UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

OPTIMIZACIÓN DE UN SISTEMA DE INDEXADO Y BÚSQUEDA DE CONTENIDOS DE AUDIO Y AUDIOVISUALES

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

David González Hernández
Tutor: Doroteo Torre Toledano

Julio 2016

OPTIMIZACIÓN DE UN SISTEMA DE INDEXADO Y BÚSQUEDA DE CONTENIDOS DE AUDIO Y AUDIOVISUALES

AUTOR: David González Hernández
TUTOR: Doroteo Torre Toledano

Biometric Recognition Group - ATVS
Departamento de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio 2016

Agradecimientos

Quiero dedicar este Trabajo de Fin de Grado a una de las dos personas más importantes de mi vida, a mi madre.

No sabes cuánto te quiero y te querré, aunque debiste de saberlo muy bien porque no había persona en este mundo que fuese capaz de amar de una forma tan inmensa como tú. Lo intento, pero no soy capaz de expresar todo lo que se me viene al corazón cuando pienso en ti, eres mi mayor privilegio y mi mayor regalo. Gracias por ser mi ángel de la guarda, por entregarme tu corazón lleno de amor verdadero y darme impulso para alcanzar lo imposible.

Papá, ni te imaginas lo que te quiero y todo lo que significas para mí, gracias por todo, por ser mi impulso para llegar a ser quien soy y porque sin ti no podría seguir adelante.

Me gustaría mencionar a mi tutor Doroteo, gracias a él tuve la oportunidad de trabajar y aprender con él durante el desarrollo de este Trabajo. Gracias por todo este tiempo y paciencia que tuviste conmigo.

No me puedo olvidar de Sergio Bernedo. Gracias por haberme acompañado durante todos estos años de carrera, gracias por todos esos momentos en los laboratorios y en la cafetería contándonos batallitas. Grandes recuerdos que nunca se olvidarán.

Gracias a todos los amigos de baloncesto, por todos esos domingos especiales a vuestro lado.

A mi amiga Vero, que siempre ha estado apoyándome en todo momento y haciéndome reír cuando uno más lo necesitaba.

Y por último, no me puedo olvidar de Valeria, gracias por todo lo que haces por mí, significa mucho más de lo que imaginas.

Mamá,

Podrá nublarse el sol eternamente;
Podrá secarse en un instante el mar;
Podrá romperse el eje de la tierra
Como un débil cristal.
¡todo sucederá! Podrá la muerte
Cubrirme con su fúnebre crespón;
Pero jamás en mí podrá apagarse
La llama de tu amor.

Te quiero.

Resumen

El objetivo de este Trabajo Fin de Grado es el de optimizar el rendimiento de un sistema de búsqueda de audio en audio. Se parte de un sistema existente desarrollado en el Proyecto de Fin de Carrera de Andrés Martín López [1], en el que el objetivo principal fue mejorar la precisión del sistema en condiciones realistas e incluso extremas de ruido, pero no se puso especial atención a aspectos de eficiencia computacional. Se implementarán una serie de algoritmos de *clustering* y búsqueda optimizada en los árboles resultantes para mejorar el tiempo de búsqueda tratando de que el efecto en la precisión del sistema sea lo más reducido posible.

La memoria comienza con una explicación de algunos sistemas de recuperación de información de audio, así como de las técnicas de robustez frente al ruido y del sistema de partida. Este sistema se divide en dos módulos, el primero se encarga de realizar una extracción de características robusta frente al ruido a partir de muestras de audio, y la segunda se encarga de buscar el mejor alineamiento temporal entre los vectores de la muestra original y la muestra grabada.

El nuevo sistema incluye dos nuevos módulos. En el primero se reorganizan los vectores de la muestra original y de la muestra grabada para tratar de optimizar los tiempos de búsqueda, y en el segundo se implementan los algoritmos de *clustering* y de búsqueda optimizada en los árboles resultantes, en este módulo se realizarán diversos estudios en los que se tratarán de conseguir un tiempo de búsqueda y precisión óptimos.

Palabras clave

Búsqueda de audio en audio, Sincronización de audios, Optimización, FLANN, vecinos cercanos, árbol K-D, árbol K-Means, algoritmos de búsqueda rápida aproximada, fingerprinting, trayectorias temporales.

Abstract

The objective of this project is to optimize an audio in audio searching system. The starting point is an existing system developed in the Andrés Martín López PFC [1], in which the main objective was to improve the accuracy of the system in real conditions even in extreme noise conditions, but no special attention was given to aspects of computational efficiency. A number of clustering algorithms and optimized searching on the resulting trees will be implemented to improve that searching time trying not to affect as much as possible the accuracy of the system.

This report begins with a brief explanation of some information retrieval of audio systems, robustness techniques against noise and the starting system. The system is divided into two modules, the first module is responsible for making robust extraction versus noise characteristic coefficients from audio samples, and the second module is responsible for performing a temporal alignment between vectors of the original sample and the recorded sample.

The new system includes two new modules. The first module is responsible for reorganizing the database and the sample recorded trying to optimize the searching time, and the second module is responsible for implementing clustering algorithms and optimized searching on the resulting trees, several studies will be made and will try to get an optimum search time and accuracy.

Keywords

Audio in audio searching system, audio synchronization, optimization, FLANN, nearest neighbors, K-D tree, K-Means tree, fast approximate matching algorithms, fingerprinting, temporary trajectories (time paths)

INDICE DE CONTENIDOS

1 Introducción ... 1
1.1 Motivación ... 1
1.2 Objetivos ... 1
1.3 Organización de la memoria ... 2
2 Estado del arte ... 3
2.1 Recuperación de la información de la señal de audio ... 3
2.1.1 Sistemas de búsqueda de audio en audio ... 3
2.1.2 Shazam ... 4
2.1.3 Audible Magic ... 5
2.2 Robustez frente al ruido ... 5
2.3 Programa de partida ... 6
2.3.1 Programa básico ... 6
2.3.2 Generación de los *fingerprints* ... 7
2.3.3 Algoritmo de búsqueda ... 8
2.3.4 Técnicas de robustez frente al ruido en el programa de partida ... 9
2.3.4.1 CMVN sobre los coeficientes MFCC ... 10
2.3.4.2 CMVN sobre los fingerprints ... 11
2.3.4.3 Análisis de trayectorias temporales ... 12
3 Diseño y Desarrollo ... 13
3.1 Binary Search Tree (BST) ... 13
3.2 Árboles K-d ... 13
3.2.1 Vecinos más cercanos en Árboles K-d ... 18
3.3 Árboles K-Means ... 19
3.4 Algoritmos escalables de vecinos más cercanos ... 21
3.4.1 The Multiple Randomized K-d Tree Algorithm ... 21
3.4.2 The Priority Search K-Means Tree Algorithm ... 23
3.4.2.1 Parámetros del algoritmo ... 25
3.5 Librería FLANN ... 26
3.5.1 flann_build_index() ... 27
3.5.2 flann_find_nearest_neighbors_index() ... 29
3.5.3 flann_free_index() ... 29
3.5.4 flann_set_distance_type() ... 29
3.6 Nuevo diseño del programa ... 30
3.7 Ficheros de prueba ... 32
4 Pruebas y Resultados ... 35
4.1 Análisis del programa de partida ... 35
4.1.1 Resultados del programa de partida ... 35
4.2 Análisis del nuevo diseño ... 36
4.2.1 Resultados: Algoritmo de Árboles K-d Aleatorios ... 37
4.2.2 Resultados: Algoritmo de Árboles K-Means con búsqueda de prioridad ... 37
4.3 Análisis de los resultados ... 38
5 Conclusiones y trabajo futuro ... 40
5.1 Conclusiones ... 40
5.2 Trabajo futuro ... 40
Referencias ... - 1 -

INDICE DE FIGURAS

FIGURA 2.1: CONSTELACIÓN DE PICOS DE UN ESPECTOGRAMA. [10] 4

FIGURA 2.2: ESQUEMA BÁSICO DEL SISTEMA DE PARTIDA. [1] 7

FIGURA 2.3: GENERACIÓN DE LOS FINGERPRINTS. [1] 8

FIGURA 2.4: ALGORITMO DE BÚSQUEDA PROPUESTO. [1] 9

FIGURA 2.5: APLICACIÓN DE LA TÉCNICA CMVN SOBRE LOS COEFICIENTES MFCC. [1] 10

FIGURA 2.6: APLICACIÓN DE LA TÉCNICA CMVN SOBRE LOS FINGERPRINTS. [1] 11

FIGURA 3.1: EJEMPLO DE ÁRBOL K-D. [3] 14

FIGURA 3.2: CONJUNTO DE DATOS PARA CONSTRUIR EL ÁRBOL K-D. [3] 14

FIGURA 3.3: NODO RAÍZ DEL ÁRBOL K-D. [3] 15

FIGURA 3.4: CREACIÓN DEL SUBÁRBOL DERECHO. [3] 16

FIGURA 3.5: ÁRBOL K-D FINAL. [3] 16

FIGURA 3.6: ELIMINACIÓN DEL SUBÁRBOL IZQUIERDO EN LA BÚSQUEDA. [3] 17

FIGURA 3.7: ELIMINACIÓN DEL SIGUIENTE SUBÁRBOL. [3] 17

FIGURA 3.8: RESULTADO FINAL DE LA BÚSQUEDA. [3] 18

FIGURA 3.9: RADIO DE BÚSQUEDA PARA ENCONTRAR LOS VECINOS MÁS CERCANOS. [3] 18

FIGURA 3.10: FUNCIONAMIENTO DEL ALGORITMO K-MEANS. [5] 20

FIGURA 3.11: EJEMPLO DE ÁRBOL K-MEANS. [15] 20

FIGURA 3.12: CONSTRUCCIÓN DEL ÁRBOL K-D. [6] Y [7] 21

FIGURA 3.13: BÚSQUEDA EN EL ÁRBOL K-D. [6] Y [7] 22

FIGURA 3.14: EJEMPLOS DE ÁRBOLES K-D ALEATORIOS. [6] Y [7]. 22

FIGURA 3.15: VELOCIDAD OBTENIDA USANDO MÚLTIPLES ÁRBOLES K-D. (RESULTADOS OBTENIDOS A PARTIR DE LAS PRUEBAS REALIZADAS POR MARIUS MUJA). [6] Y [7]. 23

FIGURA 3.16: CONSTRUCCIÓN DEL ÁRBOL K-MEANS. [6] Y [7]. 24

FIGURA 3.17: BÚSQUEDA EN EL ÁRBOL K-MEANS. [6] Y [7]. 25

Figura 3.18: Influencia del número de iteraciones del algoritmo K-Means en los tiempos de búsqueda, se muestra el tiempo de búsqueda relativo comparado con el caso usando convergencia máxima. 26

Figura 3.19: Fichero makefile 27

Figura 3.20: Organización de la estructura. [8]. 28

Figura 3.21: Tipos de log. [8]. 28

Figura 3.22: Estructura del fingerprint de la base de datos del programa de partida. 30

Figura 3.23: Estructura reorganizada del fingerprint de la base de datos. 31

Figura 3.24: Estructura del fingerprint de la muestra grabada del programa de partida. 31

Figura 3.25: Estructura reorganizada del fingerprint de la muestra grabada. 32

INDICE DE TABLAS

TABLA 4-1: LATENCIAS DE LAS MUESTRAS GRABADAS. [1]. 35

TABLA 4-2: TIEMPO DE EJECUCIÓN DEL PROGRAMA ORIGINAL. 36

TABLA 4-3: PORCENTAJE DE ACIERTO DEL PROGRAMA DE PARTIDA CON RECORTES DEL AUDIO ORIGINAL. 36

TABLA 4-4: PORCENTAJE DE ACIERTO DE LA MUESTRAS DEL IPHONE.......... 36

TABLA 4-5: PORCENTAJE DE ACIERTO DE LAS MUESTRAS DEL HTC. 36

TABLA 4-6: TIEMPO DE EJECUCIÓN APLICANDO EL ALGORITMO K-D. 37

TABLA 4-7: PORCENTAJE DE ACIERTO APLICANDO EL ALGORITMO K-D. 37

TABLA 4-8: TIEMPO DE EJECUCIÓN APLICANDO EL ALGORITMO K-MEANS. 38

TABLA 4-9: PORCENTAJE DE ACIERTO APLICANDO EL ALGORITMO K-MEANS. 38

1 Introducción

1.1 Motivación

En los últimos años hemos podido ver cómo ha aumentado el uso de aplicaciones que tratan de recuperar información musical, son sistemas capaces de identificar pequeños ficheros de audio grabados mediante el teléfono móvil. Esto permite que cualquier usuario que esté escuchando una canción pueda identificarla y saber de cuál se trata. El ejemplo más conocido de este tipo de sistemas es Shazam [10], que es capaz de identificar rápidamente cualquier tipo de ficheros musicales.

No solo existen sistemas para recuperar información musical, también existen aplicaciones que tratan de sincronizar audios, es decir, no solamente se identifica el fichero al que pertenece, sino que también indica el instante en el que transcurre sobre el mismo. Respecto a este tipo de tecnología, la compañía AudibleMagic desarrolla diversos productos relacionados con la sincronización de audio.

Debido a la popularidad que está obteniendo este tipo de aplicaciones, en el PFC de Andrés Martín López [1] se partió de un sistema básico incapaz de manejar ficheros ruidosos y se propusieron varios algoritmos de robustez frente al ruido para que este tipo de sistemas tengan mecanismos de defensa frente a este tipo de señal no deseada. Tras este PFC se consiguió un sistema con una gran precisión y un alto grado de robustez frente al ruido. Pero no solamente se quiere que sea preciso el sistema, sino que el usuario quiere que esa búsqueda sea rápida y saber cuánto antes lo que está escuchando.

Para que esto sea posible, se va a investigar una serie de algoritmos que permitan reducir este tiempo de búsqueda sin que afecte, o que afecte lo menos posible, a la precisión del sistema inicial. Además, debido a la gran variedad de aplicaciones que puede tener este tipo de sistemas, es necesario hacer un análisis con distintos tipos de condiciones, así como con diferentes dispositivos móviles, para así crear una solución que abarque el mayor número de situaciones posibles.

1.2 Objetivos

El objetivo general de este trabajo es el de optimizar el rendimiento del sistema de indexado y búsqueda de fragmentos de audio grabados con el micrófono de un móvil, para así conseguir una búsqueda mucho más rápida y un algoritmo más escalable, esto es, que este algoritmo nos permita hacer búsquedas en mayores cantidades de contenidos de audio sin que aumente linealmente el tiempo de búsqueda.

En primer lugar, se evaluará la precisión y eficiencia del sistema de partida. Para ello calcularemos la precisión en las búsquedas y el tiempo que tarda en detectar un fragmento de audio grabado en la base de datos.

En segundo lugar, aplicaremos una serie de algoritmos con los que modificaremos el programa de partida para que este tiempo de búsqueda sea inferior. Para ello

modificaremos la forma en la que se organiza la base de datos y la forma en la que se busca el fragmento de audio grabado en la base de datos.

Finalmente, el resultado esperado será un sistema en el que la tasa de acierto seguirá siendo elevada con una precisión de pocos milisegundos y el tiempo de búsqueda será mucho menor. Y mucho más importante, el tiempo de búsqueda no aumentará linealmente con el tamaño de la base de datos de búsqueda como hasta ahora, lo que permitirá mejorar la escalabilidad del algoritmo.

1.3 Organización de la memoria

Capítulo 1: Introducción

En este capítulo se detallan las motivaciones por las cuales se ha llevado a cabo este TFG, así como los objetivos principales que se pretenden conseguir.

Capítulo 2: Estado del arte

Este capítulo contiene un estado del arte detallado sobre las técnicas de recuperación de información musical, así como técnicas de robustez frente al ruido y una descripción detallada del programa de partida.

Capítulo 3: Diseño y Desarrollo

En este capítulo se proponen todas las mejoras que se incorporan al programa. Primero se comienza con una descripción detallada de la teoría de árboles K-d y K-Means. Posteriormente se introducen dos tipos de algoritmos con los que se tratará de optimizar el tiempo de búsqueda. Finalmente, se detallará la librería usada, los ficheros de audio que se utilizarán para las pruebas y el nuevo diseño que tendrá el programa.

Capítulo 4: Pruebas y Resultados

En este capítulo es en el que se ponen a prueba todas las mejoras incluidas en el apartado anterior. Primero se evaluará el potencial del programa de partida, tanto tiempos de ejecución como el porcentaje de acierto. Posteriormente se realizarán pruebas con cada uno de los algoritmos propuestos en el apartado anterior y se comprobará si se logra la mejora deseada y se alcanzan los objetivos deseados.

Capítulo 5: Conclusiones y trabajo futuro

En este último capítulo se exponen las conclusiones obtenidas en este TFG. Se hará una breve mención a los aspectos que puedan ser estudiados en un trabajo futuro, con el objetivo de mejorar las prestaciones del programa desarrollado.

2 Estado del arte

2.1 Recuperación de la información de la señal de audio

La señal de audio es una gran fuente de información y es posible extraer gran cantidad de información de ella. Existen tecnologías que se dedican a extraer información de estas señales, cada una de las cuales se centra en extraer un tipo de información distinta. Este tipo de tecnologías se denominan ASC (Audio Signal Classification), las cuales preprocesan la señal y mediante distintas técnicas se obtienen una serie de características. En general se emplean técnicas de reconocimiento de patrones.

La señal de audio se puede clasificar de muchas maneras en función del contenido que interese, y por ello existe un tipo de aplicación de extracción de información de audio para cada información específica. Una de las aplicaciones más conocidas son la de reconocimiento de música o de voz, un claro ejemplo es la aplicación Shazam, que se explicará más adelante. También se puede clasificar en función del género musical, identificar acentos, idiomas, etc.

Los sistemas que analizan el audio en tiempo real se caracterizan por tener una respuesta rápida tras analizar el audio de entrada, este procesado suele ser costoso computacionalmente, por lo que para que sea efectivo se trabaja con cadenas de datos relativamente cortas. Por eso se descompone el espectro de la señal en unos parámetros que lo caractericen fielmente, existen muchos tipos distintos de coeficientes, pero los más usados debido a su robustez y precisión son los denominados coeficientes MFCC (Mel Frequency Cepstral Coefficients.) Estos parámetros se obtienen analizando la señal de audio mediante el banco de filtros MEL [14].

2.1.1 Sistemas de búsqueda de audio en audio

Este tipo de sistemas de búsqueda tienen como objetivo la búsqueda de coincidencias de dos pistas de audio. Es un tipo de sistema ASC en el que se clasifican los audios mediante la similitud entre ellos. La idea de estos sistemas es conseguir una coincidencia entre el audio sin perturbaciones y el mismo audio contaminado por ruido y distorsión.

Existen distintos tipos de sistemas de búsqueda de audio en audio:

- **Sistemas de búsqueda unitaria**

Son aquellos sistemas que realizan una búsqueda de una señal de audio que contenga una muestra grabada, y la respuesta es una coincidencia entre ellos.

En este tipo de sistemas se encuentran aquellos que reconocen señales musicales como la aplicación Shazam o Soundhound.

- **Sistemas de sincronismo**

Es un sistema similar a los mencionados anteriormente, pero no solamente indican una coincidencia probable, sino un instante de tiempo de ocurrencia de una muestra sobre la original. En este tipo de sistemas destaca Audible Magic y su producto Media Synchronization. El sistema de este trabajo de fin de grado pertenece a este grupo.

2.1.2 Shazam

Shazam es una aplicación para móviles cuya funcionalidad es la identificación de música. Con el micrófono que viene incorporado en un dispositivo portátil (típicamente Smartphone o Tablet) se graba una pequeña muestra del audio y con ella se crea una huella digital (fingerprint) y se compara con su base de datos hasta encontrar una coincidencia, en el que caso que la haya. Si la búsqueda tuvo éxito, se recibe información acerca de la canción en cuestión y la posibilidad de comprarla o compartir tu búsqueda.

La detección que realiza este sistema se basa en analizar espectrogramas. Para cada canción se obtienen los picos de energía más significativos obteniendo una constelación de puntos para cada una. Estos picos en el espectro serán los menos vulnerables a señales indeseadas, tales como el ruido o distorsión.

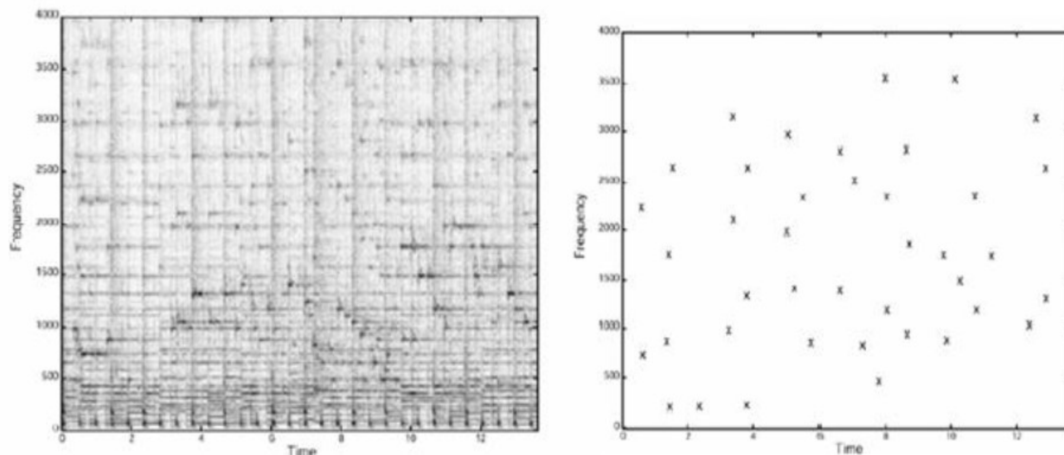

Figura 2.1: Constelación de picos de un espectograma. [10]

Por lo tanto, la búsqueda se reduce a encontrar una constelación de puntos dentro de la base de datos propia de Shazam. Si se produce una serie de correspondencias alineadas en el tiempo, Shazam calcula un histograma que representa la latencia de dichas correspondencias, si muestra un valor máximo muy definido será que se ha encontrado una coincidencia.

2.1.3 Audible Magic

AudibleMagic es una compañía que desarrolla software de reconocimiento de contenido multimedia (ACR, Automatic Content Recognition) [11]. Se creó con el objetivo de aportar nuevas técnicas para la identificación de audio, desde identificar cualquier tipo de audio hasta la sincronización.

La aplicación Media Synchronization permite sincronizar audio mediante la captura del mismo mediante el micrófono de un teléfono móvil. Se basa en la sincronización de una señal de audio desconocida con un conjunto de muestras de referencia que contiene la base de datos. Permite la activación y seguimiento de actividades en el tiempo, así como la participación interactiva del usuario y la aparición de publicidad en tiempo real. El sistema es bastante rápido y preciso, teniendo una resolución de 25ms, lo que corresponde aproximadamente a un fotograma.

2.2 Robustez frente al ruido

El ruido es uno de los factores que corrompen fácilmente cualquier señal de audio, degrada la calidad de la señal y puede acarrear grandes consecuencias negativas en el procesado de la misma. Se han desarrollado diversas técnicas que tienen como objetivo eliminar la mayor cantidad de ruido posible.

A continuación se explicará brevemente los diversos tipos de ruido:

- **Aditivo**
  Como ruido aditivo, se considera todo tipo de ruido que proviene de cualquier tipo de fuente que coexiste en el mismo entorno, y que se puede sumar tanto directamente a la señal como en el canal, este último debido a la no idealidad del mismo.

- **Convolutivo**
  Cuando una señal pasa por un canal no ideal se suele producir distorsiones que en general se pueden modelar como la convolución de la señal con la respuesta al impulso del canal, por lo que no es una señal que se añada a la original.

Para combatir este tipo de ruido en las señales de audio resulta bastante eficiente aplicar la técnica conocida como CMVN (Cepstral Mean and Variance Normalization) [12], ya que tiene como objetivo abordar el problema del ruido aditivo y convolutivo.

El ruido convolutivo se traduce en un desplazamiento constante de los parámetros MFCC de la señal, por lo que la idea será eliminar ese desplazamiento, esto se consigue substrayendo la media del vector. Este desplazamiento se elimina mediante la técnica CMN (Cepstral Mean Normalization). Con la técnica CMVN se tienen en cuenta también las varianzas, aparte de eliminar ese desplazamiento, se consigue un valor unitario de varianza y se compensa parte del efecto del ruido aditivo.

2.3 Programa de partida

El objetivo del programa es la búsqueda y localización de un fragmento de audio en una base de datos de audio. Por un lado tenemos la base de datos, formada en nuestro caso por una película de alrededor de dos horas de duración. Por otro lado tenemos una muestra grabada a través del micrófono de un móvil de un fragmento de dicha película. La grabación se realizó en una sala de cine e intencionadamente con una gran cantidad de ruido.

Primero se va a explicar el programa básico de partida para entender mejor cómo funciona realmente el programa.

2.3.1 Programa básico

La forma en la que se van a tratar estas señales es bastante parecida. Para ello se generarán huellas espectrales (*fingerprints*) de estos audios, estos *fingerprints* están formados por conjuntos de vectores de coeficientes MFCC. El número de coeficientes que normalmente se trabaja para las técnicas de reconocimiento de voz es de trece, por lo que este será el número de coeficientes que se emplearán en el programa.

El *fingerprint* del audio original se procesa de forma offline, ya que esta huella se generará al principio del programa para así poder usarla como nuestra base de datos, será el *fingerprint* en el que se buscarán las muestras de audio grabadas. Los *fingerprints* de las muestras grabadas se procesan de forma online y tendrán un tamaño mucho menor que el del original, ya que se tomarán ocho segundos de las muestras grabadas y se buscarán en el *fingerprint* original.

La respuesta que da el programa es un valor de tiempo del audio original, es decir, nos indica el momento en el cual ambas muestras de audio son más parecidas, este momento es medido desde el inicio del audio original.

Si en lugar de tomar como audio original una única grabación se concatenasen muchas grabaciones diferentes, el instante de tiempo nos indicaría qué grabación se ha reproducido y en qué instante de tiempo de dicha grabación estaríamos, consiguiendo por tanto un funcionamiento similar al de un programa como Shazam.

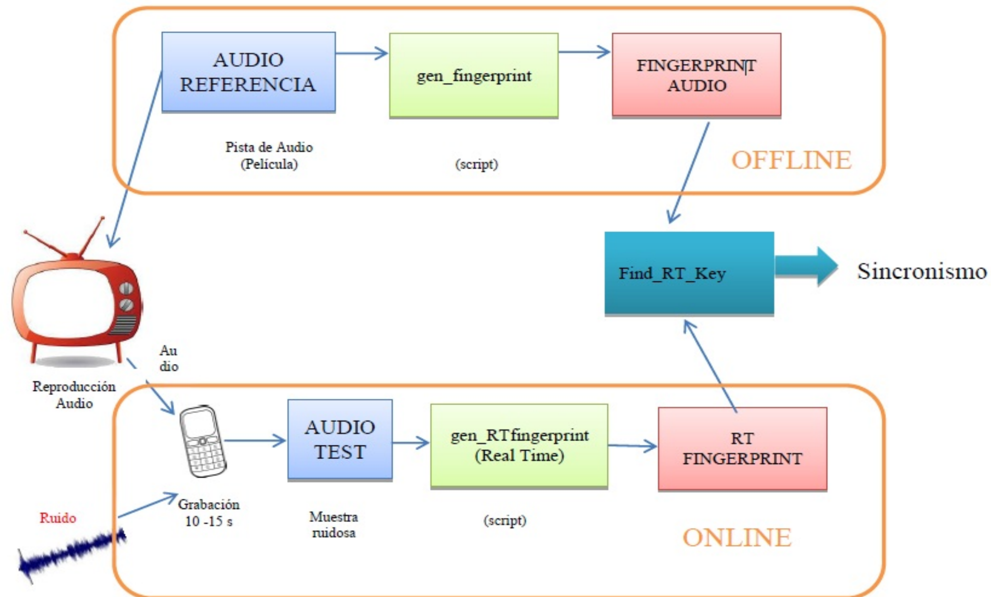

Figura 2.2: Esquema básico del sistema de partida. [1]

2.3.2 Generación de los *fingerprints*

Como se ha comentado en el apartado anterior, los *fingerprints* son ficheros de datos que contienen los vectores de los coeficientes MFCC. Para calcular estos vectores se aplican ventanas de 32 ms. solapadas al 50%, y se calculan los coeficientes MFCC cada 16ms, más adelante explicaré el porqué de este valor. Como el audio original tiene un tamaño mucho mayor que el de la muestra grabadas, para que el tamaño del *fingerprint* no sea excesivamente grande, se calcula un promedio de los MFCCs en una ventana deslizante de 0.8 s. con desplazamientos de 0.4 segundos. Respecto a la muestra grabada, se calcula también el promedio en una ventana deslizante de 0.8s, pero con desplazamientos de 16 ms, lo que es una precisión bastante aceptable.

En la siguiente figura se muestra el esquema de cómo el programa genera los *fingerprints*.

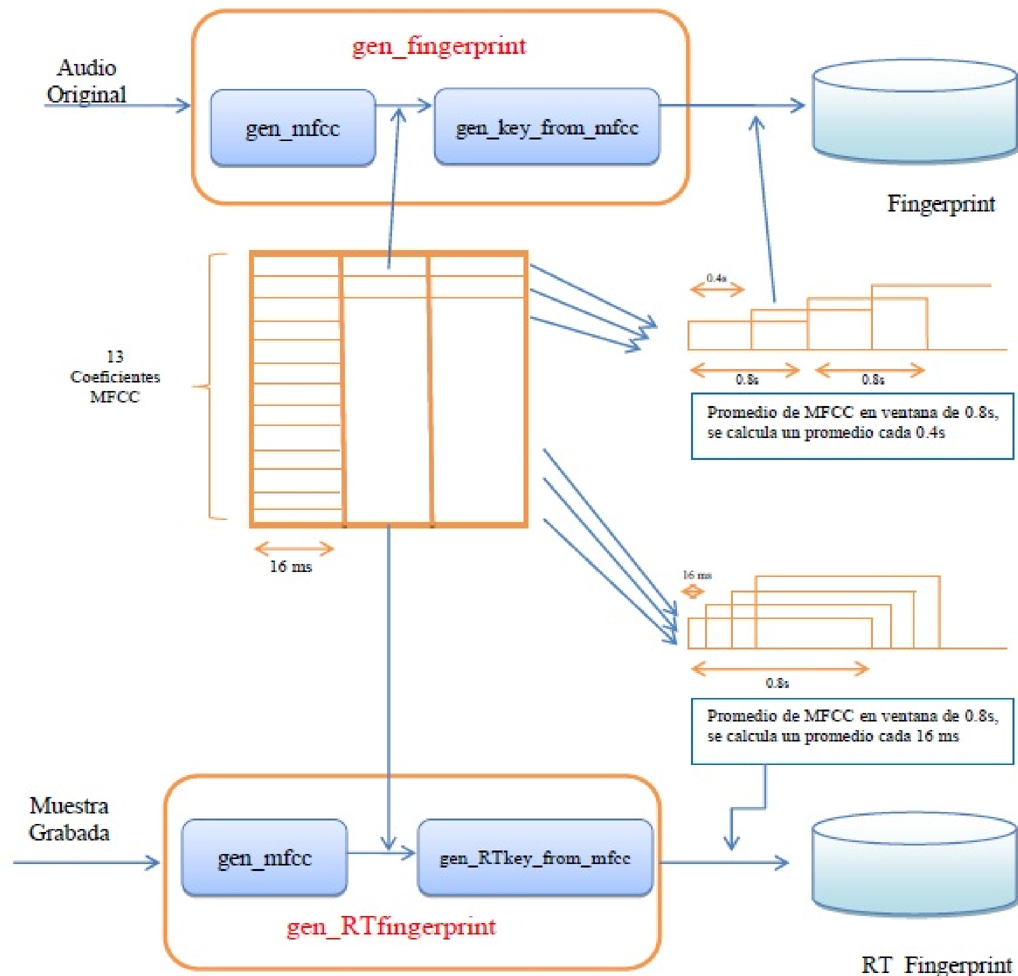

Figura 2.3: Generación de los fingerprints. [1]

2.3.3 Algoritmo de búsqueda

El algoritmo que se emplea en el programa es un algoritmo de búsqueda lineal, esto es, calcula todas las distancias entre cada par de vectores. En el audio original se calculan promedios cada 0.4s y en las muestras grabadas cada 16ms, por lo que cada promedio del audio original será comparado con los 25 (= 0.4s / 0.016s) promedios de las muestras grabadas.

La rapidez de este algoritmo de búsqueda no es bastante satisfactoria, ya que se buscaba una alta precisión y no una gran velocidad de cálculo. En esto último es en lo que nos centraremos en este trabajo, en reducir la velocidad manteniendo una buena precisión.

Las búsquedas lineales que realiza este algoritmo entre el audio original y las muestras grabadas nos da como resultado la distancia mínima. Se ha experimentado previamente con dos tipos de distancias: Distancia City Block y Distancia Euclídea. En estudios previos

no se encontraron grandes diferencias entre estas distancias, por lo que se tendía a utilizar distancia City Block por ser más eficiente.

A continuación se muestra un gráfico del algoritmo de búsqueda lineal.

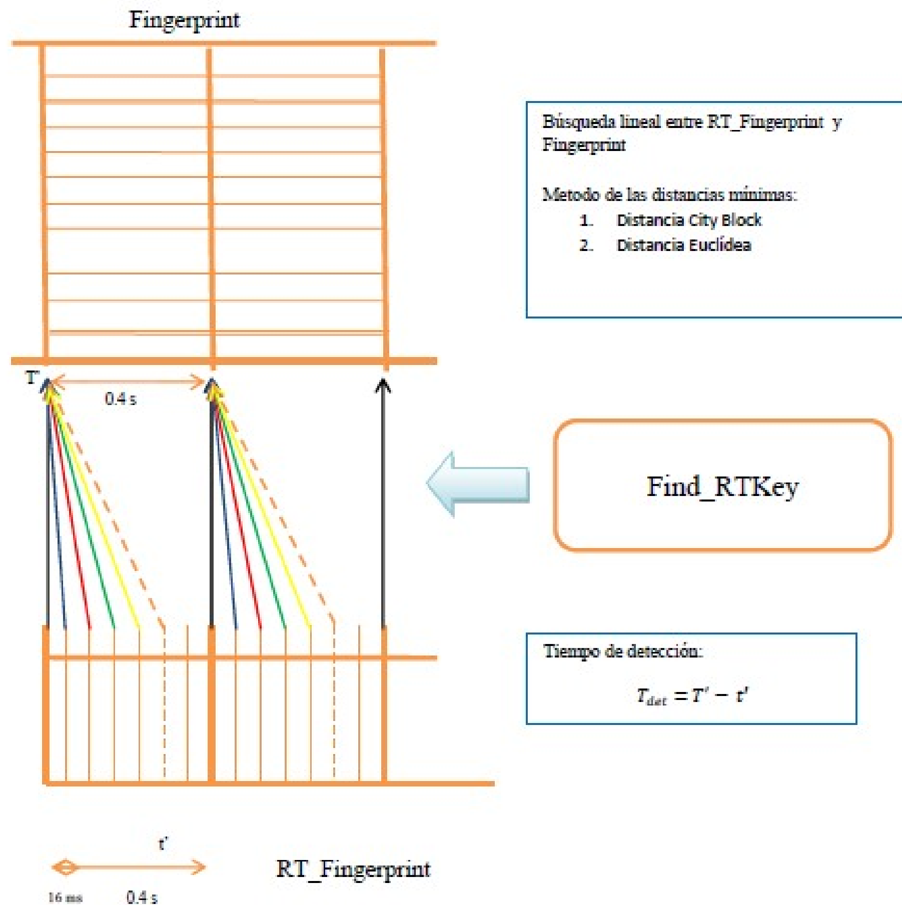

Figura 2.4: Algoritmo de búsqueda propuesto. [1]

2.3.4 Técnicas de robustez frente al ruido en el programa de partida

Inicialmente se partió de un programa sin ningún tipo de robustez frente al ruido [1], por lo que la precisión y acierto de las muestras grabadas en entornos ruidosos era muy baja. Para mejorar esto, se añadieron varios tipos de mejoras, estas son: Técnicas de normalización y un algoritmo propio que estudia el comportamiento de las trayectorias formadas por las detecciones.

En este apartado se van a describir en detalle todos estos tipos de mejora aplicadas.

2.3.4.1 CMVN sobre los coeficientes MFCC

En esta primera mejora se aplicó la técnica CMVN (*Cepstral Mean and Variance Normalization*) [12], con ella se realiza una transformación lineal a los vectores de los coeficientes MFCC, el objetivo es conseguir que los estadísticos de estos vectores tengan varianza unitaria y media cero. Gracias a esta técnica se reduce mucho el efecto que tiene el ruido en las muestras grabadas, sobre todo se reduce el ruido estacionario y gaussiano, este último es el que tiende a desplazar la media y la varianza.

La normalización que se aplica se lleva a cabo sobre cada ventana, como cada ventana tiene una duración de 0.8 segundos y los coeficientes MFCC se calculan cada 16 ms, tenemos que esto equivale a 50 frames. Por lo tanto, se realiza la normalización de cada vector de coeficientes de cada frame.

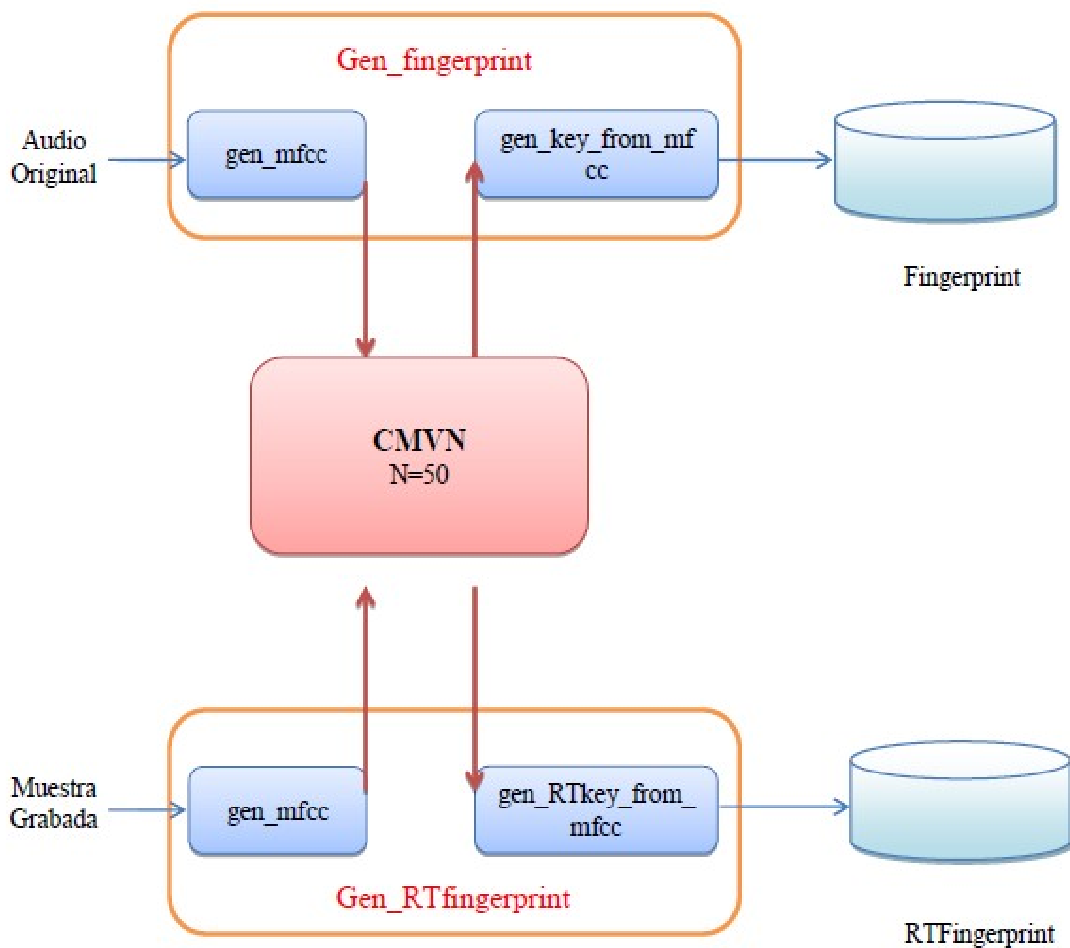

Figura 2.5: Aplicación de la técnica CMVN sobre los coeficientes MFCC. [1]

2.3.4.2 CMVN sobre los fingerprints

En esta segunda mejora se aplicó la misma técnica comentada en el apartado anterior, pero aplicada sobre los *fingerprints* generados. Con esto se consiguió una doble normalización, compensando en mayor medida el efecto del ruido en las muestras grabadas. Debido a esta mejora, se consiguieron unos resultados mucho mejores que solamente aplicando la técnica CMVN sobre los coeficientes MFCC.

En este caso el número de frames sobre los que se va a normalizar depende de cada *fingerprint*, es decir, el número de frames para el audio original será mucho menor que el de la muestra grabada. Y en este caso la ventana no es de 0.8 segundos sino de 8 segundos, ya que el tamaño del audio de las muestras grabadas es de esta duración.

Recordemos que para el audio original calculamos promedios cada 0.4s, por lo que para saber la cantidad de frames sobre los que normalizar, basta con hacer el siguiente cálculo:

$$8s / (0.4s/frame) = 20 \text{ frames}$$

Y para el audio de las muestras grabadas se calculan promedios cada 16ms, por lo tanto, si hacemos la operación de arriba pero con este nuevo valor tenemos:

$$8s / (0.016s/frame) = 500 \text{ frames}$$

Finalmente, lo que tenemos es que para el audio original aplicaremos la técnica sobre 20 frames y para las muestras grabadas sobre 500 frames.

En la siguiente figura se muestra el esquema final del programa de partida.

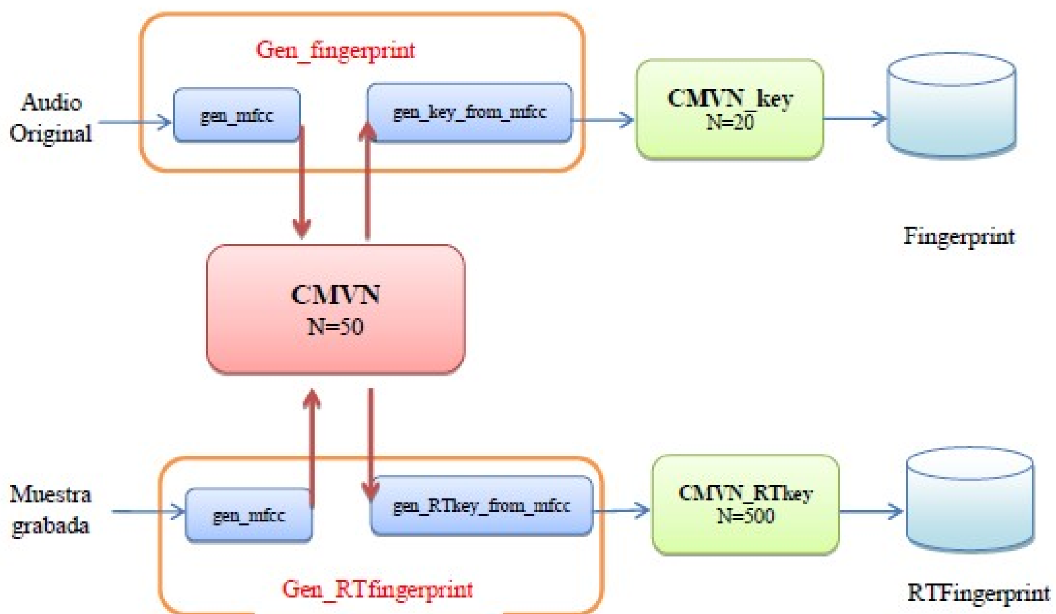

Figura 2.6: Aplicación de la técnica CMVN sobre los fingerprints. [1]

2.3.4.3 Análisis de trayectorias temporales

Respecto a esta última mejora, el programa inicialmente calculaba la distancia más pequeña, pero esto no aseguraba que la detección fuese correcta. Con este análisis ahora se tienen en cuenta las M-Best distancias más pequeñas, con esto se puede averiguar si en una detección que se ha calculado como errónea, la detección que en realidad sería la correcta se encuentre dentro de las M distancias más probables, y si es así, se aplica un algoritmo basado en el análisis de trayectorias de las detecciones obtenidas. Se obtendrá como resultado una distancia que más se adapte a esa detección, aunque no tiene porqué ser la distancia más pequeña.

Para entender mejor el porqué de la incorporación de esta mejora, se va a explicar su funcionamiento paso a paso.

- Para las M-Best detecciones se define una ventana deslizante de N frames, la longitud de esta ventana dependerá de la trayectoria que se va a estudiar.

- Se calculan todas las trayectorias posibles que siguen una evolución lineal dentro de la ventana creada.

- Se calcula la distancia mínima total de todas las trayectorias obtenidas.

- Finalmente, la salida que se obtiene será la detección cuya distancia mínima total sea la más pequeña.

Debido a este análisis podemos conocer la detección correcta cuando se detectó una detección incorrecta. Si no se consigue ninguna trayectoria disponible, lo más seguro es que esa detección sea errónea por un nivel de ruido elevado, por lo que lo más prudente será no ofrecer ningún resultado y grabar algo más de audio para tratar de encontrar una detección correcta.

3 Diseño y Desarrollo

Este TFG consiste de varias fases. Primero se analizará el programa de partida completo, tanto la precisión de los resultados como el tiempo de ejecución total. Después se analizarán las técnicas de cálculo eficiente de vecinos más cercanos y en particular se analizará la librería FLANN (*Fast Library for Approximate Nearest Neighbors*) [6], [7] y [8], esta contiene dos tipos de algoritmos que se explicarán en este apartado. Una vez analizadas estas técnicas se procederá a aplicarlas al sistema de partida con alguna ligera modificación en la organización de los fingerprints, con ello se tratará de acelerar la búsqueda sin mermar en exceso la precisión. Por último, se evaluarán los resultados comparando los tiempos y precisión del sistema de partida con el sistema optimizado.

Para que la búsqueda sea más rápida, en vez de hacer una búsqueda exhaustiva, como hace el sistema de partida, se tratarán dos tipos de árboles para hacer una búsqueda aproximada de vecinos más cercanos, uno es el árbol K-d que explicaremos en este apartado, y el otro es el árbol K-Means que explicaremos más adelante. El porqué del uso de árboles es debido a que será la forma en la que se organice la base de datos, por lo que a la hora de buscar las muestras grabadas en el audio original, el tiempo de búsqueda será menos que el tiempo de búsqueda del programa de partida.

3.1 Binary Search Tree (BST)

Antes de entrar a explicar cómo es y cómo funciona un árbol K-d, es necesario explicar brevemente otro tipo de árbol, denominado Árbol Binario de Búsqueda (*Binary Search Tree, BST*).

Un Árbol Binario de Búsqueda consiste en un árbol cuyos nodos tienen como mucho dos hijos. El subárbol izquierdo de cualquier nodo, siempre que no esté vacío, contiene valores menores que el que contiene dicho nodo, y el subárbol derecho, también siempre que no esté vacío, contiene valores mayores. Dicho de otra forma, si el nodo raíz es un 5, todos los nodos en el subárbol izquierdo tendrán valores menores de 5, y en el subárbol derecho estos valores serán mayores.

El interés de este tipo de árboles radica en que en ellos se realiza un recorrido en orden, primero se trata el subárbol izquierdo, después el nodo actual, y por último el subárbol derecho. Debido a esto la búsqueda de algún elemento en el árbol suele realizarse de forma bastante eficiente.

3.2 Árboles K-d

Una vez que se conoce cómo se almacenan los datos en los Árboles Binarios de Búsqueda, se va a mostrar un ejemplo en tres dimensiones de cómo se almacenan los datos en un árbol del tipo K-d como muestran los artículos [2] y [3].

La siguiente figura muestra el árbol que se usará como ejemplo.

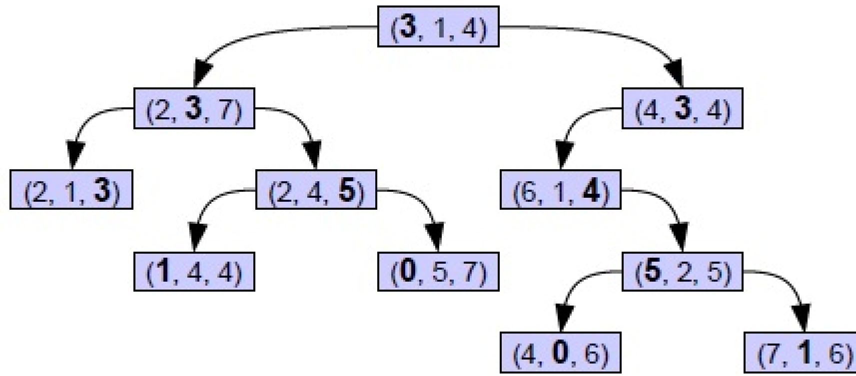

Figura 3.1: Ejemplo de árbol K-d. [3]

Se ha remarcado en negrita una componente de cada nodo en cada nivel. La razón por la cual se han remarcado estos valores, es porque al tener los nodos varias dimensiones, se discrimina en cada nivel el espacio remarcado. Por ejemplo, en este caso se ha remarcado la primera componente del nodo raíz, por lo que todas las primeras componentes del subárbol izquierdo serán menores y en el derecho serán al menos mayores. Lo mismo ocurre según se vaya bajando de nivel en el árbol, pero con distintas componentes.

La razón por la que estos árboles almacenan sus datos de esta manera es debido a un gran significado geométrico, en el que explotando esta estructura es posible realizar la búsqueda de vecinos más cercanos de forma extremadamente eficiente (en un tiempo mejor que O(n)).

Ahora tenemos el siguiente conjunto de datos, y con él se quiere construir un Árbol Binario de Búsqueda con esos puntos.

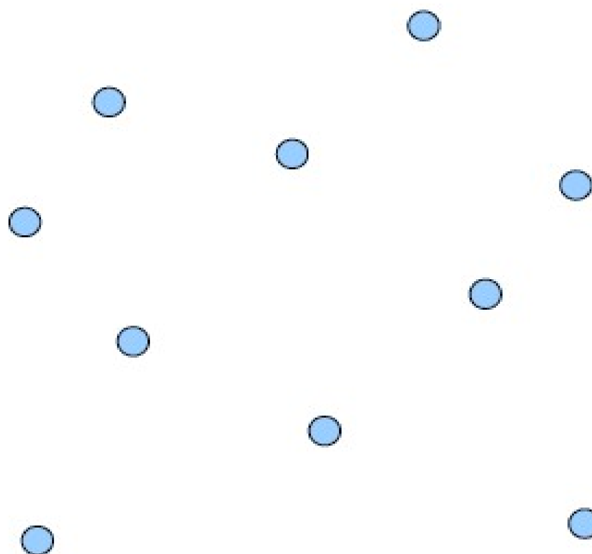

Figura 3.2: Conjunto de datos para construir el árbol K-d. [3]

Si construimos el árbol de la manera que se ha explicado anteriormente, no es fácil saber qué puntos en el espacio son mayores o menores que nuestro nodo elegido como raíz. Por lo tanto se procederá de la siguiente manera.

Se escoge un nodo cualquiera como nodo raíz y se divide el plano en dos regiones con una línea recta cualquiera atravesando el nodo elegido. Si se hace este proceso recursivamente con el resto de nodos se conseguirá nuestro Árbol Binario de Búsqueda. Esta técnica es conocida como *Binary Space Partitioning (BSP)* [13], por lo que un Árbol K-d, más que ser un Árbol Binario de Búsqueda es un tipo de Árbol BSP.

Este tipo de árboles no se limitan solamente a espacios bidimensionales, sirven también para espacios más grandes. En concreto, cuando se divide una dimensión, generalmente se denomina *splitting hyperplane*.

Ahora que se conoce el funcionamiento de los Árboles K-d, se va a proceder a construir el árbol mediante el conjunto de datos proporcionado.

Se elige un nodo cualquiera como nodo raíz y se divide el conjunto de datos en dos a través de un hiperplano vertical, en el subárbol izquierdo se encuentran los nodos con la componente en x menor que el nodo raíz, y en el subárbol derecho el valor de la componente en x es mayor.

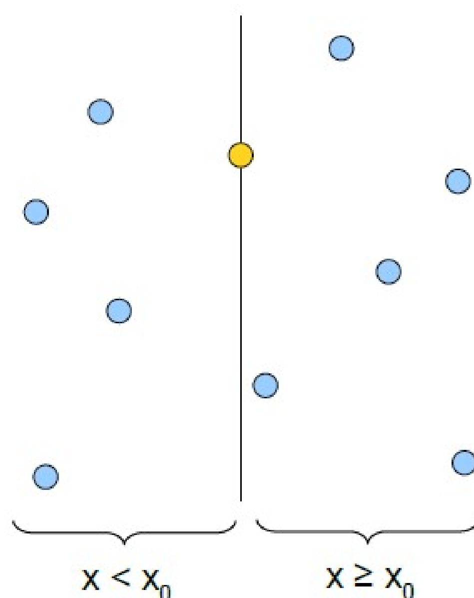

Figura 3.3: Nodo raíz del árbol K-d. [3]

A continuación se construirá la parte del subárbol derecho. De los nodos del subárbol derecho se elige nuevamente uno aleatoriamente y se divide el conjunto de datos mediante un hiperplano horizontal. En este caso es horizontal porque se dividen los datos a través de la componente y.

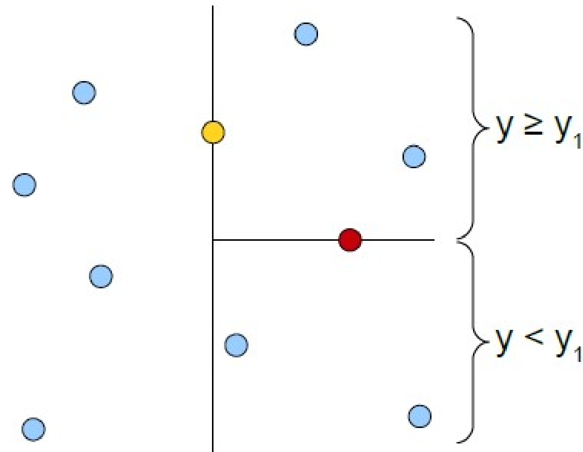

Figura 3.4: Creación del subárbol derecho. [3]

Por último se realizan todos estos pasos hasta que no que queden más nodos. El resultado es el siguiente.

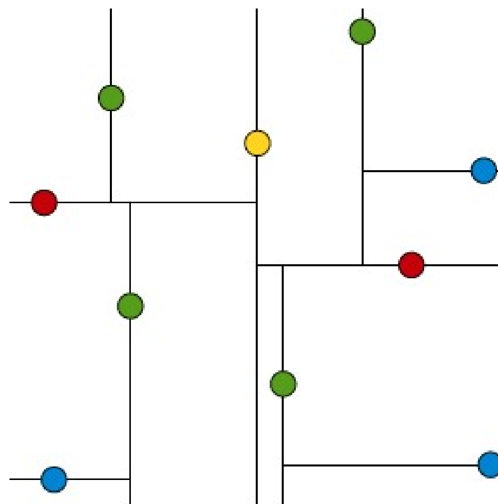

Figura 3.5: Árbol K-d final. [3]

Para entender mejor el significado geométrico de este tipo de árboles, se va a proceder a buscar un punto en particular en el árbol. Por ejemplo, se elige el nodo de abajo a la derecha. Como se sabe que este nodo se encuentra en el subárbol derecho, se pueden ignorar todos los puntos correspondientes al subárbol izquierdo, por lo que solamente se tiene que explorar el subárbol derecho.

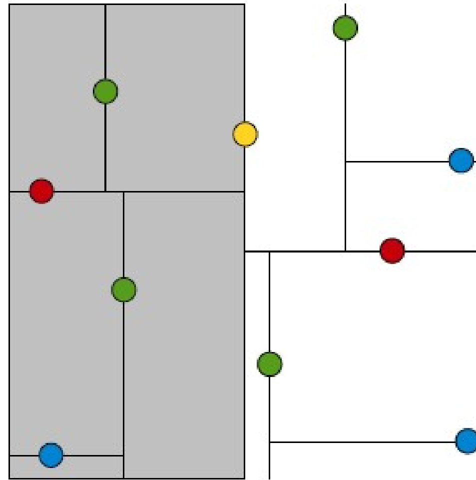

Figura 3.6: Eliminación del subárbol izquierdo en la búsqueda. [3]

En este subárbol derecho se busca si el nodo a encontrar está en el subárbol de arriba o de debajo. Como se sabe que el nodo es el de debajo a la derecha, se descarta el subárbol superior.

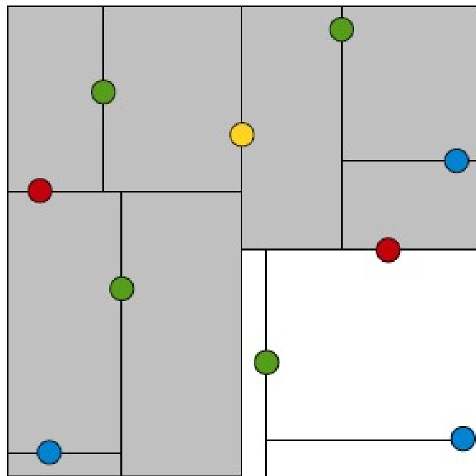

Figura 3.7: Eliminación del siguiente subárbol. [3]

Por último, solamente falta ver si el nodo se encuentra en el nuevo subárbol izquierdo o derecho. Como se encuentra en el subárbol derecho, se descarta el izquierdo. El gráfico final es el siguiente.

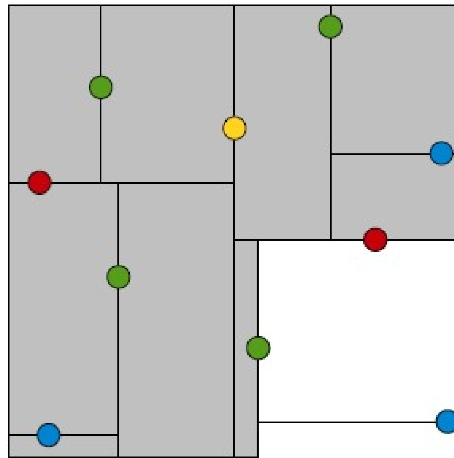

Figura 3.8: Resultado final de la búsqueda. [3]

En este punto se ha alcanzado el nodo que se quería encontrar y por lo tanto el algoritmo de búsqueda ha finalizado.

3.2.1 Vecinos más cercanos en Árboles K-d

Una vez se conoce el método de búsqueda de un punto del conjunto de la base de datos, ahora se explicará la manera en la que se hace una búsqueda de un punto cualquiera. Para ello se aplicará el método de los vecinos más cercanos.

Dado un árbol K-d y un punto cualquiera en el espacio, a este punto se le denominará punto de test, se trata de buscar qué punto del árbol es el más cercano al punto de test elegido.

A continuación se muestra un ejemplo de cómo funciona la búsqueda.

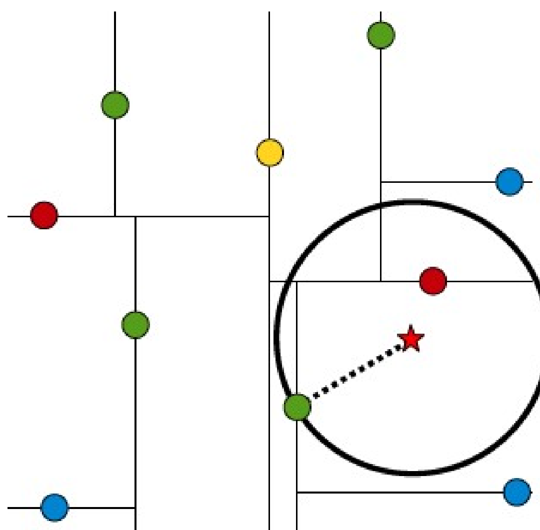

Figura 3.9: Radio de búsqueda para encontrar los vecinos más cercanos. [3]

Se tiene el punto de test, en este caso lo marcamos con una estrella, y se construye una hiperesfera centrada en el punto de test y que cruce por el punto del conjunto de datos. El vecino más cercano debe de encontrarse dentro de esa hiperesfera.

Si la hiperesfera se encuentra completamente en un lado del hiperplano, todos los puntos al otro lado del hiperplano no podrán ser los vecinos más cercanos del punto de test y se eliminarán. Si la hiperesfera cruza algún hiperplano, se tendrán que comprobar todos los correspondientes subárboles que corte y ver si los vecinos más cercanos están dentro de la hiperesfera.

Este tipo de algoritmo se ejecuta en tiempo del orden de O(log n), siendo n el número de nodos del conjunto de datos inicial y un árbol K-d construido de forma aleatoria.

3.3 Árboles K-Means

El algoritmo K-Means [4], también conocido como el algoritmo de Lloyd generalizado, es un método comúnmente usado para particionar automáticamente un conjunto de datos en K grupos. Se eligen K vectores que serán los centroides de cada grupo y, el paso siguiente, es redefinir los nuevos centroides iterativamente de la siguiente manera:

- Se asigna a cada vector su centroide más próximo,
- Se calculan de nuevo los nuevos centroides, hallando su media y asignando de nuevo cada vector a cada centroide,
- El algoritmo converge cuando no haya nuevas asignaciones.

Una regla que se debe cumplir, es que el número elegido de grupos, que es K, debe ser menor que el número total de un conjunto de datos dado. Ya que si se escoge un K muy cercano al número total, se tendrá un grupo por cada dato y no sería eficiente.

También la forma en la que se escogen los centroides inicialmente hará que se tengan grupos distintos, dependiendo de qué método se escoja. Como se ha comentado anteriormente, normalmente la elección de estos centroides se realiza de forma aleatoria, pero más adelante se verá que también hay otros tipos de métodos distintos.

A continuación se muestra un ejemplo básico de cómo funciona el algoritmo K-Means.

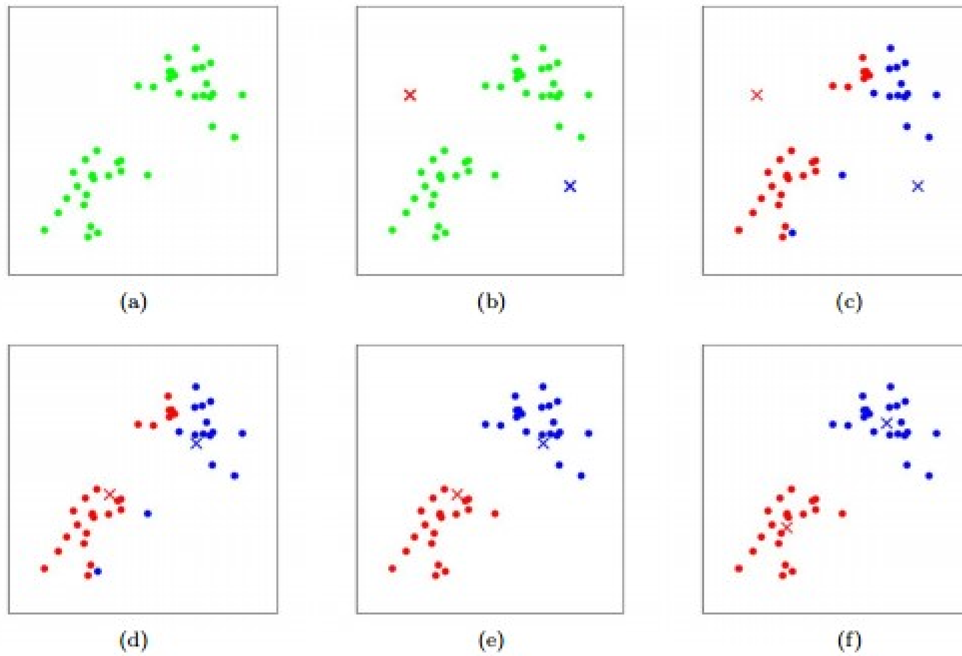

Figura 3.10: Funcionamiento del algoritmo K-Means. [5]

Se puede observar en la figura adjunta lo explicado sobre el algoritmo K-Means. Se tiene un conjunto de datos (a); se eligen aleatoriamente los centroides, en este caso se escogen dos (b); se procede a calcular iterativamente los nuevos centroides hasta que no haya nuevas asignaciones de vectores, teniendo dos grupos claramente diferenciados (c), (d), (e) y (f).

A la hora de crear el árbol K-Means, K indica el número de hijos que tendrá cada nodo del árbol, en vez de indicar el número de centroides. Por lo tanto, dado un conjunto de datos, el árbol se genera de la siguiente manera. Se divide el conjunto de datos en K grupos mediante el algoritmo K-Means. El mismo proceso se realiza recursivamente para cada conjunto de datos perteneciente a cada centroide, diviendo ese conjunto en otros K conjuntos. En el siguiente ejemplo gráfico se muestra el resultado de aplicar este algoritmo.

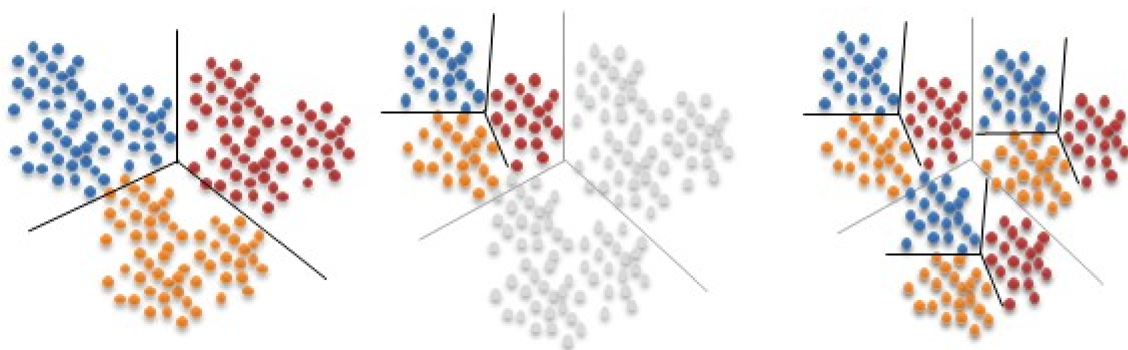

Figura 3.11: Ejemplo de árbol K-Means. [15]

3.4 Algoritmos escalables de vecinos más cercanos

En este trabajo se van a aplicar dos algoritmos de búsqueda rápida mediante el uso del método de vecinos más cercanos.

Como se ha explicado anteriormente, la búsqueda exacta de todos los puntos en grandes conjuntos de datos es demasiado costosa para la mayoría de aplicaciones, por ello se ha generado un gran interés en los algoritmos de búsqueda aproximada de vecinos más cercanos. En algunas ocasiones no se encuentran los vecinos más óptimos, pero la velocidad de búsqueda respecto a una búsqueda exhaustiva es bastante menor.

Para aplicar estos algoritmos se utilizará la librería FLANN (*Fast Library for Approximate Nearest Neighbors*) como se explica en [6] y [7]. En estos trabajos se evaluaron muchos tipos de algoritmos, pero se encontró que solamente dos dieron un mejor rendimiento: *The Priority Search K-Means Tree Algorithm* y *The Multiple Randomized K-d Trees Algorithm*.

A continuación se va a describir el funcionamiento de cada algoritmo.

3.4.1 The Multiple Randomized K-d Tree Algorithm

The Randomized K-d Tree Algorithm es un algoritmo de búsqueda aproximada de vecinos más cercanos que construye varios árboles K-d aleatorios que son buscados en paralelo. Estos árboles son construidos de forma similar a un clásico árbol K-d, con la diferencia de que el clásico divide los datos en la dimensión con mayor varianza, y para este algoritmo se elige aleatoriamente el número de dimensiones. Se fijó en 5 porque con este número de dimensiones se conseguía el mejor rendimiento [6].

```
Algorithm 1 Building the randomized k-d forest
Input: features dataset D, number of trees N
Output: the randomized kd-trees data structure
procedure BUILDRANDOMIZEDKDTREES(D,N)
 1: trees ← {}
 2: for i ← {1..N} do
 3:     trees_i ← BUILDKDTREE(D)
 4: end for
 5: return trees
procedure BUILDKDTREE(D)
 1: if |D| == 1 then
 2:     CREATELEAFNODE(D)
 3: else
 4:     axesMeans[1..d] ← mean of features in D along each dimension
 5:     axesVariances[1..d] ← variance of features in D along each dimension
 6:     maxVarianceAxes[1..N_D] ← top N_D dimensions with highest variance
 7:     splitDim ← random dimension from maxVarianceAxes
 8:     splitValue ← axesMeans[splitDim]
 9:     (D_1, D_2) ← split dataset on splitDim using splitValue
10:     T_1 ← BUILDKDTREE(D_1)
11:     T_2 ← BUILDKDTREE(D_2)
12:     CREATEINNERNODE(splitDim, splitValue, T_1, T_2)
13: end if
```

Figura 3.12: Construcción del árbol K-d. [6] y [7]

Cuando se busca en los árboles K-d aleatorios, una única cola de prioridad se mantiene a través de todos los árboles. La cola de prioridad es ordenada mediante el aumento de la distancia a la frontera de decisión de cada rama en la cola, por lo que la búsqueda explorará primero las hojas más cercanas de todos los árboles. Una vez que un punto del conjunto de datos ha sido examinado en un árbol, es decir, comparado con un punto de búsqueda, es marcada para no ser examinado en el resto de árboles. El grado de aproximación es determinado por el número máximo de hojas a visitar, devolviendo los vecinos más cercanos a ese punto de búsqueda.

Algorithm 2 Searching the randomized k-d forest

Input: randomized k-d trees T_i, query point Q, number of neighbours K, maximum number of points to examine L
Output: approximate K nearest neighbours of query point
procedure SEARCHKDTREES(T_i,Q,K,L)
1: $count \leftarrow 0$
2: $PQ \leftarrow$ empty priority queue
3: $R \leftarrow$ empty priority queue
4: **for** each tree T_i **do**
5: call TRAVERSEKDTREE(Q,T_i,PQ,R)
6: **end for**
7: **while** PQ not empty **and** $count < L$ **do**
8: $T \leftarrow$ top of PQ
9: call TRAVERSEKDTREE(Q,T,PQ,R)
10: **end while**
11: **return** K top points from R
procedure TRAVERSEKDTREE(Q,T,PQ,R)
1: **if** T is a leaf node **then**
2: add point in T to R
3: $count \leftarrow count + 1$
4: **else**
5: $splitDim \leftarrow T.splitDim$
6: $splitValue \leftarrow T.splitValue$
7: **if** $Q(splitDim) < splitValue$ **then**
8: $closestNode \leftarrow T.left$
9: $otherNode \leftarrow T.right$
10: **else**
11: $closestNode \leftarrow T.right$
12: $otherNode \leftarrow T.left$
13: **end if**
14: add $otherNode$ to PQ
15: call TRAVERSEKDTREE(Q,$closestNode$,PQ,R)
16: **end if**

Figura 3.13: Búsqueda en el árbol K-d. [6] y [7]

La siguiente figura muestra un ejemplo gráfico de cómo son los árboles K-d aleatorios.

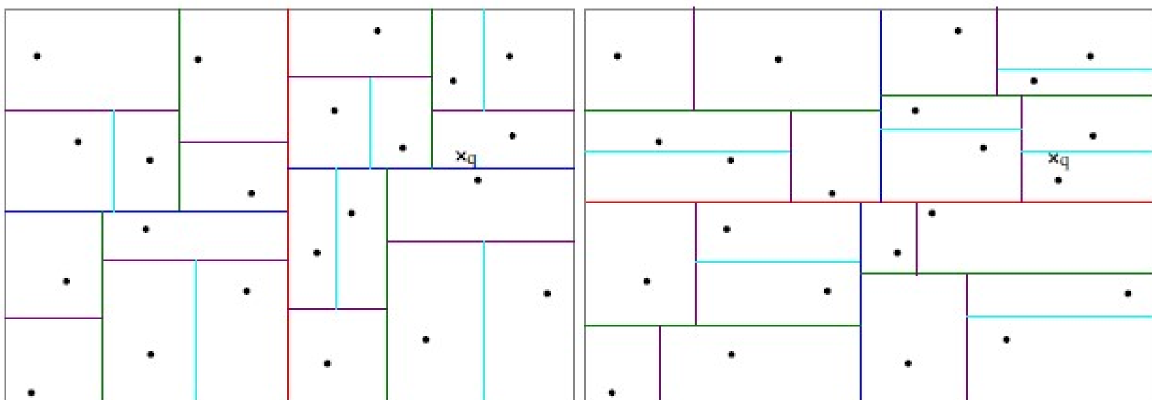

Figura 3.14: Ejemplos de árboles K-d aleatorios. [6] y [7].

Esta figura da una intuición de por qué explorar múltiples árboles K-d aleatorios mejora el rendimiento de búsqueda. Cuando nuestro punto de búsqueda está cercano a un hiperplano, su vecino más cercano residirá a cada lado del hiperplano con casi la misma probabilidad, pero si reside en el otro lado del hiperplano, se necesitará mayor exploración. Por lo tanto, usando múltiples descomposiciones aleatorias, incrementa la probabilidad de que el punto de búsqueda esté en la misma celda en varios árboles.

Respecto al rendimiento de este algoritmo, podemos ver en la siguiente figura como mejora respecto a una búsqueda lineal. El rendimiento mejora significativamente hasta cierto punto (alrededor de 20 árboles), a partir de ahí puede variar el rendimiento, aunque en general mejora un poco. Y respecto al uso de memoria, aumenta linealmente con el número de árboles, por lo que tampoco conviene crear muchos árboles K-d aleatorios.
Si queremos un alto grado de precisión, alrededor del 95%, el rendimiento será menor que si queremos un grado de precisión menor, por ejemplo, del 70%.

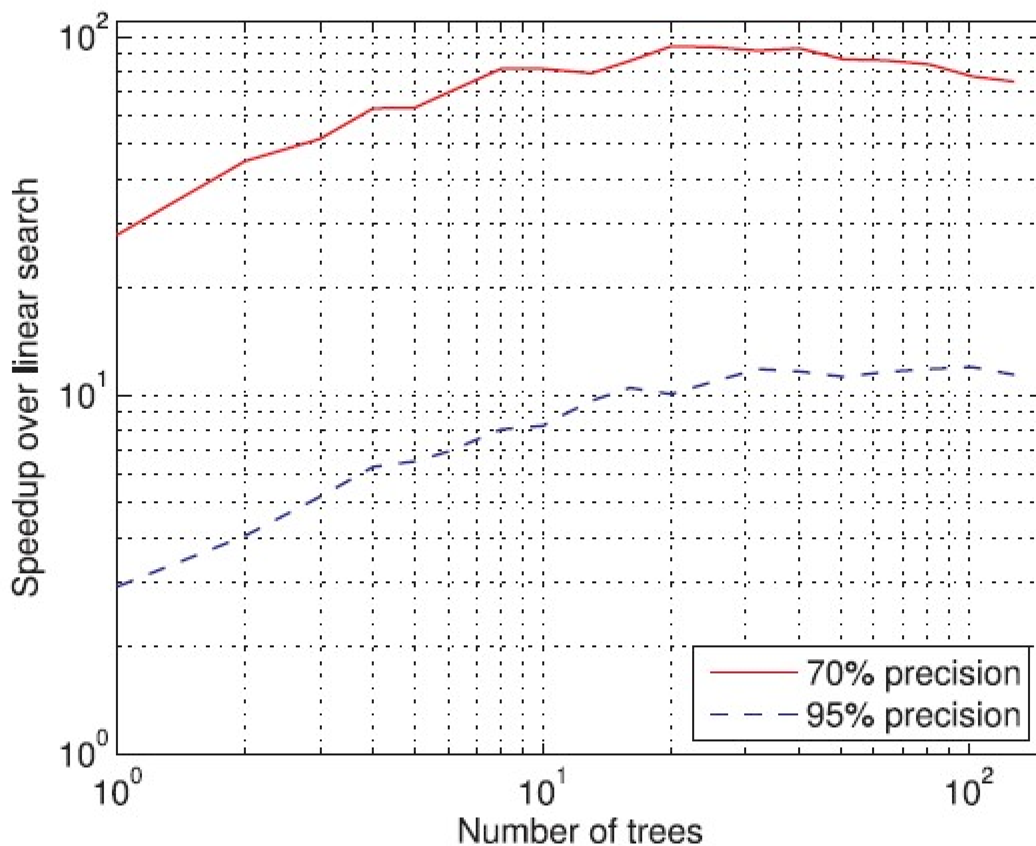

Figura 3.15: Velocidad obtenida usando múltiples árboles K-d. (Resultados obtenidos a partir de las pruebas realizadas por Marius Muja). [6] y [7].

3.4.2 The Priority Search K-Means Tree Algorithm

The Priority Search K-Means Tree Algorithm se ha encontrado muy efectivo en este tipo de situaciones, es decir, en situaciones en las que la base de datos es grande y se requieren unos resultados con una precisión alta [7].

Este algoritmo intenta explotar la estructura natural del conjunto de datos, agrupándolos usando la distancia completa en todas las dimensiones, a diferencia de los árboles K-d aleatorios que dividía los datos en una dimensión a la vez.

El árbol construido mediante este algoritmo divide el conjunto de datos en cada nivel en K regiones distintas usando el algoritmo K-Means, se repite el método recursivamente hasta que el número de puntos en una región es menor que K, como se muestra en la siguiente figura.

Algorithm 3 Building the priority search k-means tree

Input: features dataset D, branching factor K, maximum iterations I_{max}, centre selection algorithm to use C_{alg}
Output: k-means tree

```
1: if |D| < K then
2:     create leaf node with the points in D
3: else
4:     P ← select K points from D using the C_alg algorithm
5:     converged ← false
6:     iterations ← 0
7:     while not converged and iterations < I_max do
8:         C ← cluster the points in D around nearest centres P
9:         P_new ← means of clusters in C
10:        if P=P_new then
11:            converged ← true
12:        end if
13:        P ← P_new
14:        iterations ← iterations + 1
15:    end while
16:    for each cluster C_i ∈ C do
17:        create non-leaf node with center P_i
18:        recursively apply the algorithm to the points in C_i
19:    end for
20: end if
```

Figura 3.16: Construcción del árbol K-Means. [6] y [7].

Este algoritmo explora el árbol K-Means usando la estrategia *best-bin-first*, por analogía a lo que se ha encontrado para mejorar significativamente el rendimiento de las búsquedas de los árboles K-d.

Inicialmente se recorre el árbol desde el nodo raíz hasta la hoja más cercana a nuestro punto de búsqueda y se va añadiendo a una cola de prioridad todas las ramas inexploradas. Esta cola de prioridad es ordenada de forma ascendente respecto a las distancias del punto al límite de la rama que se ha añadido a la cola. Después del recorrido inicial hecho, el algoritmo sigue recorriendo el árbol, pero siempre empezando en la rama más alta de la cola de prioridad, es decir, la rama con el centroide más cercano al punto de búsqueda. En cada recorrido el algoritmo sigue añadiendo a la cola de prioridad aquellas ramas que todavía no se han explorado.

Algorithm 4 Searching the priority search k-means tree

Input: k-means tree T, query point Q, maximum number of points to examine L
Output: K nearest approximate neighbours of query point
procedure SEARCHKMEANSTREE(T,Q,L)
1: $count \leftarrow 0$
2: $PQ \leftarrow$ *empty priority queue*
3: $R \leftarrow$ *empty priority queue*
4: *call* TRAVERSETREE(T,PQ,R)
5: **while** PQ not empty **and** $count < L$ **do**
6: $N \leftarrow$ *top of* PQ
7: *call* TRAVERSETREE(N,PQ,R)
8: **end while**
9: **return** K *top points from* R
procedure TRAVERSEKMEANSTREE(N,PQ,R)
1: **if** *node* N *is a leaf node* **then**
2: *search all the points in* N *and add them to* R
3: $count \leftarrow count + |N|$
4: **else**
5: $C \leftarrow$ *child nodes of* N
6: $C_q \leftarrow$ *closest node of* C *to query* Q
7: $C_p \leftarrow C \backslash C_q$
8: *add all nodes in* C_p *to* PQ
9: *call* TRAVERSETREE(C_q,PQ,R)
10: **end if**

Figura 3.17: Búsqueda en el árbol K-Means. [6] y [7].

El grado de aproximación es controlado de la misma manera que en los árboles K-d aleatorios, pero parando la búsqueda justo después de que se haya alcanzado un número predeterminado de puntos que han sido examinados.

3.4.2.1 Parámetros del algoritmo

El número de conjuntos K a usar cuando se particiona el conjunto de datos es un parámetro del algoritmo, llamado *branching factor*, y es importante elegir cuánto es K porque es importante para obtener un buen rendimiento a la hora de realizar la búsqueda.

Otro parámetro es I_{max}, que es el número máximo de iteraciones que queramos que realice el algoritmo. Una desventaja de este algoritmo es el tiempo que tarda en construir el árbol respecto al tiempo de construcción del árbol K-d aleatorio. El tiempo de construcción puede ser significantemente reducido si I_{max} es pequeño en vez de dejar que se construya el árbol hasta que converja. Pero en [7] se llegó a la conclusión que incluso con un número pequeño de iteraciones, el rendimiento de la búsqueda de vecinos más cercanos es similar al del árbol construido hasta que converja.

Por último, respecto a la selección inicial de los centroides, parámetro C_{alg}, podemos elegir tres algoritmos distintos: selección aleatoria, algoritmo de Gonzales' y el algoritmo KMeans++. En [7] se recomienda elegir la selección aleatoria debido a que ha sido con el que mejor resultado se ha obtenido.

Por último, en la siguiente figura se muestra el rendimiento de este algoritmo respecto al número de iteraciones que se escojan.

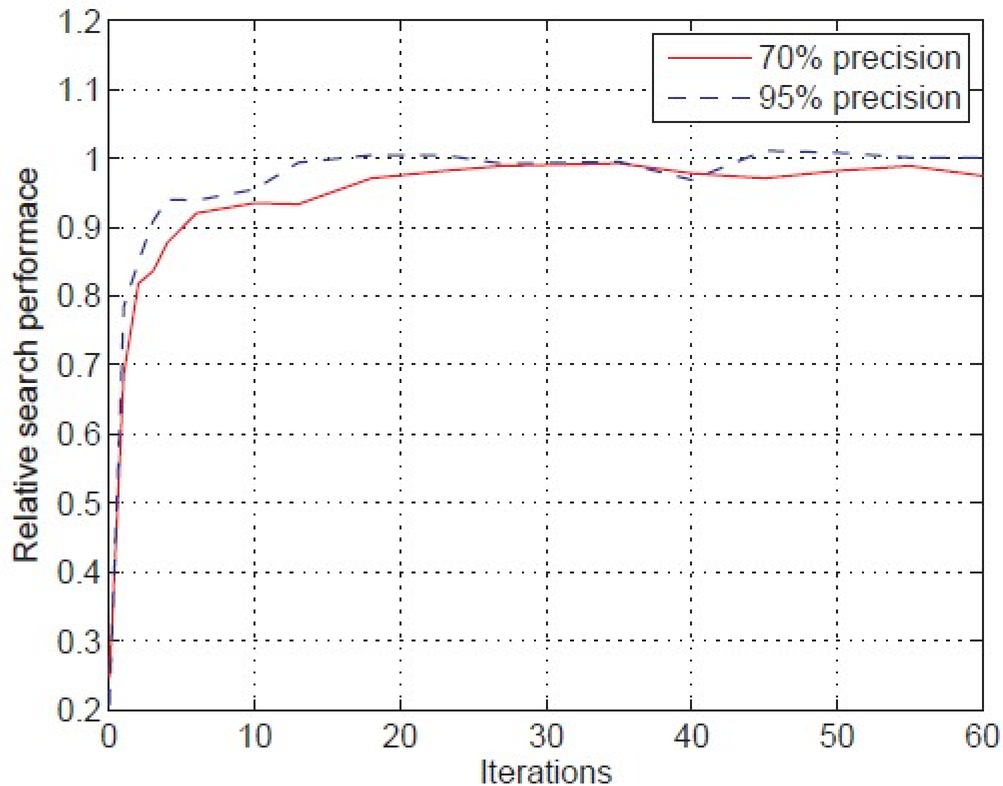

Figura 3.18: Influencia del número de iteraciones del algoritmo K-Means en los tiempos de búsqueda, se muestra el tiempo de búsqueda relativo comparado con el caso usando convergencia máxima.

3.5 Librería FLANN

La librería FLANN (*Fast Library for Approximate Nearest Neighbors*) es una librería que implementa los algoritmos comentados en el punto anterior. Está escrita en lenguaje C++, pero puede ser usada en otros lenguajes gracias a que ofrece interfaces a otros lenguajes como C, MATLAB y Python.

Para que la librería funcione adecuadamente en nuestro sistema, tenemos que hacer lo siguiente:

1. Compilar la librería a través de la herramienta *cmake*, con ella podemos elegir con qué lenguaje se programará el código.
2. Linkamos la librería con nuestro sistema. Para ello se completará el archivo *makefile* de la siguiente manera:

```
CC=gcc
CFLAGS=-c -Wall -I.
LDFLAGS=
SOURCES=gen_key_from_mfcc.c  gen_RTkey_from_mfcc.c  find_RTkey.c Analisis_trayectorias.c
OBJECTS=$(SOURCES:.c=.o)
EXECUTABLES=$(OBJECTS:.o=)

all: $(EXECUTABLES)

$(EXECUTABLES): $(OBJECTS)
    $(CC) $(LDFLAGS) $@.o -o $@

.c.o:
    $(CC) $(CFLAGS) $< -o $@

clean:
    rm $(OBJECTS) $(EXECUTABLES)
```

Figura 3.19: Fichero makefile

Esta librería contiene múltiples funciones, pero solamente se detallarán aquellas que se usen en este trabajo.

3.5.1 flann_build_index()

Esta función construye los índices del conjunto de datos que se quiere analizar. La función tiene los siguientes parámetros:

flann_index_t flann_build_index (float* dataset, int rows, int cols, float* speedup, struct FLANNParameters* flann_params)

- dataset: Conjunto de datos a analizar.
- rows: Número de filas de Dataset. Este número dependerá del tamaño del conjunto de datos.
- cols: Número de columnas de Dataset. En nuestro caso no serán los 13 coeficientes que se aplicaron en el PFC, en este caso dependerá de si estamos tratando la base de datos o los ficheros generados. En el capítulo 4.2 se explicará cuántas columnas se aplicarán en cada caso.
- speedup: Devuelve la velocidad aproximada que tarda el algoritmo respecto a una búsqueda lineal.
- flann_params: es una estructura que contiene todos los parámetros a usar en la función. Esta estructura está compuesta de la siguiente manera:

```
struct FLANNParameters {
  enum flann_algorithm_t algorithm;        /* the algorithm to use */

  /* search parameters */
  int checks;             /* how many leafs (features) to check in one search */
  float cb_index;         /* cluster boundary index. Used when searching the
                           kmeans tree */

  /*  kdtree index parameters */
  int trees;                 /* number of randomized trees to use (for kdtree) */

  /* kmeans index parameters */
  int branching;             /* branching factor (for kmeans tree) */
  int iterations;            /* max iterations to perform in one kmeans cluetering
                              (kmeans tree) */
  enum flann_centers_init_t centers_init;  /* algorithm used for picking the initial
                                cluster centers for kmeans tree */

  /* autotuned index parameters */
  float target_precision;    /* precision desired (used for autotuning, -1 otherwise) */
  float build_weight;        /* build tree time weighting factor */
  float memory_weight;       /* index memory weigthing factor */
  float sample_fraction;     /* what fraction of the dataset to use for autotuning */

  /* LSH parameters */
  unsigned int table_number_; /** The number of hash tables to use */
  unsigned int key_size_;     /** The length of the key in the hash tables */
  unsigned int multi_probe_level_; /** Number of levels to use in multi-probe LSH, 0 for standard LSH */

  /* other parameters */
  enum flann_log_level_t log_level;   /* determines the verbosity of each flann function */
  long random_seed;                         /* random seed to use */
};
```

Figura 3.20: Organización de la estructura. [8].

El tipo enumerado *flan_algorithm_t algorithm* se emplea para seleccionar el algoritmo de búsqueda entre los que se puede elegir una búsqueda lineal o los dos métodos que se emplearán, el algoritmo K-d y K-Means.

El tipo enumerado *flan_centers_init_t centers_init* se emplea para escoger qué tipo de algoritmo se ejecutará para elegir los centroides en el algoritmo K-Means.

Por último, otro tipo enumerado importante es el *flan_log_level_t log_level*.

```
enum flann_log_level_t {
    FLANN_LOG_NONE = 0,
    FLANN_LOG_FATAL = 1,
    FLANN_LOG_ERROR = 2,
    FLANN_LOG_WARN = 3,
    FLANN_LOG_INFO = 4
};
```

Figura 3.21: Tipos de log. [8].

Este tipo de enumerado controla el nivel de detalle de los mensajes generados por las funciones de la librería FLANN.

3.5.2 flann_find_nearest_neighbors_index()

Esta función busca los vecinos más cercanos de un conjunto de datos de muestras grabadas usando un índice previamente construido y referenciado. El conjunto de datos que contiene las muestras grabadas tiene que tener el mismo número de columnas que nuestra base de datos.

La función busca un determinado número de vecinos más cercanos para cada punto del conjunto de datos de las muestras grabadas. Los parámetros son los siguientes:

int flann_find_nearest_neighbors_index(FLANN_INDEX index_id, float* testset, int trows, int* índices, float* dists, int nn, struct FLANNParameters* flann_params)

- index_id: Índices de la base de datos construidos mediante la función anterior.
- testset: Conjunto de datos que se corresponden con los *fingerprints* de las muestras grabadas.
- trows: Número de filas del testset. Dependerá del tamaño de las muestras.
- indices: Nos devuelve los índices encontrados.
- dists: Nos devuelve la distancia del testset respecto a la base de datos.
- nn: Número de vecinos más cercanos que queremos buscar.
- flann_params: Es la estructura explicada en el punto anterior.

3.5.3 flann_free_index()

Esta función borra el índice previamente construido y libera toda la memoria que ha sido usada por él. Los parámetros son los siguientes:

int flann_free_index(FLANN_INDEX iindex_id, struct FLANNParameters* flann_params)

- index_id: Índice previamente construido que queremos liberar y borrar.
- flann_params: Estructura comentada en el punto 3.5.1.

3.5.4 flann_set_distance_type()

Esta función escoge la distancia a aplicar por los distintos algoritmos disponibles. Los parámetros son los siguientes:

void flann_set_distance_type(enum flann_distance_t distance_type, int order)

- distance_type: Indica el tipo de distancia que se va a aplicar. Es un tipo enumerado que nos permite elegir entre distancia Euclídea o Manhattan entre otras.
- order: solamente es usado para calcular la distancia Minkowski, para elegir el orden de esta distancia.

3.6 Nuevo diseño del programa

El objetivo del programa es el de optimizar el tiempo de sincronización de dos pistas de audio, una de ellas contiene el audio de la fuente original, y la otra contiene una muestra grabada con el micrófono de un teléfono móvil. Lo que se desea es que el tiempo de búsqueda sea el mínimo posible, al utilizar una búsqueda vecinos más cercanos se obtendrá una reducción de tiempo, pero la precisión será posiblemente menor que en el programa de partida.

En este caso las huellas de los audios se tratarán de forma distinta, se generarán de la misma manera pero después se reorganizarán para mejorar la optimización.

La huella del audio original se modificará de la siguiente forma:

- Se genera la huella del audio original y se obtiene la siguiente estructura

Figura 3.22: Estructura del fingerprint de la base de datos del programa de partida.

Cada frame contiene sus correspondientes coeficientes MFCC y N indica el último frame de la huella.

- Se modifica esta estructura para obtener la siguiente

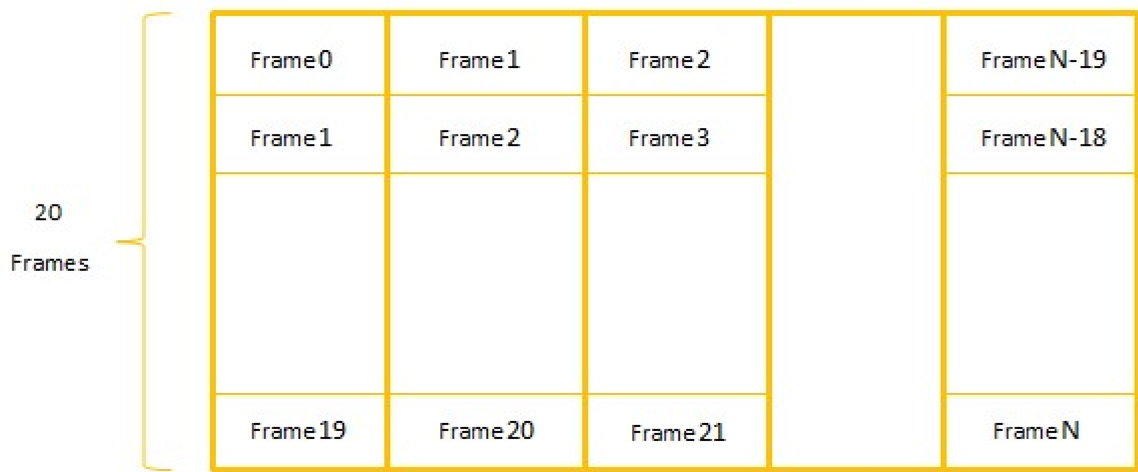

Figura 3.23: Estructura reorganizada del fingerprint de la base de datos.

Esta estructura está ordenada de forma que el primer frame está compuesto por los primeros 20 frames de la huella original, es decir, cada nuevo frame es un fragmento de 8 segundos del original. Los siguientes frames comienzan cada 16 ms, esto es, el primer frame empieza en el segundo cero, el segundo frame empieza a los 16 ms, el tercer frame a los 32 ms, y así sucesivamente.

En el caso de la huella de cada muestra grabada la modificación será de la siguiente manera:

- Se genera la huella de la muestra grabada y se obtiene la siguiente estructura, recordemos que cada muestra grabada tiene una duración de 8 segundos

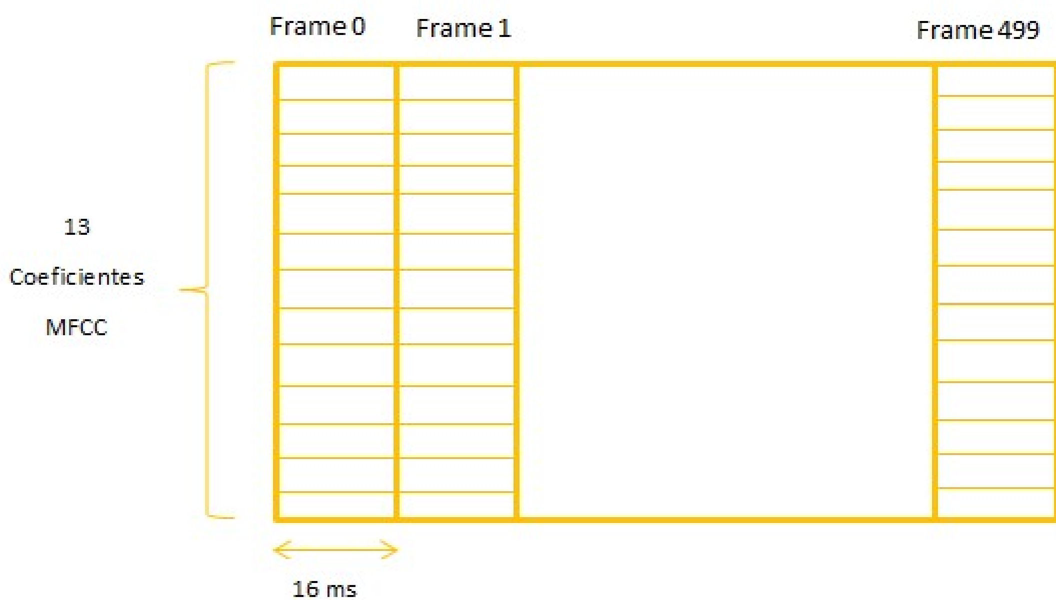

Figura 3.24: Estructura del fingerprint de la muestra grabada del programa de partida.

- Se modifica esta estructura de la siguiente forma

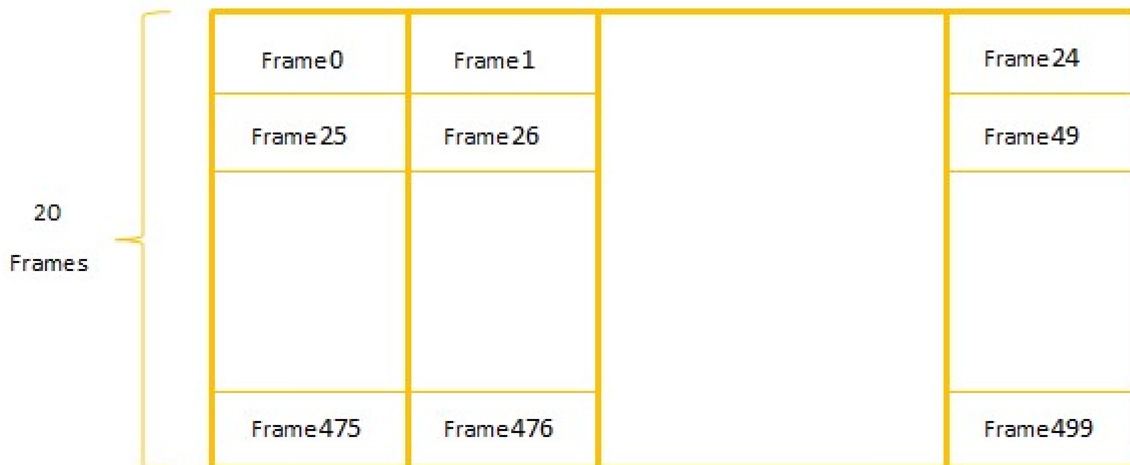

Figura 3.25: Estructura reorganizada del fingerprint de la muestra grabada.

En este caso, esta estructura está ordenada de forma que cada frame está compuesto de 20 frames de la huella de la muestra grabada, y cada uno de esos 20 frames son los correspondientes a las medias calculadas cada 0.4 segundos, es decir, el frame 0 es el calculado en un instante cualquiera de la muestra grabada, el frame 25 es el calculado a los 0.4 segundos de ese instante, el frame 50 es el calculado a los 0.8 segundos del instante inicial, y así sucesivamente. Se ha decidido reordenarlo de esta manera ya que es una forma más eficiente de buscar los vectores de las muestras en la base de datos nueva generada.

A continuación se busca la nueva huella de la muestra grabada en la base de datos reorganizada a través de la librería FLANN, ésta nos devuelve las distancias de cada frame y su correspondiente detección en el tiempo.

3.7 Ficheros de prueba

Para el desarrollo de este TFG se ha hecho el análisis del audio de referencia que se aplicó en el PFC de Andrés [1]. Este audio es una película de una duración aproximada de dos horas y para su análisis se tomaron muestras aleatorias grabadas mediante los micrófonos de dos tipos dispositivos móviles, estos dispositivos son un iPhone y un HTC. Estas muestras se tomaron de la siguiente manera:

- **Regazo**: El dispositivo se encuentra sobre el regazo sin ningún tipo de obstáculo, por lo que el ruido recogido es mínimo.

- **Bolsillo**: En este caso la muestra se toma con el dispositivo introducido dentro del bolsillo, teniendo parte de ropa y cuerpo bloqueándolo, el nivel de audio es menor y se captura más ruido.

- **Bolso**: La última situación es la peor de todas, en este caso se introduce el dispositivo en un bolso. El audio recogido es mínimo y el ruido es muy elevado, debido por ejemplo a la acción de meterlo en el bolso y cerrar su cremallera.

Se tiene una muestra de cada dispositivo para cada una de estas situaciones.

4 Pruebas y Resultados

4.1 Análisis del programa de partida

El primer paso para el desarrollo de este TFG consiste en realizar un análisis del rendimiento del programa de partida, con ello se podrán extraer unas primeras conclusiones, así como ver cuáles son los márgenes de mejora. Para ello, primero se realizarán pruebas sobre el audio original con recortes aleatorios del propio audio original, para así comprobar que el funcionamiento del sistema es el correcto. Y el siguiente paso es realizar pruebas sobre el audio original con recortes de las muestras grabadas con los dispositivos móviles.

Las pruebas que se van a realizar dan como resultado unos valores de tiempo de sincronismo. Para poder evaluar correctamente la precisión del sistema con las muestras grabadas hay que saber antes los valores de desfase entre ambas muestras para poder compararlos con los obtenidos de realizar la búsqueda. Para ello, en el PFC [1] se calculó la correlación cruzada de cada una de las muestras grabadas, siendo estos valores los siguientes:

Muestra	Latencia (en segundos)
Regazo iPhone	44.4836
Bolsillo iPhone	37.46612
Bolso iPhone	35.33082
Regazo HTC	50.6164
Bolsillo HTC	31.2384
Bolso HTC	28.576

Tabla 4-1: Latencias de las muestras grabadas. [1].

Con estos valores se consigue localizar el desplazamiento de cada una de las muestras respecto al inicio del audio original con una precisión bastante elevada, y sirve como base para realizar las medidas de rendimiento posteriores.

4.1.1 Resultados del programa de partida

Primero se realiza una prueba para medir el tiempo que tarda en ejecutar la búsqueda de los recortes del audio original sobre la propia pista de audio de la película.

Los resultados obtenidos se muestran en la siguiente tabla:

Muestras grabadas	Tiempo de ejecución (segundos)	Número de muestras
Recortes Original	2436	20000
Regazo iPhone	1878	3200
Bolsillo iPhone	1942	3200
Bolso iPhone	1916	3200
Regazo HTC	1907	3200
Bolsillo HTC	1971	3200
Bolso HTC	1997	3200

Tabla 4-2: Tiempo de ejecución del programa original.

Y por último se realiza una prueba para medir la precisión del sistema completo. Los resultados que se obtuvieron son los siguientes:

Original	% acierto	100

Tabla 4-3: Porcentaje de acierto del programa de partida con recortes del audio original.

Para terminar, se realizan pruebas sobre el resto de muestras grabadas:

iPhone		
Distancia	First Order	Euclídea
Muestras	% acierto	
Regazo	100	100
Bolsillo	100	100
Bolso	76.8	77.3

Tabla 4-4: Porcentaje de acierto de la muestras del iPhone.

HTC		
Distancia	First Order	Euclídea
Muestras	% acierto	
Regazo	100	100
Bolsillo	84.2	84.6
Bolso	57.4	57.1

Tabla 4-5: Porcentaje de acierto de las muestras del HTC.

4.2 Análisis del nuevo diseño

Ahora se realizará el análisis sobre el nuevo diseño del programa. Para ello se procederá a seguir los mismos pasos que se realizaron para analizar el programa de partida. Primero se analizará con el algoritmo de los árboles K-d y luego con el algoritmo de los árboles K-Means.

4.2.1 Resultados: Algoritmo de Árboles K-d Aleatorios

En primer lugar se medirá el tiempo que tarda el nuevo sistema en ejecutar las muestras, tanto los recortes del audio original como las muestras grabadas. Se realizarán medidas con un número distinto de árboles, ya que el algoritmo nos permite elegir el número de árboles que se desean. En estas medidas de tiempo está incluido el tiempo que se tarda en generar la base de datos y la búsqueda de las muestras grabadas.

Muestras grabadas	Tiempo de ejecución (segundos)				Número de muestras
	8 Árboles	9 Árboles	10 Árboles	20 Árboles	
Recortes Original	28	18	28	38	20000
Regazo iPhone	11	10	11	19	3200
Bolsillo iPhone	11	9	12	19	3200
Bolso iPhone	10	11	13	19	3200
Regazo HTC	10	9	13	21	3200
Bolsillo HTC	10	10	13	19	3200
Bolso HTC	11	11	13	21	3200

Tabla 4-6: Tiempo de ejecución aplicando el algoritmo K-d.

Muestras grabadas	Porcentaje de acierto (%)				Número de muestras
	8 Árboles	9 Árboles	10 Árboles	20 Árboles	
Recortes Original	96.5	96.5	96.4	96.6	20000
Regazo iPhone	85	84.9	84.6	98	3200
Bolsillo iPhone	80.6	80.6	81.4	85.2	3200
Bolso iPhone	14.2	14.3	14.6	24.4	3200
Regazo HTC	98.8	98.8	98.4	98.6	3200
Bolsillo HTC	14.8	14.7	14.8	14.8	3200
Bolso HTC	15.6	15.6	15.6	15.6	3200

Tabla 4-7: Porcentaje de acierto aplicando el algoritmo K-d.

4.2.2 Resultados: Algoritmo de Árboles K-Means con búsqueda de prioridad

En primer lugar se medirá el tiempo que tarda el nuevo sistema en ejecutar las muestras, tanto los recortes del audio original como las muestras grabadas. Como con el algoritmo K-d, en los tiempos de ejecución está incluido el tiempo que tarda en generarse la base de datos y la búsqueda de las muestras grabadas. Se realizarán medidas variando los parámetros que nos permite el algoritmo K-Means, estos parámetros son el número de clusters, el número de iteraciones que se realizarán para construir el árbol y el tipo de algoritmo para seleccionar los centroides. En este último parámetro se escogió el elegir los centroides aleatoriamente, ya que es el que mejores resultados ha proporcionado.

Muestras grabadas	Tiempo de ejecución (segundos)				Número de muestras
	4k – 10i	4k – 20i	8k – 20i	8k – 20i	
Recortes Original	381	382	381	425	20000
Regazo iPhone	290	286	291	310	3200
Bolsillo iPhone	296	299	297	322	3200
Bolso iPhone	295	295	298	324	3200
Regazo HTC	292	291	291	316	3200
Bolsillo HTC	296	294	297	331	3200
Bolso HTC	315	310	329	324	3200

Tabla 4-8: Tiempo de ejecución aplicando el algoritmo K-Means.

Muestras grabadas	Porcentaje de acierto (%)				Número de muestras
	4k – 10i	4k – 20i	8k – 20i	8k – 20i	
Recortes Original	96.5	96.5	96.2	95.9	20000
Regazo iPhone	98.2	100	82.6	87.6	3200
Bolsillo iPhone	97.4	90	96.2	96	3200
Bolso iPhone	54.4	75.6	68.8	60	3200
Regazo HTC	100	99.8	96	90.6	3200
Bolsillo HTC	14.4	21	35.4	7.2	3200
Bolso HTC	58.8	68.8	53	60	3200

Tabla 4-9: Porcentaje de acierto aplicando el algoritmo K-Means.

4.3 Análisis de los resultados

Como se comentó anteriormente, el objetivo de este TFG es el de optimizar el tiempo de búsqueda mediante la aplicación de algoritmos de búsqueda aproximada de vecinos más cercanos. Esto se ha conseguido con estos dos algoritmos, con el algoritmo de árboles K-d se ha conseguido reducir el tiempo de búsqueda en dos órdenes de magnitud, mientras que con el algoritmo de árboles K-Means solamente se ha conseguido reducir en un orden de magnitud.

En ambos algoritmos el tiempo de búsqueda es similar, la gran diferencia es a la hora de generar la base de datos, con el algoritmo K-Means el tiempo es mucho mayor.

Respecto al porcentaje de acierto, el rendimiento es un poco menor, ya que en este TFG se aplica una búsqueda aproximada de vecinos más cercanos, mientras que en el PFC se aplicó una búsqueda lineal exhaustiva. En las muestras grabadas en el regazo el porcentaje se mantiene cercano al 100% en la mayoría de los casos, mientras que en las muestras tomadas en el bolsillo y en el bolso el porcentaje disminuye considerablemente. Con el algoritmo K-d, el porcentaje en las muestras tomadas en el bolso mediante el iPhone es bastante baja, mientras que en el HTC son bajas tanto para las muestras tomadas en el bolsillo y en el bolso.

Algo parecido ocurre con el algoritmo K-Means, con este algoritmo se consigue mayor porcentaje de acierto en comparación con el algoritmo K-d, en este caso los peores resultados se obtienen con las muestras tomadas en el bolsillo con el HTC. El resto de porcentajes con este algoritmo se acercan bastante al conseguido con el programa de partida.

5 Conclusiones y trabajo futuro

5.1 Conclusiones

Durante este trabajo se han ido estudiando diversas situaciones que se plantearon inicialmente. Mediante la aplicación de los dos algoritmos se ha conseguido el objetivo principal de este trabajo, el de optimizar el tiempo de búsqueda sin que afecte demasiado al rendimiento del sistema.

El algoritmo de árboles K-d ha resultado ser muy bueno en tiempos de búsqueda, ya que se ha reducido en dos órdenes de magnitud ese tiempo. Aunque afecta notablemente en el rendimiento del sistema en las muestras más ruidosas, como son las muestras tomadas en el bolsillo y en el bolso.

Con el algoritmo de árboles K-Means también se consiguió reducir el tiempo de búsqueda, pero en este caso en un orden de magnitud, ya que a la hora de generar la base de datos el tiempo es mucho mayor. Pero en este caso el rendimiento del sistema no se ve tan afectado como con el algoritmo K-d.

Con todo esto, el objetivo principal fijado para este trabajo se puede considerar satisfecho, consiguiendo un tiempo de búsqueda mucho más asequible que el conseguido en el programa de partida.

5.2 Trabajo futuro

A pesar de conseguir el objetivo principal del trabajo, existen importantes vías de mejora, así como de investigación respecto a distintos aspectos.

Uno de los aspectos que admite un mayor margen de mejora es el rendimiento del sistema. Al aplicar una búsqueda aproximada mediante vecinos más cercanos, el sistema no es tan preciso como el programa de partida, que aplicaba una búsqueda lineal exhaustiva. Para mejorar este rendimiento se puede aplicar el algoritmo desarrollado en el PFC, el algoritmo de análisis de trayectorias,

Otros aspectos que se podrían estudiar, sería el de aplicar otras técnicas de normalización, por ejemplo se puede aplicar una técnica de normalización más avanzada como es la técnica CSN (Cepstral Shape Normalization), que es más robusta en condiciones ruidosas.

Referencias

[1] Andrés Martín López, "*Mejora de la robustez frente al ruido en un sistema de búsqueda rápida de audio en audio*", Proyecto Fin de Carrera, UAM, Mayo 2015. Accesible en http://arantxa.ii.uam.es/~jms/pfcsteleco/index.html#2014-15

[2] Andrew W. Moore, "*An introductory tutorial on kd-trees*", Carnegie Mellon University, 1991. Accesible en https://www.ri.cmu.edu/pub_files/pub1/moore_andrew_1991_1/moore_andrew_1991_1.pdf

[3] Standard C++ Programming Laboratory, "*Assignment 3: KDTree*", Stanford University, 1-10, 2015

[4] J. Ortega-García, "*Tratamiento de Señales de Voz y Audio*", ATVS-Universidad Autónoma de Madrid, 2015, pp. 4.23-4.24. Transparencias de la asignatura TSVA.

[5] Chris Piech, "*K-Means*", 2013, Accesible en:http://stanford.edu/~cpiech/cs221/handouts/kmeans.html.

[6] Marius Muja, David G. Lowe, "*Scalable Nearest Neighbor Algorithms for High Dimensional Data*", Pattern Analysis and Machine Intelligence (PAMI), Vol. 36, 2014, pp. 1-6.

[7] Marius Muja, "*Scalable Nearest Neighbour Methods For High Dimensional Data*", Phd thesis, The University Of British Columbia, April, 2013. Accesible en: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.459.3003&rep=rep1&type=pdf

[8] Marius Muja, David Lowe, "*FLANN – Fast Library for Approximate Nearest Neighbors, User Manual*", The University Of British Columbia, January, 2013. Accesible en: http://www.cs.ubc.ca/research/flann/

[9] William H. Press, Saul A.Teukolsky, William T. Vetterling, Brian P. Flannery. "*Numerical Recipes in C – The art of Scientific Computing 2nd Edition*". Cambridge University Press, 1992.

[10] Avery Li-Chun Wang, "*An Industrial-Strength Audio Search Algorithm*", Shazam Entertainment Ltd. Accesible en: https://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf

[11] Audible Magic, "*Automatic Content Recognition (ACR).[online]*", Accesible en https://www.audiblemagic.com/

[12] S. Tiberwala, H. Hermansky, "*Multi-Band and Adaptation Approaches to Robust Speech Recognition*", In Proc. EUROSPEECH'97, Rhodes, Greece, pp. 2619-2622, 1997.

[13] Bruce F. Naylor, "*Binary Space Partitioning Trees*", Spatial Labs Inc. Accesible en: https://www.researchgate.net/profile/Bruce_Naylor2/publication/238348725_A_Tutorial_on_Binary_Space_Partitioning_Trees/links/55cc86bd08aeca747d6c2e15.pdf

[14] Jacob Benesty, M. M. Sondhi, Yiteng Huang, "*Springer Handbook of Speech Processing*", Springer, 2008.

[15] Lu Yu, "*[Research] Bag-Of-Features for visual recognition*", 2013. Accesible en https://littlecheesecake.wordpress.com/2013/04/24/research-bag-of-features-for-visual-recognition/