsize)))
```

```
    # Find the MAAs:
    MAAs = np.max(np.abs(sampleframes), axis=0)

    # While there are still samples greater than 3* the median value of the MAAs, then remove those spikes:
    while np.nonzero((MAAs > np.median(MAAs) * 3))[0].size != 0:  #
        # Find the window with the max MAA:
        val = np.max(MAAs)
        window_num = np.argmax(MAAs)

        # Find the position of the spike within that window:
        val = np.max(np.abs(sampleframes[:, window_num]))
        spike_position = np.argmax(sampleframes[:, window_num])

        # Finding zero crossings (where there may not be actual 0 values, just a change from positive to negative):
        zero_crossings = np.hstack((np.abs(np.diff(np.sign(sampleframes[:, window_num]))) > 1, np.array([0])))

        # Find the start of the spike, finding the last zero crossing before spike position. If that is empty, take the start of the window:
        sp1 = np.where(zero_crossings[0:spike_position - 1] == 1)
        spike_start = np.max([1, sp1[0][-1]])

        # Find the end of the spike, finding the first zero crossing after spike position. If that is empty, take the end of the window:
        sp2 = np.where(zero_crossings[spike_position:-1] == 1) + spike_position + 1
        spike_end = np.min([sp2[0][0], windowsize])

        # Set to Zero
```

```
        sampleframes[spike_start:spike_end, window_num] = 0.0001

        # Recalculate MAAs
        MAAs = np.max(np.abs(sampleframes), axis=0)

    despiked_signal = np.reshape(sampleframes, (windowsize *
int(trimmed_signal.size / windowsize)))
    # Add the trailing samples back to the signal:
    if trailingsamples != 0:
        despiked_signal = np.hstack((despiked_signal, original_signal[-
trailingsamples:]))

    return despiked_signal
def getHeartRateSchmidt(audio_data, Fs):
    """
    Derive the heart rate and the sytolic time interval from a PCG recording.
    This is used in the duration-dependant HMM-based segmentation of the PCG
    recording.
    This method is based on analysis of the autocorrelation function, and the
    positions of the peaks therein.
    This code is derived from the paper:
    S. E. Schmidt et al., "Segmentation of heart sound recordings by a
    duration-dependent hidden Markov model," Physiol. Meas., vol. 31,
    no. 4, pp. 513-29, Apr. 2010.

    Developed by David Springer for comparison purposes in the paper:
    D. Springer et al., "Logistic Regression-HSMM-based Heart Sound
    Segmentation," IEEE Trans. Biomed. Eng., In Press, 2015.
    INPUTS:
```

```
    audio_data: The raw audio data from the PCG recording
    Fs: the sampling frequency of the audio recording
    OUTPUTS:
    heartRate: the heart rate of the PCG in beats per minute
    systolicTimeInterval: the duration of systole, as derived from the
    autocorrelation function, in seconds

    Copyright (C) 2016  David Springer
    dave.springer@gmail.com

    Ported to Python by Pablo Vásquez
    """
    """
    Get heatrate:
    From Schmidt: "The duration of the heart cycle is estimated as the time from lag zero  to the highest peaks between
    500 and 2000 ms in the resulting autocorrelation"  This is performed after filtering and spike removal:
    25-400Hz 4th order Butterworth band pass
    """
    audio_data = butterworth_low_pass_filter(audio_data, 2, 400, Fs, figures=False)
    audio_data = butterworth_high_pass_filter(audio_data, 2, 25, Fs, figures=False)

    # Spike removal from the original paper:
    audio_data = schmidt_spike_removal(audio_data, Fs)

    # Find the homomorphic envelope
    homomorphic_envelope = Homomorphic_Envelope_with_Hilbert(audio_data, Fs)
    # Find the autocorrelation:
```

```
    y = homomorphic_envelope - np.mean(homomorphic_envelope)
    c = np.correlate(y, y, "full")
    signal_autocorrelation = c[len(homomorphic_envelope) + 1:-1]

    """ Set the max and min search indices
     This sets the search for the highest peak in the autocorrelation to be  between 120 (0.5*Fs) and 30 (2*Fs) BPM
     """
    min_index = int(0.5 * Fs)
    max_index = int(2 * Fs)
    index = np.argmax(signal_autocorrelation[min_index:max_index])
    true_index = index + min_index - 1
    heartRate = 60 / (true_index / Fs)

    """ Find the systolic time interval:
    From Schmidt: "The systolic duration is defined as the time from lag zero
    to the highest peak in the interval between 200 ms and half of the heart
    cycle duration" """
    max_sys_duration = int(np.round(((60 / heartRate) * Fs) / 2))
    min_sys_duration = int(np.round(0.2 * Fs))
    pos = np.argmax(signal_autocorrelation[min_sys_duration:max_sys_duration])
    systolicTimeInterval = (min_sys_duration + pos) / Fs

    return heartRate, systolicTimeInterval

def get_duration_distributions(heartrate, systolic_time):
    """
    This function calculates the duration distributions for each heart cycle state, and the minimum and maximum times for each state.
```

Inputs:

heartrate is the calculated average heart rate over the entire recording systolic_time is the systolic time interval

Outputs:

d_distributions is a 4 (the number of states) dimensional vector of gaussian mixture models

(one dimensional in this case), representing the mean and std deviation of the duration in each state.

The max and min values are self-explanatory.

This code is implemented as outlined in the paper:

S. E. Schmidt et al., "Segmentation of heart sound recordings by a duration-dependent hidden Markov model," Physiol. Meas., vol. 31,

no. 4, pp. 513-29, Apr. 2010.

Developed by David Springer for comparison purposes in the paper: D. Springer et al., "Logistic Regression-HSMM-based Heart Sound

Segmentation," IEEE Trans. Biomed. Eng., In Press, 2015.

Ported to Python by Pablo Vásquez

"""

```
schmidt_options = default_Schmidt_HSMM_options()
mean_S1 = np.round(0.122 * schmidt_options["audio_segmentation_Fs"])
std_S1 = np.round(0.022 * schmidt_options["audio_segmentation_Fs"])
mean_S2 = np.round(0.094 * schmidt_options["audio_segmentation_Fs"])
std_S2 = np.round(0.022 * schmidt_options["audio_segmentation_Fs"])
mean_systole = np.round(systolic_time * schmidt_options["audio_segmentation_Fs"]) - mean_S1
std_systole = (25 / 1000) * schmidt_options["audio_segmentation_Fs"]
```

```
    mean_diastole = ((60 / heartrate) - systolic_time - 0.094) *
schmidt_options["audio_segmentation_Fs"]
    std_diastole = 0.07 * mean_diastole + (6 / 1000) *
schmidt_options["audio_segmentation_Fs"]

    # Cell array for the mean and covariance of the duration distributions:
    d_distributions = np.zeros(shape=(4, 2))

    # Assign mean and covariance values to d_distributions:
    d_distributions[0, 0] = mean_S1
    d_distributions[0, 1] = std_S1 ** 2
    d_distributions[1, 0] = mean_systole
    d_distributions[1, 1] = std_systole ** 2
    d_distributions[2, 0] = mean_S2
    d_distributions[2, 1] = std_S2 ** 2
    d_distributions[3, 0] = mean_diastole
    d_distributions[3, 1] = std_diastole ** 2

    # Min systole and diastole times
    min_systole = mean_systole - 3 * (std_systole + std_S1)
    max_systole = mean_systole + 3 * (std_systole + std_S1)
    min_diastole = mean_diastole - 3 * std_diastole
    max_diastole = mean_diastole + 3 * std_diastole

    # Setting the Min and Max values for the S1 and S2 sounds:
    # If the minimum lengths are less than a 50th of the sampling frequency, set to a
50th of the sampling frequency:
    min_S1 = mean_S1 - 3 * std_S1
    if min_S1 < (schmidt_options["audio_segmentation_Fs"] / 50):
```

```
        min_S1 = schmidt_options["audio_segmentation_Fs"] / 50
    min_S2 = mean_S2 - 3 * std_S2
    if min_S2 < (schmidt_options["audio_segmentation_Fs"] / 50):
        min_S2 = schmidt_options["audio_segmentation_Fs"] / 50
    max_S1 = mean_S1 + 3 * std_S1
    max_S2 = mean_S2 + 3 * std_S2
    return [d_distributions, max_S1, min_S1, max_S2, min_S2, max_systole,
min_systole, max_diastole, min_diastole]
def viterbiDecodePCG(observation_sequence, pi_vector, b_matrix, heartrate,
systolic_time, Fs):
    """
    This function calculates the delta and psi matrices associated with the duration-
dependant Viterbi decoding algorithm. This algorithm is outlined
    in:
    L. R. Rabiner, ?A tutorial on hidden Markov models and selected  applications in
speech recognition,? Proc. IEEE, vol. 77, no. 2, pp.
    257?286, Feb. 1989.

    This code is implemented as outlined in the paper:   S. E. Schmidt et al.,
"Segmentation of heart sound recordings by a
    duration-dependent hidden Markov model," Physiol. Meas., vol. 31,     no. 4, pp.
513-29, Apr. 2010.

    Developed by David Springer for comparison purposes in the paper:     D. Springer
et al., "Logistic Regression-HSMM-based Heart Sound
    Segmentation," IEEE Trans. Biomed. Eng., In Press, 2015.

    INPUTS:
    observation_sequence: The sequence of extracted features pi_vector: the array of
initial state probabilities, dervived from
    "trainSchmidtSegmentationAlgorithm".
```

```
    b_matrix: the observation probabilities, dervived from
"trainSchmidtSegmentationAlgorithm".
    heartrate: the heart rate of the PCG, extracted using  "getHeartRateSchmidt"
    systolic_time: the duration of systole, extracted using  "getHeartRateSchmidt"
    Fs: the sampling frequency of the observation_sequence
    Outputs:
    delta: the matrix of highest probability along a single path, at time t, which accounts
for the first t observations and ends in state S_i
    psi: the matrix of states that maximised delta for each t and j
    qt: the optomised state sequence

    Ported to Python by Pablo Vásquez
    """
    # Preliminary
    schmidt_options = default_Schmidt_HSMM_options()
    T = len(observation_sequence)
    N = 4  # Number of states
    realmin = 10 ** (-30)

    duration_sum = np.zeros(shape=(N,))

    # Setting the maximum duration of a single state. This is set to an entire  heart
cycle:
    max_duration_D = int(np.round((1 * (60 / heartrate)) * Fs))

    """ Initialising the variables that are needed to find the optimal state path along  the
observation sequence.
    delta_t(j), as defined on page 264 of Rabiner, is the best score (highest probability)
along a single path, at
    time t, which accounts for the first t observations and ends in State s_j."""
```

```
    delta = np.ones(shape=(T, N)) * -np.inf

    """The argument that maximises the transition between states (this is  basically the previous state that had the
    highest transition probability to the current state) is tracked using the psi variable."""
    psi = np.zeros(shape=(T, N))

    """An additional variable, that is not included on page 264 or Rabiner, is the state duration that maximises the delta variable. This is essential
    #for the duration dependant HMM."""
    psi_duration = np.zeros(shape=(T, N))

    # Setting up observation probs
    observation_probs = np.zeros(shape=(T, N))

    """ From the multivariate normal B-matrix, find the probability of the  features derived from
     each sample being in each state:"""
    for state_n in range(0, N):
        observation_probs[:, state_n] = multivariate_normal.pdf(observation_sequence, mean=b_matrix[state_n, 0],
                                                cov=b_matrix[state_n, 1])
    """Setting up state duration probabilities, using Gaussian distributions:  Schmidt's paper makes use of Gaussian timing
    distributions for each state to model the expected duration of each state:"""
    d_distributions, max_S1, min_S1, max_S2, min_S2, max_systole, min_systole, max_diastole, min_diastole = get_duration_distributions(
        heartrate, systolic_time)
```

```
    """ To speed up computation, compute the probability of each state being d
samples in duration
    ahead of computation, and save these values in a matrix:"""
    duration_probs = np.zeros(shape=(N, max_duration_D))


    for state_j in range(0, N):
        for d in range(0, max_duration_D):
            if state_j == 0:
                duration_probs[state_j, d] = multivariate_normal.pdf(d,
mean=d_distributions[state_j, 0],
                                                     cov=d_distributions[state_j, 1])
                # TODO Justify minimum length
                if d < min_S1 or d > max_S1:
                    duration_probs[state_j, d] = realmin


            elif state_j == 2:
                duration_probs[state_j, d] = multivariate_normal.pdf(d,
mean=d_distributions[state_j, 0],
                                                     cov=d_distributions[state_j, 1])
                # TODO Justify minimum length
                if d < min_S2 or d > max_S2:
                    duration_probs[state_j, d] = realmin


            elif state_j == 1:
                duration_probs[state_j, d] = multivariate_normal.pdf(d,
mean=d_distributions[state_j, 0],
 cov=d_distributions[state_j, 1])


                # TODO Justify minimum length
                if d < min_systole or d > max_systole:
```

```
                duration_probs[state_j, d] = realmin


            elif state_j == 3:
                duration_probs[state_j, d] = multivariate_normal.pdf(d,
mean=d_distributions[state_j, 0],
 cov=d_distributions[state_j, 1])


                # TODO Justify minimum length
                if d < min_diastole or d > max_diastole:
                    duration_probs[state_j, d] = realmin


        duration_sum[state_j] = duration_probs[state_j, :].sum()
    # Perform the actual Viterbi Recursion:
    qt = np.zeros(shape=(len(delta),))
    # Initialisation Step
    # Equation 32a and 69a:
    delta[0, :] = np.log(pi_vector) + np.log(observation_probs[0, :])
    """ first value is the probability of initially being in each state * probability of
observation 1
    coming from each state   Equation 32b"""
    psi[0, :] = -1


    """The state duration probabilities are now used. Change the a_matrix to have
zeros along the diagonal, therefore, only  relying on the duration probabilities and
     observation probabilities to influence change in states: This would only be valid in
sequences where the transition between states follows a distinct order."""
    a_matrix = np.array([[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1], [1, 0, 0, 0]])


    # Run the core Viterbi algorithm
    if schmidt_options["use_mex"]:
```

```
        path = "C:\\Users\\pjvas\\Documents\\Python thesis\\recordings\\Sample_code\\"
        #viterbi_Schmidt = ctypes.CDLL(path + 'viterbi_Schmidt.so')
        #[delta, psi, psi_duration] = viterbi_Schmidt.viterbi(N, T, a_matrix,
max_duration_D, delta, observation_probs, duration_probs, psi)
        viterbi_Schmidt = ctypes.CDLL(path + 'viterbi_Schmidt.so')
        viterbi =viterbi_Schmidt.viterbi
        viterbi.argtypes=[ctypes.c_int,
                          ctypes.c_int,
                          np.ctypeslib.ndarray(shape=(4,4), dtype=np.double),
                          np.ctypeslib.ndpointer(dtype=np.float64),
                          np.ctypeslib.ndpointer(dtype=np.float64),
                          np.ctypeslib.ndpointer(dtype=np.float64),
                          np.ctypeslib.ndpointer(dtype=np.float64)]
        [delta, psi, psi_duration] = viterbi(N, T, a_matrix, max_duration_D, delta,
observation_probs, duration_probs, psi)


    else:
        """Recursion
        For the first D steps, neeeds to calculate each delta separately:
        Equations 33a and 33b and 69a, b, c etc: again, ommitting the p(d), as state
could have started before t = 1
        For first samples of the signal, less than the max duration of a state:
        As the state duration probabilities cannot be used yet, find each state  by the
transition and observation
        probabilities only: For the rest of the signal, where t > max_duration of a state,
the state  duration probabilities
        can now be used, as we know that we are at least a maximum duration into the
signal. This loop differs from the one above, as we now
        also search over all the possible durations we could be in each state with
variable d = 1:max_duration_D. Therefore, we now search over
```

```
        an "analysis window" of time d, taking into account the probability of a state lasting for time d, as well as seeing all the observations

        within that window in one state. Change the a_matrix to have zeros along the diagonal, therefore, only relying on the duration probabilities and

        observation probabilities to influence change in states:"""
        for t in range(1, T):
            for j in range(0, N):
                emission_probs = 0
                for d in range(0, int(max_duration_D)):
                    if t - d > 0:
                        """The start of the analysis window, which is the current time step, minus d, the time

                        horizon we are currently looking back, plus 1. The analysis window can be seen to be starting

                        one step back each time the variable d is increased."""
                        start = t - d - 1


                        """Find the max_delta and index of that from the previous step and the transition to the current step:

                        This is the first half of the expression of equation 33a from Rabiner:"""

                        #[max_delta, max_index] = np.max(delta[start, :] + np.log(a_matrix[:, j].T))

                        max_delta= np.max(delta[start, :] + np.log(a_matrix[:, j].T))

                        max_index = np.argmax(delta[start, :] + np.log(a_matrix[:, j].T))


                        """Find the normalised probabilities of the observations at only the time point at the start

                        of the time window:"""

                        probs = np.prod(observation_probs[start:t, j], axis=0)
```

```
                """Keep a running total of the emmission probabilities as the start point
of the time window
                is moved back one step at a time. This is the probability of seeing all
the observations in
                the analysis window in state j:"""
                if probs == 0 or np.isnan(probs):
                    probs = realmin

                emission_probs = np.log(probs)

                """Find the total probability of transitioning from the last state to this
one, with the
                observations and being in the same state for the analysis window. This
is the
                duration-dependant variation of equation 33a from Rabiner:"""
                delta_temp = max_delta + emission_probs + np.log((duration_probs[j,
d]/duration_sum[j]))

                """Unlike equation 33a from Rabiner, the maximum delta could come
from multiple d values,
                or from multiple size of the analysis window. Therefore, only keep the
maximum delta value
                over the entire analysis window: If this probability is greater than the
last greatest,
                update the delta matrix and the time duration variable:"""

                if delta_temp > delta[t, j]:
                    delta[t, j] = delta_temp
                    psi[t, j] = max_index
                    psi_duration[t, j] = d
    """ Termination
```

1) Find the last most probable state

2) From the psi matrix, find the most likely preceding state

3) Find the duration of the last state from the psi_duration matrix

4) From the onset to the offset of this state, set to the most likely state

5) Repeat steps 2 - 5 until reached the beginning of the signal

The initial steps 1-4 are equation 34b in Rabiner. 1) finds P*, the most likely last state in the sequence,

2) finds the state that precedes the last most likely state, 3) finds the onset in time of the last state(included

due to the duration-dependency) and 4) sets the most likely last state to the q_t variable."""

```
#val = np.max(delta[T-1, :])
state = np.argmax(delta[T-1, :])  # 1
offset = T-1  # 2
preceding_state = psi[int(offset), int(state)]  # 3
onset = offset - psi_duration[int(offset), int(state)]
qt[int(onset):int(offset)] = state  # 4

"""The state is then updated to the preceding state, found above, which must end when the last most
likely state started in the observation sequence:"""
state = preceding_state
count = 0

# While the onset of the state is larger than the maximum duration specified:
while onset > 1:
    offset = onset - 1
    preceding_state = psi[int(offset), int(state)]
    onset =offset - psi_duration[int(offset), int(state)]
```

```
        if onset < 1:
            onset = 0
        qt[int(onset):int(offset)+1] = state
        state = preceding_state
        count = count + 1
        if count > 10000:
            break
    return [delta, psi, qt]
def runSchmidtSegmentationAlgorithm(audio_data, Fs, B_matrix, pi_vector,
figures=False):
    """
    runSchmidtSegmentationAlgorithm(audio_data, Fs, B_matrix, pi_vector, figures):
    A function to assign states to a PCG recording based on a trained
    duration-dependant HMM

    INPUTS:
    audio_data: The raw audio data from the PCG recording
    Fs: the sampling frequency of the audio recording
    B_matrix: the observation matrix for the HMM, trained in the
    "trainSchmidtSegmentationAlgorithm.m" function
    pi_vector: the initial state distribution, also trained in the
    "trainSchmidtSegmentationAlgorithm.m" function
    figures: (optional) boolean variable for displaying figures

    OUTPUTS:
    assigned_states: the array of state values assigned to the original
    audio_data (in the original sampling frequency).

    This code is derived from the paper:
```

```
    S. E. Schmidt et al., "Segmentation of heart sound recordings by a
    duration-dependent hidden Markov model," Physiol. Meas., vol. 31,
    no. 4, pp. 513-29, Apr. 2010.

    Developed by David Springer for comparison purposes in the paper:
    D. Springer et al., ?Logistic Regression-HSMM-based Heart Sound
    Segmentation,? IEEE Trans. Biomed. Eng., In Press, 2015.
    Ported to Python by Pablo Vásquez
    """
    # Get PCG Features:
    [PCG_Features, featuresFs] = getSchmidtPCGFeatures(audio_data, Fs)
    # Get PCG heart rate
    [heartRate, systolicTimeInterval] = getHeartRateSchmidt(audio_data, Fs)
    [delta, psi, qt] = viterbiDecodePCG(PCG_Features, pi_vector, B_matrix, heartRate,
systolicTimeInterval, featuresFs)

    assigned_states = expand_qt(qt, featuresFs, Fs, len(audio_data))

    if figures:
        plt.figure()
        plt.title('Derived state sequence')
        t1 = np.linspace(1, len(audio_data)) / Fs
        plt.plot(t1, audio_data, color="black", label="Audio data")
        plt.plot(t1, assigned_states, color="red", label="Derived States")
        plt.legend(loc='best')
    return assigned_states
def trainBandPiMatricesSchmidt(state_observation_values):
    """
    Initialise the B_matrix as a 4x2 cell array. Each of the four states has
```

```
    two cell arrays - the first (B_matrix{state_i,1}) holds the mean value
    for the observations for that state. The second entry,
    (B_matrix{state_i,2}) holds the covariance matrix for the observations
    for each state.
    """
    B_matrix = np.empty(shape=(4, 2))
    """
    Set pi_vector
    The true value of the pi vector, which are the initial state
    probabilities, are dependant on the heart rate of each PCG, and the
    individual sound duration for each patient. Therefore, instead of setting
    a patient-dependant pi_vector, simplify by setting all states as equally
    probable:
    """
    pi_vector = np.array([0.25, 0.25, 0.25, 0.25])

    # Derive  B  matrix mean and covariance
    statei_values = [[], [], [], []]

    for PCGi in range(0, len(state_observation_values)):
        statei_values[0] = np.hstack((statei_values[0],
state_observation_values[PCGi][0]))
        statei_values[1] = np.hstack((statei_values[1],
state_observation_values[PCGi][1]))
        statei_values[2] = np.hstack((statei_values[2],
state_observation_values[PCGi][2]))
        statei_values[3] = np.hstack((statei_values[3],
state_observation_values[PCGi][3]))

    # Assign  mean and covariance values  to    B_matrix:
```

```
    for state_i in range(0, 4):
        B_matrix[state_i, 0] = np.mean(statei_values[state_i])
        B_matrix[state_i, 1] = np.cov(statei_values[state_i])


    return B_matrix, pi_vector
def labelPCGStates(envelope, s1_positions, s2_positions, samplingFrequency,
figures=False):
    """
        This function assigns the state labels to a PCG record.  This is based on ECG
markers, derived from the R peak and end-T wave locations.


        Inputs:
        envelope: The PCG recording envelope (found in getSchmidtPCGFeatures.m)
        s1_positions: The locations of the R peaks (in samples)
        s2_positions: The locations of the end-T waves (in samples)
        samplingFrequency: The sampling frequency of the PCG recording
        figures (optional): boolean variable dictating the display of figures


        Output:
        states: An array of the state label for each sample in the feature
        vector. The total number of states is 4. Therefore, this is an array of
        values between 1 and 4, such as: [1,1,1,1,2,2,2,3,3,3,3,4,4,4,4,4,1,1,1],
        illustrating the "true" state label for each sample in the features.
        State 1 = S1 sound
        State 2 = systole
        State 3 = S2 sound
        State 4 = diastole
```

```
        This code was developed by David Springer for comparison purposes in the
paper:

        D. Springer et al., ?Logistic Regression-HSMM-based Heart Sound
Segmentation,? IEEE Trans. Biomed. Eng., In Press, 2015.

        where a novel segmentation approach is compared to the paper by Schmidt et
al:

        S. E. Schmidt et al., "Segmentation of heart sound recordings by a duration-
dependent hidden Markov model," Physiol. Meas., vol. 31,

        no. 4, pp. 513-29, Apr. 2010.

        Ported to python by pablo Vásquez
        """

    states = np.zeros(shape=(len(envelope),), dtype=int)

    print("envelope size:{}, max s1: {}, max s2 {}: ".format(envelope.size,
                                                    np.max(s1_positions), np.max(s2_positions)))

    # Timing   durations  from Schmidt:

    mean_S1 = 0.122 * samplingFrequency

    std_S1 = 0.022 * samplingFrequency

    mean_S2 = 0.092 * samplingFrequency

    std_S2 = 0.022 * samplingFrequency

    """ Setting the duration from each R-peak to(R - peak + mean_S1) as the  first
state:

        The R - peak in thec  ECGc    coincides with the start of the S1 sound (A.G.

        Tilkian and M.B.Conover, Understanding heart sounds and murmurs: with an
introduction

        to lung sounds, 4th   ed.Saunders, 2001.)

        Therefore, the duration   from each R -peak  to  the  mean_S1  sound  duration

        later were labelled as the  "true" positions of  the S1  sounds:

        """

    for i in range(0, len(s1_positions)):

        # Set an upper bound, in case the  window  extends over  the length  of  the
signal:
```

```
        upper_bound = int(np.round(np.min(np.array([len(states), s1_positions[i] +
mean_S1]))))


        # Set the states between the start of the R peak and the upper bound as state 1:
        states[np.max([0, s1_positions[i]]): np.min([upper_bound, len(states)])] = 1
    """
        Set S2 as state 3 depending on position of end T-wave peak in ECG:
        The second heart sound occurs at approximately the same time as the
        end-T-wave (A. G. Tilkian and M. B. Conover, Understanding heart sounds
        and murmurs: with an introduction to lung sounds, 4th ed. Saunders, 2001.)
        Therefore, for each end-T-wave, find the peak in the envelope around the
        end-T-wave, setting a window centered on this peak as the second heart
        sound state:
        """
    for i in range(0, len(s2_positions)):
        # find search window of envelope:
        # T - end + - mean + 1sd
        # Set upper and lower bounds, to avoid errors of searching outside size of the
signal
        lower_bound = int(np.max([s2_positions[i] - np.floor((mean_S2 + std_S2)), 0]))
        upper_bound = int(np.min([len(states), np.ceil(s2_positions[i] +
np.floor(mean_S2 + std_S2))
                                  ]))
        search_window = envelope[lower_bound:upper_bound] *
(states[lower_bound:upper_bound] != 1)
        if search_window.size == 0:
            print(
                "i : {}. lower bound: {}, upper bound: {} , search windows size: {}".format(i,
lower_bound, upper_bound,
                                                                    search_window))
```

```
        # Find the maximum value of the envelope in the search window:
        S2_index = np.argmax(search_window) + lower_bound
        # Find the actual index in the envelope of the maximum peak:
        # Make sure this has a max value of the length of the signal:
        S2_index = np.min([len(states), S2_index])
        # Set the states to state 3, centered on the S2 peak, +- 1 / 2 of the
        # expected S2c sound duration.Again, making sure it does not try to set a
        # value outside of the length of the signal:
        upper_bound = int(np.min([len(states), np.ceil(S2_index + (mean_S2 / 2))]))
        states[int(np.max([np.ceil(S2_index - (mean_S2 / 2)), 0])): upper_bound] = 3
        # Set the spaces between state 3 and the next R peak as state 4:
        if i <= len(s2_positions):
            # We need to find the next R pak after  this S2  sound So, subtract the
            # position of  this  S2 rom the S1 positions
            diffs = s1_positions - s2_positions[i]
            # Exclude  those that are negative(meaning before this S2 occurred) by
setting
            # them to infinity.They are then excluded  when finding the minimum later
            diffs[diffs < 0] = np.iinfo(np.int32).max
            # If the array is empty, then no  S1s after this S2, so set to end of signal:
            if np.nonzero(diffs < np.iinfo(np.int32).max)[0].size == 0:
                end_pos = len(states)
            else:
                # else, send the to the minimum diff - 1
                index = np.argmin(diffs)
                end_pos = s1_positions[index] - 1
            states[int(np.ceil(S2_index + ((mean_S2 + (0 * std_S2)) / 2))): end_pos] = 4
    """
```

Setting the first and last sections of the signal.

As all states are derived from either R -peak or end - T - wave locations, the first affirmed state in the will be state 1 or state 3. Therefore, until this state, the first state should always be set to 4 or 2:"""

```
# Find the first  step up:
first_location_of_definite_state = np.nonzero(states != 0)[0][0] - 1
if first_location_of_definite_state > 0:

    if states[first_location_of_definite_state + 1] == 1:
        states[0: first_location_of_definite_state] = 4

    if states[first_location_of_definite_state + 1] == 3:
        states[0: first_location_of_definite_state] = 2

# Find the last step down:
last_location_of_definite_state = np.nonzero(states != 0)[0][-1]
if last_location_of_definite_state > 0:
    if states[-last_location_of_definite_state] == 1:
        states[-last_location_of_definite_state:] = 2

    if states[-last_location_of_definite_state] == 3:
        states[-last_location_of_definite_state:] = 4

states = states[:len(envelope)]
# Set everywhere else as state 2:
states[states == 0] = 2
# Plotting figures
if figures:
```

```
        plt.figure()
        plt.title('Envelope and labelled states')
        plt.plot(envelope, label='Envelope')
        plt.plot(states, color='red', label='States')
        plt.legend(loc='best')

    return states
def trainSchmidtSegmentationAlgorithm(PCGCellArray, annotationsArray, Fs, figures=False):
    """Training the emissions matrix, B_matrix, and initial distribution,
    pi_vector, for the Schmidt HMM segmentation algorithm.

    Inputs:
        PCGCellArray: A 1XN cell array of the N audio signals. For evaluation
        purposes, these signals should be from a distinct training set of
        recordings, while the algorithm should be evaluated on a separate test
        set of recordings, which are recorded from a completely different set of
        patients (for example, if there are numerous recordings from each patint).

        annotationsArray: a Nx2 cell array: position (n,1) = the positions of the
        R-peaks and postion (n,2) = the positions of the end-T-waves both in SAMPLES)

        Fs: The sampling frequency
        figures (optional): boolean variable dictating the disaplay of figures.

    Outputs:
        The B_matrix and pi arrays for an HMM - as Schmidt et al's algorithm is a
        duration dependant HMM, there is no need to calculate the A_matrix, as
        the transition between states is only dependant on the state durations.
```

```
    This code is derived from the paper:
    S. E. Schmidt et al., "Segmentation of heart sound recordings by a
    duration-dependent hidden Markov model," Physiol. Meas., vol. 31,
    no. 4, pp. 513-29, Apr. 2010.

    Developed by David Springer for comparison purposes in the paper:
    D. Springer et al., ?Logistic Regression-HSMM-based Heart Sound
    Segmentation,? IEEE Trans. Biomed. Eng., In Press, 2015.

    Copyright (C) 2016  David Springer"""

    # Options
    numberOfStates = 4
    numberOfPCGs = len(PCGCellArray)
    # A matrix of the values from each state in each of the PCG recordings:
    state_observation_values = list()
    for PCGi in range(0, numberOfPCGs):
        print("PCGi no {}".format(PCGi))
        PCG_audio = PCGCellArray[PCGi]
        S1_locations = annotationsArray[PCGi][0] # Couldn't read .dat files
        S2_locations = annotationsArray[PCGi][1]
        PCG_Features, featuresFs = getSchmidtPCGFeatures(PCG_audio, Fs)
        PCG_states = labelPCGStates(PCG_Features, S1_locations, S2_locations,
featuresFs)
        # Plotting assigned states:
        if figures:
            fig = plt.figure()
            plt.title('Assigned states to PCG')
```

```
        t1 = np.linspace(0, len(PCG_audio), len(PCG_audio)) / Fs
        t2 = np.linspace(0, len(PCG_Features), len(PCG_Features)) / featuresFs
        plt.plot(t1, PCG_audio, color='black', label='Audio')
        plt.plot(t2, PCG_Features, color='blue', label='Features')
        plt.plot(t2, PCG_states, color='red', label='States')
        plt.legend(loc='best')

    # Group together all observations from the same state in the PCG recordings:
    state_observation_values.append([[], [], []])
    for state_i in range(numberOfStates):
        state_observation_values[PCGi][state_i] = PCG_Features[PCG_states ==
state_i + 1]

        # Train the B and pi matrices after all the PCG recordings have been labelled:

[B_matrix, pi_vector] = trainBandPiMatricesSchmidt(state_observation_values)
return B_matrix, pi_vector
```