Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Secuenciamiento automático de *playlists* musicales utilizando modelos auto-supervisados y heurísticas TSP

Tesis de Licenciatura en Ciencias de la Computación

Tomás Valentín Fedi

Director: Pablo Riera

Buenos Aires, 2024

La música está presente en muchos momentos de nuestras vidas. Tanto en conciertos como en fiestas, el artista o *DJ* le presta mucha importancia no solo a las canciones que decide reproducir en estos eventos, sino que también al orden en que lo hace. Por ejemplo, una banda de *Rock and Roll* puede querer evitar tocar solo ritmos movidos para que el público no se agote al poco tiempo de iniciado el concierto. En cambio, es probable que decida alternar con otros ritmos más lentos. El mismo razonamiento suele acompañarnos cuando confeccionamos *playlists* con la ayuda de nuestras plataformas de música favoritas para diferentes situaciones. El criterio a la hora de ordenarla puede variar, pero, por lo general, se busca que los saltos entre canciones sean suaves. En este trabajo, nos preguntamos si es posible automatizar el ordenamiento mediante una herramienta que únicamente conozca el audio de las canciones. También nos preguntamos qué tan similar lo haría en comparación con la subjetividad humana, sin tener registro de nuestros gustos o de otra información disponible. Para la construcción de la misma, experimentamos con métodos de extracción de atributos (MFCCs y modelos de *Deep Learning*) y con heurísticas para resolver el problema del viajante de comercio (TSP). Propusimos distintas estrategias para evaluar el desempeño de la herramienta. Observamos que, utilizando un modelo y una heurística en particular, se logra una correlación con las listas de reproducción que fueron ordenadas manualmente por humanos.

Palabras claves: música, *playlist*, orden, secuenciamiento, audio, grafo, TSP.

Music is part of many moments of our lives. Both in concerts and parties, the artist or DJ pays a lot of attention, not only to the songs he decides to play during these events, but also to the order in which he decides to play them. For instance, a Rock and Roll band may want to avoid playing only upbeat rhythms so that the audience does not get exhausted soon after the show starts. Instead, it may decide to alternate with slower rhythms. The same reasoning usually accompanies us when we compile playlists with the help of our favorite music platforms to suit different situations. The ordering criteria may vary, but the jumps between songs are generally intended to be smooth. In this research, the question addressed is whether it is possible to automate this process using a tool that only knows the audio of the songs. We also wonder how similar it would be in comparison to human subjectivity, with no record of our tastes or other available information. To build it, we experimented with attribute extraction methods (MFCCs and Deep Learning models) and with heuristics to solve the traveling salesman problem (TSP). We proposed different strategies to evaluate the performance of the tool. We observed that using a particular model and heuristic, some correlation is achieved with playlists that were manually sorted by humans.

Keywords: music, playlist, order, sequencing, audio, graph, TSP.

AGRADECIMIENTOS

Quiero agradecer principalmente a mi familia por todo el amor y el apoyo.

A Darío, mi papá, por despertarme la curiosidad por las computadoras y por ayudarme a ser perseverante.

A Marilú, mi mamá, por inculcarme la importancia de estudiar y a divertirse resolviendo problemas.

A Lucía, mi novia, por las risas, por enseñarme a ser paciente y a tener confianza en mi mismo.

A Pablo, mi director de tesis, por todo el esfuerzo y la mentoría para llevar este trabajo adelante.

A Miriam, por recomendarme que estudie esta hermosa carrera en la Facultad de Ciencias Exactas y Naturales.

También quiero agradecerle a mis amigos de cursada, por las divertidas charlas en los pasillos y las tardes haciendo trabajos prácticos y preparando parciales.

A todo el personal del Instituto Politécnico Modelo, principalmente a Enrique Toriggia, Nicolás Pruscino, Lucas Saclier, Nicolás Bazan y Claudio Vaccalluzzo.

A Nicolás y Rodrigo de Tupaca. A Johnny, Matías, Leandro y Andrés de Southworks.

Finalmente, le quiero agradecer a la Universidad de Buenos Aires y a cada uno de sus docentes, por darme una educación de primera calidad, con clases llenas de pasión, conocimiento y desafíos.

A mis abuelos, que me cuidan desde lo más alto.

Índice general

1. INTRODUCCIÓN

1.1. Motivación

La creación de *playlists* o listas de reproducción es una de las funcionalidades más utilizadas en cualquier plataforma de música digital [1]. Los usuarios invierten tiempo en seleccionar las canciones o pistas que desean escuchar en un evento dado: hacer ejercicio, viajar en la ruta, celebrar una reunión, etc. Estas *playlists* pueden contener un amplio repertorio de artistas y géneros distintos (sobre todo en estos últimos tiempos donde las *playlists* colaborativas están de moda) o pueden ser más bien consistentes, dependiendo del uso para el que se hayan concebido. Por ejemplo, para una sesión de estudio un usuario puede querer escuchar solo canciones instrumentales que acompañen al oyente, mientras que para un cumpleaños familiar los usuarios que participan pueden elegir las canciones más populares de sus respectivas épocas.

Una vez seleccionadas las canciones que conformarán una *playlist*, hay que decidir en qué orden queremos que se reproduzcan. En cualquier escenario de los previamente mencionados, suele ser una propiedad deseable que el salto de una canción a la otra sea poco notable. De hecho, los *DJ* de fiestas se especializan en generar transiciones para lograr largas sesiones de escucha sin interrupciones [Bit+17].

A pesar de esto, la mayoría de plataformas de música digital ofrecen criterios de ordenamiento según la fecha en la que se agregan las pistas, el orden alfabético o el modo aleatorio. Quizás, el más novedoso sea el modo aleatorio inteligente de Spotify, que introduce canciones que no están en la *playlist* original para suavizar las transiciones. No obstante, por experiencia personal y de otros usuarios [2], en repetidas ocasiones este modo puede agregar demasiadas canciones que no están en el listado original y, así, se pierde de escuchar muchas de las que sí figuraban.

Problema 1: Dado un conjunto de canciones, generar un **secuenciamiento** (es decir, ordenarlas) para lograr una sesión de escucha homogénea (con la menor cantidad de saltos bruscos posibles entre pistas).

Desde luego, resolver el problema se vuelve exponencialmente más difícil conforme la cantidad de canciones del conjunto de entrada crece. Puede parecer tarea fácil para conjuntos con menos de diez canciones, pero si queremos lograr sesiones de escucha prologadas, tendremos conjuntos mucho mayores. El lector puede hacer el ejercicio de buscar en su biblioteca de alguna plataforma de música digital alguna *playlist* que tenga más de veinte canciones e intentar ordenarla para lograr el efecto deseado. Por ende, se vuelve de interés resolver este problema de manera automatizada.

[1] "In 2020 alone, 1 billion new playlists were created, showing just how often people love to create brand new playlists to listen to and share with friends and family." Fuente: `https://thesocialshepherd.com/blog/spotify-statistics`

[2] `https://www.reddit.com/r/spotify/comments/17wlnur/spotify_only_wants_you_to_use_smart_shuffle_now/`

¿Qué significa que dos canciones sean parecidas? A simple vista puede parecer obvio, pero necesitamos explicitar una función para medir la distancia entre dos pistas. Dicha función podría definirse, por ejemplo, utilizando meta-datos comunes de cada canción: artista, género y año de grabación, entre otros. También se pueden analizar meta-datos relacionados con el contenido de la canción, como pueden ser la tonalidad o el tempo. En este trabajo, se emplearán distintos métodos para medir la distancia entre el **contenido** de dichas pistas. Nos basaremos, únicamente, en la señal de audio pare medir la distancia. Esta noción se formaliza en la sección 2.3.3.

Claro está que, dado que no agregamos ni quitamos canciones del conjunto de entrada, la homogeneidad del resultado está ligada de forma directa a tal conjunto. Es decir, si todas las canciones de entrada son muy diferentes entre sí, cualquier ordenamiento sobre estas resultará heterogéneo. Así mismo, si todas las pistas son muy parecidas entre sí, cualquier ordenamiento resultará homogéneo. El problema se vuelve interesante cuando hay varios subconjuntos de canciones similares. Esto se explora en la sección 3.2.

En este trabajo, vamos a utilizar la siguiente versión simplificada del problema. La motivación es que para comparar contra listas de reproducción existentes, necesitamos que ambas empiecen desde la misma canción.

Problema 2: Dado un conjunto de canciones y una canción inicial, generar un **secuenciamiento** (es decir, ordenarlas) para lograr una sesión de escucha homogénea (con la menor cantidad de saltos bruscos posibles entre pistas).

1.2. Revisión de literatura

El artículo "Automated Generation of Music Playlists: Survey and Experiments" de Bonney y Jannach [BJ14] define, primero, una *playlist* como una **secuencia de canciones**. En la literatura previa no hay una definición consistente, por lo que trabajos anteriores emplearon el mismo término para referirse a un conjunto de canciones (sin ordenar). Nosotros utilizamos la primera definición (con orden). Luego, se define el problema de generar *playlists* como: dado (i) un conjunto de canciones, (ii) una base de datos con información sobre estas (como género, tempo, popularidad, etc.) y (iii) las características buscadas en la *playlist* resultante, crear una secuencia de pistas que cumplan con dichas características de la mejor manera posible. Se puede notar que el trabajo define, entonces, un problema más amplio que el nuestro, dado que asumimos que la característica buscada en la *playlist* resultante es que sea lo más homogénea posible, además de que no utilizamos ningún meta-dato. El artículo también hace una revisión de los diferentes métodos para resolver el problema. Dichos métodos son agrupados según estrategia en siete categorías distintas: algoritmos basados en similitud, filtros colaborativos, minería de patrones, modelos estadísticos, razonamiento basado en casos, optimización discreta e híbridos. Nuestro método pertenece a la primera categoría, dado que estamos seleccionando y ordenando las canciones con base en qué tan similares son, y empleamos, como criterio de calidad, la coherencia entre estas. Las otras categorías presentan enfoques alternativos. Por ejemplo, la minería de patrones consiste en observar canciones que aparecen juntas frecuentemente en varias *playlists*.

El primer trabajo que propuso generar listas de reproducción basadas en similitud extrayendo los atributos relevantes de las señales de audio de las canciones, "Content-Based Playlist Generation: Exploratory Experiments" de Logan [Log02], vino de la mano de la llegada del formato MP3. Este introduce la noción de representar la base de datos de canciones como un grafo, en el cual el peso de sus aristas mide qué tan similares son las señales de audio de las canciones asociadas a sus ejes. De esta forma, generar la *playlist* consiste en, dada una canción inicial (eje), encontrar un camino en el grafo que minimice la distancia total recorrida. Así, generar la *playlist* más homogénea posible con las canciones en la base de datos se vuelve equivalente a resolver el problema del viajante de comercio (TSP) en el grafo que representa dicha base de datos. Tal noción estará presente en nuestro método para resolver el problema.

Esta misma idea se exploró en trabajos posteriores. En particular, "Generating Similarity-Based Playlists Using Traveling Salesman Algorithms" de Pohle, Pampalk y Widmer [PPW05] utiliza coeficientes cepstrales en las frecuencias de Mel (MFCCs) para extraer la información de las señales de audio de las canciones e implementa heurísticas TSP para encontrar el recorrido, como vecino más cercano, árboles generadores mínimos o LKH [Hel00], que es una optimización de la heurística Lin-Kernighan. Para comparar los resultados entre estas heurísticas, se utilizó la correlación entre el género de las canciones. Es decir, si dos pistas continuas pertenecen al mismo género, no hay salto brusco. Este trabajo define una versión distinta del problema, donde se utiliza un conjunto inicial de más de 3000 canciones y al elegir una canción inicial, se buscan caminos de largo n. Con esta definición del problema, otra forma de evaluar los resultados es mediante la entropía de Shannon (siendo p la probabilidad de pertenecer a un género), ya que al dejar afuera canciones del conjunto original de entrada, es una forma de medir que tan cohesiva es la selección de canciones.

Algunos trabajos más recientes emplean modelos de inteligencia artificial para extraer información de las señales de audio, que en los trabajos anteriores todavía no habían sido desarrollados. En particular, "Automatic Playlist Sequencing and Transitions" de Bittner, Gu, Hernandez, Humphrey, Jehan, McCurry y Montecchio[Bit+17] menciona que se entrenó una red neuronal convolucional (CNN) para generar *embeddings* (vector que codifica los atributos extraídos de una señal de audio) que agrupen las canciones por género, aunque se incluye información extra como el tempo o la tonalidad. También, se implementan heurísticas para resolver TSP, como vecino más cercano. Finalmente, para evaluar los resultados obtenidos, se utilizaron curadores profesionales (*DJs*). Se les pidió escuchar, para un mismo conjunto de canciones, tanto la *playlist* generada por el algoritmo como una generada por un ordenamiento aleatorio (sin saber cuál es cuál). Luego, debían elegir la que mejor secuenciada estuviera y contar la cantidad de saltos bruscos que hubieran escuchado en ambas, a fin de comparar cuál tenía menos. En los resultados reportados, los curadores eligieron la lista de reproducción generada por el algoritmo en la mayoría de los casos. Asimismo, se observó que había más saltos bruscos (con notable diferencia para algunos conjuntos de canciones) en las *playlists* generadas aleatoriamente. El trabajo también estudia cómo generar las transiciones (por ejemplo, utilizando un *fade out*) de manera automática entre las pistas, que, si bien está relacionado con la generación de secuenciamientos, es otro problema en sí mismo y queda por fuera del alcance de esta tesis.

Otros trabajos recientes como "Automatic *playlist* generation using Convolutional Neural Networks and Recurrent Neural Networks" de Irene, Borrelli, Zanoni, Buccoli y Sarti [Ire+19] exploran la idea de entrenar un modelo de inteligencia artificial para generar el secuenciamiento en lugar de solamente extraer el audio. Si bien este método entraría en la categoría de modelo estadístico, dado que se utilizan cadenas de Markov para intentar predecir la siguiente canción en la lista dadas las pistas anteriores, es interesante la idea de usar un *dataset* de *playlists* curadas como fuente de verdad para evaluar qué tan bien predice el modelo. De hecho, será el mismo *dataset* "Art of the Mix 2011" [ML12], el que utilizaremos para nuestros experimentos.

En los últimos años, han surgido varios modelos de inteligencia artificial que han sido entrenados de forma auto-supervisada para resolver la tarea de extracción de atributos de señales de audio. Tenemos "BYOL-A" de Niizumi, Takeuchi, Ohishi, Harada y Kashino[Nii+21], que combina técnicas de normalización e intensificación de segmentos de audio para lograr buenos resultados en varias tareas. Por otro lado, "Music2Vec" de Li, Yuan, Zhang, Ma, Lin, Chen, Ragni, Yin, Hu, He, Benetos, Gyenge, Liu y Fu[Li+22], nace como un *framework* para procesar música, que implementa novedosas técnicas de aprendizaje auto-supervisado para lograr buenos resultados con mucho menos parámetros. También tenemos "MERT" de Li, Yuan, Zhang, Ma, Chen, Yin, Lin, Ragni, Benetos, Gyenge, Dannenberg, Liu, Chen, Xia, Shi, Huang, Guo y Fu [Li+23], que propone un modelo acústico para entender la música con un entrenamiento auto-supervisado a gran escala. Finalmente, "EnCodecMAE" de Pepino, Riera y Ferrer [PRF24], que explora el uso de EnCodec, un *codec* de audio neuronal y un *masked autoencoder* (MAE). Estos modelos por si solos no resuelven el problema de secuenciar *playlists*. Sin embargo, permiten obtener representaciones numéricas de los audios de las canciones que, en principio, codifican más información que los clásicos MFCCs. En "Comparative Analysis of Pretrained Audio Representations in Music Recommender Systems" de Tamm y Aljanaki [TA24], se realiza una comparación entre los modelos al emplearlos en sistemas de recomendación de música. En dicho trabajo, se reportan resultados muy favorables para los modelos MERT y EnCodecMAE en contraste con Music2Vec o MFCCs.

En resumen, vamos a abordar el problema de secuenciamiento de canciones, combinando el uso de redes neuronales para extraer características del audio con heurísticas para resolver el problema del viajante de comercio y vamos a utilizar un *dataset* de listas de reproducción reales para la evaluación de resultados.

1.3. Objetivo

El objetivo de este trabajo es construir una herramienta que resuelva el problema 2 (dado un conjunto de canciones y una canción inicial, generar un secuenciamiento para lograr una sesión de escucha homogénea) de manera automatizada. Para esto, nos basamos en las técnicas utilizadas en trabajos anteriores para generar una *pipeline* de procesamiento como se muestra en la figura 1.1. Esta, dada una *playlist* de entrada, consigue las muestras de audio de sus canciones. Luego, usa un modelo de extracción de atributos de señales de audio para generar los *embeddings*. Utilizando estos, se construye el grafo con todas las distancias entre las pistas. Finalmente, genera un secuenciamiento a partir de la primera canción de la lista original mediante una heurística TSP.

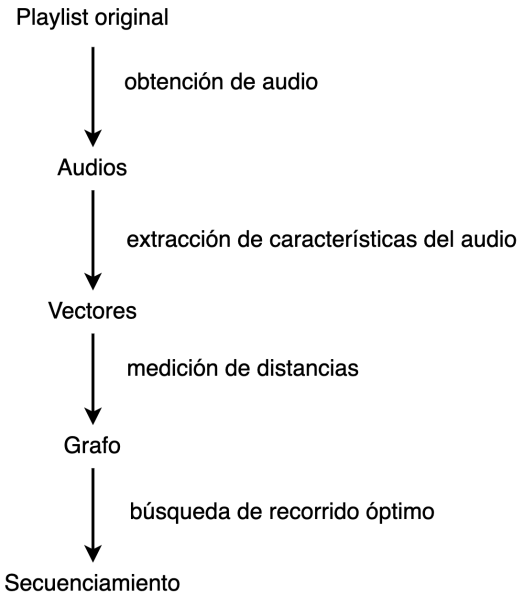


Fig. 1.1: Ilustración del pipeline de procesamiento

Nótese que, si bien la entrada es, en realidad, una lista de reproducción (ordenada) y no conjuntos de canciones (sin ordenar), solamente utilizamos el orden original para establecer la canción de partida, dado que una vez construido el grafo se pierde dicho orden. Sería sencillo, entonces, modificar la implementación para recibir como entrada un conjunto de canciones y una canción de partida para crear el secuenciamiento. Con esta metodología se apunta entonces a resolver el problema dado por la definición 2.

Veamos un ejemplo concreto a fin de agregar mayor claridad antes de avanzar a la siguiente sección. En la figura 1.2 podemos ver una lista de reproducción de cuatro canciones: “Nos siguen pegando abajo” de Charly Garcia, “Por una cabeza” de Carlos Gardel, “Giros” de Fito Paez y “Milonga del ángel” de Astor Piazzolla.

En el orden actual, las primeras dos transiciones resultan un poco bruscas de escuchar, ya que “Nos siguen pegando abajo” tiene características acústicas muy diferentes de “Por una cabeza”, siendo la primera más movida que la segunda, con sonidos más intensos:

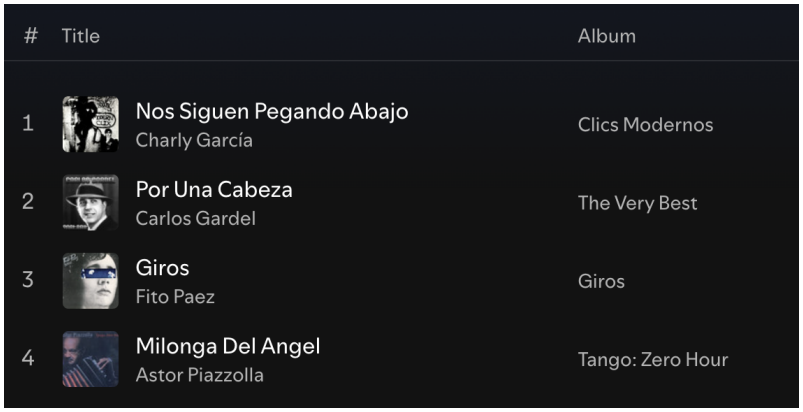


Fig. 1.2: Lista de reproducción con 4 canciones

guitarra eléctrica, caja de ritmos digital o una melodía de voz más efusiva, entre otros detalles. Lo mismo ocurre en la segunda transición, aunque "Giros" tiene un ritmo menos movido y la instrumental se asemeja un poco más a un tango, dado que incorpora otros elementos como un bandoneón. Es por eso que la última transición resulta menos brusca, ya que "Milonga del ángel" está interpretada principalmente con un bandoneón. Si procesamos los audios de las canciones utilizando el modelo *EnCodecMAE* y medimos las distancias resultantes entre cada par de canciones del conjunto, podemos observar que estas reflejan lo expresado en el párrafo anterior. En la figura 1.3 se puede ver como "Nos siguen pegando abajo" es mucho más cercana acústicamente a "Giros", y como esta a su vez está más cerca de "Milonga del angel" que de "Por una cabeza".

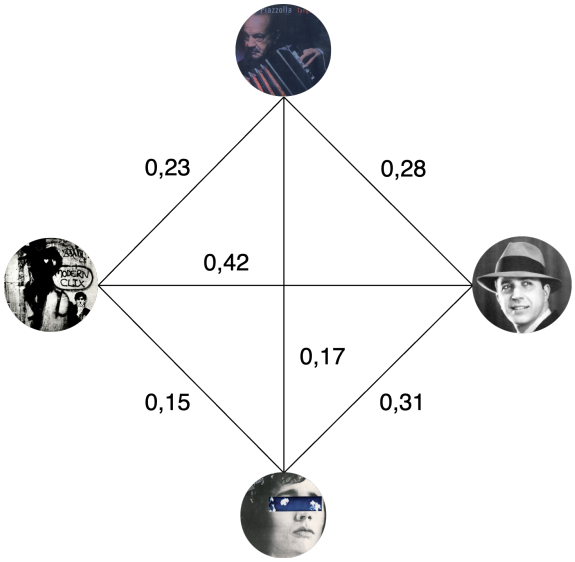


Fig. 1.3: Grafo para el conjunto de canciones, generado con el modelo EnCodecMAE. A la izquierda "Nos siguen pegando abajo". A la derecha "Por una cabeza". Arriba "Milonga del ángel". Abajo "Giros".

Se concluye entonces que, si movemos "Giros" a la segunda posición de la lista y "Milonga del ángel" a la tercera, la sesión de escucha de esta *playlist* se vuelve más homogénea, sin tantos saltos bruscos. De hecho, la transición de "Milonga del ángel" a

"Por una cabeza" resulta también muy natural. La lista resultante se puede observar en la figura 1.4.

Fig. 1.4: Secuenciamiento de la lista inicial para minimizar saltos bruscos

Esta lista, no es casualidad, es el resultado equivalente a resolver el problema del viajante de comercio (TSP) para el grafo que representa el conjunto de canciones, partiendo desde "Nos siguen pegando abajo". El lector puede ver en la figura 1.5 que el camino resultante es el mejor posible, ya que cualquier otro supera el peso total (suma de distancias) de este.

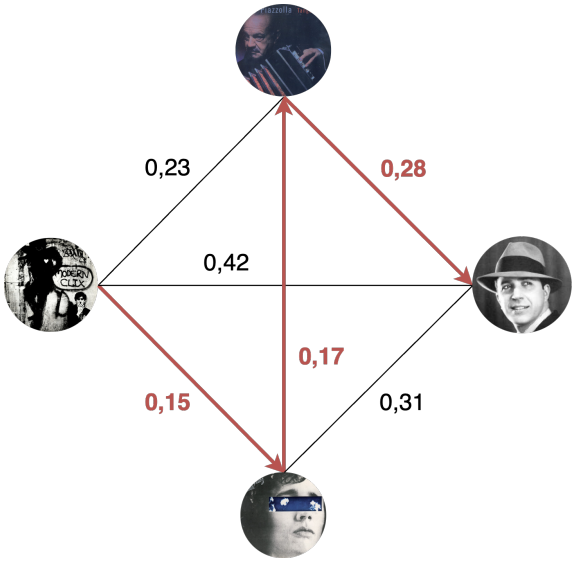


Fig. 1.5: Camino que minimiza la suma de distancias

Para el desarrollo del programa, se utilizó Python como lenguaje de programación debido a la extensa oferta de bibliotecas para trabajar en este dominio. El resultado fue un *script* que procesa las listas de reproducción del *dataset* mencionado con anterioridad y genera varios secuenciamientos por cada *playlist*. Cada secuenciamiento corresponde a una elección de hiperparámetros (modelo o heurística) detallados a lo largo de la sección 2.2. Esto se realizó para facilitar la experimentación. Pero, si se quisiera poner la herramienta disponible para que usuarios pudieran ordenar sus *playlists*, se pueden fijar los valores

que mejores resultados nos dieron para nuestros experimentos, los cuales se hayan en la sección 3. La herramienta también puede recibir como entrada el identificador de una lista de reproducción de Spotify y ejecutar el mismo proceso.

2. METODOLOGÍA

2.1. Datos

Si bien podríamos intentar dar una definición formal de "salto brusco" basándonos en características de las canciones como la tonalidad y el tempo de la canción, así cómo también utilizando meta-datos cómo artista, género o año de publicación, nosotros optamos por utilizar un *dataset* de *playlists* curadas como fuente de verdad para medir los resultados de nuestra herramienta. Cuando decimos que las *playlists* son curadas, nos referimos al hecho de que fueron concebidas por usuarios reales. Los mismos experimentos de este trabajo se podrían haber realizado con algún *dataset* de *playlists* generadas automáticamente mediante algún algoritmo. Por ejemplo, *Discover Weekly* de Spotify, que se actualiza todas las semanas según los gustos del usuario [1]. Sin embargo, debido a que nos interesa medir qué tanto se asemeja lo que hace nuestro algoritmo a lo que hacen los usuarios cuando construyen listas de reproducción, se descartó el uso de *playlists* generadas por otros algoritmos para evaluar resultados.

Para este trabajo, utilizamos el ya mencionado *dataset* **Art of the Mix 2011**, que consiste en 101.343 listas de reproducción curadas por usuarios entre 22/01/1998 y 17/06/2011. Es interesante advertir que no hay un criterio estricto impuesto a los usuarios en la confección de las *playlists* para este *dataset*. Esto implica que no se garantiza que estas hayan sido concebidas buscando que sean lo más homogéneas posible en cuanto al orden de sus canciones. Sin embargo, resulta útil para la experimentación, dado que si los secuenciamientos obtenidos por nuestro método son correlativos a los del *dataset*, significa que, efectivamente, varios usuarios utilizaron la homogeneidad como criterio a la hora de generar sus listas de reproducción.

El *dataset* es un archivo en formato JSON que representa cada *playlist* como una lista, donde cada elemento es, a su vez, una lista de dos elementos: el primero, una tupla con el nombre de la canción y el nombre del artista; el segundo, el identificador de la canción en el Million Song Dataset [2]. Aunque en varias canciones este campo no se encuentra disponible, por lo que decidimos no utilizarlo. Tampoco incluye meta-datos como la duración de la canción, el nombre del álbum, el año de lanzamiento o el género. Esto dificulta la tarea de corroborar que los audios obtenidos corresponden efectivamente a las canciones deseadas. En la sección 2.2.1 se detalla este proceso.

Hay otros campos disponibles que también decidimos no usar, como la categoría de la *playlist*, la fecha de confección o el usuario que la creó. En el caso de la categoría, porque según lo observado hay muy pocas y se repiten mucho. Tampoco son muy significativas, por ejemplo *Mixed Genre*. En cuanto a los demás campos, tampoco consideramos que aportaran valor al análisis.

[1] `https://medium.com/the-sound-of-ai/spotifys-discover-weekly-explained-breaking-from-your-music-bubble-or-maybe-not-b506da144123`

[2] `http://millionsongdataset.com/`

De las 101.343 *playlists* del *dataset*, para este trabajo se consiguieron los audios de 973 *playlists* completas. Esto se hizo para limitar el alcance del trabajo, ya que el proceso de descarga no es instantáneo y son muchas canciones. Más detalles sobre el proceso de descarga se presentan en la sección 2.2.1.

2.2. Procesamiento

2.2.1. Obtención de fragmentos de canciones

El primer desafío en la construcción de la herramienta fue conseguir las muestras de las canciones que conforman las listas de reproducción del *dataset*. Dado que, como mencionamos en la sección anterior, solo contamos con el nombre de la canción y de su artista, necesitamos un mecanismo para realizar una búsqueda con estos valores y obtener los audios correspondientes a los resultados.

Primero se consideró utilizar la API de Spotify, la cual ofrece un *endpoint* para búsquedas [3] donde se obtiene el identificador de las canciones y otro *endpoint* para obtener información sobre un *track* dado su identificador [4]. Entre la información obtenida de este último, se encuentra un campo que contiene una URL a un archivo con una muestra de 30 segundos del *track*. Esto parecía ideal y de hecho se menciona como método utilizado en uno de los trabajos más recientes [Ire+19], por lo cual se optó como primer método.

La implementación tuvo sus desafíos, ya que la búsqueda de la API resultó ser muy inferior a la de la plataforma. Como consecuencia, usamos algoritmos para remover palabras que pudieran contener errores tipográficos o discrepancias con el nombre almacenado en la plataforma hasta conseguir un resultado. Una vez finalizada la implementación, nos encontramos con que en la gran mayoría de *playlists* tenían al menos una canción cuya muestra no estaba disponible, por lo que se descartó esta solución.

Optamos, entonces, por cambiar de estrategia y utilizar la API de Youtube Music para realizar la búsqueda de las canciones. El término que se utiliza para cada una es: artista - título. En la práctica encontramos que utilizar un separador puede ser muy efectivo para conseguir buenos resultados, sobre todo en casos donde tanto el artista como el titulo de la canción contienen varias palabras. A su vez la API nos permite filtrar los resultados por canciones, ya que en principio la plataforma incorpora videos musicales, los cuales queremos evitar debido a la gran cantidad de ruido que agregan a la pista original. Los resultados de las búsquedas se encuentran ordenados por popularidad, lo que incrementa las probabilidades de descargar la pista deseada.

Habiendo obtenido los resultados de nuestra búsqueda, se iteran (en orden de popularidad) y se intenta descargar el audio correspondiente mediante línea de comando. Una vez conseguida una muestra exitosa para la canción buscada, se detiene la iteración. El resultado de la descarga está, originalmente, en formato WEBM y con alta calidad. A fin de ahorrar espacio en disco, se convierte el audio a formato MP3 y se re-samplea a *24kHz*, ya que es la máxima frecuencia que exigen los modelos utilizados para extraer los

[3] `https://developer.spotify.com/documentation/web-api/reference/search`
[4] `https://developer.spotify.com/documentation/web-api/reference/get-track`

atributos. De hecho, casi todos trabajan con una frecuencia menor excepto MERT.

Finalmente, se recorta el audio original, a treinta segundos a partir del primer minuto de canción. Esto se realizó a fin de tener una muestra representativa de toda la pista que ocupe menos espacio en disco. Si la canción original durase menos de un minuto y medio, se toman los treinta segundos iniciales. Si la canción durase menos de treinta segundos, se utiliza toda la pista.

2.2.2. Generación de *embeddings*

Para generar los *embeddings*, como se mencionó previamente, nos interesa probar con varios modelos auto-supervisados para compararlos en el rendimiento para esta tarea.

A su vez, cada modelo tiene varias capas donde se generan los *embeddings*, y, según la bibliografía, uno debería elegir empíricamente qué capa funciona mejor para la tarea que se debe realizar.

El modelo BYOL-A utiliza como *encoder* una red neuronal convolucional (CNN), donde durante la experimentación la dimensionalidad de los *embeddings* era un hiperparámetro. Dado que, en la publicación del modelo, los autores advierten que la dimensionalidad que obtuvo los mejores resultados fue 2048, se utiliza esta para la implementación final.

El modelo Music2Vec utiliza una CNN para codificar la señal, luego pasa esos *tokens* por una red *Transformer* de 12 capas, con una dimensionalidad de 768 cada una.

El modelo EnCodecMAE consiste, primero, del EnCodec *encoder*, que es otra CNN, y, luego, del *Masked Autoencoder* (MAE) *encoder*, que es otra red *Transformer* de 10 capas para el modelo base, también con una dimensionalidad de 768 cada una.

El modelo MERT-v1-330M también utiliza una red *Transformer*. Esta vez, es de 24 capas con una dimensionalidad de 1024 cada una.

Es así, que decidimos probar con las siguientes combinaciones:

- Music2Vec: tiene 12 capas. Elegimos las capas 0, 5 y 11.
- EnCodecMAE: tiene 10 capas. Elegimos las capas 0, 5 y 9.
- BYOL-A: tiene 5 capas. Elegimos las capas 0 y 4.
- MERT: tiene 24 capas. Eleginmos las capas 0, 6 y 18.

Los *embeddings* de cada modelo varían en dimensionalidad, pero no suelen ser muy extensos. El más grande consta de 2048 números de punto flotante, por lo que se optó por almacenar los *embeddings* en disco a medida que se generaban. Esto se llevó a cabo para evitar volver a computarlos en cada corrida, debido a que se trata de un proceso lento. El proceso utiliza la GPU de la computadora que ejecuta la herramienta con el fin de acelerarlo, ya que si se utilizara la CPU se tardaría mucho más. Los *embeddings* resultantes se almacenan en formato .pt, ya que son tensores de PyTorch, que ofrece una

interfaz amigable para guardarlos y volverlos a cargar.

Asimismo, implementamos Coeficientes Cepstrales en las Frencuencias de Mel (MFCCs), pues en la experimentación nos va a interesar compararlos. Puntualmente, queremos ver si las representaciones de los modelos logran mejores resultados para nuestra tarea que los MFCCs. Estos últimos se basan en la escala de Mel (del inglés, *Melody*), que es una representación de las frecuencias que captura su relevancia para el oído humano. Esto se debe a que nuestra percepción de las frecuencias tiene una naturaleza logarítmica. Para obtenerlos, primero se divide la señal en ventanas muy pequeñas (el estándar es 93 milisegundos para audios sampleados a 22050 Hz). Luego, para cada ventana, se utiliza la Transformada de Fourier de Tiempo Reducido (STFT), obteniendo así un espectrograma (matriz que en cada columna representa las frecuencias identificadas para la ventana correspondiente en la señal original). Una vez que se tiene el espectrograma, se aplica un filtro de Mel, que se construye a raíz de identificar las frecuencias mínima y máxima, partirlas en 128 intervalos iguales y transformarlas utilizando la escala de Mel. Esto da como resultado lo que se conoce como un espectrograma de Mel. Finalmente, se aplica la Transformada de Coseno Discreta (DCT) sobre el espectrograma, y se obtiene así una matriz que codifica los atributos del audio original. Se utilizan 20 coeficientes, que es el valor por defecto [5].

[5] Para la implementación se utilizó librosa.feature.mfcc

2.2.3. Búsqueda del mejor secuenciamiento

Una vez que tenemos los *embeddings*, se genera el grafo que mencionamos con anterioridad al computar las distancias entre los pares de *embeddings* de una *playlist*. Para tomar la distancia entre dos *embeddings*, se utilizó la similitud coseno, para la que PyTorch trae una funcionalidad que la computa con los tensores como entrada [6]. También, se implementó la distancia euclídea. No obstante, se descartó para reducir el alcance de este trabajo, ya que la distancia coseno es la más utilizada para este tipo de problemas. Para representar el grafo utilizamos una matriz de adyacencias, la que se almacena también en disco en formato JSON una vez generada para ahorrar tiempo de cómputo. Se llevó a cabo de esta manera debido a que, si se necesita el grafo de una *playlist* que ya fue generado previamente, se lee del disco en lugar de computarse de nuevo.

Con el grafo ya computado, procedemos a correr distintas heurísticas de TSP que se implementaron para conseguir un camino, empezando desde el vértice correspondiente a la primera canción en la *playlist* original. Las heurísticas implementadas son:

- Vecino más cercano (Nearest Neighbour o NN) [1].
- Arista más corta (Shortest Edge) [2].
- Árboles generadores mínimos (Minimum Spanning Tree o MinSpan) [3].
- Búsqueda Tabú (Tabu Search) con 3 vecinos, vetando hasta 5 turnos los movimientos utilizados y con un máximo de 50 iteraciones [4].

Notamos $G = (V, X)$ al grafo asociado a una lista de reproducción de tamaño n, donde $V = \{0, 1, ..., n-1\}$ son los vértices (cada vértice es una canción) y $X = \{(0, 1), (0, 2), ..., (n-2, n-1)\}$ son las aristas. También notamos $peso : X \to \mathbb{R}$ a la función que obtiene el peso asociado a una arista, es decir, la distancia entre los *embeddings* de dos canciones de la *playlist*. Por último, notamos $d_G(v)$ al grado del vértice v en el grafo G, es decir, con cuantos otros vértices está conectado v.

Algorithm 1 Vecino más cercano

```
procedure VECINOMASCERCANO(G = (V, X))
    v ← 0
    H ← [v]
    while |H| < n do
        w ← arg mín{peso(v, w), w ∈ V − H}
        H ← H + w
        v ← w
    end while
    return H
end procedure
```

Un árbol generador de un grafo $G = (V, X)$ es otro grafo AG tal que sus mismos vértices son V y sus aristas son un subconjunto de X de manera tal que no se forman ciclos. Al agregar cualquier arista de X que no esté en el AG, se forma un ciclo y ese ciclo

[6] `https://pytorch.org/docs/stable/generated/torch.nn.functional.cosine_similarity.html`

Algorithm 2 Arista más corta

procedure ARISTAMASCORTA(G = (V, X))
  $X_t \leftarrow \emptyset$
  $i \leftarrow 1$
  while $i \leq n - 1$ **do**
    $e \leftarrow \arg\min\{peso(u, v), (u, v) \in X \wedge d_{X_t}(u) \leq 1 \wedge d_{X_t}(v) \leq 1 \wedge no\ forma\ ciclo\}$
    $X_t \leftarrow X_t \cup \{e\}$
    $i \leftarrow i + 1$
  end while
  $X_t \leftarrow X_t \cup \{(u, v)\}$ con $d_{X_t}(u) = 1 \wedge d_{X_t}(v) = 1$
  $v \leftarrow 0$
  $H \leftarrow [v]$
  while $|H| < n$ **do**
    $w \leftarrow \text{dameUno}\{w, (v, w) \in X_t \wedge w \notin H\}$
    $H \leftarrow H + w$
    $v \leftarrow w$
  end while
  return H
end procedure

contiene a la arista nueva.

Un árbol generador mínimo AGM es, entonces, un árbol generador de un grafo pesado que minimiza la suma de los pesos de sus aristas. Cualquier otro árbol generador del mismo grafo tiene una suma de pesos igual o mayor que su AGM.

Para construir un AGM a partir de un grafo G, implementamos el algoritmo de Kruskal, que consiste en ordenar los pesos de las aristas de menor a mayor e ir seleccionando aquellas que no formen un ciclo. Es similar a lo que hicimos en el algoritmo 2, con la diferencia de que no se tienen en cuenta los grados de los vértices. Es decir, en un AGM puede haber un vértice conectado a más de dos vértices distintos.

Algorithm 3 Árbol generador mínimo

procedure ÁRBOLGENERADORMÍNIMO(G = (V,X))
  $T \leftarrow AGM(G)$
  $E \leftarrow$ duplicar las aristas de T
  $D \leftarrow$ recorrer E usando DFS
  $H \leftarrow$ armar el camino siguiendo el orden dado por D
  return H
end procedure

Está demostrado que si el grafo de entrada es euclideano, es decir, para cualquier tripla de vértices i, j, k se cumple la desigualdad triangular $peso(i, k) \leq peso(i, j) + peso(j, k)$, entonces la solución encontrada por la heurística es 1-aproximada. Esto significa que el peso del camino encontrado, dividido por el peso del camino óptimo, es menor o igual que dos.

Sin embargo, en "Generating Similarity-Based Playlists Using Traveling Salesman Algorithms" de Pohle, Pampalk y Widmer [PPW05], se implementa esta heurística pese a no cumplirse la desigualdad en los grafos de entrada y reporta resultados favorables, por lo que nosotros optamos por proceder de la misma forma.

La búsqueda tabú consiste en partir desde una camino inicial para mejorarlo mediante intercambios de aristas. Nosotros implementamos la vecindad 2-opt, que consiste en reemplazar dos aristas $(i, i+1)$ y $(j, j+1)$, por la aristas (i, j) y $(i+1, j+1)$. Notar que esto invierte el orden de los vértices del camino entre $i+1$ y j.

Para evitar caer en un máximo local, se utiliza una lista tabú, que funciona como una memoria a corto plazo donde se almacenan los movimientos realizados a fin de vetarlos por una cantidad definida de iteraciones.

Algorithm 4 Búsqueda Tabú

procedure BÚSQUEDATABÚ(G = (V,X), caminoInicial, cantMaxIteraciones, cantIteracionesTabú, tamañoVecindad)
  $mejorCamino \leftarrow$ caminoInicial
  $T \leftarrow$ inicializar lista tabú con la cantidad de iteraciones en 0 para todas las aristas.
  $i \leftarrow 0$
  while $i <$ cantMaxIteraciones **do**
    $vecindad \leftarrow$ generar movimientos 2-opt de hasta tamañoVecindad
    $mejorMovimiento \leftarrow$ movimiento que minimiza el peso del camino y no está en T
    $mejorCamino \leftarrow$ aplicar $mejorMovimiento$ a $mejorCamino$
    $T \leftarrow$ marcar $mejorMovimiento$ como tabú por cantIteracionesTabú
    $T \leftarrow$ actualizar iteraciones restantes
  end while
  return $mejorCamino$
end procedure

2.3. Metodología de evaluación

2.3.1. Preliminares

Definimos una *playlist* p de tamaño n como $p = [t_1, ..., t_n]$, donde t_i es la i-ésima canción de la lista de reproducción. Asimismo, notaremos al largo de la *playlist* p como $|p|$. Es decir, $p = [t_1, ..., t_n]$ entonces $|p| = n$. Cada canción se identifica con la posición que ocupa en la *playlist* original, es decir $t_i = i - 1 \in \mathbb{N} \cup \{0\}$. De esta forma, $p = [0, 1, 2, ..., n - 1]$. Notamos p' a un secuenciamiento de p. Por ejemplo, si p es una *playlist* de 5 canciones $[0, 1, 2, 3, 4]$, entonces un posible secuenciamiento p' podría ser $[0, 3, 1, 4, 2]$. Definimos también $cabeza(p) = t_1$ y $cola(p) = [t_2, ..., t_n]$. También utilizamos p_i para referirnos a la i-ésima canción t_i.

Para generar secuenciamientos aleatorios, definimos la función *mezcla*, que toma como entrada una lista y retorna otra del mismo largo y con los mismos elementos, pero no necesariamente en sus mismas posiciones [7]. Notamos entonces $r(p) = cabeza(p) + mezcla(cola(p))$ a un secuenciamiento aleatorio de la playlist p. La primer canción se deja fija para que la comparación tenga sentido, ya que todas las heurísticas implementadas saben desde qué canción comenzar a secuenciar.

2.3.2. Distancias entre secuencias

Una función para medir la distancia entre dos cadenas del mismo largo es la **distancia de Hamming** [Ham50], la que cuenta la cantidad de posiciones para las que hay elementos distintos en ambas cadenas.

$$hamming(p, p') = \sum_{i=1}^{|p|} neq(p_i, p'_i) \tag{2.1}$$

donde

$$neq(a, b) = \begin{cases} 0 & \text{si } a = b \\ 1 & \text{si } a \neq b \end{cases}$$

Notemos que en el caso en que la *playlist* original $p = [0, 1, 2, 3, 4]$ y el secuenciamiento $p' = [0, 2, 3, 4, 1]$, $hamming(p, p') = 4$, dado todas las canciones de $cola(p)$ están en posiciones que no son las originales. Sin embargo, podríamos decir que el secuenciamiento estuvo muy cerca de la *playlist* original, ya que bastaría con un *shift* a la derecha de $cola(p)$ para que quede exactamente igual.

Por ese motivo, otra distancia muy utilizada para este tipo de problemas es la distancia de edición o **distancia de Levenshtein** [Lev65], que mide la cantidad de ediciones que hay que hacer para convertir una cadena en otra. Las posibles ediciones son insertar un elemento, eliminarlo , o sustituirlo.

[7] Para la implementación se utilizó la función *shuffle* de la biblioteca estándar de Python para trabajar con aleatoreidad

$$levenshtein(p, p') = \begin{cases} |p| & \text{si } |p'| = 0 \\ |p'| & \text{si } |p| = 0 \\ levenshtein(cola(p), cola(p')) & \text{si } cabeza(p) = cabeza(p') \\ 1 + \text{mín} \begin{cases} levenshtein(cola(p), p') \\ levenshtein(p, cola(p')) \\ levenshtein(cola(p), cola(p')) \end{cases} & \text{si no} \end{cases} \tag{2.2}$$

Volviendo al ejemplo anterior, si $p = [0, 1, 2, 3, 4]$ y $p' = [0, 2, 3, 4, 1]$, $levenshtein(p, p') = 2$, pues las ediciones necesarias para convertir p' en p son eliminar el 1 del final e insertarlo entre el 0 y el 2.

Se puede ver de la definición que en el peor caso la distancia entre dos cadenas es igual al largo de la cadena más larga, ya que basta con sustituir todas las posiciones de la cadena más corta e insertar las posiciones restantes. De esta manera, si comparamos siempre cadenas del mismo largo, sabemos que la distancia va a ser, como máximo, ese mismo largo.

2.3.3. Distancias de contenido

Notamos al *embedding* de una canción t generado por un modelo m en la capa c_i como $m_{c_i}(t)$. De esta forma, podemos pensar en un modelo y en una capa del mismo como una función $m_{c_i} : \mathbb{N} \cup \{0\} \Rightarrow \mathbb{R}^n$. La dimensionalidad del *embedding* dependerá del modelo elegido, por ejemplo: $Music2Vec_5 : \mathbb{N} \cup \{0\} \Rightarrow \mathbb{R}^{768}$, $BYOLA_4 : \mathbb{N} \cup \{0\} \Rightarrow \mathbb{R}^{2048}$.

Además, notamos la distancia entre dos *embeddings* e_i y e_j como $d(e_i, e_j)$ y esta está dada por la función de **similitud coseno** [HKP12], la cual se define como

$$S_c(e_i, e_j) = \frac{e_i \cdot e_j}{||e_i||_2 \cdot ||e_j||_2} \in [-1, 1] \tag{2.3}$$

Dado que la similitud coseno vale -1 cuando los *embeddings* son opuestos y 1 cuando son idénticos, se define la **distancia coseno** mediante la ecuación 2.4, que vale 0 cuando los *embeddings* son idénticos y 2 cuando son opuestos.

$$d(e_i, e_j) = 1 - S_c(e_i, e_j) \in [0, 2] \tag{2.4}$$

Definimos, entonces, el peso de una *playlist* p en el espacio dimensional generado por el modelo m como la suma de distancias entre cada par de canciones continuas de la lista. Dado que, para evaluar resultados, vamos a comparar los pesos de *playlists* con diferente largo, nos va a interesar normalizarlos. Para esto, podemos utilizar la cota superior para cada distancia por la ecuación 2.4:

$$W(p, m_{c_i}) = \frac{\sum_{i=1}^{|p|-1} d(m_{c_i}(p_i), m_{c_i}(p_{i+1}))}{2(|p| - 1)} \tag{2.5}$$

Ahora bien, W puede tomar valores pequeños tanto si todas las canciones de la playlist tienen distancias muy pequeñas entre sí como si esta fue ordenada cuidadosamente para

que cada transición sea suave. A fin de poder medir esto, tomamos para cada lista y para cada modelo una cantidad n_rand de muestras aleatorias y dividimos su promedio por el peso original. Si este es mayor a uno, significa que las variaciones aleatorias de las mismas canciones generan caminos más pesados, y por ende el orden original tiene transiciones más suaves. Esto se expresa en la ecuación 2.6:

$$\tilde{W}(p, m_{c_i}) = \frac{\frac{\sum_{n\_rand} W(r(p), m_{c_i})}{n\_rand}}{W(p, m_{c_i})} \tag{2.6}$$

2.3.4. Entropía de Shannon

Otra forma de caracterizar a las *playlists* es utilizar la entropía de Shannon en función de los géneros de las canciones. Si bien para calcularla recurrimos a meta-datos y no al contenido del audio, este numero igualmente puede servir para medir qué tan predecibles pueden ser las listas con las que estamos trabajando (similar a lo que se hace en [PPW05] para medir la coherencia de las listas). La fórmula para calcularla es la siguiente:

$$entropia(p) = -\sum_{p_i \in p} prob(genero(p_i))\ log_2(prob(genero(p_i))) \tag{2.7}$$

Ejemplo: tenemos una playlist de 5 canciones cuyos géneros son:

- 0: {Rock, Prog Rock}
- 1: {Prog Rock, Grunge}
- 2: {Pop}
- 3: {Hip-hop, Nu metal}
- 4: {Pop, Hip-hop}

Entonces, las probabilidades de que una canción pertenezca a cada género son:

- Rock: $\frac{1}{5}$
- Prog Rock: $\frac{2}{5}$
- Grunge: $\frac{1}{5}$
- Pop: $\frac{2}{5}$
- Hip-hop: $\frac{2}{5}$
- Nu metal: $\frac{1}{5}$

Y podemos calcular la entropía de esta playlist como:

$$-\frac{1}{5}\log_2\left(\frac{1}{5}\right) \times 3 - \frac{2}{5}\log_2\left(\frac{2}{5}\right) \times 3 = 4{,}1794705708$$

Lo cual es bastante alto. Supongamos otra playlist más coherente:

- 0: {Hip-hop, Pop}
- 1: {Hip-hop, Rap, Pop}
- 2: {Hip-hop, Pop}
- 3: {Hip-hop}
- 4: {Hip-hop, Pop}

Entonces las probabilidades de que una canción pertenezca a cada género son:

- Pop: $\frac{4}{5}$
- Hip-hop: 1
- Rap: $\frac{1}{5}$

Y podemos calcular la entropía de esta playlist como:

$$-\frac{4}{5}\log_2\left(\frac{4}{5}\right) - \log_2(1) - \frac{1}{5}\log_2\left(\frac{1}{5}\right) = 0{,}72192809488$$

Que nos da un número menor a 1, por lo cual es mejor que la distribución uniforme.

2.3.5. Saltos de género

Para aproximarnos a la cantidad de saltos bruscos que posee una lista de reproducción, podemos contar cuantos saltos hay en los cuales la canción actual no tiene ningún género en común con la siguiente:

$$genre\_fluctuations(p) = \sum_{i=1}^{|p|-1} genres(p_i) \cap genres(p_{i+1}) = \varnothing \tag{2.8}$$

Los géneros son a nivel de artista, por lo que pueden no ser muy precisos. Sin embargo, sirven para dar una noción, ya que dos canciones con características acústicas similares tienen una alta probabilidad de pertenecer al mismo género musical.

2.3.6. Puntajes

Para este trabajo utilizaremos dos puntajes principales para medir qué tan cerca la *playlist* p' está de la *playlist* original p: el puntaje de edición y el puntaje de edición pesado.

El puntaje de edición consiste en normalizar la distancia de Levenshtein. Esto se debe a que queremos tener un número que nos diga qué tan bien resultó el algoritmo sin importar el largo de la lista de reproducción. Como el método comienza siempre desde la primera canción, sabemos que, en el peor caso, la distancia de Levenshtein es $|p| - 1$.

$$score(p') = 1 - \frac{levenshtein(p, p')}{|p| - 1} \tag{2.9}$$

Se ve que, de esta manera, $score(p') \in [0, 1]$, donde 0 implica que es necesario editar todas las canciones de $cola(p)$ para que se convierta en $cola(p')$, mientras que 1 implica que ya son idénticas.

Ahora bien, *score* penaliza de igual manera si el secuenciamiento puso una canción muy similar como si puso una diametralmente distinta, con tal de que no sea la misma canción que había en la *playlist* original. Para mitigar esto, proponemos una **distancia de edición pesada**, en la cual el peso de cada edición es equivalente a la distancia entre sus *embeddings*:

$$weighted\_edit\_distance(p, p', m_{c_i}) = \sum_{1}^{|p|} d(m_{c_i}(p_i), m_{c_i}(p'_i)) \tag{2.10}$$

Luego, utilizamos la lógica del puntaje anterior para establecer uno que use esta distancia pesada:

$$weighted\_score(p', m_{c_i}) = 1 - \frac{weighted\_edit\_distance(p, p', m_{c_i})}{2(|p| - 1)} \tag{2.11}$$

Al igual que en la ecuación 2.5, dado que las distancias entre *embeddings* valen 2 como mucho, esto tiene el efecto de que $weighted\_score(p') \in [0, 1]$ y el significado del puntaje es análogo al anterior. Notar que en 2.5 también dividíamos por $2(|p| - 1)$, dado que las distancias eran entre canciones consecutivas de una misma *playlist*. En cambio, ahora estamos tomando las distancias entre dos listas de reproducción diferentes. Sin embargo, todas las heurísticas para recorrer el grafo saben que deben empezar desde la primera canción de la *playlist* original. Por ende, como siempre la primera canción es la correcta, tenemos, como mucho, una distancia de $|p| - 1$, por lo que el denominador queda igual.

Ahora bien, es verdad que la cantidad de ordenamientos posibles crece de manera exponencial con cada pista que se agrega a la *playlist*. Es decir, obtener un puntaje de 0,5 para una lista de reproducción $p1$ de 5 canciones se podría considerar bajo, dado que hay $4! = 24$ combinaciones posibles para ordenar $cola(p1)$. En cambio, obtener un puntaje de 0,5 en una *playlist* $p2$ de 50 canciones se podría considerar alto, ya que habremos ubicado con éxito 24 canciones de $cola(p2)$ (recordemos que la cabeza siempre está bien ubicada) cuando hay 49! combinaciones posibles para $cola(p2)$.

Esto nos llevó a pensar en otra normalización si utilizamos el número combinatorio entre el largo de la *playlist* y la distancia obtenida. Para esto, calculamos la probabilidad de asignar de forma correcta k canciones en una lista de reproducción de tamaño $n+1$ (de esta forma notaremos brevemente a n como el largo de $cola(p)$). Esto es: cuántas combinaciones de k canciones puedo tomar de las n, multiplicado por la cantidad de combinaciones en las que las $n-k$ canciones restantes estén **todas** mal asignadas, sobre la cantidad total de permutaciones.

La cantidad de combinaciones está dada por $\binom{n}{k}$, mientras que la cantidad total de permutaciones está dada por $n!$. Ahora, para calcular la cantidad de combinaciones donde hay $n - k$ canciones mal asignadas, utilizamos el número **subfactorial**, que calcula, para un largo de cadena m, la cantidad de desarreglos posibles. Un desarreglo es una permutación donde ninguno de los elementos de la secuencia original queda ubicado en su posición

original.

Por ejemplo, para una *playlist* de 5 canciones $p = [0, 1, 2, 3, 4]$, tenemos los siguientes desarreglos para $cola(p)$: $[0, 2, 1, 4, 3]$, $[0, 2, 3, 4, 1]$, $[0, 3, 1, 4, 2]$, $[0, 3, 4, 1, 2]$, $[0, 3, 4, 2, 1]$, $[0, 4, 1, 2, 3]$, $[0, 4, 3, 1, 2]$ y $[0, 4, 3, 2, 1]$. En todas las demás permutaciones de $cola(p)$ hay al menos un elemento que se encuentra en su posición original.

Notamos D_m al subfactorial de m y tiene la siguiente fórmula cerrada:

$$D_m = m! \sum_{i=0}^{m} \frac{(-1)^i}{i!} \tag{2.12}$$

Esta se puede obtener de restar a la cantidad total de permutaciones $m!$ todas aquellas donde, al menos, un elemento aparece bien ubicado, y para eso hay que sumar las combinaciones donde aparece exactamente un elemento bien ubicado, luego donde aparecen dos, y así de manera sucesiva. En el capítulo 5 de *Concrete Mathematics* [GKP94] se encuentra una demostración de la fórmula. Usando el ejemplo anterior, tenemos que $D_4 = 9$. Notar que $D_0 = 0! = 1$ y $D_1 = 1 - 1 = 0$.

Al juntar todo, notamos $P(k, n)$ a la probabilidad de asignar correctamente k canciones de n:

$$P(k, n) = \frac{\binom{n}{k} \times D_{n-k}}{n!} \tag{2.13}$$

Siguiendo con el ejemplo, la probabilidad de asignar bien 2 canciones de 4 es $P(2, 4) = \frac{\binom{4}{2} \times D_2}{4!} = \frac{6 \times 1}{24} = 0{,}25$. Esto es correcto dado que de 24 permutaciones posibles de $cola(p)$, solo en 6 se asignan 2 bien y 2 mal (recordemos que no contamos la primera posición): $[0, 1, 2, 4, 3]$, $[0, 1, 4, 3, 2]$, $[0, 1, 3, 2, 4]$, $[0, 4, 2, 3, 1]$, $[0, 3, 2, 1, 4]$, $[0, 2, 1, 3, 4]$.

También, podemos observar que, dado que tenemos la distancia de Hamming computada, que indica cuántas canciones quedaron mal colocadas, podemos calcular la probabilidad de asignar **mal** k pistas de n, dado que es lo mismo que calcular la probabilidad de asignar bien $n - k$ canciones. Notamos entonces $P'(k, n)$ a la probabilidad de asignar mal k canciones de n y lo calculamos de la siguiente forma:

$$P'(k, n) = P(n - k, n) = \frac{\binom{n}{n-k} \times D_{n-(n-k)}}{n!} = \frac{\binom{n}{n-k} \times D_k}{n!} \tag{2.14}$$

Calculemos todas las probabilidades para $cola(p)$:

$P'(0, 4)$	$\frac{1}{24}$
$P'(1, 4)$	0
$P'(2, 4)$	$\frac{6}{24}$
$P'(3, 4)$	$\frac{8}{24}$
$P'(4, 4)$	$\frac{9}{24}$

Notemos que $P'(1, 4) = 0$ porque $D_1 = 0$, lo cual es correcto dado que si hay una canción mal ubicada, indefectiblemente tiene que haber otra que lo esté también. También, se advierte que la suma de las probabilidades es 1, dado que son todas las combinaciones

posibles.

Finalmente, podemos utilizar esta probabilidad normalizada para obtener un puntaje. Para esto tomamos la probabilidad más alta como referencia. Algo curioso sucede con playlists de tamaño mayor a 5, y es que la probabilidad de asignar mal todas las canciones (utilizando la ecuación 2.14) es ligeramente menor a la de asignar mal todas las canciones excepto una. Por ejemplo $P(5,5) = \frac{1\times44}{120}$ y $P(4,5) = \frac{5\times9}{120}$.

$$comb\_score(p') = \frac{\log P'(hamming(p',p), |p|-1) - \log P'(|p|-2, |p|-1)}{\log P'(0, |p|-1) - \log P'(|p|-2, |p|-1)} \tag{2.15}$$

Vemos que si $p' = p$ entonces $hamming(p', p) = 0$ y por ende $comb\_score(p') = 1$, mientras que si $hamming(p', p) = |p| - 2$ entonces $comb\_score(p') = 0$.

Otro puntaje diferente surge de la idea de contar cuantos *clusters* se forman en un secuenciamiento. Es decir, si el conjunto de canciones posee varios géneros distintos, por ejemplo cinco canciones de *Jazz*, seis de *Rock & Roll* y cuatro de *Blues*, es de esperar que en el secuenciamiento que minimiza la cantidad de saltos bruscos se encuentren las canciones agrupadas por género y que los saltos más bruscos se produzcan justamente en el cambio de género, aunque como vimos en el ejemplo de la sección 1.3 si se eligen bien las canciones que unen estos *clusters* se puede suavizar la transición de un género a otro.

Con esta idea en mente, decidimos generar un puntaje en base a la cantidad de subarreglos del secuenciamiento en los cuales, para cada par de temas consecutivos, la distancia entre sus posiciones en la *playlist* original no sea mayor a uno. Este puntaje se basa en la noción de que, si estamos comparando contra una lista homogénea que tiene varios géneros, es de esperar que haya agrupaciones por estos y que tanto el orden de los grupos no impacta tanto en la experiencia de escucha. Es verdad que hay géneros que están mas cerca de otros, pero si son todos muy distintos el orden se vuelve indiferente en la experiencia de escucha. Por ejemplo, supongamos una lista $p = [0, 1, 2, 3, 4, 5, 6, 7, 8]$, donde $[0, 1, 2]$ son canciones de música clásica, $[3, 4, 5]$ de música electrónica y $[6, 7, 8]$ de salsa. Tomemos un secuenciamiento $p' = [0, 1, 2, 8, 7, 6, 3, 4, 5]$. En este caso, pasamos de música clásica a salsa, que es igualmente válido que pasar de clásica a electrónica, ya que no hay elementos en común entre estos tres géneros. A su vez, el orden relativo de cada género tampoco impacta demasiado en la experiencia de escucha en este caso, por ende consideramos escuchar las canciones de salsa en el orden inverso es igualmente válido. Para conseguir estos subarreglos, utilizamos el algoritmo 5.

Para priorizar los secuenciamientos con subarreglos de mayor longitud, se define el puntaje 2.16, donde utilizamos el largo de cada subarreglo como exponente. Este es el único puntaje que no se encuentra normalizado, ya que por cómo lo definimos habría que dividir por 2^n para que quede entre 0 y 1, y al ser este un número tan grande quedarían todos los resultados muy cerca del 0.

$$subarray\_score(p') = \sum_{s \in subarreglos(p')} 2^{|s|-1} \tag{2.16}$$

En línea con lo anterior, otro puntaje que puede servir para obtener más información sobre la naturaleza del orden en las listas de reproducción es utilizar los géneros de las

Algorithm 5 Subarreglos

```
procedure SUBARREGLOS(p)
    subarreglos ← []
    i ← 0
    while i < |p| − 1 do
        subarregloActual ← [p[i]]
        while i < |p| − 1 & abs(p[i] − p[i + 1]) = 1 do
            subarregloActual.push(p[i + 1])
            i ← i + 1
        end while
        if |subarregloActual| > 1 then
            subarreglos.push(subarregloActual)
        end if
        i ← i + 1
    end while
    return subarreglos
end procedure
```

canciones. Podemos contar la cantidad de veces que se cambia de género en una lista y utilizarlo para construir un puntaje donde el valor más alto implica que no hay cambio de género en toda la lista y el valor más bajo implica que cada par de canciones pertenecen a un género distinto. Esto se expresa mediante la ecuación 2.17

$$genre\_fluctuation\_score(p') = 1 - \frac{genre\_fluctuations(p')}{|p'| - 1} \tag{2.17}$$

Para todos los puntajes presentados anteriormente, generamos una variante donde les restamos el promedio de los resultados obtenidos para instancias aleatorias. Esto, a fin de poder visualizar mejor los resultados obtenidos.

$$adjusted\_score(p') = score(p') - \frac{\sum_{n\_rand} score(r(p))}{n\_rand} \tag{2.18}$$

$$adjusted\_weighted\_score(p', m_{c_i}) = weighted\_score(p', m_{c_i}) - \frac{\sum_{n\_rand} weighted\_score(r(p), m_{c_i})}{n\_rand} \tag{2.19}$$

$$adjusted\_comb\_score(p') = comb\_score(p') - \frac{\sum_{n\_rand} comb\_score(r(p))}{n\_rand} \tag{2.20}$$

$$adjusted\_subarray\_score(p') = subarray\_score(p') - \frac{\sum_{n\_rand} subarray\_score(r(p))}{n\_rand} \tag{2.21}$$

3. EXPERIMENTACIÓN

En este capítulo planteamos las hipótesis del trabajo (algunas fueron introducidas previamente como interrogantes) y hacemos uso de la herramienta que construimos para poder validarlas o falsearlas según los resultados obtenidos. Se describe también la configuración de cada experimento y un análisis de los resultados que se alcanzaron.

3.1. Configuración general de los experimentos

En todos los experimentos se utilizaron los audios correspondientes a 973 *playlists* del Art of The Mix 2011 *dataset* que se mencionaron en la sección 2.1. Los *embeddings* correspondientes fueron generados con fragmentos de treinta segundos a partir del minuto de canción. Para construir el grafo se empleó la función de distancia coseno dada por la ecuación 2.4 para calcular los pesos de las aristas pertenecientes a dos canciones.

Por cada *playlist*, se ejecutaron todos los modelos listados en 2.2.2 y, por cada uno, se ejecutaron todas las heurísticas mencionadas en 2.2.3. Sin embargo, cada experimento luego puede fijar el modelo o la heurística a fin de explorar y comparar los resultados. Esto se aclara en la configuración de cada experimento.

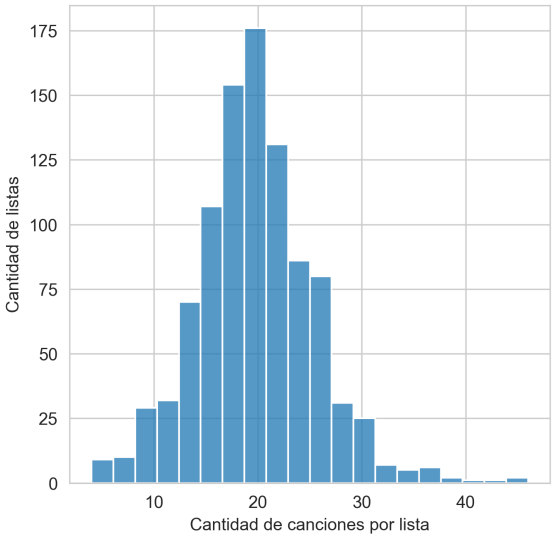
Para evaluar los resultados se usaron los puntajes mencionados en 2.3.6. En algunos experimentos también se hizo un ajuste de los resultados, que consiste en restarle al puntaje de una ejecución, el promedio obtenido por los secuenciamientos aleatorios de ese mismo puntaje, para el mismo largo de *playlist*.

3.2. Análisis exploratorio

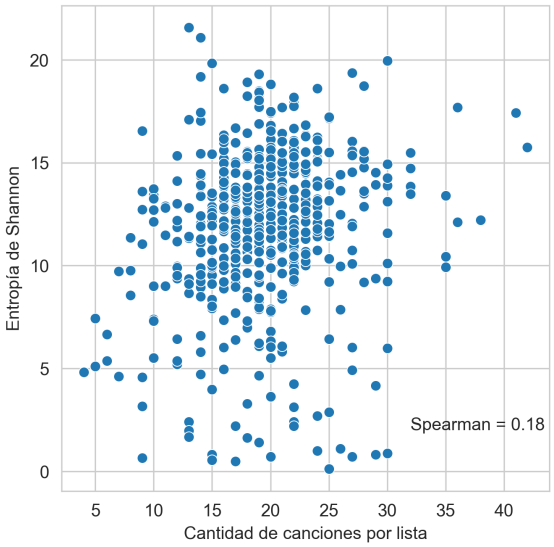
A fin de comprender la naturaleza del *dataset*, realizamos algunas pruebas antes de analizar los resultados de nuestros algoritmos. Lo primero que quisimos observar fue la distribución de las *playlists* en cuanto a su tamaño. Recordemos que cuantas más canciones tenga la lista a secuenciar, menor es la probabilidad de reproducir el orden original. Como se puede ver en la figura 3.1a, las listas de reproducción del conjunto de datos sigue una distribución acampanada, estando la mayoría de estas concentradas entre diez y treinta canciones.

También calculamos la entropía de Shannon de estas listas, para medir qué tanta variación de géneros hay y que correlación hay con el largo de las mismas. Es de esperar que para *playlists* más largas haya una mayor cantidad de géneros. En la figura 3.1b se ve que hay dispersión entre las listas. Calculamos el coeficiente de correlación de Spearman y este da un valor relativamente bajo $(0, 18)$, por lo que no podemos esperar que haya demasiada relación entre la cantidad de canciones y la cantidad de géneros.

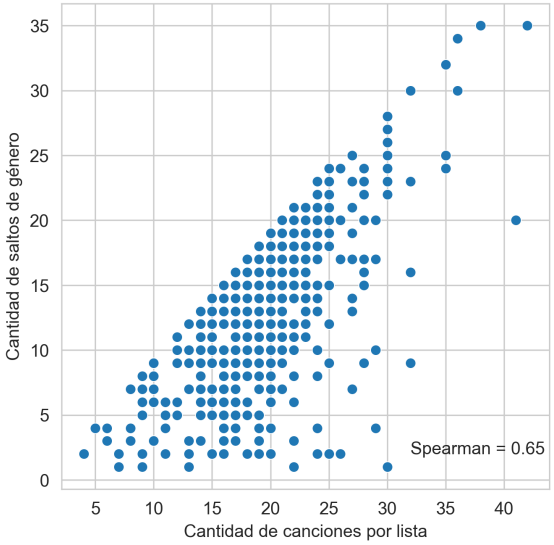
Otro valor a explorar para entender la homogeneidad de las *playlists* originales es la cantidad de saltos de género que contienen. Para ver esto, calculamos $genre\_fluctuation$ para todas las listas utilizando la ecuación 2.8. El resultado se puede ver en la figura



(a) Distribución del largo



(b) Dispersión de la Entropía



(c) Dispersión de géneros

Fig. 3.1: Análisis de las listas de reproducción obtenidas del conjunto de datos.

3.1c. Cómo es de esperarse, hay una mayor correlación entre la cantidad de canciones y la cantidad de saltos, aunque se observa una cantidad considerable de listas que se encuentran por debajo de la diagonal, al menos hasta treinta canciones. Estas pueden ser muy homogéneas debido a que agrupan las canciones por género, aunque, si hay muy pocos géneros en total, cualquier ordenamiento aleatorio también lo sería.

También calculamos W (suma de distancias) de todas las *playlists* que conseguimos descargar del *dataset*, utilizando la ecuación 2.5 y todos los modelos de extracción de audio implementados. Podemos observar de la figura 3.2 que los valores de W se mueven en rangos muy distintos para las mismas *playlists*. Esto podría deberse a que algunos modelos logran capturar mejor lo que el oído humano percibe como similar que otros (asumiendo que la mayoría de las listas están ordenadas por similitud). Otro motivo válido para explicar esta diferencia en los rangos es que simplemente las distancias se mueven en diferentes escalas. Es decir, un modelo X podría decir que las canciones A y B están a una distancia de $0,1$ y que las canciones B y C están a una distancia de $0,2$, mientras que otro modelo Y podría medir $0,3$ para las primeras y $0,6$ para las segundas. Ambos modelos miden lo mismo, solo que en diferente escala.

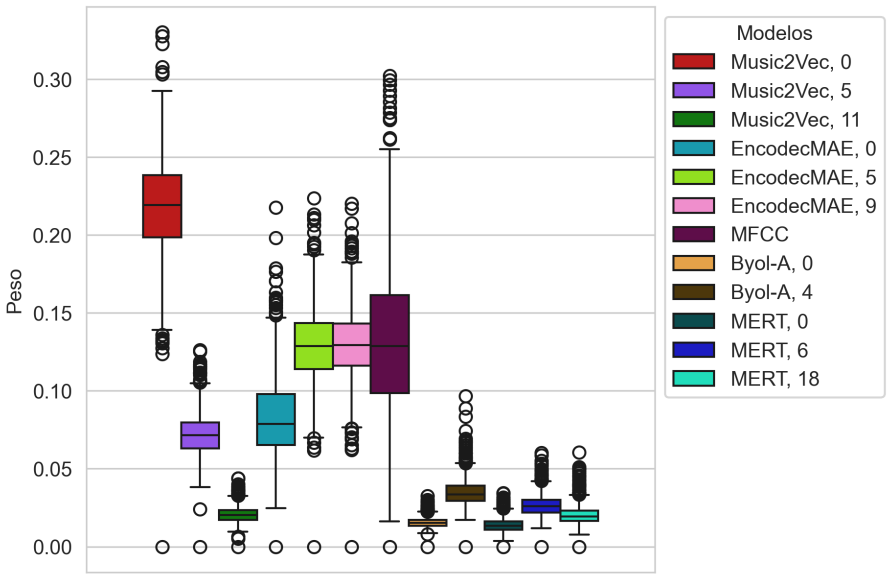


Fig. 3.2: Pesos de las *playlists* originales por modelo

Para ver en cuál de estos dos escenarios nos encontramos, procedemos a calcular $\tilde{W}$ mediante la ecuación 2.6, ya que si nos encontramos en el primer escenario (modelos con menores pesos capturan muy bien el oído humano), deberíamos observar que, al cambiar el orden de las canciones de la *playlist* original al azar, los pesos de los recorridos totales deberían incrementarse notablemente, por lo que el cociente debería dar un número superior a uno. Cómo podemos observar en la figura 3.3, la mediana para todos los modelos está bastante cerca de uno, aunque es verdad que tenemos varios *outliers* para arriba, pero todos los modelos parecen haberse alineado bastante, con lo que nos inclinamos a pensar que nos encontramos en el segundo escenario.

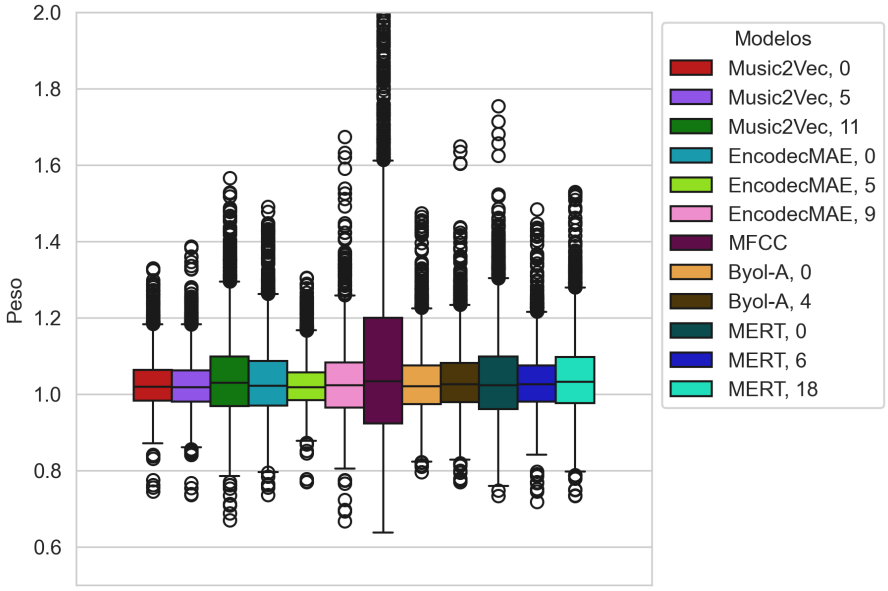
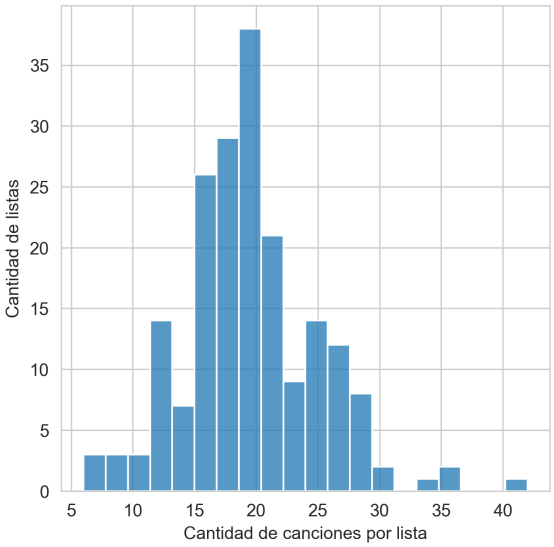


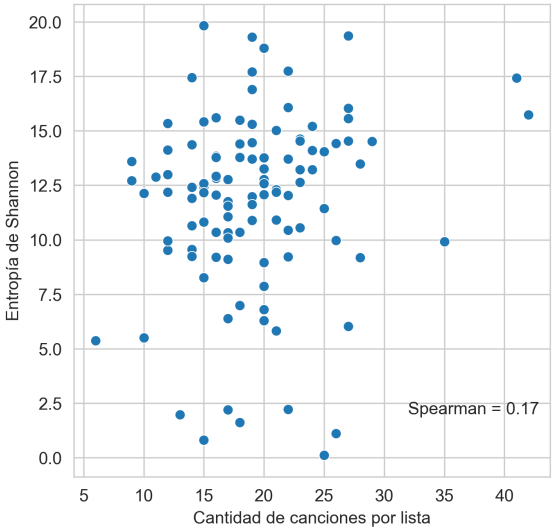
Fig. 3.3: Pesos de las *playlists* originales ordenadas al azar, divididos por los pesos originales, separados por modelo

De hecho, podemos analizar las distribuciones de un subconjunto para ver si se observa una correlación más clara. Tomamos en este caso el veinte porciento de las listas con mayor valor de $\tilde{W}$. Se puede ver en la figura 3.4a que los tamaños de las listas en este subconjunto no varían mucho, lo cual es bueno ya que podemos descartar que esto se deba simplemente a una menor cantidad de canciones. Algo similar se ve en las figuras 3.4b y 3.4c y es que la dispersión hasta se incrementa levemente, por lo que descartamos que haya muchas listas en este subconjunto con pocos géneros para explorar.

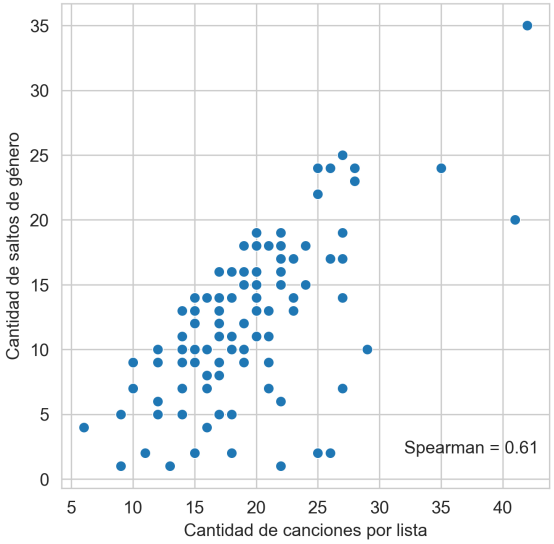
Por último, para finalizar con el análisis exploratorio y pasar a evaluar los resultados obtenidos por nuestra herramienta, podemos observar en la figura 3.5 un mapa de calor para medir la correlación de W entre los modelos implementados. Se puede observar una alta correlación entre la mayoría de modelos, lo que implica que miden distancias similares. La excepción principal es $MFCC$. Esto creemos que se debe a que usa una normalización distinta. También podemos notar que $EnCodecMAE_9$ también tiene una correlación baja con la mayoría de los demás modelos.



(a) Distribución del largo



(b) Dispersión de la Entropía



(c) Dispersión de géneros

Fig. 3.4: Análisis del subconjunto cuya diferencia de pesos con instancias aleatorias es mayor.

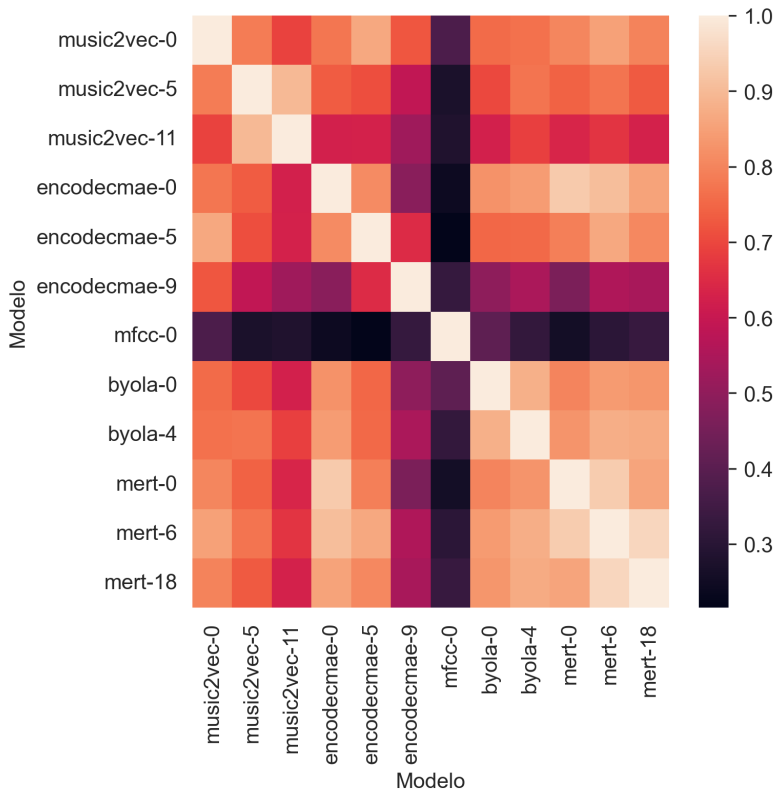


Fig. 3.5: Correlaciones entre los pesos normalizados de las *playlists* originales entre modelos

3.3. Evaluación de caminos obtenidos

3.3.1. Configuración

Para este experimento utilizamos el modelo $MERT_0$, dado que observamos resultados muy interesantes con este modelo mientras construíamos la herramienta. Para encontrar los caminos, se utilizó la heurística de vecino más cercano sobre el grafo dado que es la heurística más similar a cómo ordenamos los humanos, partiendo desde la primera canción de la *playlist* original.

Calculamos W y $genre\_fluctuation\_score$ de los secuenciamientos encontrados por nuestra herramienta utilizando las ecuaciones 2.5 y 2.17 y los comparamos contra los ordenamientos originales y aleatorios (veinte instancias).

El análisis se hace sobre un subconjunto de listas de reproducción. De las 973 disponibles, nos quedamos con el veinte porciento (195 listas aproximadamente) que mayor valor de $\tilde{W}$ tenían. Esto se hizo para que la comparación tuviera más sentido, ya que, si la *playlist* de base no tiene algún criterio de ordenamiento (da lo mismo que aleatorio), seguro nuestra herramienta encuentra un camino con menor peso, dado que está optimizado para minimizar esa función. A este subconjunto lo llamamos **listas homogéneas**.

3.3.2. Hipótesis

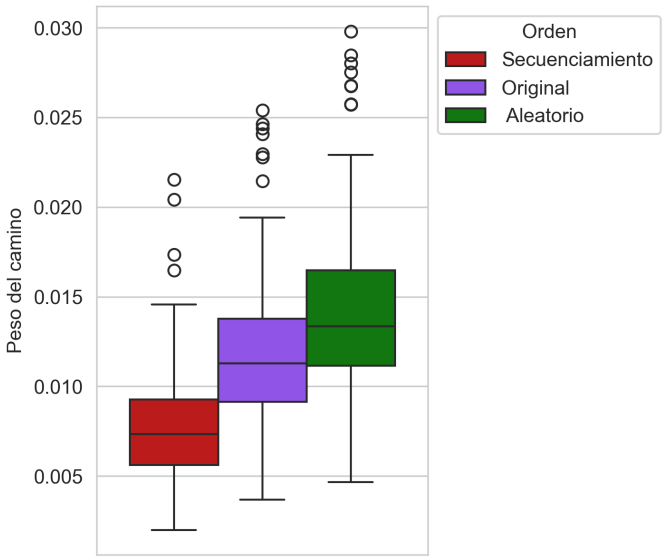
Esperamos ver que nuestra herramienta genere secuenciamientos al menos tan homogéneos como en las listas originales. Esto se debe a que la heurística elegida es golosa y tiene una naturaleza muy similar a la que empleamos los humanos a la hora de ordenar las canciones.

3.3.3. Resultados

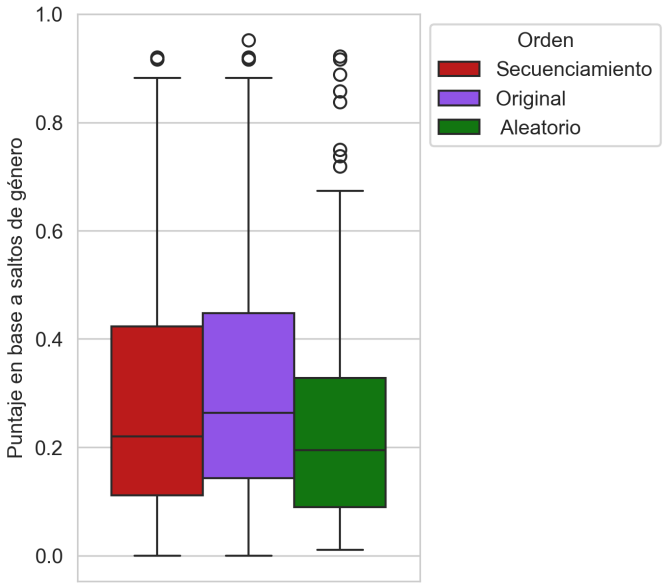
Podemos ver en la figura 3.6a que los pesos de los caminos obtenidos por nuestro algoritmo son, en promedio, más bajos que los de las *playlists* originales.

También, podemos ver en la figura 3.6b que los caminos obtenidos por nuestro algoritmo tienen, en promedio, más fluctuaciones de género que las listas originales, aunque tienen menos que las instancias aleatorias.

Con estos resultados podemos afirmar que los secuenciamientos generados por nuestra herramienta, utilizando $MERT_0$ como modelo y la heurística del vecino más cercano, son más homogéneos que las instancias aleatorias. Con respecto a la hipótesis, no podemos afirmar que nuestros secuenciamientos tengan una menor cantidad de saltos bruscos que las *playlists* originales, si bien los pesos de los caminos en promedio son menores, debido a que poseen una mayor cantidad de saltos de género que todavía no podemos justificar.



(a) Comparación de W (distancia total del recorrido). Valores más bajos indican mejores resultados.



(b) Comparación de *genre_fluctuation_score* (saltos de género). Valores más altos indican mejores resultados.

Fig. 3.6: Resultados con $MERT_0$ y heurística de vecino más cercano sobre el subconjunto de listas homogéneas.

3.4. Evaluación de modelos

3.4.1. Configuración

Calculamos los puntajes mencionados en la sección 2.3.6 para las listas homogéneas (subconjunto con valores más altos de $\tilde{W}$), debido a que las tomamos como fuente de verdad. Los puntajes se calcularon para todos los modelos de extracción de audio implementados, mencionados en la sección 2.2.2. La heurística que se usó para compararlos fue la del vecino más cercano, por el mismo motivo que para el experimento anterior. Para todos los puntajes excepto *adjusted_weighted_score* pudimos generar mil instancias aleatorias para cada tamaño de *playlist*, dado que no se necesitan computar los *embeddings* de las canciones y por ende el costo computacional de la simulación es mucho más bajo.

3.4.2. Hipótesis

Dado que en el experimento anterior vimos que las *playlists* generadas por nuestra herramienta tienen, en promedio, menor peso total y mayor cantidad de fluctuaciones de género que las listas originales, no podemos esperar que se parezcan del todo entre sí, por lo que la mayoría de los puntajes propuestos deberían dar valores bajos.

3.4.3. Resultados

Como podemos observar en la figura 3.7, once modelos obtuvieron un *adjusted_score* cercano al cero. Esto era de esperarse, ya que, al observar que aun el mejor porcentaje de listas tiene un peso total mayor al de nuestros secuenciamientos, es de esperarse que estos no se parezcan demasiado. Sobre todo, utilizando la distancia de Levenshtein, que penaliza de igual manera todas las ediciones.

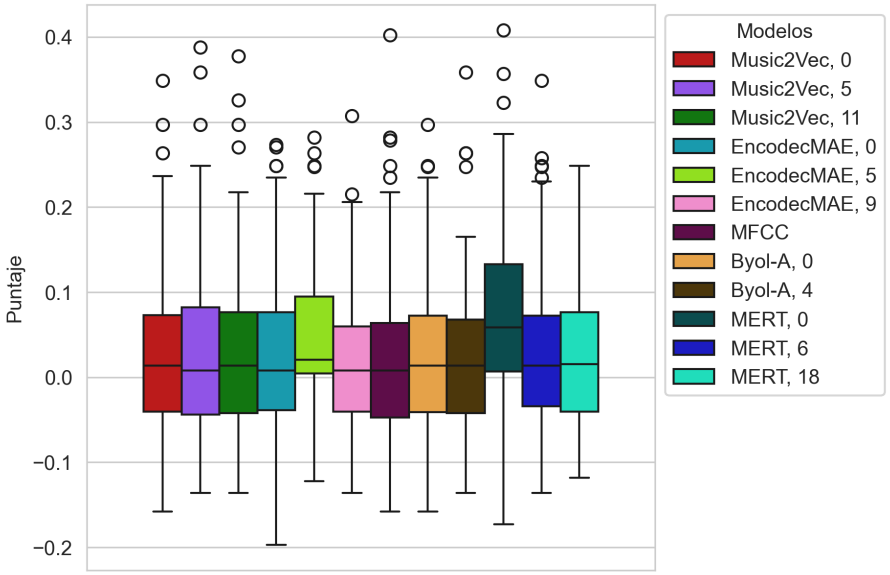


Fig. 3.7: Comparación entre modelos de *adjusted_score* (distancia de Levenshtein) sobre el subconjunto de listas homogéneas. Heurística de vecino más cercano. Valores más altos indican mejores resultados.

Sin embargo, podemos notar que el modelo $MERT_0$ tiene una mediana de *adjusted_score* mayor a la de todos los demás modelos. Pese a no ser un puntaje demasiado alto tampoco, elegimos estudiarlo más en detalle. Podemos observar en el gráfico 3.8 que los secuenciamientos generados por nuestra herramienta con este modelo son, en promedio, más parecidos a las *playlists* originales del *dataset* que el ordenamiento aleatorio. Este es un resultado muy interesante porque muestra indicios de una correlación entre los criterios utilizados a la hora de generar las listas y el criterio utilizado por nuestra herramienta para ordenarlas, según el espacio dimensional del modelo.

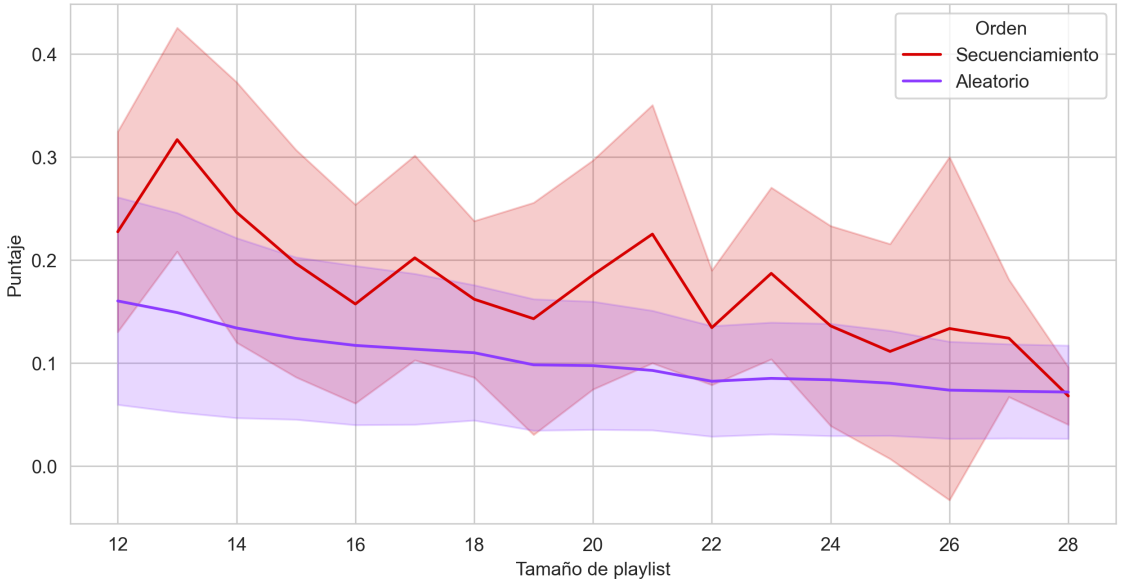


Fig. 3.8: Comparación de *score* (distancia de Levenshtein) entre secuenciamientos generados por la herramienta y permutaciones aleatorias (con la primer canción fija). Modelo $MERT_0$ y heurística de vecino más cercano.

Se plantea entonces el interrogante de qué comportamiento será el que captura $MERT_0$, que es distinto a todos los demás modelos. Para intentar entender mejor lo que sucede, exploramos los resultados de los otros puntajes propuestos. En la figura 3.9 podemos ver que, para *adjusted_weighted_score*, el resultado no es el mismo, sino que es $MFCC$ el que obtiene valores más altos en promedio.

Sin embargo, si miramos el detalle de $MERT_0$ en la figura 3.10, vemos también que está por encima de los secuenciamientos aleatorios, aunque se acercan mucho para algunos tamaños de *playlists*. Esto se explica porque $MERT_0$ es un modelo que se mueve en una escala muy pequeña de distancias, por lo que es de esperar que ambos puntajes estén cerca. Lo más difícil de justificar es la alta variación de estos, ya que no decrecen de manera consistente conforme avanza el tamaño de las listas. Aquí la normalización del puntaje puede influir, ya que, al trabajar con distancias muy pequeñas, la división por dos puede quedar muy grande.

Siguiendo con la comparación con las listas originales del conjunto de datos, analizamos los resultados obtenidos mediante el puntaje *adjusted_comb_score* que surge de las

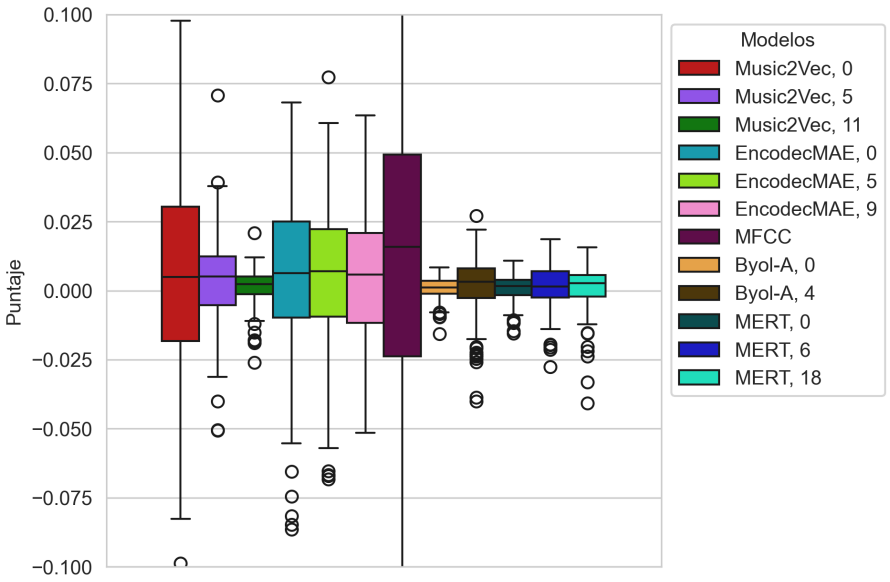


Fig. 3.9: Comparación entre modelos de *adjusted_weighted_score* (distancia de *embeddings*) sobre el subconjunto de listas homogéneas. Heurística de vecino más cercano. Valores más altos indican mejores resultados.

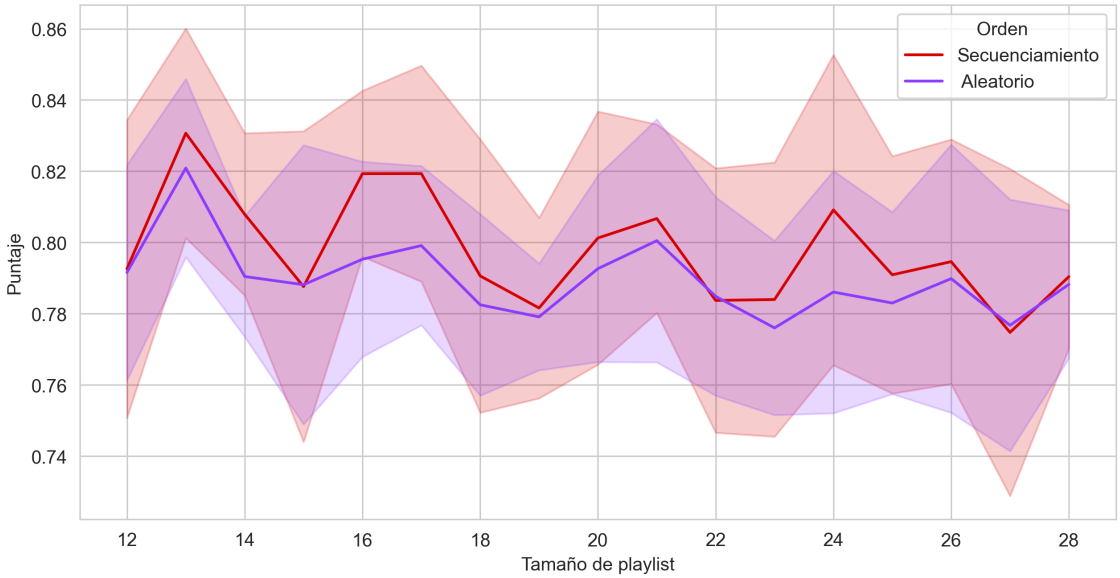


Fig. 3.10: Comparación de *weighted_score* (distancia de *embeddings*) entre secuenciamientos generados por la herramienta y permutaciones aleatorias (con la primer canción fija). Modelo $MERT_0$ y heurística de vecino más cercano.

probabilidades de obtener ciertos secuenciamientos. La intuición nos dice que, al igual que sucede con *adjusted_score*, el modelo $MERT_0$ debería tener ventaja por sobre los demás. Al generar secuenciamientos más parecidos a las listas originales en función de la distancia de Levenshtein, es de esperar que este puntaje lo favorezca. Las probabilidades de asignar canciones en las posiciones originales disminuyen conforme crece el tamaño de la lista.

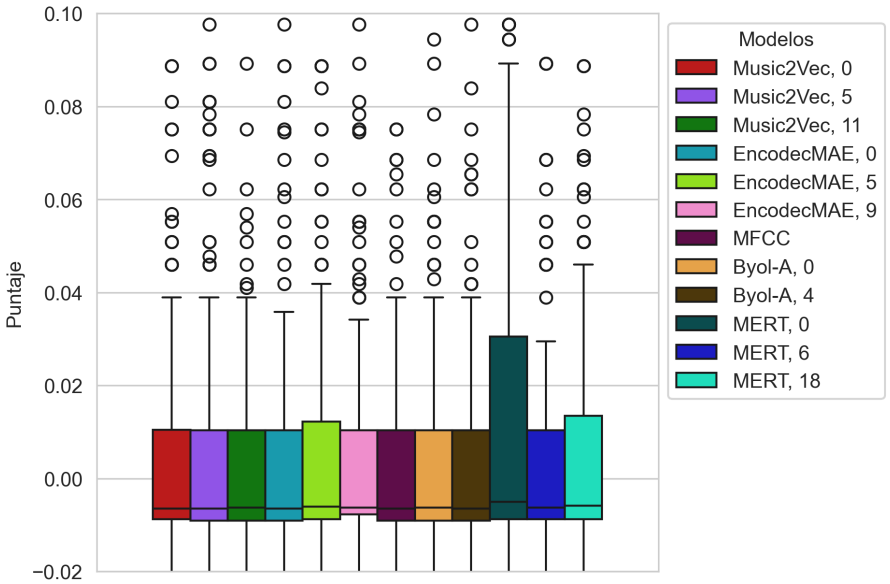


Fig. 3.11: Comparación entre modelos de *adjusted_comb_score* (probabilidad de obtener el secuenciamiento logrado) sobre el subconjunto de listas homogéneas. Heurística de vecino más cercano. Valores más altos indican mejores resultados.

Como se puede ver en la figura 3.11, la mediana de todos los modelos está apenas por debajo de cero. Una conclusión que podemos extraer de este resultado es que, al ser muy alta la cantidad de combinaciones, cuesta más distinguir entre resultados no tan buenos. Esto es lo opuesto al efecto buscado, ya que pretendíamos con este puntaje darle más valor a las canciones que fueron bien colocadas. También se ve que $MERT_0$ alcanza en su percentil 75 un máximo que los otros modelos no, por lo que sigue siendo el modelo que mejor captura similitudes con las listas originales.

Si observamos los puntajes a lo largo de los tamaños de las listas, vemos en la figura 3.12 que para las instancias aleatorias decrece conforme las listas son más largas, con menor velocidad que utilizando el puntaje original. Los mismos saltos se pueden observar aquí para nuestros secuenciamientos (aunque, en esta ocasión son más pronunciados, y para algunos tamaños de listas llega incluso a decrecer hasta la media del puntaje obtenido por las instancias aleatorias).

Para terminar de comparar los modelos, calculamos *adjusted_subarray_score* para medir cuántos *clusters* se generan por cada modelo. Aquí no podemos basarnos en los resultados anteriores para formular una hipótesis, dado que podría suceder que un modelo genere varios de estos subarreglos donde las canciones están en el orden inverso y, por ende, cuentan como ediciones.

Se puede ver en la figura 3.13 que los puntajes obtenidos por los diferentes modelos para este puntaje están mucho más parejos, siendo $Music2Vec_0$, $EnCodecMAE_5$ y $MERT_0$ los que alcanzan máximos en sus percentiles 75. De hecho, se observa que la mediana es igual para todos los modelos.

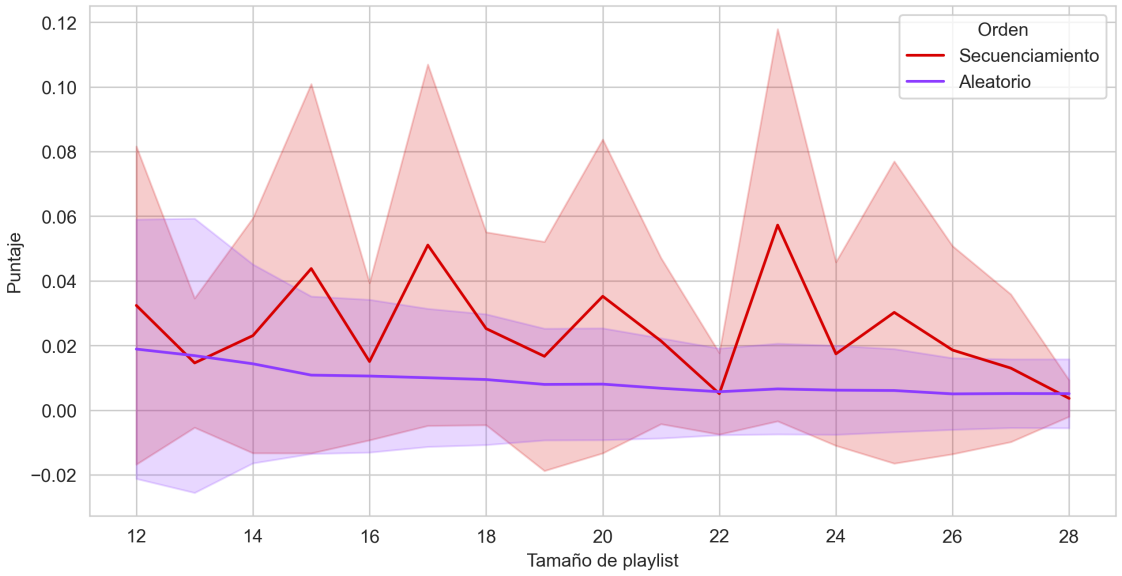


Fig. 3.12: Comparación de *comb_score* (probabilidad de obtener el secuenciamiento logrado) entre secuenciamientos generados por la herramienta y permutaciones aleatorias (con la primer canción fija). Modelo $MERT_0$ y heurística de vecino más cercano.

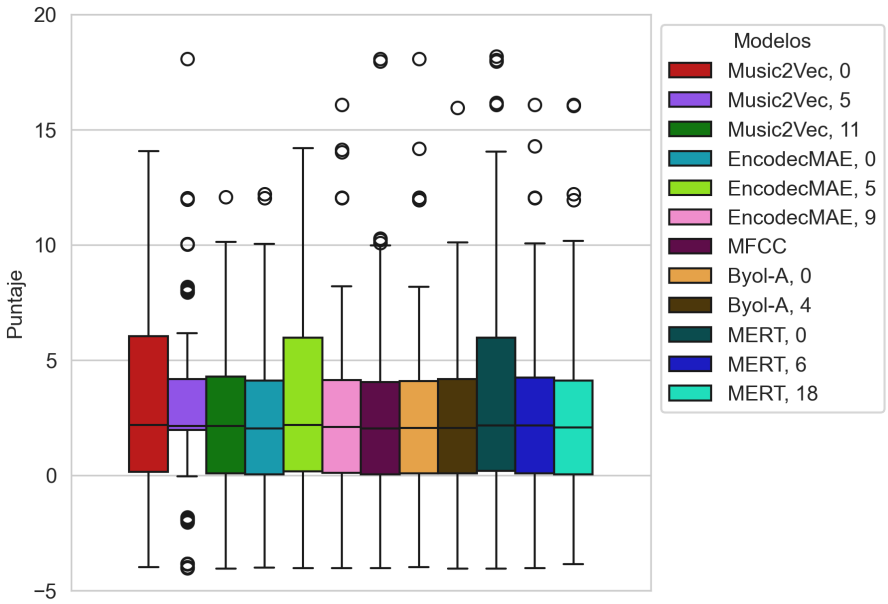


Fig. 3.13: Comparación entre modelos de *adjusted_subarray_score* (*clusters* de canciones que respetan la cercanía en la lista original) sobre el subconjunto de listas homogéneas. Heurística de vecino más cercano. Valores más altos indican mejores resultados.

Si de nuevo miramos en detalle lo que sucede con $MERT_0$ para este puntaje a lo largo de los diferentes tamaños de listas, podemos ver en la figura 3.14 que nuestros secuenciamientos obtienen un puntaje muy superior a las instancias aleatorias, indicando así una tendencia de la herramienta a agrupar canciones que también estaban agrupadas en las

listas originales, aunque no precisamente en el mismo orden, ya que sino veríamos en el gráfico puntajes muchos más altos debido a la naturaleza exponencial de su cálculo.

Estos resultados muestran otra propiedad interesante de las listas originales y de la naturaleza del problema en sí mismo: es más sencillo agrupar las canciones que se parecen más, pero cuando hay que saltar de un grupo al otro, la elección puede no ser evidente. Así mismo, el orden de las canciones dentro de cada grupo, tampoco lo es. Para medir esto, se podrían utilizar algunas *playlists* testigo a fin de observar:

1. Qué tanto varía el peso total del camino alterando el orden de las canciones de un mismo grupo.

2. Cuánta distancia hay entre la última canción de un grupo y la primera de los otros. De esto podría surgir un umbral de seguridad a la hora de dar el salto, y buscar otras estrategias para analizar los grupos restantes en caso de que la elección no sea clara.

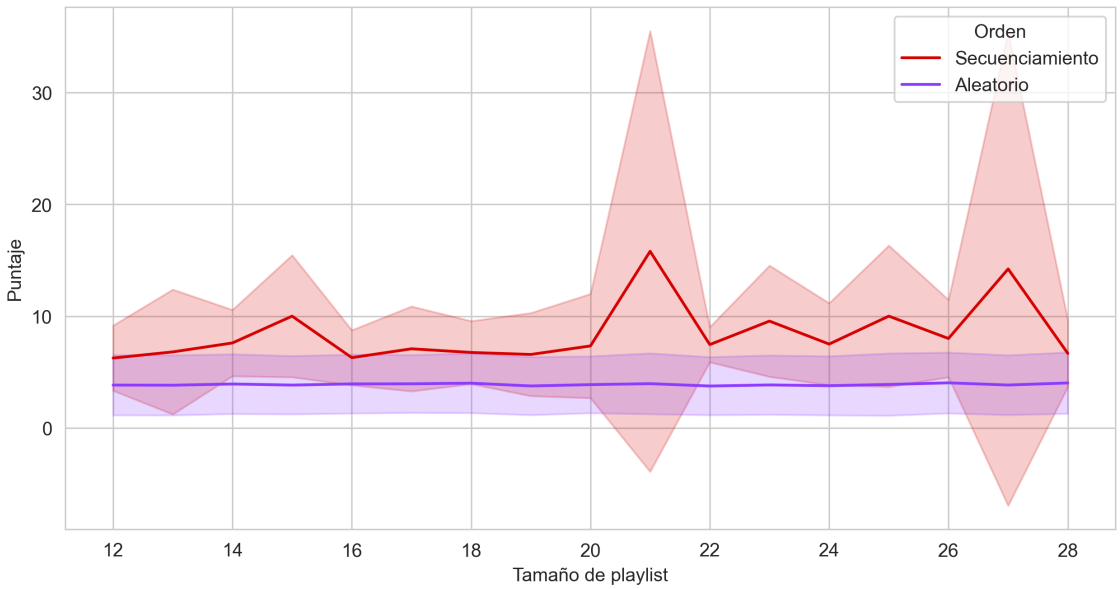


Fig. 3.14: Comparación de *subarray_score* (*clusters* de canciones que respetan la cercanía en la lista original) entre secuenciamientos generados por la herramienta y permutaciones aleatorias (con la primer canción fija). Modelo $MERT_0$ y heurística de vecino más cercano.

Concluimos, entonces, que este último puntaje es más justo para evaluar el rendimiento de la tarea, y el que más potencial tiene para seguir explorando el problema.

3.5. Evaluación de heurísticas

3.5.1. Configuración

Repetimos la configuración del experimento anterior, es decir, los puntajes mencionados en la sección 2.3.6 para las listas homogéneas (subconjunto con valores más altos de $\tilde{W}$). La diferencia radica en que en esta sección observaremos los resultados obtenidos por las diferentes heurísticas. El modelo que se elige para este análisis es $MERT_0$, dado que es el que obtuvo los mejores resultados en el experimento anterior y queremos ver si otra heurística puede mejorarlos aún más. De nuevo, utilizamos mil instancias aleatorias para todos los puntajes, excepto *adjusted_weighted_score*. El camino inicial elegido para la búsqueda tabú es una permutación al azar, dado que no quisimos que los resultados dependieran del comportamiento de otra heurística.

3.5.2. Hipótesis

Mencionamos previamente que esperamos que la heurística de vecinos más cercanos genere los secuenciamientos más parecidos a las listas originales, debido a la naturaleza de la heurística. Sin embargo, no podemos afirmar que esta consiga las listas de menor peso, ya que es una heurística golosa y, por elegir el vecino más cercano en las primeras iteraciones, puede agregar aristas de mucho peso en las iteraciones finales.

Es por esto que esperamos ver un W más bajo de los secuenciamientos generados por las otras heurísticas, principalmente por la búsqueda tabú, que optimiza una solución inicial mediante intercambios de canciones (inclusive puede revisitar movimientos ya realizados luego de un cierto tiempo, para escapar de mínimos locales).

3.5.3. Resultados

Como podemos ver en la figura 3.15, nuestra hipótesis con respecto a los pesos totales de los secuenciamientos generados no resultó verdadera, ya que la heurística del vecino más cercano consigue los secuenciamientos de menor peso total, contrario a la intuición que teníamos.

La heurística de arista más corta obtiene el segundo mejor resultado para este puntaje. En tercer lugar, está la heurística de árbol generador mínimo y finalmente la búsqueda tabú. Esta última, al ser configurable, quizás con más iteraciones o con una mayor cantidad de movimientos a explorar, podría conseguir caminos mejores. Se advierte igual que la diferencia entre todas las heurísticas es muy pequeña.

En cuanto al puntaje de edición, podemos observar en la figura 3.16 que, tal como esperábamos, la heurística del vecino más cercano es la que genera secuenciamientos más parecidos a las listas originales del conjunto de listas con menor $\tilde{W}$. La heurística de árbol generador mínimo es la segunda mejor, superando a la heurística de la arista más corta, pese a que en el cálculo de W sucede lo contrario. Esto se explica porque, como mencionamos con anterioridad, las listas originales no eran necesariamente las de menor peso. La búsqueda tabú de nuevo obtiene el peor puntaje.

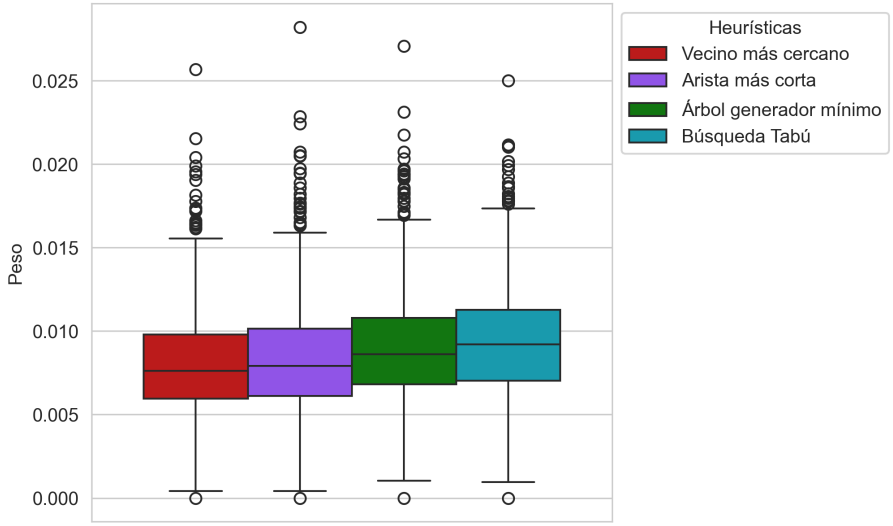


Fig. 3.15: Comparación de valores de W (distancia total del recorrido) entre las heurísticas implementadas. Valores más bajos indican mejores resultados. Modelo $MERT_0$.

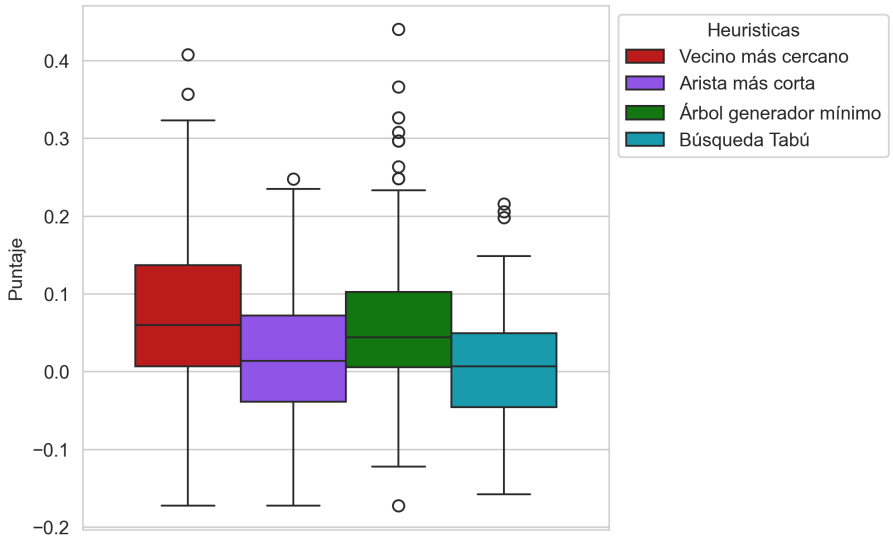


Fig. 3.16: Comparación entre heurísticas de *adjusted_score* (distancia de Levenshtein) sobre el subconjunto de listas homogéneas. Modelo $MERT_0$. Valores más altos indican mejores resultados.

Si observamos los puntajes de edición pesada en la figura 3.17, podemos observar que, curiosamente, se respeta el orden de los puntajes de edición, aunque con la edición pesada varía menos.

Este resultado puede llamar la atención, porque vimos que los W seguían un orden distinto. Sin embargo, como aquí estamos midiendo la distancia entre los *embeddings* posición a posición, tiene sentido que las heurísticas que consiguieron secuenciamientos similares a las listas originales, logren un mejor puntaje.

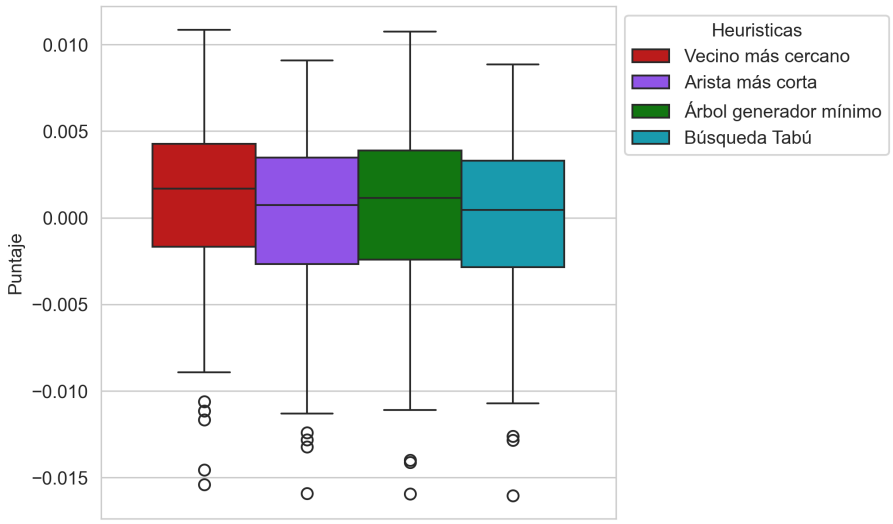


Fig. 3.17: Comparación entre heurísticas de *adjusted_weighted_score* (distancia de *embeddings*) sobre el subconjunto de listas homogéneas. Modelo $MERT_0$. Valores más altos indican mejores resultados.

Sin embargo, este comportamiento no se observó cuando analizamos los modelos, ya que el orden de los modelos que obtuvieron un mejor puntaje de edición no necesariamente obtuvieron los mejores puntajes de edición pesada. Esto implica que, el hecho de que los secuenciamientos sean más parecidos en cuanto a cantidad de ediciones, no alcanza para justificar que la distancia de los *embeddings* también se parezca. En los casos donde la canción requiere dos ediciones para colocarla en su posición original, no se distingue si la que estaba en su lugar era similar acústicamente o no.

Luego tenemos el puntaje combinatorio. Según lo observado durante el análisis de los modelos, existe una correlación con la distancia de edición normal, por lo que podemos esperar ver el mismo patrón para las heurísticas. Efectivamente, si vemos los resultados obtenidos en la figura 3.18, se observa que el orden de las heurísticas se repite, aunque la diferencia entre arista más corta y árbol generador mínimo aquí es mucho menor. En cambio, la búsqueda tabú aquí se ve todavía más distanciada de las otras, teniendo un puntaje muy cercano al cero para la gran mayoría de instancias. Esto podría explicarse porque cada movimiento 2-opt invierte el orden de los vértices intermedios.

Finalmente, tenemos el puntaje basado en subarreglos. Cuando analizamos los resultados de los modelos, vimos que todos obtienen resultados bastante parecidos utilizando este puntaje, a diferencia de los demás. Podemos esperar entonces que suceda lo mismo con las heurísticas.

Si observamos los resultados en la figura 3.19, podemos ver que la heurística de vecino más cercano y la heurística de árbol generador mínimo obtienen resultados muy similares, y apenas por debajo están las otras heurísticas. Es llamativo cómo en este análisis sucede lo mismo que pasó cuando observamos el detalle de los modelos para este puntaje, y es que

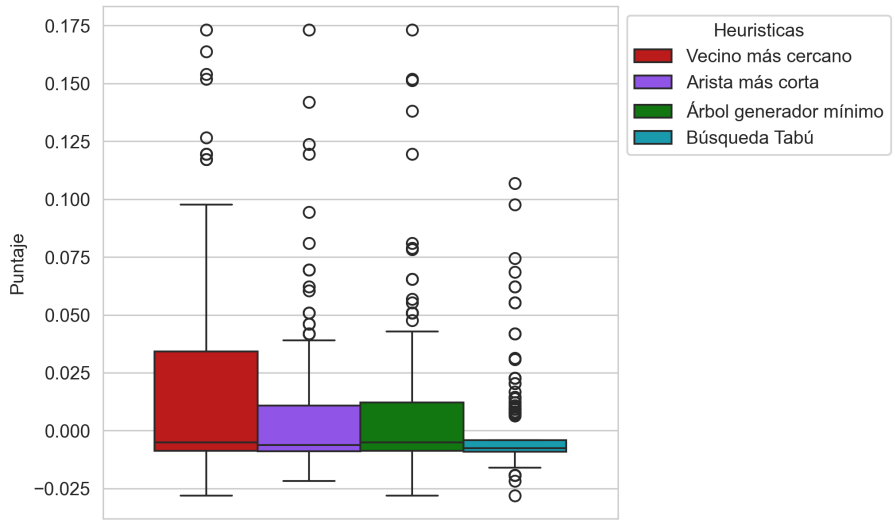


Fig. 3.18: Comparación entre heurísticas de *adjusted_comb_score* (probabilidad de obtener el secuenciamiento logrado) sobre el subconjunto de listas homogéneas. Modelo $MERT_0$. Valores más altos indican mejores resultados.

las medianas se encuentran alineadas entre sí. La intuición para este fenómeno es que al definir el puntaje como la sumatoria de $2^{|subarreglo|}$ no hay mucha variedad en los valores posibles. También podría estar pasando que hay secuencias que siempre se encuentran juntas, sin importar la combinación de hiperparámetros. Queda como trabajo futuro continuar con este análisis.

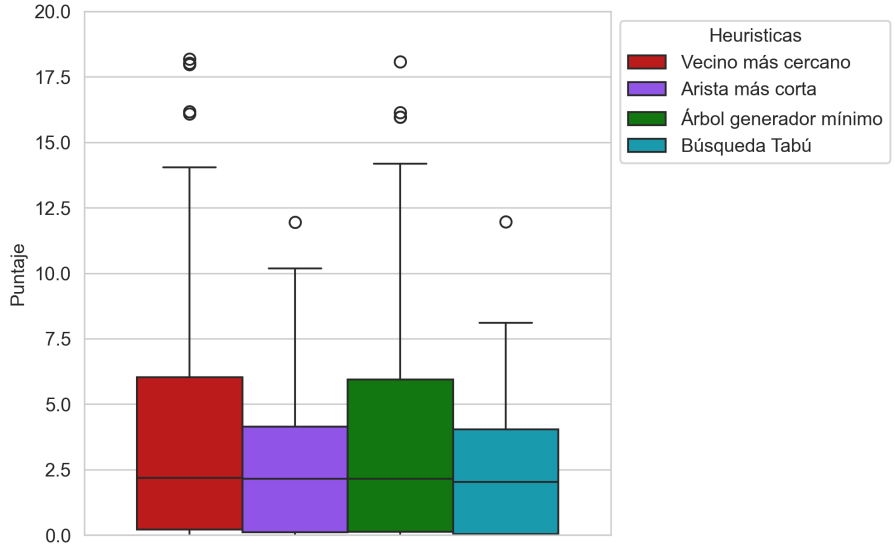


Fig. 3.19: Comparación entre heurísticas de *adjusted_subarray_score* (*clusters* de canciones que respetan la cercanía en la lista original) sobre el subconjunto de listas homogéneas. Modelo $MERT_0$. Valores más altos indican mejores resultados.

Concluimos entonces que la heurística del vecino más cercano es la más indicada para

resolver el problema, al menos en listas de entre 10 y 30 canciones. Quizás con *playlists* de más de 100 canciones y modificando los parámetros, se podría ver una mejora de la búsqueda tabú con respecto a las demás heurísticas.

4. CONCLUSIONES

En este trabajo, implementamos una herramienta capaz de tomar como entrada un conjunto de canciones y producir como salida un ordenamiento de estas para que la sesión de escucha sea homogénea.

Se evaluaron diferentes y novedosos modelos para extraer atributos de las señales de audio, así como las heurísticas más conocidas para resolver el problema del viajante de comercio. Se realizó un estudio exhaustivo de los resultados, y se propusieron puntajes que sirven para evaluar el éxito en la tarea de ordenar una *playlist* si utilizamos como criterio la homogeneidad de la misma. Se plantearon diferentes hipótesis basadas en la intuición, y se propusieron diversos experimentos con los puntajes previamente mencionadas para aceptarlas o rechazarlas.

Concluimos que utilizando la primera capa del modelo MERT en conjunto con la heurística del vecino más cercano se obtienen secuenciamientos similares a las listas originales, cuando estas son homogéneas. También concluimos que Las listas originales no minimizan la suma de distancias entre *embeddings* para los modelos implementados. Finalmente, concluimos que al evaluar los resultados para esta tarea mediante la observación de las agrupaciones de canciones que se forman, logramos mejor separación de las permutaciones aleatorias.

4.0.1. Trabajo futuro

En relación a las conclusiones extraídas, se podría hacer una investigación más profunda de los datos: entender cómo se generan estas agrupaciones por género en las listas originales y qué relación existe con el peso total de estas. También se podrían explorar otros metadatos como la tonalidad, el tempo, etc., para comprender si hay alguna correlación que no se observó durante nuestro análisis.

Asimismo, se podría representar la distancia entre géneros y usarla en conjunto con la distancia de *embeddings*. De esta manera, es probable que la herramienta encuentre resultados todavía más similares a las listas originales.

También se podría indagar más sobre el proceso de entrenamiento del modelo MERT para entender por qué la primera capa logra resultados superiores. Es decir, profundizar para entender qué patrón está captando mejor que los demás modelos.

Por último, una vez comprendida la naturaleza de las listas del *dataset*, se podría entrenar un modelo para resolver específicamente el problema de secuenciar *playlists* de manera homogénea.

BIBLIOGRAFíA

[Bit+17] R. Bittner et al. «Automatic Playlist Sequencing and Transitions». En: *Proceedings of the 18th ISMIR Conference, Suzhou, China* (2017), págs. 442-448. DOI: https://doi.org/10.5281/zenodo.1417028.

[BJ14] Geoffray Bonnin y Dietmar Jannach. «Automated Generation of Music Playlists: Survey and Experiments». En: *ACM Computing Surveys* 47 (ene. de 2014), págs. 1-35. DOI: 10.1145/2652481.

[GKP94] Robert M. Graham, Donald E. Knuth y Oren Patashnik. *Concrete mathematics*. Addison-Wesley, 1994.

[Ham50] R. W. Hamming. «Error detecting and error correcting codes». En: *Bell System Technical Journal* 29.2 (1950), págs. 147-160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[Hel00] Keld Helsgaun. «An effective implementation of the Lin–Kernighan traveling salesman heuristic». En: *European Journal of Operational Research* 126.1 (2000), págs. 106-130. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(99)00284-2. URL: https://www.sciencedirect.com/science/article/pii/S0377221799002842.

[HKP12] Jiawei Han, Micheline Kamber y Jian Pei. «2 - Getting to Know Your Data». En: *Data Mining (Third Edition)*. Ed. por Jiawei Han, Micheline Kamber y Jian Pei. Third Edition. The Morgan Kaufmann Series in Data Management Systems. Boston: Morgan Kaufmann, 2012, págs. 39-82. ISBN: 978-0-12-381479-1. DOI: https://doi.org/10.1016/B978-0-12-381479-1.00002-2. URL: https://www.sciencedirect.com/science/article/pii/B9780123814791000022.

[Ire+19] Rosilde Tatiana Irene et al. «Automatic playlist generation using Convolutional Neural Networks and Recurrent Neural Networks». En: *2019 27th European Signal Processing Conference (EUSIPCO)*. 2019, págs. 1-5. DOI: 10.23919/EUSIPCO.2019.8903002.

[Lev65] V. I. Levenshtein. «Binary codes capable of correcting deletions, insertions, and reversals». En: *Dokl. Akad. Nauk SSSR* 163.4 (1965), págs. 845-848.

[Li+22] Yizhi Li et al. *MAP-Music2Vec: A Simple and Effective Baseline for Self-Supervised Music Audio Representation Learning*. 2022. arXiv: 2212.02508 [cs.SD]. URL: https://arxiv.org/abs/2212.02508.

[Li+23] Yizhi Li et al. *MERT: Acoustic Music Understanding Model with Large-Scale Self-supervised Training*. 2023. arXiv: 2306.00107 [cs.SD].

[Log02] Beth Logan. «Content-Based Playlist Generation: Exploratory Experiments.» En: *ISMIR*. Vol. 2. 2002, págs. 295-296.

[ML12] B. McFee y G. R. G. Lanckriet. «Hypergraph models of playlist dialects». En: *13th International Symposium for Music Information Retrieval (ISMIR2012)*. 2012.

[Nii+21] Daisuke Niizumi et al. *BYOL for Audio: Self-Supervised Learning for General-Purpose Audio Representation*. 2021. arXiv: 2103.06695 [eess.AS]. URL: https://arxiv.org/abs/2103.06695.

[PPW05] T. Pohle, E. Pampalk y G. Widmer. «GENERATING SIMILARITY-BASED PLAYLISTS USING TRAVELING SALESMAN ALGORITHMS.» En: *Proc. Of the 8th Int. Conference on Digital Audio Effects (DAFx'05)*. (2005).

[PRF24] Leonardo Pepino, Pablo Riera y Luciana Ferrer. *EnCodecMAE: Leveraging neural codecs for universal audio representation learning*. 2024. arXiv: 2309.07391 [cs.SD]. URL: https://arxiv.org/abs/2309.07391.

[TA24] Yan-Martin Tamm y Anna Aljanaki. «Comparative Analysis of Pretrained Audio Representations in Music Recommender Systems». En: *arXiv preprint arXiv:2409.08987* (2024).