

UNIVERSITÀ DI MODENA E REGGIO EMILIA  
*Dipartimento di Scienze Fisiche, Informatiche e Matematiche*  
*Corso di Laurea in Informatica*

Tesi di Laurea  
IMPLEMENTAZIONE E ANALISI DI UN METODO DI  
GENERAZIONE DI PLAYLIST PERSONALIZZATE

**Relatore:**  
**Prof. Manuela Montangero**

**Laureando:**  
**Franceschini**  
**Alessandro**

**Matricola n.**  
**165337**

*Anno Accademico 2023/2024*

# Indice

<b>Introduzione</b>	<b>4</b>
<b>1 Stato dell'arte</b>	<b>5</b>
1.1 Lavori Correlati . . . . .	5
1.1.1 Clustering . . . . .	5
1.1.2 Panoramica sui sistemi di raccomandazione musicale . . . . .	6
1.1.3 Deep Learning nei sistemi di raccomandazione musicale . . . . .	9
1.1.4 Valutazione . . . . .	11
1.1.5 Principali sfide . . . . .	11
1.1.6 Raccomandazioni musicali sequence-aware . . . . .	12
1.1.7 Altri lavori . . . . .	16
1.1.8 Conclusioni . . . . .	18
<b>2 La nuova proposta</b>	<b>19</b>
2.1 Descrizione generale . . . . .	19
2.2 Fase 1 - Elaborazione della cronologia di ascolto . . . . .	21
2.3 Fase 1 - Calcolo delle abitudini di ascolto . . . . .	22
2.4 Fase 1 - Clustering . . . . .	22
2.5 Fase 1 - Generazione dei song-set . . . . .	25
2.6 Fase 2 - Ordinamento del song-set . . . . .	28
<b>3 Implementazione</b>	<b>33</b>
3.1 Introduzione . . . . .	33
3.2 Interazione con Spotify . . . . .	34
3.2.1 Memorizzazione delle credenziali . . . . .	34
3.2.2 Scelta delle credenziali . . . . .	36
3.2.3 Recupero delle raccomandazioni . . . . .	37
3.2.4 Recupero delle caratteristiche audio dei brani . . . . .	38
3.3 Recupero degli ascolti recenti . . . . .	43
3.4 Calcolo dei periodi . . . . .	51
3.5 Calcolo della cronologia di ascolto . . . . .	53
3.6 Calcolo delle abitudini di ascolto . . . . .	55
3.6.1 Calcolo di dNTNA e dNTKA per ciascun periodo . . . . .	56
3.6.2 Calcolo di NTNA e NTKA . . . . .	58
3.7 Clustering . . . . .	59
3.7.1 Algoritmo K-means . . . . .	59
3.7.2 Algoritmo Furthers-Point-First (FPF) . . . . .	61
3.7.3 Calcolo dei clustering . . . . .	65
3.8 Generazione dei song-set . . . . .	67

3.8.1	Euristica lineare . . . . .	68
3.8.2	Euristica sferica . . . . .	69
3.8.3	Applicazione delle euristiche . . . . .	74
3.8.4	Calcolo dei migliori song-set . . . . .	76
3.9	Calcolo dei pattern di cronologia . . . . .	81
3.10	Scelta dei migliori pattern di cronologia . . . . .	86
3.10.1	Scelta pseudocasuale del pattern . . . . .	90
3.10.2	Sovrapposizione dei pattern . . . . .	91
3.10.3	Scelta del pattern più simile . . . . .	92
3.11	Ordinamento dei song-set . . . . .	93
3.12	Recupero delle canzoni ordinate . . . . .	98
3.13	Salvataggio delle playlist . . . . .	100
3.14	Metodi alternativi . . . . .	103
3.14.1	REC-1 . . . . .	103
3.14.2	REC-2 . . . . .	105
3.14.3	HYB-1 . . . . .	107
3.15	Calcolo della Playlist Pattern Distance . . . . .	111
<b>4</b>	<b>Sperimentazione</b>	<b>118</b>
4.1	Descrizione del dataset . . . . .	119
4.1.1	Raccolta dei dati . . . . .	119
4.1.2	Analisi della cronologia . . . . .	120
4.2	Metodi alternativi . . . . .	128
4.3	Generazione delle playlist . . . . .	128
4.3.1	Preparazione dei dati . . . . .	129
4.3.2	Generazione della playlist tramite SMART . . . . .	129
4.3.3	Generazione delle playlist tramite i metodi alternativi . . . . .	129
4.4	Calcolo della Playlist Pattern Distance . . . . .	130
4.5	Analisi dei risultati . . . . .	130
4.5.1	Confronto basato sulla Playlist Pattern Distance . . . . .	130
4.5.2	Confronto basato sulla lunghezza della cronologia . . . . .	140
4.5.3	Confronto basato sulla lunghezza della playlist . . . . .	147
4.5.4	Altre differenze tra le playlist . . . . .	147
4.6	Altri grafici . . . . .	151
4.6.1	Analisi della cronologia di ascolto . . . . .	151
4.6.2	Playlist Pattern Distance . . . . .	163
4.7	Conclusioni . . . . .	173
4.7.1	Playlist Pattern Distance . . . . .	173
4.7.2	Lunghezza della cronologia . . . . .	173
4.7.3	Lunghezza delle playlist . . . . .	174
4.7.4	Altre metriche di confronto . . . . .	174

<b>5 Conclusioni</b>	<b>175</b>
5.1 Riepilogo delle motivazioni e degli scopi della tesi . . . . .	175
5.2 Risultati ottenuti . . . . .	175
5.3 Sviluppi futuri . . . . .	175
<b>Bibliografia</b>	<b>177</b>
<b>A Appendice</b>	<b>183</b>

# Introduzione

Le playlist musicali hanno acquisito sempre più popolarità nel tempo, diventando oggi il principale modo di ascoltare musica per molti utenti. Di conseguenza, esse sono diventate un elemento essenziale nei servizi di streaming musicale come Spotify. Inoltre, le playlist possono incrementare la popolarità degli artisti, rappresentando per loro una fonte significativa di guadagno. Senza le playlist, gli utenti rischiano di ascoltare sempre le stesse canzoni già conosciute, vanificando il potenziale di un ampio catalogo musicale e minacciando gli interessi economici dei fornitori di servizi di streaming.

Per questi motivi, la generazione automatica di playlist musicali è un argomento di crescente interesse sia nell'ambito accademico che commerciale. Le piattaforme di streaming musicale, come Spotify, offrono vasti cataloghi di brani che gli utenti possono esplorare e ascoltare. Tuttavia, la vasta quantità di scelta può rendere difficile per gli utenti scoprire nuova musica adatta ai loro gusti e preferenze personali.

Il contributo di questa tesi consiste nell'implementazione e nella sperimentazione di un metodo innovativo per la generazione di playlist personalizzate. Il metodo, denominato SMART (Sensitive Music Arrangement and Recommendation Technique), viene descritto formalmente nella Sezione 1, e si basa su algoritmi di clustering e su un'analisi approfondita delle abitudini di ascolto degli utenti. Attraverso l'uso di dati storici di ascolto e caratteristiche audio dei brani, l'obiettivo è quello di creare playlist che non solo rispecchino i gusti musicali dell'utente, ma che introducano anche nuovi brani potenzialmente apprezzati, fornendo un'esperienza di ascolto completa e un grado di personalizzazione superiore rispetto ai sistemi di raccomandazione esistenti.

I principali servizi di streaming musicale, tra cui Spotify, offrono agli utenti una lista di playlist personalizzate. Tuttavia, gli utenti devono comunque ispezionare manualmente le playlist una per una per trovare quella più adatta al momento della giornata e all'attività che stanno svolgendo.

Per questo motivo, SMART mira a eliminare il compito, spesso noioso, di scegliere quale musica ascoltare, offrendo agli utenti un'esperienza di ascolto soddisfacente e senza impegno, che si adatti il più possibile all'attività svolta e al momento dell'ascolto.

In sintesi, questa tesi si propone di contribuire al miglioramento delle tecniche di generazione automatica di playlist, offrendo una soluzione che combina tecniche di apprendimento automatico (clustering) con una profonda analisi delle preferenze musicali degli utenti.

I contenuti rimanenti di questa tesi sono strutturati come segue: nel Capitolo 1 verrà presentato lo stato dell'arte, che include i principali contributi accademici relativi alla generazione di playlist musicali. Il Capitolo 2 descrive formalmente SMART. L'implementazione di SMART verrà discussa nel Capitolo 3. Successivamente, nel Capitolo 4 mostreremo gli esperimenti che sono stati condotti per valutare l'efficacia del metodo proposto. Infine, i risultati ottenuti verranno analizzati e discussi nel Capitolo 5, mettendo in evidenza i punti di forza e le possibili aree di miglioramento.

# 1 Stato dell'arte

In questo capitolo ci occuperemo, nella maniera più accurata possibile, di fornire ai lettori una panoramica sul contesto teorico preesistente, analizzando le ricerche e le teorie più rilevanti nel campo, al fine di stabilire un quadro completo delle conoscenze attuali e delle sfide ancora da affrontare.

## 1.1 Lavori Correlati

L'idea tradizionale di playlist come semplici sequenze di brani musicali prodotte manualmente è ormai superata. Negli ultimi anni, l'attenzione si è spostata verso la generazione automatica di playlist, che mira a fornire agli utenti esperienze di ascolto personalizzate, tramite numerosi studi e ricerche focalizzati su questo tema.

In questa sezione, inizieremo con una panoramica sui sistemi di raccomandazione musicale. Successivamente, discuteremo i contributi più rilevanti presenti in letteratura sulla generazione di playlist, con particolare attenzione alle raccomandazioni sensibili all'ordinamento delle canzoni (sequence-aware).

### 1.1.1 Clustering

Il clustering è essenzialmente un tipo di metodo di apprendimento non supervisionato. Un metodo di apprendimento non supervisionato è un metodo nel quale traiamo riferimenti da insiemi di dati costituiti da dati di input senza risposte etichettate. Generalmente, viene utilizzato come processo per trovare una struttura significativa, processi sottostanti esplicativi, caratteristiche generative e raggruppamenti intrinseci in un insieme di esempi. Il clustering è il compito di dividere la popolazione o i punti dati in un certo numero di gruppi, facendo in modo che i punti presenti in un determinato gruppo risultino tra di loro simili e che i punti appartenenti a gruppi diversi risultino tra di loro dissimili. È essenzialmente una raccolta di oggetti sulla base di somiglianza e dissomiglianza tra di essi. L'utilizzo di algoritmi di clustering risulta particolarmente utile in tutte quelle situazioni in cui è necessario determinare il raggruppamento intrinseco tra dati non etichettati.

K-means è probabilmente l'algoritmo di clustering più conosciuto ed utilizzato in letteratura. Il funzionamento è molto semplice: inizialmente, si selezionano casualmente  $k$  punti come centroidi iniziali dei cluster. Ogni punto del dataset da clusterizzare viene assegnato al gruppo del centroide più vicino. Successivamente, i centroidi di ciascun cluster vengono ricalcolati e viene ripetuta l'assegnazione dei punti del dataset. Questi due passaggi vengono ripetuti fino a quando il numero desiderato di cluster non è raggiunto.

Further-Point-First (FPF) è un algoritmo di clustering meno noto, ma che ha dimostrato di essere molto efficace in contesti multimediali [13, 15]. FPF si basa su una strategia di clustering iterativa piuttosto semplice: all'inizio, tutti i punti del dataset

sono inclusi in un unico cluster e la selezione del centroide avviene casualmente tra di essi. Successivamente, un nuovo cluster viene aggiunto in due passaggi:

1. Il centroide del nuovo cluster è il punto più distante dal suo centroide.
2. Tutti i punti che sono più vicini al nuovo centroide rispetto a quello precedente vengono spostati nel nuovo cluster.

Questi due passaggi vengono ripetuti fino al raggiungimento del numero desiderato di cluster. Una volta che il clustering tramite FPF è giunto al termine, calcoliamo i centroidi di ciascun cluster. *to be used in the next step.* In effetti, i centroidi potrebbero non essere punti del cluster ma avere una distanza media da tutti i punti del cluster, mentre i centri potrebbero essere punti che si trovano all'esterno del cluster.

### 1.1.2 Panoramica sui sistemi di raccomandazione musicale

Per generare playlist personalizzate è necessario un Music Recommendation System (MRS), ossia un sistema di raccomandazioni specifico per l'ambito musicale. Tutte le tecniche di raccomandazione musicale esistenti si basano sul prevedere il comportamento degli utenti, sfruttando le loro interazioni con i servizi e/o le informazioni da loro fornite in merito alle loro preferenze musicali. Esistono diversi tipi di raccomandazioni musicali[9]:

- Raccomandazioni non personalizzate e non contestualizzate: in questo caso, ad un determinato input corrisponde sempre lo stesso output, indipendentemente dall'utente. Le raccomandazioni di questo tipo sono basate soltanto sulla popolarità di una canzone o di un artista.
- Raccomandazioni contestualizzate ma non personalizzate: Anche in questo caso, un dato input produce lo stesso output per tutti gli utenti. Esempi di raccomandazioni di questo tipo sono stazioni radio basate su una singola traccia o su un singolo artista.
- Raccomandazioni personalizzate ma non contestualizzate: viene raccomandato un insieme di tracce ottenute analizzando il comportamento dell'utente sulla piattaforma musicale.
- Raccomandazioni personalizzate e contestualizzate: la letteratura si concentra su approcci di questo tipo. In questo modo è possibile, ad esempio, generare una playlist basandosi su informazioni come lo stato d'animo dell'utente in un determinato momento e le sue preferenze passate.

Un MRS può essere implementato in diversi modi [14]:

**Content-Based Filtering (CBF)** Utilizza tecniche di apprendimento automatico per generare raccomandazioni confrontando il contenuto delle canzoni con il profilo dell’utente. Gli elementi vengono rappresentati mediante insiemi di caratteristiche, assegnabili manualmente o automaticamente, per rendere il confronto significativo. Questo approccio dipende fortemente dalla correttezza con cui gli elementi sono rappresentati, è soggetto all’effetto ”glass-ceiling” e potrebbe considerare erroneamente come simili elementi notevolmente diversi tra loro.

**Collaborative Filtering (CF)** Procedura mediante la quale viene effettuata una previsione automatica sugli interessi di un utente raccogliendo informazioni sui gusti o sulle preferenze di molti utenti. Se un utente A e un utente B hanno valutato positivamente lo stesso oggetto (canzone), è probabile che se l’utente A ascolta una nuova canzone e la valuta positivamente, anche l’utente B potrebbe apprezzarla. Questo semplice approccio, nella pratica, presenta alcune limitazioni:

- Cold Start Problem: risulta difficile generare raccomandazioni per nuovi utenti o nuovi oggetti, poiché non ci sono dati sufficienti.
- Scalabilità: con un grande numero di utenti e oggetti, il calcolo delle similarità può diventare computazionalmente costoso.
- sparsità dei Dati: Spesso le matrici di valutazione sono sparse, cioè la maggior parte delle voci è vuota, il che può rendere difficile trovare somiglianze significative.

**Metadata-Based Filtering** Rappresenta un approccio più semplice e tradizionale, raccomandando canzoni agli utenti basandosi sui metadati come il genere, il nome dell’artista e il nome dell’album. Questo sistema non tiene in considerazione informazioni sugli utenti per generare raccomandazioni, limitando così la personalizzazione.

**Emotion-Based Filtering** Determina le emozioni suscite da una canzone basandosi sulle sue caratteristiche acustiche, con l’obiettivo di raccomandare brani capaci di evocare emozioni simili a quelle che l’utente potrebbe provare in un dato momento. Questo sistema richiede un grande campione di dati e un notevole lavoro umano, inoltre, le canzoni potrebbero suscitare emozioni diverse in persone diverse, portando a delle ambiguità nei dataset.

**Context-Based Model** Utilizza informazioni dai social media per comprendere la percezione di una canzone da parte del pubblico, generando raccomandazioni musicali pertinenti in base a questo contesto. Questo modello è efficiente anche su piccoli dataset, poiché utilizza la cronologia di ascolto per raccogliere informazioni sull’utente e raccomanda canzoni simili in base all’interesse che queste suscitano sui principali

social media. Alternativamente, il contesto può essere ricavato analizzando la località dell’utente, supponendo che utenti vicini ascoltino musica simile.

**Neural Collaborative Filtering (NCF)** Utilizza le reti neurali profonde (Deep Neural Network) per svelare relazioni più complesse tra utenti e oggetti. Questi modelli affrontano il problema di linearità riscontrabile nelle recenti varianti di CF (come la Matrix Factorization, MF).

**Graph-Based Models** Modelli che aggregano le interazioni passate tra utenti e oggetti come un grafo bipartito per esaminare le relazioni sottostanti. Gli approcci di raccomandazione musicale basati su grafi utilizzano approcci basati su random walk, auto-encoder grafici e GNN (Graph Neural Network).

**Sistemi ibridi** A livello commerciale, la maggior parte dei sistemi di raccomandazione musicale (MRS) si basa su modelli “content-driven”[27], combinando tecniche di filtraggio collaborativo (CF) e approcci content-based (CB) per consigliare musica agli utenti. Il problema principale dei MRS basati interamente su CF è che il profilo utente è indipendente dagli attributi descrittivi degli oggetti che gli utenti hanno apprezzato in passato. Di conseguenza, modelli di questo tipo generano raccomandazioni trascurando grandi quantitativi di conoscenza in-domain. Per questo motivo, la quasi totalità dei MRS è basata su un approccio ibrido, in cui le informazioni catturate dal filtraggio collaborativo vengono unite a conoscenza in-domain, generando raccomandazioni più rappresentative del modo con cui gli umani percepiscono la musica. Inoltre, questi modelli ibridi permettono di generare raccomandazioni anche in assenza di sufficienti interazioni con gli utenti (problema di cold start).

I contenuti in-domain, sfruttati dai MRS per migliorare le raccomandazioni basate su CF, seguono un modello a “cipolla” (Onion-Model). L’Onion-Model è costituito da cinque diversi strati di categorie di contenuto. Gli strati più interni descrivono i contenuti in modo strettamente oggettivo, mentre quelli più esterni, che vengono aggiunti a mano a mano che si raccolgono informazioni, descrivono contenuti in modo più soggettivo. Gli strati, elencati dal più interno al più esterno, sono i seguenti: audio (segnale grezzo), EMD (Embedded Metadata, come titolo, artista e album), EGC (Expert-Generated Content), UGC (User-Generated Content) e DC (Derivative Content). L’Onion-Model viene sfruttato, in modi diversi, in tutti i tipi di MRS.

### 1.1.3 Deep Learning nei sistemi di raccomandazione musicale

**Introduzione** Il deep learning è una branca dell’intelligenza artificiale che ha rivoluzionato molti campi dell’informatica, inclusi il riconoscimento delle immagini, il riconoscimento del linguaggio naturale, il controllo autonomo e molto altro. Questa tecnologia ha reso possibili avanzamenti significativi in settori come la medicina, l’automazione industriale e la ricerca scientifica. Il deep learning si basa su reti neurali artificiali profonde, ovvero reti neurali composte da molti strati di unità di calcolo chiamate neuroni. Questi strati permettono al sistema di apprendere rappresentazioni sempre più complesse dei dati, consentendo di riconoscere modelli e caratteristiche che sarebbero difficili da individuare con altri approcci più tradizionali.

**Motivazione** La generazione di raccomandazioni musicali presenta diverse peculiarità rispetto ad altri contesti di applicazione dei sistemi di raccomandazione:

- Durata: una canzone è molto più breve di un film o una vacanza.
- Numero di item: esistono decine di milioni di canzoni da cui scegliere. La disponibilità di item in altri contesti è molto più limitata.
- Estrazione delle caratteristiche: grazie ai progressi scientifici avvenuti nella MIC (Music Information Retrieval) e nella elaborazione dei segnali audio, l’estrazione di caratteristiche nel contesto musicale gioca un ruolo molto più importante rispetto agli altri domini di applicazione.
- Raccomandazioni ripetute: è probabile che un utente ascolti più volte la stessa canzone, ma è improbabile che guardi più volte lo stesso film.
- Passività: l’ascolto di musica è spesso abbinato allo svolgimento di altre attività da parte dell’utente, al contrario di ciò che accade in altri contesti.

L’abbondanza di canzoni abbinata alla breve durata delle stesse implica che produrre raccomandazioni imperfette non influisca in maniera eccessivamente negativa sulla esperienza dell’utente, al contrario di ciò che potrebbe succedere in altri contesti (come una

vacanza). Inoltre, le architetture DL trattano gli oggetti in maniera probabilistica, e il loro output è costituito da un vettore contenente le probabilità di “fit” degli oggetti. Per questo motivo, l’utilizzo di DL permette di ottenere raccomandazioni ripetute, in contrasto con quello che succede negli altri sistemi di raccomandazione. Infine, l’ascolto di musica è solitamente un’attività passiva. Per questo motivo, se un utente completa l’ascolto di una canzone non è detto che questa gli sia piaciuta. Di conseguenza, è necessario generare raccomandazioni prendendo in considerazione dati contestuali, come dati provenienti da activity tracker o da altre app utilizzate durante l’ascolto. Le architetture basate su DL possono occuparsi anche di questo. Questi aspetti rendono particolarmente adatto l’utilizzo di tecniche di deep learning nei sistemi di raccomandazione musicale.

**Implementazione** I sistemi di raccomandazione musicale basati su DL fanno uso di DNN (Deep Neural Network) per generare rappresentazioni di canzoni e artisti a partire dai loro metadati e dal loro contenuto audio. Queste rappresentazioni vengono poi utilizzate nei sistemi CBF (Content Based Filtering), integrate in approcci di Matrix Factorization (MF), o sfruttate per costruire sistemi di raccomandazione ibridi. Le principali tecnologie di DL adottate in letteratura per generare raccomandazioni musicali sono Convolutional Neural Network (CNN), Deep Belief Network (DBN) e Multilayer Perceptron (MLP).

Per quanto riguarda la generazione di sequenze di canzoni, in letteratura sono presenti due approcci: Next Track Recommendation (NTR), per consigliare all’utente una canzone da ascoltare dopo aver terminato quella corrente, e Automatic Playlist Continuation (APC), per produrre sequenze di canzoni di lunghezza arbitraria. In questo contesto, le principali tecnologie utilizzate sono i Recurrent Neural Network (RNN), gli Attentive Neural Networks (ANN) e i Convolutional Neural Network (CNN). Prima dell’adozione estesa dei modelli di deep learning, venivano utilizzati metodi più semplici come k-nearest neighbors (k-NN), regole di associazione e pattern sequenziali.

**Sfide** Le principali sfide nell’utilizzo di tecniche di Deep Learning all’interno dei MRS sono le seguenti:

- Trasparenza: attualmente, l’utilizzo di DL nel contesto musicale non consente di comprendere il motivo per cui una canzone è stata consigliata all’utente.
- Pochi dataset multimodali e molte metriche per valutare i risultati ottenuti: i ricercatori sono costretti a costruirsi manualmente i dataset e, di conseguenza, a creare metriche di valutazione personalizzate, portando a una mancanza di convenzioni.
- Prospettiva centrata sul sistema: gli approcci basati su DL utilizzano grandi quantità di dati per creare modelli che siano in grado di generare musica sensibile alla sequenza di ascolto. Tuttavia, questi modelli non prendono in considerazione lo scopo della sessione di ascolto dell’utente o le sue intenzioni di creare una playlist.

#### 1.1.4 Valutazione

Per valutare un sistema di raccomandazione, è necessario tenere in considerazione i seguenti fattori di qualità [9]:

- Diversità: le raccomandazioni devono essere sufficientemente diverse tra loro.
- Coerenza: tracce adiacenti devono essere omogenee tra loro e coerenti con le altre tracce nella playlist.
- Scopo: le raccomandazioni devono tenere in considerazione il motivo per cui l'utente desidera ascoltare musica. Ad esempio, le raccomandazioni destinate a chi vuole scoprire nuova musica dovrebbero differire da quelle destinate a chi desidera ascoltare brani familiari.
- Contesto: le raccomandazioni dovrebbero tenere in considerazione il contesto di ascolto. Ad esempio, le raccomandazioni musicali per un allenamento dovrebbero differire da quelle per una sessione di lettura.

Per misurare l'efficacia di un servizio di raccomandazione, si possono effettuare diverse analisi. Si può misurare l'adozione del servizio, determinando quale frazione degli utenti lo utilizza in modo significativo nel tempo. Per servizi come una stazione radio personalizzata, si può analizzare il comportamento specifico degli utenti e il loro feedback sulle raccomandazioni, ad esempio, se gli utenti saltano certi brani raccomandati o continuano ad ascoltarli. Tali misurazioni possono essere confrontate in test sul campo (A/B test), accompagnati eventualmente da sondaggi agli utenti.

#### 1.1.5 Principali sfide

Generare raccomandazioni musicali in modo efficace è un compito complesso, e richiede di considerare alcuni fattori:

- Diversità delle raccomandazioni: il desiderio di inclusione di brani meno conosciuti nelle proprie raccomandazioni varia da utente a utente. Sviluppare un MRS senza considerare questo aspetto soggettivo risulterebbe in una personalizzazione parziale, peggiorando potenzialmente l'esperienza degli utenti.
- Spiegare all'utente il motivo per cui gli è stata consigliata una determinata canzone: questo aspetto è particolarmente problematico nei sistemi basati su Deep Learning.
- Integrare informazioni contestuali per generare migliori raccomandazioni: è necessario determinare quali informazioni considerare, come ottenerle e come integrarle nel sistema di raccomandazione.

- Generare raccomandazioni musicali sequenziali: presentare all’utente una lista di canzoni ordinate in maniera opportuna permetterebbe di generare intere playlist personalizzate o estendere playlist esistenti, migliorando sensibilmente l’esperienza dell’utente.
- Migliorare la scalabilità e l’efficienza degli algoritmi.
- Mitigare il problema di cold start: è necessario determinare come raccomandare canzoni agli utenti in caso questi non abbiano interagito a sufficienza con la piattaforma.
- Gestire diverse varianti della stessa canzone: questo aspetto, che dipende fortemente dall’utente target, può avere una forte influenza sulla sua valutazione della raccomandazione.
- Gestire scenari specifici: ad esempio, raccomandazioni musicali destinate a un suonatore di piano devono essere differenti da quelle destinate a un gruppo di persone che viaggia in macchina. In questi casi, è necessario tenere in considerazione dati aggiuntivi che cambiano in base alla situazione.
- Stakeholder multipli: i sistemi di raccomandazione musicali tradizionali sono ottimizzati per beneficiare un solo gruppo di interesse, come gli azionisti della compagnia di streaming, i creatori di contenuti, i fornitori di contenuti o i consumatori di musica. Un approccio Multi-Stakeholder permetterebbe di considerare in modo congiunto le esigenze di tutti i partecipanti al sistema, bilanciando i loro obiettivi.

#### **1.1.6 Raccomandazioni musicali sequence-aware**

Una delle principali sfide da affrontare nello sviluppo di un MRS è la sequenzialità dei brani raccomandati: considerare questo aspetto, che risulta particolarmente importante nel contesto teorico di questa tesi, permetterebbe di presentare all’utente una lista di canzoni ordinate in maniera opportuna, migliorando sensibilmente il suo grado di soddisfazione. In letteratura sono presenti due approcci:

- Next Track Recommendation (NTR): questa tecnica permette di determinare la prossima traccia da ascoltare basandosi su informazioni relative alla traccia corrente. Questa funzionalità permette, ad esempio, di implementare una radio con un quantitativo virtualmente infinito di musica e permette di assistere gli utenti nel processo di creazione di playlist. Le tecniche next-track recommendation sono solitamente basate sul filtraggio collaborativo.
- Automatic Playlist Continuation (APC): tecnica per produrre sequenze di canzoni di lunghezza arbitraria. La letteratura presenta pochi studi in questo ambito.

**Next-Track Recommendation (NTR)** Per quanto riguarda il Next Track Recommendation (NTR), possiamo classificare gli algoritmi esistenti in questo modo[8]:

- Sequence Learning: questi metodi sfruttano la natura sequenziale delle azioni degli utenti. Possiamo distinguere le seguenti sottoclassi:
  - Frequent Pattern Mining (FPM): metodo che consiste nell'estrazione di pattern di canzoni che occorrono frequentemente nelle sessioni utente.
  - Sequence modeling: la modellazione delle sequenze permette di comprendere come le interazioni passate di un utente influenzino quelle future. In questo contesto, gli algoritmi utilizzano modelli di Markov, per prevedere le transizioni tra stati basati sulle azioni recenti degli utenti, oppure Recurrent Neural Networks (RNNs), per memorizzare e prevedere sequenze complesse di azioni utente.
  - Distributed item representations: rappresentazioni dense e a bassa dimensione degli item, costruite a partire da sequenze che possono preservare le relazioni sequenziali tra gli items stessi. Dato il contesto della sessione corrente dell'utente, le raccomandazioni sugli item successivi possono essere generate navigando nello spazio di embedding in modo stocastico oppure cercando i vicini più prossimi agli ultimi item esplorati dall'utente.
  - Supervised models with sliding window: una finestra mobile di dimensione  $W$  viene fatta scorrere su ogni sequenza. A ogni passo, tutti gli item all'interno della finestra vengono utilizzati per costruire le caratteristiche del problema di apprendimento supervisionato, e l'identificatore dell'item immediatamente successivo viene utilizzato come variabile target. Il problema della previsione delle sequenze viene così convertito in un problema di classificazione multiclass o multietichetta, se sono consentiti più item target.
- Matrix completion: si tratta di una tecnica utilizzata nei sistemi di raccomandazione per riempire una matrice incompleta di valutazioni degli utenti su vari articoli. Nonostante questa tecnica non sia ottimale, in quanto non tiene in considerazione le raccomandazioni ripetute, esistono casi specifici in cui può essere efficacemente utilizzata:
  - Purchase Interval Matrix Factorization (PIMF): questo modello fattorizza la matrice utilizzando una funzione di perdita ponderata che massimizza l'utilità prevista per l'utente, pesata sugli intervalli di tempo osservati tra ciascuna coppia di item. Il modello risultante può prevedere la rilevanza personalizzata di un item in base al momento in cui le raccomandazioni vengono generate.
  - Generazione Personalizzata di Storie: le storie vengono rappresentate come grafi di prefissi, in cui ogni nodo rappresenta un prefisso di una storia (sequenza di point plots). In questo contesto, viene utilizzata una "prefix-rating

matrix” da completare, dove gli item sono sostituiti da possibili prefissi di storie e gli utenti forniscono valutazioni solo ad alcuni di questi prefissi. La fattorizzazione della matrice viene utilizzata per prevedere le voci mancanti nella matrice delle valutazioni dei prefissi. La storia completa con il punteggio più alto discendente dalla ”storia attuale” dell’utente viene utilizzata per suggerire il prossimo point plot. Il processo continua iterativamente (fattorizzazione, raccomandazione, valutazione) fino a che la storia non raggiunge una conclusione.

- Metodi ibridi: combinano la flessibilità del Sequence learning con la robustezza delle tecniche di fattorizzazione delle matrici. Possiamo distinguere le seguenti sottoclassi:
  - Factorized Personalized Markov Chains (FPMS): può essere visto come una Catena di Markov di primo ordine la cui matrice di transizione è fattorizzata congiuntamente con una matrice utente-item bidimensionale standard. La fattorizzazione congiunta consente di inferire le transizioni non osservate nella Catena di Markov dalle coppie di transizione di altri utenti. Al momento della raccomandazione, gli item sono classificati in base alla loro probabilità di essere il prossimo item dato l’ultimo item con cui l’utente ha interagito.
  - LDA con sequence learning: il Latent Dirichlet Allocation (LDA) viene utilizzato per estrarre argomenti latenti dalle playlist, dove le playlist sono considerate come documenti e le canzoni come parole per il LDA. Viene quindi applicato il mining dei pattern sequenziali per trovare pattern di argomenti latenti nelle playlist. Al momento della raccomandazione, i pattern frequenti vengono mappati agli argomenti estratti dalla sessione di ascolto e utilizzati per filtrare le raccomandazioni generate da un classico sistema di raccomandazione user-based KNN.
- Metodi Graph-based: le preferenze a lungo termine e quelle recenti di un utente vengono fuse in un grafo bipartito a due lati, il Session Based Temporal Graph (STG). Un lato del grafo connette gli utenti agli item con cui hanno interagito (preferenze a lungo termine). L’altro lato del grafo connette gli identificatori delle sessioni agli item con cui l’utente ha interagito durante la sessione (preferenze recenti). Gli archi del grafo sono pesati in modo tale da riflettere più accuratamente l’influenza delle preferenze a breve e a lungo termine. Le relazioni tra gli item sono propagate tramite Injected Preference Fusion (IPF). Per generare le raccomandazioni, l’STG è attraversato tramite Temporal Personalized Random Walk (TPRW) per generare raccomandazioni session-aware.

Queste tecniche avanzate sono onerose dal punto di vista computazionale, ma non sempre si dimostrano più efficaci di tecniche più semplici[9]

**Automatic Playlist Continuation (APC)** Come anticipato, per quanto riguarda l’Automatic Playlist Continuation (APC), la letteratura è molto più carente rispetto al Next Track Recommendation. Di seguito analizziamo i principali lavori in questo contesto.

In uno studio del 2006[2], Pauws et al. hanno proposto un algoritmo che sfrutta tecniche di ricerca locale, in particolar modo il Simulated annealing (SA), per generare playlist che si allineino con le preferenze musicali degli utenti. Il loro approccio definisce formalmente un modello per il problema della generazione automatica di playlist, riconoscendo la sua struttura NP-difficile (NP-hard). Gli autori hanno introdotto tecniche euristiche come riduzione del dominio delle canzoni (song domain reduction) e voto parziale dei vincoli (partial constraint voting) per migliorare l’efficienza e la qualità delle playlist generate. I loro risultati dimostrano miglioramenti significativi rispetto ai metodi tradizionali di soddisfacimento dei vincoli (constraint satisfaction).

In uno studio del 2015[6], Jannach et al. propongono un approccio algoritmico e uno schema di ottimizzazione per generare continuazioni di playlist che siano coerenti con i brani appena ascoltati. Utilizzando collezioni di playlist condivise, metadati musicali e preferenze degli utenti, il loro metodo seleziona brani con alta accuratezza e applica un processo di riordinamento per ottimizzare la coerenza con la sequenza di ascolto recente. I risultati empirici dimostrano che la combinazione di segnali di input diversi migliora l’accuratezza della selezione dei brani e che la tecnica di riordinamento aiuta a bilanciare gli obiettivi di qualità e ad aumentare ulteriormente l’accuratezza.

In uno studio del 2016[7], Bittner et al hanno sviluppato metodi per la sequenziazione automatica delle playlist e l’aggiunta di transizioni in stile DJ tra i brani. La loro ricerca affronta la problematica della creazione di una sequenza ottimale dei brani in una playlist, modellata come un problema di attraversamento di grafi, e delle transizioni tra i brani, trattate come un problema di ottimizzazione. Utilizzando i dati degli ascoltatori e valutazioni soggettive da parte di curatori professionisti, i loro metodi sono stati progettati per migliorare l’esperienza di ascolto automatizzando la sequenza e le transizioni dei brani in modo simile a quanto farebbe un DJ professionista. Tuttavia, questo lavoro si focalizza soltanto su lla musica party, e potrebbe non essere adatto ad altri tipi di musica.

Nel 2018[10], Shih e Chi hanno sviluppato un sistema per la generazione automatica di playlist musicali in modo flessibile, sfruttando tecniche di deep learning e di reinforcement learning. Gli autori hanno definito formalmente la generazione di playlist come un problema di language modeling, utilizzando un modello di linguaggio RNN migliorato tramite algoritmi di policy gradient. Il principale vantaggio di questo approccio è la flessibilità: tramite l’utilizzo di multiple reward functions, è possibile spingere il modello a generare playlist in base alle necessità specifiche dell’utente, come freshness (canzoni più o meno datate), novelty (canzoni familiari o nuove), e diversity (canzoni dello stesso artista o di artisti diversi). Il metodo ha ottenuto performance migliori rispetto a modelli

pre-addestrati, tuttavia, ulteriori lavori sono necessari per integrare in questo sistema le caratteristiche audio dei brani.

Nel 2019[12], Liebman et al. hanno introdotto un framework, chiamato DJ-MC, che si adatta alle preferenze di ordinamento delle canzoni dell’utente, apprese in tempo reale. Il processo di generazione di playlist è modellato tramite Markov Decision Process (MDP), e si fa uso del roll-out di Monte Carlo per valutare e pianificare le sequenze di brani che massimizzeranno la soddisfazione dell’utente. Le canzoni sono rappresentate tramite 34 features, ottenute partendo dal Million Song Dataset. La valutazione è avvenuta assegnando casualmente a ciascuno dei 47 partecipanti un metodo scelto tra DJ-MC o un approccio GREEDY. Ciascun brano è stato riprodotto per 60 secondi. Al termine della riproduzione di ogni brano, gli studenti hanno rilasciato un feedback sulla qualità della transizione tramite un’interfaccia grafica. I risultati hanno mostrato che DJ-MC produce un reward di transizione medio più alto rispetto a GREEDY, con un miglioramento statisticamente significativo ( $p \leq 0.05$ ). Nonostante i buoni risultati, questo approccio si differenzia dagli altri metodi presenti in letteratura poiché non analizza i dati di ascolto storici dell’utente. Invece di essere progettato per generare playlist complete, si concentra sull’adattamento in tempo reale alle preferenze musicali dell’utente, apprendendo dai suoi feedback durante l’ascolto.

### 1.1.7 Altri lavori

Dopo aver esaminato i principali studi presenti in letteratura sulla generazione di raccomandazioni musicali sequence-aware, in questa sezione presenteremo sinteticamente i lavori più rilevanti che mirano a personalizzare le raccomandazioni considerando dati contestuali, allontanandoci leggermente dal contesto teorico principale di questa tesi.

**Approccio basato sullo stato affettivo** Mallol-Ragolta et al.[18] propongono un sistema che utilizza uno spazio circumplex arousal-valence per creare playlist che seguono un percorso emozionale. Utilizzando un approccio K-nearest neighbor, il sistema sceglie canzoni che gradualmente attraversano lo spazio affettivo, partendo da uno stato emozionale iniziale fino a raggiungere lo stato desiderato.

**Utilizzo di chatbot e analisi del sentimento** Nair et al.[15] sviluppano un sistema di raccomandazione basato su chatbot interattivo che analizza le emozioni dell’utente attraverso una serie di domande. Le risposte degli utenti vengono utilizzate per generare una playlist su misura attraverso l’API di Spotify. Questo sistema classifica le emozioni in tre categorie: Positivo, Negativo e Neutro, con l’obiettivo di espandere ulteriormente lo spettro emotivo in futuro.

**Riconoscimento delle emozioni** Upadhyay et al.[21] presentano un generatore di playlist basato sul riconoscimento delle emozioni facciali attraverso una webcam. Ut-

lizzando un modello di rete neurale convoluzionale (CNN), il sistema rileva l'umore dell'utente e genera una playlist corrispondente. Questo approccio semplifica il processo di selezione delle playlist in base all'umore rilevato, con un'accuratezza del 97.42% sui dati di addestramento.

In un altro studio, Shaurya Gaur[23] propone un metodo innovativo basato sul K-Nearest Neighbor (KNN) per la generazione automatica di playlist musicali che seguono un percorso emotivo dinamico.

**Apprendimento per rinforzo** Tomasi et al.[25] propongono un approccio basato sull'apprendimento per rinforzo (RL) per la generazione automatica di playlist. Utilizzando un Deep Q-Network modificato, il sistema è in grado di fare raccomandazioni da ampi set di candidati, massimizzando le metriche di consumo. I risultati mostrano un miglioramento significativo della soddisfazione degli utenti rispetto ai metodi di base durante i test A/B online.

**Generazione di playlist basata su espressioni facciali** Patel e Gupta [16] utilizzano un approccio basato sulle espressioni facciali e sulla cronologia delle selezioni musicali dell'utente per generare playlist personalizzate. L'uso di CNN per il rilevamento delle emozioni e di reti neurali artificiali (ANN) per la classificazione delle canzoni ha portato a un'accuratezza dell'84% nel rilevamento delle emozioni e dell'82% nella classificazione delle canzoni.

**Utilizzo di Knowledge Graph** Sakurai et al.[20] propongono un metodo innovativo che combina grafi di conoscenza e apprendimento per rinforzo per la generazione di playlist musicali. Utilizzando dati di grafi di conoscenza per ottimizzare l'apprendimento, il sistema guida gli utenti verso nuovi brani musicali che soddisfano le loro preferenze uniche. I risultati sperimentali dimostrano che il metodo proposto supera i metodi esistenti nella generazione di playlist, sebbene ci siano ancora problemi da risolvere, come la verifica dell'efficacia del modello con utenti reali e il perfezionamento della funzione di ricompensa.

**Generazione di playlist context-aware** Con l'avvento e la diffusione di dispositivi smart e activity tracker, che monitorano costantemente lo stato fisiologico degli utenti, diventa possibile creare raccomandazioni musicali altamente personalizzate in base al contesto d'uso. Alcuni studi emergenti si stanno occupando proprio di questo.

Lo studio "Context Aware Music Recommendation and Playlist Generation" [28] di Elias Mann, esplora l'uso di contesti fisiologici e situazionali per migliorare i sistemi di raccomandazione musicale.

### 1.1.8 Conclusioni

Gli studi osservati mostrano spunti promettenti per migliorare i sistemi di raccomandazione musicale. Tuttavia, molti di questi metodi richiedono che gli utenti interagiscano con dispositivi hardware (webcam) o software (chatbot) oppure che indossino uno smart-watch o altri dispositivi smart in grado di monitorare costantemente il loro stato fisiologico. Questi requisiti possono limitare l'applicabilità del sistema, poiché non tutti gli utenti possiedono o indossano regolarmente tali dispositivi, soprattutto mentre ascoltano musica.

Inoltre, si è osservato che la maggior parte delle soluzioni attuali offrono solo una personalizzazione parziale, limitandosi a considerare quali brani musicali riprodurre, senza tener conto dell'ordinamento delle canzoni.

Queste osservazioni hanno portato alla proposta di un metodo che è in grado di comprendere non solo quali brani l'utente preferisce, ma anche quando e come preferisce ascoltarli, al fine di soddisfare al meglio il suo gusto musicale.

## 2 La nuova proposta

Questo capitolo mira a descrivere il metodo SMART discusso nella introduzione. Per prima cosa forniremo una descrizione ad alto livello del metodo, concentrandoci sui principali passaggi e concetti coinvolti. Successivamente, approfondiremo la nostra analisi con una descrizione più dettagliata, esaminando specificamente gli algoritmi e le tecniche utilizzate per la generazione delle playlist.

### 2.1 Descrizione generale

Per generare playlist personalizzate è necessario capire quali brani piacciono all’utente. Tuttavia, per ottenere una personalizzazione maggiore, è necessario comprendere anche quando e come gli utenti ascoltano musica. La letteratura analizzata nel Capitolo 1 mostra come le playlist personalizzate vengano generate considerando solo il ”cosa” (i brani), mentre il ”quando” e il ”come” non sono stati presi in considerazione dagli algoritmi esistenti. Questa personalizzazione parziale potrebbe generare playlist che piacciono all’utente, ma che potrebbero non essere appropriate per il momento specifico in cui desidera ascoltare musica. Ad esempio, un utente potrebbe voler ascoltare musica tranquilla nei momenti di concentrazione e musica più energica durante l’attività fisica. Oltre alla potenziale inappropriatezza delle playlist per certi momenti della giornata, è possibile che l’ordine delle canzoni presenti non soddisfi le preferenze dell’utente. In particolare, alcuni utenti potrebbero preferire alternare brani con ritmi diversi, mentre altri potrebbero preferire una sequenza di brani con ritmi simili.

SMART mira a generare playlist altamente personalizzate considerando cosa, quando e come gli utenti ascoltano musica. Il funzionamento prevede la generazione di una singola playlist da riprodurre quando l’utente accede alla piattaforma musicale. Tale playlist è generata per soddisfare le abitudini di ascolto dell’utente, ovvero, è sensibile al periodo della giornata in cui l’utente decide di ascoltare musica ed è ordinata secondo le sue preferenze di ascolto per tale periodo.

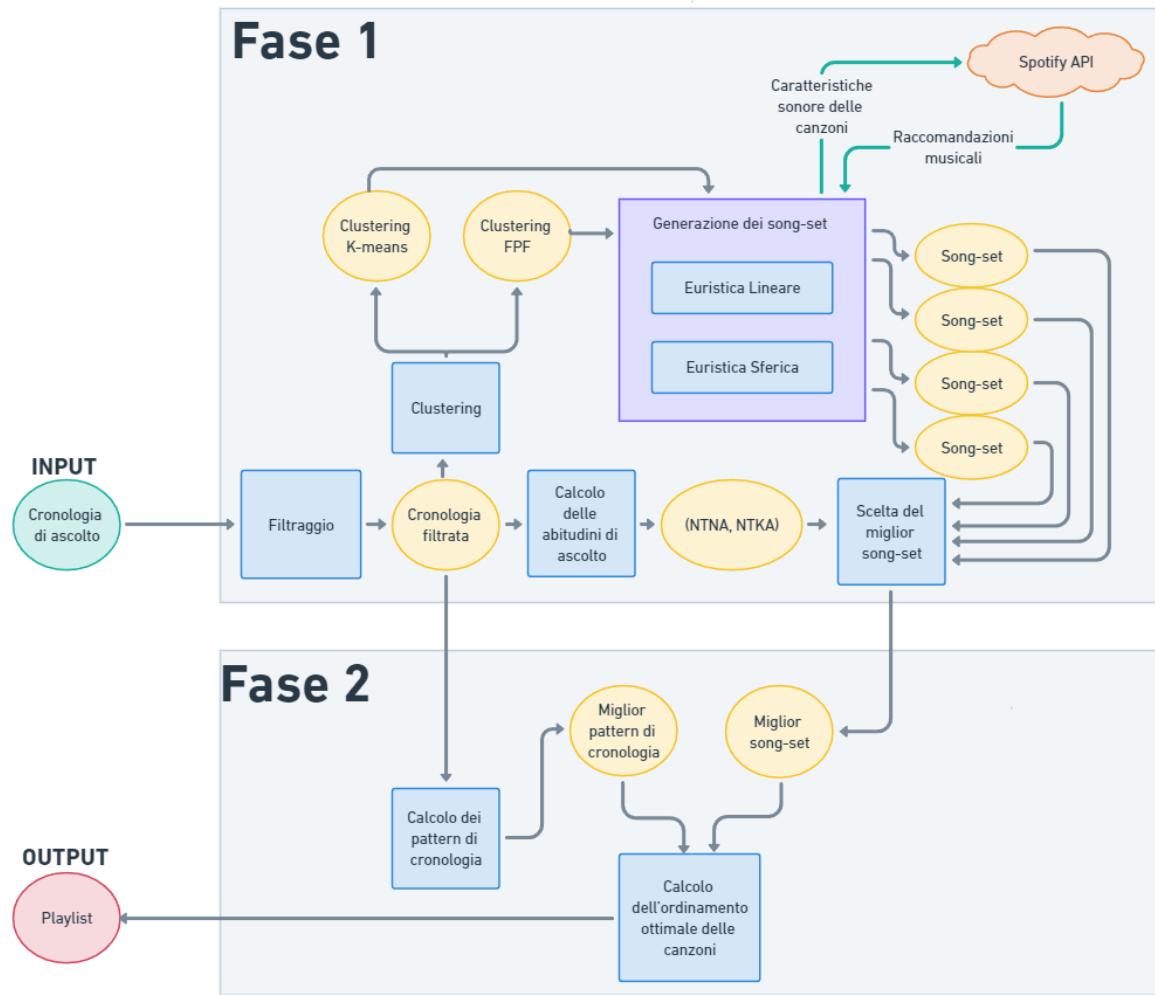


Figura 1: Architettura ad alto livello di SMART

La Figura 1 mostra l’architettura generale di SMART. È composto da due fasi principali: la prima fase è responsabile della produzione di un insieme di brani (song-set) che riflette le abitudini di ascolto dell’utente per il periodo (quando) in cui desidera ascoltare musica. La prima fase è costituita dai seguenti moduli:

1. elaborazione della cronologia di ascolto;
2. calcolo delle abitudini di ascolto;
3. clustering (Raggruppamento);
4. generazione dei song-set e scelta di quello migliore.

Questi moduli, gestendo opportunamente la cronologia di ascolto dell’utente, producono un insieme di brani simili (song-set) a quelli che l’utente è solito ascoltare in un determinato periodo della giornata. Con ”brani simili” ci si riferisce a due concetti: si ricerca una selezione di brani con caratteristiche sonore simili a quelli precedentemente riprodotti dall’utente e si mira a ottenere un insieme di canzoni che presenti una percentuale simile di musica nuova e conosciuta rispetto alle abitudini di ascolto dell’utente in un dato momento della giornata. È importante notare che questa fase si basa sul sistema di raccomandazione musicale di Spotify, che ci permette di trasformare una canzone nella cronologia di ascolto, rappresentata come una sequenza di caratteristiche audio di basso livello, in una canzone simile. La seconda fase si occupa di ordinare l’insieme di brani prodotto dalla prima fase, al fine di imitare il modo (come) in cui l’utente ascolta musica rispetto alle sue abitudini di ascolto.

## 2.2 Fase 1 - Elaborazione della cronologia di ascolto

Questo modulo interagisce con la cronologia di ascolto dell’utente, rappresentata come una sequenza ordinata di canzoni riprodotte in un intervallo temporale, solitamente di mesi o anni. Ciascuna canzone nella cronologia è descritta tramite alcuni metadati ad alto livello, come i nomi degli artisti, il titolo della canzone e il genere. Per prima cosa, questo modulo trasforma ogni canzone presente nella cronologia, accedendo a un database di canzoni, in una sequenza di caratteristiche audio di basso livello. Formalmente, supponiamo che la cronologia di ascolti di un utente sia composta da  $h$  tracce. Ogni canzone  $s$  è trasformata in un vettore di caratteristiche  $\mathbf{x}_s \in \mathbb{R}^f$ , dove  $f$  è il numero di caratteristiche audio che costituiscono ciascun vettore. Di conseguenza, la cronologia di ascolto può essere vista come una matrice  $H$  con  $h$  righe e  $f$  colonne. Nel nostro caso,  $f$  è pari a 12, come illustrato nella Figura 2.

Dopo essere stata trasformata in una sequenza di vettori di caratteristiche audio, la cronologia di ascolto viene filtrata. Il filtraggio avviene come segue: supponendo che un utente si connetta alla piattaforma musicale durante il tempo  $t$  del giorno  $d$ , consideriamo

soltanto le  $h_d \leq h$  canzoni riprodotte dall'utente durante il periodo  $P = [t, t + v]$ . Di conseguenza, indichiamo con  $H_d \in \mathbb{R}^{h_d \times f}$  la matrice che contiene soltanto quelle righe di  $H$  che rappresentano le abitudini di ascolto dell'utente durante il periodo  $P$ .

Il filtraggio viene effettuato includendo soltanto le canzoni ascoltate dall'utente nei periodi passati corrispondenti al momento in cui si è connesso alla piattaforma. Ad esempio, se l'utente si connette alle 17 di Giovedì, la cronologia filtrata conterrà tutte le canzoni ascoltate nel passato dalle 17 alle 18 di Giovedì. La lunghezza del periodo  $v$  è fissata a un'ora. In questo modo, la generazione del song-set verrà effettuata a partire soltanto da quelle canzoni che sono state ascoltate dall'utente nel periodo di riferimento.

### 2.3 Fase 1 - Calcolo delle abitudini di ascolto

Questo modulo, dato un periodo di riferimento, si occupa di calcolare le abitudini di ascolto di un utente, aggiungendo ulteriore personalizzazione. Supponendo che l'utente si connetta alla piattaforma musicale durante il tempo  $t$  del giorno  $d$ , siamo interessati a conoscere la musica che l'utente ha ascoltato partendo dal tempo  $t$ . A questo proposito, sia  $P$  il periodo che inizia al tempo  $t$  e termina al tempo  $t + v$ :  $P = [t, t + v]$ . Assumiamo inoltre che la cronologia di ascolti non sia vuota durante il periodo  $P$ . Per ogni giorno  $i$  all'interno della cronologia di ascolto, consideriamo il sottoinsieme  $H_i \subseteq H$  di canzoni ascoltate durante il periodo  $P$  nel giorno  $i$ , e calcoliamo la percentuale di nuove canzoni riprodotte dall'utente, facendo una distinzione tra artisti sconosciuti e artisti conosciuti. Le nuove canzoni riprodotte dall'utente consistono in quei brani che non sono mai stati riprodotti prima del giorno  $i$ . Analogamente, gli artisti conosciuti (risp. sconosciuti) consistono negli artisti per cui sono stati (risp. non sono stati) riprodotti brani prima del giorno  $i$ . Più dettagliatamente, calcoliamo:

$$NTNA = 100 \times \left( \frac{\sum_{i \in Days} dNTNA_i}{h} \right) \quad (1)$$

$$NTKA = 100 \times \left( \frac{\sum_{i \in Days} dNTKA_i}{h} \right) \quad (2)$$

Dove,  $Days$  è l'insieme di giorni nella cronologia di ascolto,  $dNTNA_i$  rappresenta il numero di nuove tracce di artisti sconosciuti riprodotte durante il giorno  $i$  nel periodo  $P$ , e  $dNTKA_i$  rappresenta il numero di nuove tracce di artisti conosciuti riprodotte durante il giorno  $i$  nel periodo  $P$ . La coppia  $P_h = (NTNA, NTKA)$  rappresenta le abitudini di ascolto dell'utente durante il periodo  $P$ , per quanto riguarda la nuova musica, e verrà utilizzato per personalizzare la playlist.

### 2.4 Fase 1 - Clustering

Come anticipato nella Sezione 2.2, a ciascuna canzone presente nella cronologia di ascolto sono associate le 12 caratteristiche audio elencate nella Figura 2. L'obiettivo di questo

<b>Acousticness</b>	"Acusticità" della canzone [0,1]
<b>Danceability</b>	"Ballabilità" della canzone [0,1]
<b>Energy</b>	Percezione di intensità e di attività nella canzone [0,1]
<b>Instrumentalness</b>	Presenza di voci nella canzone [0,1]
<b>Key</b>	Classe di altezze della canzone in notazione intera [-1,11]
<b>Liveness</b>	Presenza di pubblico nella canzone [0,1]
<b>Loudness</b>	Rumorosità della canzone in decibels
<b>Mode</b>	Modo della canzone (maggiore o minore) [0 o 1]
<b>Speechiness</b>	Presenza di parole pronunciate nella canzone [0,1]
<b>Tempo</b>	BPM (Beats per Minute) stimati della canzone
<b>Time_signature</b>	Numero di movimenti per battuta della canzone [3,7]
<b>Valence</b>	Positività della canzone [0,1]

Figura 2: Caratteristiche audio di basso livello utilizzate per rappresentare le canzoni

modulo è quello di comprendere cosa l’utente desidera ascoltare. Ci occuperemo di questo utilizzando la cronologia di ascolto filtrata, prodotta in output dal primo modulo (discusso nella Sezione 2.2). Nella fattispecie, la cronologia di ascolto filtrata verrà fornita come dataset in input agli algoritmi di clustering che andremo ad applicare. In questo modo, saremo in grado di capire se le canzoni riprodotte possiedono caratteristiche audio comuni o meno.

Per quanto riguarda il nostro caso di studio, andremo ad applicare alla cronologia di ascolto filtrata due differenti algoritmi di clustering: K-means e Furthers-Point-First, discussi nella Sezione 1.1.1. La scelta di applicare due algoritmi di clustering anzichè uno solo evita che le specificità di un algoritmo influenzino i risultati.

Entrambi gli algoritmi di clustering richiedono due elementi: un numero di cluster da produrre ( $k$ ) e una metrica di distanza. Per quanto riguarda la metrica di distanza, utilizziamo la distanza Euclidea:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Per quanto riguarda il numero di cluster da produrre, utilizziamo l’indice di Davies-Bouldin: questo indice, tramite un calcolo che coinvolge la coesione intra-cluster (distanza tra elementi di uno stesso cluster) e la separazione tra cluster (distanza tra elementi di

cluster distinti), ci fornisce una stima del numero appropriato di cluster da utilizzare. Di seguito elenchiamo i passi per il calcolo dell'indice di Davies Bouldin. Siano  $\{x_1, \dots, x_N\}$  i punti da clusterizzare e sia  $\{C_1, \dots, C_k\}$  la partizione da valutare (insieme dei  $k$  clusters, ognuno di cardinalità  $n_j$ , con  $j = 1, \dots, k$ ). Per ogni cluster  $C_j$ , con  $j = 1, \dots, k$ , calcoliamo:

Il centroide:

$$m_j = \frac{1}{n_j} \sum_{x_i \in C_j} (x_i)$$

La coesione intra-cluster:

$$e_j^2 = \frac{1}{n_j} \sum_{x_i \in C_j} (x_i - m_j)^T \cdot (x_i - m_j)$$

Per ogni cluster  $C_h$ , con  $h = 1, \dots, k$ , la separazione tra cluster:

$$dm(j, h) = d(m_j, m_h)$$

dove  $d(m_j, m_h)$  è la distanza euclidea tra  $m_j$  e  $m_h$ .

Successivamente, per ogni coppia di cluster  $(j, h)$ , con  $j, h = 1, \dots, k$ , calcoliamo:

$$R_{jh} = e_j + e_h + dm(j, h)$$

Per ogni cluster  $C_j$ , con  $j = 1, \dots, k$ , calcoliamo:

$$R_j = \max_{j \neq h} \{R_{jh}\}$$

L'indice di Davies-Bouldin viene determinato come:

$$DB(\{C_1, \dots, C_k\}) = \frac{1}{k} \sum_{j=1}^k R_j$$

Siccome la qualità del clustering aumenta al diminuire di questo indice, l'obiettivo è quello di eseguire i due algoritmi di clustering con il numero  $k$  di clusters che minimizzano l'indice di Davies Bauldin. Il metodo SMART esegue entrambi gli algoritmi con un numero crescente di cluster  $k$ :  $k \in [2, \sqrt{h}]$ . Ad ogni esecuzione, si tiene traccia dell'indice di Davies Bauldin e, alla fine, si mantiene, per ogni algoritmo, la soluzione che ha minimizzato tale indice.

Concludendo, questo modulo produce due clustering: il clustering  $K = k\text{-means}(H_d)$ , generato tramite k-means impiegando  $n_k$  cluster, e il clustering  $F = FPF(H_d)$ , generato tramite Furthers-Point-First impiegando  $n_f$  cluster.

## 2.5 Fase 1 - Generazione dei song-set

Questo modulo prende in input i due clustering  $K$  e  $F$  prodotti dal modulo precedente, e li utilizza per restituire un insieme di canzoni (song-set) la cui coppia ( $NTNA, NTKA$ ) è più simile possibile alla coppia  $P_h$  che era stata calcolata nella Sezione 2.3 con le equazioni 1 e 2. A questo proposito, sfruttiamo il sistema di raccomandazioni di Spotify, che richiede in input due elementi: l'ID di una traccia e un vettore di caratteristiche, non necessariamente corrispondenti alla traccia stessa. L'output del sistema di raccomandazioni consiste in un insieme di  $m$  canzoni. Definiamo  $m = \frac{\ell}{n \times 4}$  dove  $\ell$  è il numero di brani che la playlist deve avere,  $n \in \{n_k, n_F\}$  è il numero di cluster, e 4 è il numero di richieste al sistema di raccomandazione musicale per ogni cluster. Ad esempio, supponiamo che la playlist debba avere  $\ell = 48$  brani e supponiamo che il numero di cluster sia  $n_k = 4$ , allora, per ogni punto, chiediamo al sistema di raccomandazione musicale di restituire  $m = 3$  brani.

Per produrre l'input per il sistema di raccomandazioni, facciamo uso di due euristiche differenti, una lineare e una sferica: dato un clustering  $C = \{C_1, C_2, \dots, C_t\}$ , con  $C \in \{K, F\}$ , entrambe le euristiche operano cluster per cluster,  $C_i$ , con  $i = 1, \dots, t$  e producono un insieme di canzoni  $S_c$ . In pratica, ciascuna euristica produce un insieme di canzoni per ciascun clustering. Siccome abbiamo due euristiche e due clustering, alla fine avremo quattro song-set. Tra queste, si sceglierà come output di questo modulo il song-set la cui coppia ( $NTNA, NTKA$ ) è più simile possibile a  $P_h$ , utilizzando come metrica la distanza euclidea.

Vediamo nel dettaglio il funzionamento delle due euristiche:

- Euristica lineare: dato un cluster  $C_i$ , con  $i = 1, \dots, t$ , ordiniamo i suoi punti in senso decrescente di distanza dal centroide, utilizzando la distanza Euclidea. Successivamente, consideriamo i seguenti quattro punti:

1.  $c_{i_1}$ : il centroide.
2.  $c_{i_2}$ : il punto in posizione  $\left\lfloor \frac{|C_i|}{3} \right\rfloor$ .
3.  $c_{i_3}$ : il punto in posizione  $2 \left\lfloor \frac{|C_i|}{3} \right\rfloor$ .
4.  $c_{i_4}$ : il punto più lontano dal centroide.

Questi quattro punti, insieme ai relativi ID di traccia, vengono utilizzati come input per il sistema di raccomandazioni. Il centroide ( $c_{i_1}$ ) può essere considerato come una sintesi delle abitudini musicali dell'utente per l'insieme di canzoni nel cluster, mentre gli altri tre punti, allontanandosi gradualmente dal centroide, permettono di riempire il song-set anche con canzoni che non rispecchiano perfettamente i gusti musicali dell'utente. La figura 3 chiarisce l'idea graficamente.

- Euristica a sfera: dato un cluster  $C_i$ , con  $i = 1, \dots, t$ , consideriamo una ipersfera  $f$ -dimensionale centrata nel centroide del cluster, il cui raggio coincide con la massima distanza tra il centroide e i punti del cluster. Successivamente, selezioniamo casualmente un punto  $rp$  nella ipersfera, e lo utilizziamo per calcolare un nuovo punto  $p$ , le cui coordinate soddisfano i vincoli imposti dalle caratteristiche audio (consultabili nella figura 2). Questo processo è necessario in quanto la generazione casuale di un punto all'interno della ipersfera non implica necessariamente che le coordinate di quest'ultimo soddisfino i vincoli imposti dalle caratteristiche audio. Per calcolare il punto  $p$  partendo da un punto casuale  $rp$ , si opera in questo modo: per ogni coordinata il cui valore cade al di fuori dell'intervallo imposto dalla caratteristica audio corrispondente, impostiamo tale valore all'estremo più vicino dell'intervallo valido. Ad esempio, una volta generato il punto  $rp$ , se questo dovesse avere la coordinata corrispondente alla caratteristica audio «acousticness» pari a 1.2, allora dovremo porre tale coordinata pari a 1, ossia l'estremo più vicino dell'intervallo valido relativo alla caratteristica «acousticness». Siccome non è detto che il punto  $p$  che abbiamo ricavato appartenga al cluster, cerchiamo all'interno di quest'ultimo il punto  $cp$  più vicino a  $p$ . Infine, utilizziamo come input per il sistema di raccomandazioni il punto  $cp$  e il relativo ID di traccia. Ripetiamo questa procedura fino all'ottenimento di quattro punti differenti. Nel caso in cui il numero di cluster sia minore di quattro, il processo di selezione termina quando il numero di punti ottenuti tramite questo procedimento risulta uguale alla cardinalità del cluster. La figura 4 chiarisce l'idea graficamente.

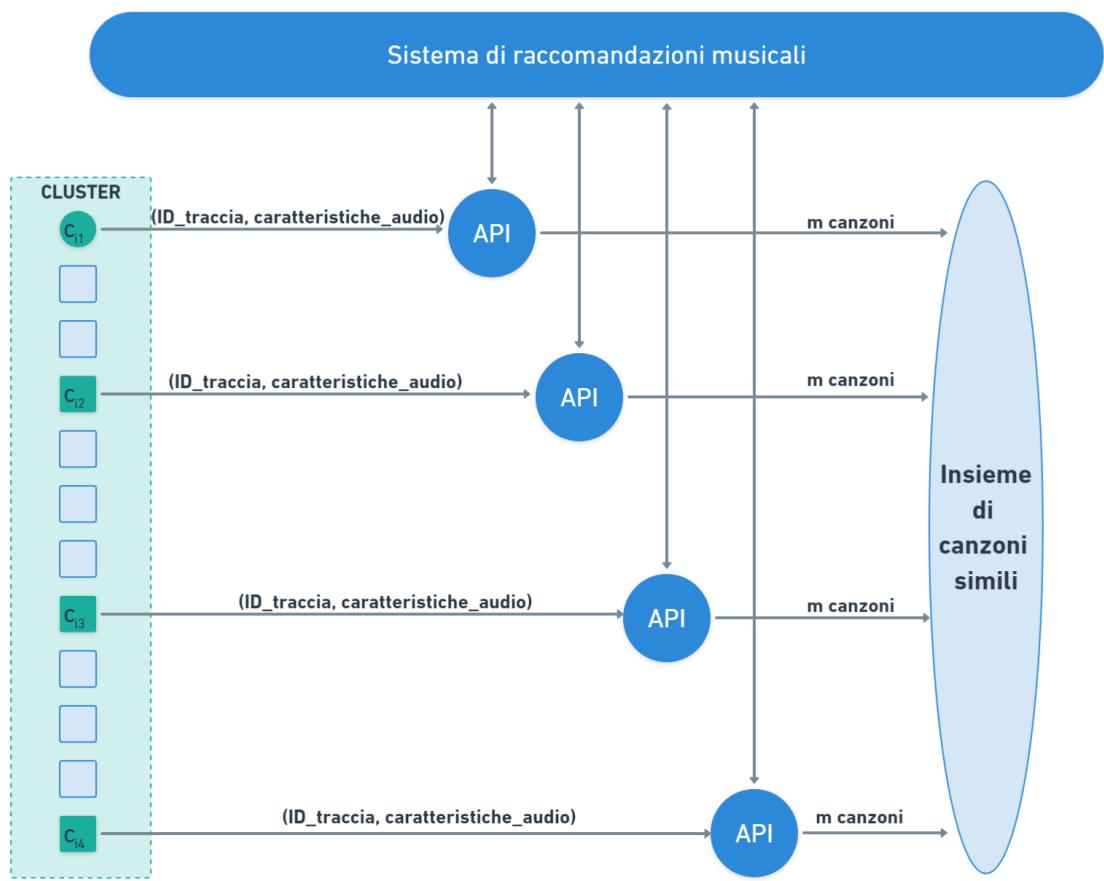


Figura 3: Euristica lineare

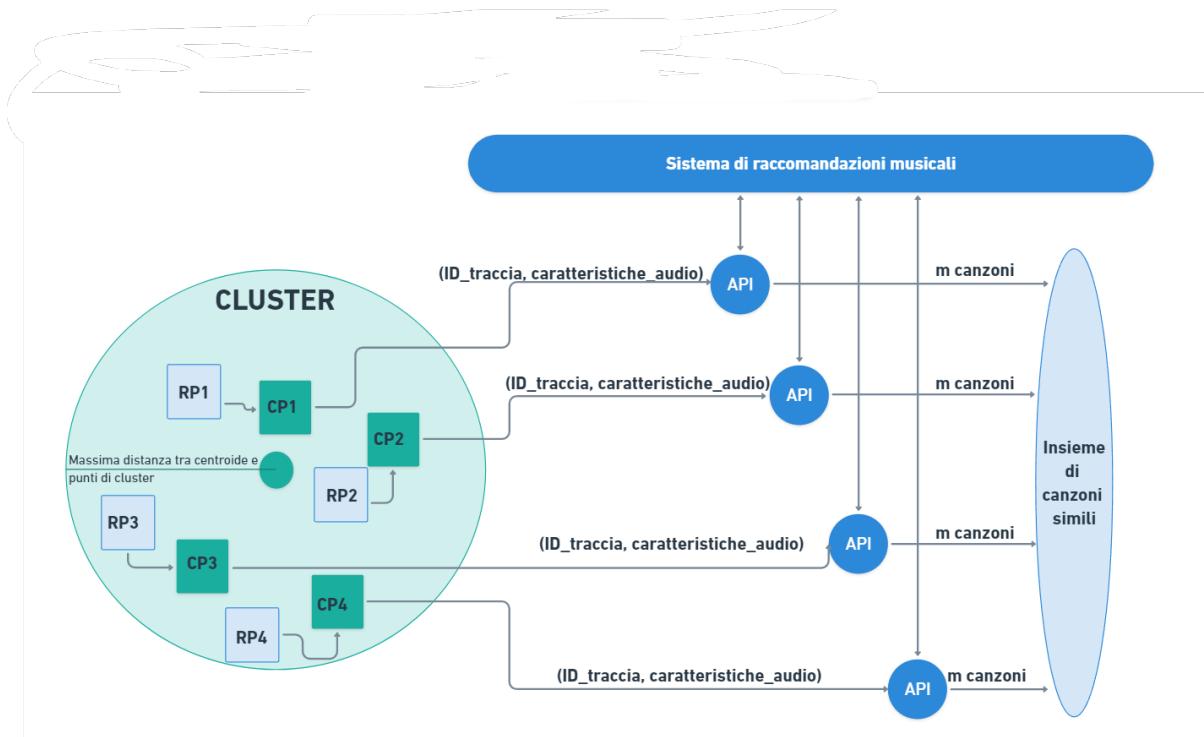


Figura 4: Euristica a sfera

## 2.6 Fase 2 - Ordinamento del song-set

La prima fase di SMART ha prodotto un song-set contenente canzoni simili a quelle che piacciono all’utente in un periodo di riferimento. La seconda fase, costituita soltanto da questo modulo, prende in input la cronologia di ascolto e il song-set restituito dalla prima fase, e restituisce una playlist che soddisfa le abitudini di ascolto dell’utente. La necessità di questo modulo è motivata dal fatto che, siccome ogni utente ha il proprio modo di ascoltare la musica, è improbabile che un insieme di canzoni privo di un ordinamento appropriato (ossia quello restituito dalla prima fase) soddisfi a pieno le abitudini di ascolto di un utente. Ad esempio, durante un’attività sportiva, alcuni utenti potrebbero preferire ascoltare musica più energica all’inizio e musica meno energica alla fine, mentre altri potrebbero preferire l’esatto opposto, ossia iniziare con musica più tranquilla e terminare con musica più energica. Per questo motivo, è necessario comprendere come un utente ascolta la musica: senza conoscere questo aspetto, la playlist non sarà mai completamente personalizzata.

La restante parte di questa sezione mira a definire formalmente il problema di ordinamento delle canzoni appena discusso, facendo interamente riferimento allo studio[17]

«Automatic and Personalized Sequencing of Music Playlists», condotto da Manuela Montangero e Marco Furini nel 2022.

Per comprendere come un utente ascolta musica, analizziamo la sua cronologia di ascolto durante il periodo  $P$ . Poiché ogni brano è descritto attraverso un vettore di caratteristiche audio (ovvero un punto nello spazio  $f$ -dimensionale, dove  $f$  è il numero totale di caratteristiche audio del vettore), la cronologia di ascolto può essere vista come un percorso che si sposta da un punto a un altro punto. Questo percorso è rappresentato dal pattern di cronologia di ascolto dell'utente ed è definito formalmente come segue:

**Definizione 1.** *Un pattern di cronologia di ascolto  $hP = \langle h_1, h_2, \dots, h_k \rangle$  di lunghezza  $k$  è una sequenza ordinata di  $k$  punti  $h_i \in \mathbb{R}^f$ , dove  $f$  è la dimensione del vettore di caratteristiche delle canzoni e dove la canzone  $h_i$  è stata riprodotta subito prima della canzone  $h_{i+1}$ , per  $i = 1, \dots, k - 1$ .*

Analogamente, possiamo definire il pattern di playlist come segue:

**Definizione 2.** *Un pattern di playlist  $\langle p_1, p_2, \dots, p_m \rangle$  di lunghezza  $m$  è una sequenza ordinata di  $m$  punti  $p_i \in \mathbb{R}^f$ , dove  $f$  è la dimensione del vettore di caratteristiche delle canzoni e dove la canzone  $p_i$  viene riprodotta nella playlist subito prima della canzone  $p_{i+1}$ , per  $i = 1, \dots, m - 1$ .*

L'obiettivo di questo modulo è quello di generare una playlist il cui pattern è più simile possibile al pattern di cronologia di ascolto, partendo dal song-set prodotto dalla prima fase.

Sia  $C$  l'insieme di canzoni prodotte dalla prima fase e sia  $hP$  il pattern di cronologia di ascolto dell'utente composto da  $k$  vertici. Per generare la playlist ordinata, questo modulo definisce l'ordine di riproduzione delle canzoni selezionate in un pattern di playlist  $P^*$ , facendo in modo che la sua forma risulti il più simile possibile a quella del pattern di cronologia di ascolto  $hP$ . Questo consente alla playlist di essere altamente personalizzata: infatti, oltre a cosa e quando, la playlist ordinata consente di imitare come un utente ascolta la musica. Per ordinare le canzoni, innanzitutto definiamo una metrica di distanza tra coppie di pattern di playlist che cattura l'idea di forme simili, e quindi formalizziamo il nostro problema come un problema di ottimizzazione che possiamo risolvere in modo ottimale con un algoritmo di programmazione dinamica (ovvero, determinare l'ordine delle canzoni nella playlist in modo che la sua forma sia la più simile possibile alla cronologia di ascolto rispetto a qualsiasi altro ordine).

**Definizione 3.** *Definizione 3.3. Sia  $d(\cdot, \cdot)$  la distanza euclidea. Date due playlist di lunghezza  $t$ ,  $P_1 = \langle p_{1,1}, p_{1,2}, \dots, p_{1,t} \rangle$  e  $P_2 = \langle p_{2,1}, p_{2,2}, \dots, p_{2,t} \rangle$ , definiamo le seguenti distanze tra  $P_1$  e  $P_2$ :*

- *Pattern Vertex Distance (VD): la somma delle distanze tra coppie di vertici nelle stesse posizioni nei due pattern:*

$$VD(P_1, P_2) = \sum_{i=1}^t d(p_{1,i}, p_{2,i});$$

- *Pattern Segment Distance (SD): la somma dei valori assoluti delle differenze delle lunghezze dei segmenti corrispondenti nei due pattern:*

$$SD(P_1, P_2) = \sum_{i=1}^{t-1} |d(p_{1,i}, p_{1,i+1}) - d(p_{2,i}, p_{2,i+1})|;$$

- *Playlist Pattern Distance (PD): la somma tra la Pattern Vertex Distance (VD) e la Pattern Segment Distance (SD).*

$$PD(P_1, P_2) = VD(P_1, P_2) + SD(P_1, P_2).$$

Utilizzeremo la Playlist Pattern Distance per calcolare un punteggio di similarità tra la cronologia di ascolto dell’utente e qualsiasi altra playlist candidata. Si noti che, per definizione, la Playlist Pattern Distance è sempre un valore non negativo. Inoltre, maggiore è la Playlist Pattern Distance, maggiore è la differenza tra i due pattern (sia per quanto riguarda le canzoni che il loro ordinamento). Pertanto, dato un pattern di cronologia di ascolto  $hP$  e due pattern di playlist candidate  $P_1$  e  $P_2$ ,  $P_1$  è più simile a  $hP$  rispetto a  $P_2$  se  $PD(hP, P_1) < PD(hP, P_2)$ . Per ulteriori chiarimenti in merito all’intuizione geometrica alla base della definizione della Playlist Pattern Distance, si prega di consultare lo studio [17]. Siamo ora pronti a introdurre il problema di ottimizzazione.

**Definizione 4.** Il problema di minimizzazione della playlist pattern distance è definito come segue:

*INPUT: il pattern di cronologia di ascolto  $hP = \langle h_1, h_2, \dots, h_k \rangle$  di lunghezza  $k$  e il song-set di candidati  $C = \{c_1, c_2, \dots, c_m\} \subseteq R^f$ , con  $k \leq m$ .*

*OUTPUT: il pattern di playlist  $P^* = \langle p_1^*, p_2^*, \dots, p_k^* \rangle$  che minimizza la Playlist Pattern Distance  $PD$  con il pattern di cronologia di ascolto  $hP$ , con  $p_i \in C$ ,  $i = 1, \dots, k$ .*

In sostanza, cerchiamo il pattern di playlist  $P^*$  di lunghezza  $k$ , con vertici in  $C$ , tale che:

$$P^* = \operatorname{argmin}_{pP} PD(hP, pP).$$

La risoluzione di questo problema tramite un algoritmo di forza bruta renderebbe necessario calcolare la Playlist Pattern Distance tra  $hP$  e tutte le possibili combinazioni di pattern di playlist  $pP$ , partendo dal song-set  $C$  in input. Questo approccio non è percorribile nel nostro caso, in quanto porterebbe ad un costo computazionale esponenziale. A questo proposito, abbiamo pensato ad un algoritmo di programmazione dinamica che è in grado trovare la soluzione ottima al problema in tempo polinomiale.

Forniamo dapprima la definizione di sotto-pattern di playlist (e di cronologia):

**Definizione 5.** Dato un pattern di playlist (risp. di cronologia)  $pP = \langle p_1, \dots, p_k \rangle$  (risp.  $hP = \langle h_1, \dots, h_k \rangle$ ) e un numero intero  $i \in [1..k]$ ,  $pP_i = \langle p_1, \dots, p_i \rangle$  (risp.  $hP_i = \langle h_1, \dots, h_i \rangle$ ) è un sotto-pattern di  $pP$  (risp. di  $hP$ ) di lunghezza  $i$ .

Osserviammo che il problema possiede una sottostruttura ottima: sia  $P^* = \langle p_1^*, \dots, p_{k-1}^*, p_k^* \rangle$  la playlist ottima per  $hP = \langle h_1, \dots, h_k \rangle$ , allora il sotto-pattern  $P_{k-1}^* = \langle p_1^*, \dots, p_{k-1}^* \rangle$  deve necessariamente essere il miglior pattern per  $hP_{k-1} = \langle h_1, \dots, h_{k-1} \rangle$ , tra tutti quelli terminanti per  $p_{k-1}^*$ . Di conseguenza, definiamo il seguente sotto-problema:

**Definizione 6.** Dati  $i \in [1..k]$  e  $j \in [1..m]$ , il sotto-problema di minimizzazione della Playlist Pattern Distance per  $i$  e  $j$  consiste nel trovare un pattern di playlist  $P_{i,j}$  con i vertici scelti in  $C$ , tale che:

1. l'ultimo vertice è  $c_j$ ;
2. minimizza la Playlist Pattern Distance  $PD$  con il sotto-pattern di cronologia di ascolto di lunghezza  $i$   $hP_i = \langle h_1, h_2, \dots, h_i \rangle$ .

Definiamo  $M[i, j] = PD(hP_i, P_{i,j})$  come il valore di tale soluzione ottima.

Osserviamo che il numero totale di sotto-problemi è  $k \times m$ , e che il valore  $OPT$  della soluzione ottima relativo al problema originale è dato da

$$OPT = \min_{1 \leq j \leq m} \{M[k, j]\},$$

In pratica, tra tutti i pattern di playlist ottimi terminanti in ciascuno degli  $m$  candidati in  $C$ , selezioniamo quello che minimizza la playlist pattern distance con il pattern di cronologia di ascolto.

Per calcolare i valori  $M[i, j]$ , per tutti i  $j \in [1..m]$  e per valori crescenti di  $i$  nell'intervallo  $[1..k]$ , procediamo come segue:

- $M[1, j] = d(h_1, c_j)$ , for  $j = 1, \dots, m$ ;
- Per ogni  $i \in [2..k]$  e  $j \in [1..m]$ ,  $M[i, j]$  è calcolato mediante i valori  $M[i - 1, t]$ , con  $t = 1, \dots, m$ , nel seguente modo:

$$M[i, j] = d(h_i, c_j) + \min_{1 \leq t \leq m} \{M[i - 1, t] + |d(h_{i-1}, h_i) - d(c_t, c_j)|\}.$$

**Lemma 1.** Dato il sotto-pattern di cronologia  $hP_i$  e l'insieme candidato  $C = \{c_1, \dots, c_m\}$ ,  $M[i, j]$  è il valore della soluzione ottima del sotto-problema di minimizzazione della Playlist Pattern Distance per  $i$  e  $j$ , con  $i \in [1..k]$  e  $j \in [1..m]$ .

Per la dimostrazione, si prega di consultare lo studio [17] già più volte citato.

Per determinare la soluzione  $P^* = \langle p_1^*, \dots, p_k^* \rangle$  che ha valore  $OPT$ , utilizziamo una matrice  $V$  di dimensioni  $k \times m$ : in  $V[i, j]$  memorizziamo l'indice  $t$  che è stato utilizzato per calcolare  $M[i, j]$ , per ogni coppia di indici  $i, j$ . Successivamente, partendo dall'ultimo vertice del pattern di cronologia e procedendo all'indietro, recuperiamo tutti i vertici della soluzione ottimale. Nel dettaglio:

- Il vertice  $p_k^*$  (l'ultimo) è il candidato  $c_r \in C$  tale che  $M[k, r] = \min_{1 \leq j \leq m} M[k, j]$ .
- Per ogni  $i = k - 1, \dots, 1$ , sia  $c_t$  il candidato selezionato per la posizione  $i + 1$  nella soluzione, quindi  $r = V[i + 1, t]$  è l'indice del candidato per la posizione  $i$ , cioè abbiamo che  $p_i^* = c_r$ .

La soluzione  $P^* = \langle p_1^*, \dots, p_k^* \rangle$  costituisce l'output di questo modulo (che coincide con l'output di SMART).

Prima di passare ai prossimi capitoli, analizziamo il Costo computazionale dell'algoritmo di programmazione dinamica adottato da SMART:

- Tempo: ci sono  $O(m \cdot k)$  sottoproblemi e il costo per calcolare la soluzione ottimale per ogni sottoproblema è  $O(m + f)$  (dove  $f$  è la dimensione del vettore di caratteristiche della canzone). Quindi, il costo per calcolare tutti i valori  $M[i, j]$  è  $O(m^2 \cdot k)$ , assumendo ragionevolmente che  $f \in O(m)$ . Calcolare  $OPT$  e determinare la soluzione effettiva aggiunge un fattore additivo  $O(m + k)$ . In conclusione, il costo computazionale temporale per trovare la soluzione ottimale è dominato da  $O(m^2 \cdot k)$ .
- Spazio: abbiamo bisogno di memorizzare due matrici  $k \times m$  ( $M$  e  $V$ ), quindi il costo computazionale dello spazio è  $O(k \cdot m)$ .

## 3 Implementazione

Questo capitolo è dedicato a fornire i dettagli relativi all'implementazione del metodo SMART discusso nel Capitolo 1. Opereremo analogamente a quanto fatto nel capitolo relativo allo stato dell'arte: prima di tutto, forniremo una descrizione generale della struttura del programma, poi, analizzaremos i dettagli implementativi.

### 3.1 Introduzione

L'implementazione è avvenuta utilizzando "Python", un linguaggio di programmazione ad alto livello e orientato agli oggetti, dalla sintassi semplice e molto flessibile.

I principali file coinvolti nel progetto sono i seguenti:

- `listening_history_manager.py`: file Python che filtra la cronologia di ascolto e che gestisce l'interazione con la API di Spotify;
- `custom_cache_handler.py`: file Python che gestisce la lettura e la scrittura dei token di autorizzazione di Spotify memorizzati sottoforma di file JSON sul disco;
- `step1.py`: file Python che implementa le funzioni relative alla prima fase del metodo;
- `step2.py`: file Python che implementa le funzioni relative alla seconda fase del metodo;
- `other_methods.py`: file Python che implementa le funzioni relative ai metodi sperimentali da confrontare con SMART;
- `evaluation.py`: file Python che implementa i risultati ottenuti durante la sperimentazione per produrre grafici e fogli di calcolo;
- `playlist_generator.py`: file Python che, utilizzando tutti i precedenti moduli, genera una playlist per ciascun metodo e mette a confronto le playlist generate;
- `requirements.txt`: file testuale contenente le librerie esterne da installare per il corretto funzionamento del software;
- `README.md`: file testuale contenente le istruzioni per l'installazione del software.

Nelle prossime sezioni discuteremo nei dettagli il contenuto e il funzionamento di ciascuno di questi file.

## 3.2 Interazione con Spotify

La API web di Spotify consente la creazione di applicazioni in grado di interagire con l'omonimo servizio di streaming, permettendo il recupero dei metadati dei contenuti, l'ottenimento di raccomandazioni, la creazione e la gestione di playlist e il controllo della riproduzione. Tutte le funzioni che servono per interagire con l'API di Spotify sono implementate all'interno del modulo `listening_history_manager`.

### 3.2.1 Memorizzazione delle credenziali

Per interagire con l'API di Spotify[29] è necessario registrare una o più applicazioni presso la Dashboard degli sviluppatori, accessibile al link [developer.spotify.com/dashboard](https://developer.spotify.com/dashboard). Ciascuna app registrata presso la dashboard possiede tre informazioni necessarie per l'utilizzo della API:

- Client\_id: è un identificatore univoco assegnato a ciascuna applicazione quando viene registrata sulla piattaforma di sviluppo di Spotify. Viene utilizzato per identificare in modo univoco un'app quando vengono effettuate le richieste API a Spotify. È essenziale per l'autenticazione dell'applicazione e per ottenere l'accesso alle risorse di Spotify.
- Client\_secret: è un token segreto che viene assegnato a ciascuna applicazione durante la registrazione. Viene utilizzato per l'autenticazione dell'applicazione durante il processo di autenticazione OAuth 2.0.
- Redirect\_uri: è l'URL a cui Spotify reindirizzerà l'utente dopo che ha autorizzato l'applicazione. Dopo che l'utente ha concesso l'autorizzazione, Spotify reindirizza l'utente all'URL specificato e include i parametri necessari (come il codice di autorizzazione) nell'URL. Questo è un passaggio cruciale nel processo di autorizzazione OAuth 2.0 e consente all'applicazione di ottenere il token di accesso per l'utente.

Nella nostra implementazione, manteniamo le credenziali di tutte le app registrate all'interno del dizionario `credentials_dicts`.

```
1 import spotipy
2 from spotipy.oauth2 import SpotifyOAuth
3 from custom_cache_handler import CustomCacheFileHandler
4 credentials_dicts = {
5     'email@domain.com': [
6         {
7             'client_id': '',
8             'client_secret': '',
9             'redirect_uri': ''
10        }
11    ]
12 }
```

Questo codice utilizza la libreria `Spotify`: questa libreria fornisce un'interfaccia Python per l'API Web di Spotify, consentendo agli sviluppatori di recuperare dati, come le canzoni recentemente riprodotte dall'utente nel codice fornito. Tramite il modulo `SpotifyOAuth` di Spotify, l'utente sarà in grado di autenticarsi alla nostra applicazione tramite il protocollo di autenticazione OAuth2 utilizzando le credenziali del suo account Spotify. È importante notare che per eseguire questo codice, è necessario riempire il dizionario `credential_dicts` con delle credenziali valide, ottenibili registrando uno o più account Spotify presso la Developer Dashboard e creando, per ogni account, una o più app (al massimo cinque per account). Il dizionario `credential_dicts` ha come chiavi le e-mail di ciascun account Spotify inserito dall'utente. Ad ogni e-mail si associa la lista di credenziali delle app registrate presso la Developer Dashboard per quell'account. Le credenziali di ciascuna app sono memorizzate sottoforma di un dizionario, le cui chiavi sono `client_id`, `client_secret` e `redirect_uri`. I valori `client_id`, `client_secret` vengono generati automaticamente da Spotify al momento della creazione di un'app, mentre il valore `redirect_uri` deve essere specificato dall'utente. Le credenziali da utilizzare per l'autenticazione verranno scelte casualmente dal dizionario `credential_dicts` più volte durante l'esecuzione del programma. In questo modo, a patto che `credential_dicts` contenga sufficienti credenziali, è possibile aggirare i limiti stringenti posti da Spotify relativi al numero massimo di richieste alla API effettuabili in un determinato arco di tempo.

### 3.2.2 Scelta delle credenziali

La variabile globale `credential_dicts`, contenente delle credenziali valide, verrà utilizzata dalla funzione `change_credentials()` per scegliere casualmente un'app da utilizzare per l'autenticazione. In questo modo, le richieste alla API di Spotify verranno distribuite equamente tra le varie applicazioni registrate.

```
1 import random
2 def change_credentials():
3     scope = "playlist-modify-private user-read-recently-played"
4     while True:
5         try:
6             account, credentials =
7                 random.choice(list(credentials_dicts.items()))
8             random_credentials = random.choice(credentials)
9             account_index =
10                list(credentials_dicts.keys()).index(account)
11             credentials_index =
12                credentials.index(random_credentials)
13             credentials_index_complete = account_index *
14                 len(credentials) + credentials_index
15             custom_cache_handler =
16                 CustomCacheFileHandler(credentials_index =
17                     credentials_index_complete)
18             auth_manager = SpotifyOAuth(**random_credentials,
19                                         scope=scope,
20                                         cache_handler=custom_cache_handler)
21             spotify = spotipy.Spotify(auth_manager=auth_manager)
22             user_id = spotify.current_user()['id']
23         except spotipy.oauth2.SpotifyOauthError:
24             path = f"cache/.cache{credentials_index_complete}"
25             if os.path.exists(path):
26                 os.remove(path)
27             continue
28         except spotipy.exceptions.SpotifyException:
29             continue
30         else:
31             break
32
33     return spotify
```

Il funzionamento di `change_credentials()` è il seguente:

- Si importa la libreria `random`, utilizzata per scegliere casualmente le credenziali.

- Si definisce lo scope `scope`, che consente al programma di creare e di modificare le playlist private e di leggere l’elenco delle riproduzioni recenti dell’utente.
- Si entra in un ciclo infinito che termina una volta scelte le prime credenziali “ valide ”.
- Si utilizza il modulo `choice()` della libreria `random` per selezionare casualmente una coppia (`account, credentials`) dalla lista di coppie ottenuta da `credentials_dicts.items()`. In questo modo si sceglie casualmente un `account` `account` e la relativa lista `credentials` di credenziali da `credentials_dicts`.
- Si utilizza nuovamente `choice()` per scegliere casualmente il dizionario di credenziali `random_credentials` partendo dalla lista `credentials`.
- Si ricava l’indice `credentials_index_complete` all’interno del dizionario `credentials_dicts` delle credenziali scelte casualmente `random_credentials`. Tale indice viene passato al costruttore della classe `CustomCacheFileHandler` (implementata all’interno di `custom_cache_handler.py`, che si occupa di creare il relativo file `.cache`, evitando la necessità di autenticarsi ogni volta che si utilizza l’applicazione scelta).
- L’oggetto `custom_cache_handler`, il dizionario `random_credentials` e lo scope `scope` vengono passati al costruttore di `SpotifyOAuth` per creare l’oggetto `auth_manager`, che viene utilizzato per ottenere e gestire i token di accesso OAuth necessari per effettuare chiamate API a Spotify a nome dell’utente.
- Si passa `auth_manager` al costruttore di `spotipy.Spotify`, e si crea l’oggetto `spotify`, che verrà utilizzato per interagire con la API di Spotify, permettendo di richiedere le caratteristiche audio dei brani, le raccomandazioni di brani simili, gli ascolti recenti, e creando le playlist.
- La linea di codice `user_id = spotify.current_user()['id']` serve per assicurarsi che le credenziali scelte casualmente siano valide. In caso contrario, questa istruzione solleverà un’eccezione (`SpotifyOAuthError` o `SpotifyException`).
- Si esce dal ciclo alla prima iterazione per cui la chiamata a `current_user()` sull’oggetto `spotify` non ha sollevato eccezioni.
- Si restituisce l’oggetto `spotify`.

### 3.2.3 Recupero delle raccomandazioni

La funzione `get_recommendations` utilizza la API di Spotify per restituire canzoni simili (raccomandazioni) a quelle che vengono fornite in input tramite l’argomento `tracks`. Ecco il codice della funzione:

```

1 def get_recommendations(tracks, spotify=change_credentials(),
2                         limit=100, kwargs=None):
3     while True:
4         try:
5             ret = spotify.recommendations(seed_tracks=tracks,
6                                             limit=limit, kwargs=kwargs)
7         except Exception as e:
8             change_credentials()
9         else:
10            break
11 return ret

```

Il funzionamento è molto semplice: si entra in un ciclo while infinito in cui, tramite un blocco try-except, si tenta di invocare il metodo `recommendations` sull'oggetto `spotify`. Se l'istruzione va a buon fine, si esce forzatamente dal ciclo e si restituisce il relativo output `ret`. Altrimenti, si scelgono nuove credenziali tramite la funzione `change_credentials()`. La funzione `get_recommendations` accetta i parametri opzionali `spotify`, nel caso si voglia specificare delle credenziali specifiche, e `kwargs`, nel caso si vogliano specificare parametri aggiuntivi relativi alle caratteristiche audio che devono essere soddisfatte dalle canzoni restituite dalla API di Spotify. Il parametro opzionale `kwargs` verrà utilizzato durante la generazione dei song-set tramite le due tecniche euristiche. Per maggiori dettagli in merito all'output della chiamata API sulle raccomandazioni musicali è possibile consultare la documentazione ufficiale presso il link <https://developer.spotify.com/documentation/web-api/reference/get-recommendations>

### 3.2.4 Recupero delle caratteristiche audio dei brani

Per recuperare le caratteristiche audio dei brani utilizziamo due funzioni: `get_features()` e `check_features()`. La funzione `check_features()` prende in input una lista di tracce `tracks` e la lista dei nomi delle caratteristiche audio `feature_names`. La funzione restituisce due oggetti: `all_features` e `feature_list`. La funzione `get_features()` verifica che tutte le tracce in `tracks` contengano tutte le 12 caratteristiche audio. In tal caso, `all_features` sarà impostato a `True` e la lista `all_features` conterrà le caratteristiche audio di ciascun brano. In questo modo, si evita di effettuare chiamate inutili alla API di Spotify preservando il più possibile il "rate limit" imposto.

```

1  def check_features(tracks, feature_names):
2      all_features = True
3      feature_list = []
4
5      for track in tracks:
6          features_track = {}
7          for feature_name in feature_names:
8              if feature_name == 'speechiness':
9                  features_track['speechiness'] =
10                     track.get(feature_name, None)
11                  if not features_track['speechiness']:
12                      all_features = False
13                      break
14                  elif feature_name == 'TrackID':
15                      features_track['id'] = track.get(feature_name,
16                         None)
17                      if not features_track['id']:
18                          all_features = False
19                          break
20
21                  else:
22                      features_track[feature_name.lower()] =
23                         track.get(feature_name, None)
24                      if not features_track[feature_name.lower()]:
25                          all_features = False
26                          break
27
28                  if not all_features:
29                      feature_list = []
30                      break
31                  else:
32                      feature_list.append(features_track)
33
34      return all_features, feature_list

```

1. Inizializzazione delle variabili `all_features` e `feature_list` come `True` e lista vuota, rispettivamente.
2. Iterazione attraverso ogni elemento `track` nella lista `tracks`.
3. Per ogni `track`, viene inizializzato un dizionario vuoto `features_track`.
4. Iterazione attraverso ogni `feature_name` nella lista `feature_names`.

5. Se il `feature_name` è "speechness", il valore corrispondente viene inserito nel dizionario `features_track` come "speechiness". Se non è presente, viene impostato a `None`. Se non è presente alcun valore per "speechiness", `all_features` diventa `False` e si interrompe il ciclo.
6. Se il `feature_name` è "TrackID", il valore corrispondente viene inserito nel dizionario `features_track` come "id". Se non è presente, viene impostato a `None`. Se non è presente alcun valore per "id", `all_features` diventa `False` e si interrompe il ciclo.
7. Altrimenti, il valore corrispondente viene inserito nel dizionario `features_track` con chiave in minuscolo. Se non è presente, viene impostato a `None`. Se non è presente alcun valore per la chiave, `all_features` diventa `False` e si interrompe il ciclo.
8. Se `all_features` è ancora `True` dopo aver controllato tutte le feature per la `track`, il dizionario `features_track` viene aggiunto alla lista `feature_list`.
9. Se `all_features` è diventato `False` durante il controllo delle feature per una `track`, la lista `feature_list` viene reimpostata a vuota e il ciclo viene interrotto.
10. Alla fine delle iterazioni su tutte le `tracks`, la funzione restituisce una tupla contenente il valore booleano `all_features` e la lista `feature_list`.

Ecco l'implementazione di `get_features()`:

```

1  def get_features(tracks, spotify):
2  if isinstance(tracks[0], str):
3      try:
4          with open("data/all_features.bin", "rb") as file:
5              all_features = pickle.load(file)
6              feature_list = []
7              for track in tracks:
8                  for feature in all_features:
9                      if feature['id'] == track:
10                          feature_list.append(feature)
11                          break
12                  if len(feature_list) == len(tracks):
13                      return feature_list
14  except Exception as e:
15      pass
16  ids = tracks

```

1. Verifica se il primo elemento di `tracks` è una stringa.

2. In tal caso, prova ad aprire un file binario `all_features.bin` per caricare le caratteristiche audio precedentemente salvate.
3. Cerca le caratteristiche corrispondenti per ogni traccia in `tracks` e le aggiunge a `feature_list`.
4. Se tutte le caratteristiche richieste sono trovate, restituisce `feature_list`.
5. In caso di eccezione, continua con il resto del codice.
6. Assegna `tracks` a `ids`.

```

1  else:
2      all_features, feature_list = check_features(tracks,
3          feature_names_1)
4      if not all_features:
5          if tracks[0].get('track', None):
6              ids = [track['track']['id'] for track in tracks]
7          elif tracks[0].get('TrackID', None):
8              ids = [track['TrackID'] for track in tracks]
9          else:
10             ids = [track['id'] for track in tracks]
11     else:
12         feature_list = [{key: track[key] for key in
13                         keys_ordering} for track in feature_list]
14     return feature_list

```

Se il primo elemento di `tracks` non è una stringa, vengono eseguite le seguenti operazioni:

7. Se le caratteristiche non sono tutte disponibili, estrae gli ID delle tracce dai dizionari in `tracks`.
8. Se tutte le caratteristiche sono disponibili, formatta `feature_list` e la restituisce.

```

1  if len(ids) > 50:
2      sublists = [ids[i:i+50] for i in range(0, len(ids), 50)]
3      features = []
4      for sublist in sublists:
5          while True:
6              try:
7                  features =
8                      spotify.audio_features(tracks=sublist)
9              except Exception as e:
10                 spotify = change_credentials()
11             else:
12                 break
13             features.extend(features)

```

Se, arrivati a questo punto, `feaure_list` non è stato restituito e il numero di ID in `tracks` è maggiore di 50, si opera come segue:

9. Se il numero di ID è maggiore di 50, suddivide gli ID in sottoliste di massimo 50 elementi (massimo numero di ID di canzoni accettate dalla API di Spotify).
10. Per ogni sottolista, tenta di recuperare le caratteristiche audio tramite il metodo `audio_features` di `spotipy`.
11. In caso di eccezione, cambia le credenziali e riprova.
12. Altrimenti, continua con la sottolista successiva.

```

1  else:
2      while True:
3          try:
4              features = spotify.audio_features(tracks=ids)
5          except Exception as e:
6              spotify = change_credentials()
7          else:
8              break

```

Se, arrivati a questo punto, `feaure_list` non è stato restituito e il numero di ID in `tracks` è minore o uguale a 50, si opera come segue:

13. Si tenta di recuperare le caratteristiche audio tramite il metodo `audio_features`.
14. In caso di eccezione, cambia le credenziali e riprova.
15. Altrimenti, esce dal ciclo.

```

1  feature_list = []
2  for feature in features:
3      if feature:
4          track_features = feature.get('id', None)
5          if track_features:
6              final_features = dict(filter(lambda item:
7                  item[0] not in feature_names_to_remove,
8                  feature.items()))
9              feature_list.append(final_features)
10
11 try:
12     with open("data/all_features.bin", "rb") as file:
13         all_stored_features = pickle.load(file)
14         all_stored_features.extend(feature_list)
15
16     with open("data/all_features.bin", "wb") as file:
17         pickle.dump(list(all_stored_features), file)
18 except (EOFError, FileNotFoundError):
19     pass
20 return feature_list

```

16. Crea una lista `feature_list` per contenere le caratteristiche finali.
17. Per ogni caratteristica in `features`, se esiste un ID valido, filtra le caratteristiche da rimuovere e aggiunge il risultato a `feature_list`.
18. Tenta di aprire il file `all_features.bin`, aggiunge le nuove caratteristiche a quelle esistenti e salva tutto nel file.
19. In caso di errore di fine file o file non trovato, continua senza fare nulla.
20. Restituisce `feature_list`.

Facendo in questo modo, riusciamo a minimizzare la quantità di chiamate alla API di Spotify, velocizzando il programma e riducendo il rischio di raggiungere il limite massimo di chiamate alla API imposto da Spotify.

### 3.3 Recupero degli ascolti recenti

Il metodo SMART è basato sulla elaborazione della cronologia di ascolto degli utenti. Per calcolare e filtrare la cronologia per periodi, è necessario recuperare gli ascolti recenti dell'utente. A questo proposito, distinguiamo due differenti casistiche:

- L'utente dispone del file `StreamingHistory.json`: questo file, ottenibile soltanto tramite richiesta (funzionalità Request Data di Spotify), contiene, in formato json, la cronologia di ascolto dell'utente relativa ai tre mesi antecedenti alla richiesta;
- L'utente non è in possesso del file `StreamingHistory.json`. In questo caso sarà necessario recuperare gli ascolti recenti tramite una chiamata specifica alla API di Spotify.

Se l'utente dispone del file `StreamingHistory.json`, operiamo in questo modo:

1. Il file `StreamingHistory.json` viene passato in input alla funzione `listening_history_file_to_dict()`: questa funzione memorizza in una lista di dizionari `data` tutti gli elementi contenuti nella cronologia di ascolto.
2. La lista `data` viene passata alla funzione `filter_listening_history_file()` che restituirà l'oggetto `recently_played_songs`, ossia una lista di canzoni con le seguenti caratteristiche:
  - (a) Le canzoni sono ordinate in senso decrescente di timestamp di ascolto.
  - (b) Le canzoni ascoltate più volte consecutivamente vengono memorizzate una volta sola. In altre parole, si eliminano gli eventuali duplicati consecutivi di ciascuna canzone.
  - (c) Per essere presente, ciascuna canzone deve essere stata ascoltata per almeno il 50% della sua durata.

Ecco il codice della funzione `listening_history_file_to_dict()`:

```

1 import csv
2 def listening_history_file_to_dict(history_file):
3     data = []
4     try:
5         with open(history_file, 'r') as file:
6             reader = csv.reader(file, delimiter=',')
7             keys = next(reader)
8             for row in reader:
9                 data.append(dict(zip(keys, row)))
10    except Exception:
11        data = []
12    else:
13        if data:
14            data = filter_listening_history_file(data)
15    return data

```

Il funzionamento di `listening_history_file_to_dict()` è il seguente:

- Si importa la libreria `csv`, utilizzata per estrarre i dati dal file in input.
- Si inizializza `data` a una lista vuota.
- Si entra in un blocco try-catch nel quale si tenta di aprire in lettura il file `history_file` passato in input.
- Si crea l'oggetto iterabile `reader`, ottenuto tramite l'omonimo modulo appartenente alla libreria `csv` specificando il carattere ; come delimitatore.
- Si memorizzano in `keys` le chiavi (colonne) assegnate a ciascuna canzone presente nella cronologia. Queste chiavi, contenute nella prima riga del file, vengono estratte tramite la funzione `next()` invocata sull'iterabile `reader`.
- Si itera sulle righe `row` di `reader`, a partire dalla seconda in poi.
- Si aggiunge a `data` il dizionario che ha come chiavi `keys` e come valori `row`.
- Se in una delle precedenti istruzioni viene sollevata un'eccezione si assegna a `data`, nel caso fosse stata modificata, una lista vuota.
- Altrimenti, si verifica che `data` sia stata effettivamente riempita, e, in tal caso, si assegna a `data` il risultato della funzione `filter_listening_history(data)`.
- Si restituisce `data`.

Ecco il codice della funzione `filter_listening_history_file()`:

```
1 from datetime import datetime, timedelta
2 import math
3 def filter_listening_history_file(data):
4     valid_items = []
5     for item in data:
6         try:
7             item['endTime'] = datetime.strptime(item['endTime'],
8                                               "%Y-%m-%d %H:%M")
8             item['endTime'] = item['endTime'] -
9                 timedelta(milliseconds=int(item['msPlayed']))
10            valid_items.append(item)
11        except ValueError:
12            data.remove(item)
13    recently_played_songs = sorted(valid_items, key=lambda x:
14        x['endTime'], reverse=True)
15    unique_songs = [recently_played_songs[0]]
16
17    for i in range(1, len(recently_played_songs)):
18        current_song = recently_played_songs[i]
19        previous_song = recently_played_songs[i - 1]
20        listen_percentage = math.floor(100 *
21            (int(current_song['msPlayed']) /
22             int(current_song['msDuration']))))
23        if current_song['TrackID'] != previous_song['TrackID']
24            and listen_percentage >= 50:
25                unique_songs.append(current_song)
26    recently_played_songs = unique_songs
27    return recently_played_songs
```

Vediamo nel dettaglio come funziona `filter_listening_history_file()`:

- Si importano i moduli `datetime` e `timedelta` dalla libreria `datetime` e si importa la libreria `math`.
- Si inizializza `valid_items` a una lista vuota.
- Si itera sugli elementi `item` contenuti nell'oggetto `data` ricevuto in input.
- Si entra in un blocco try-catch in cui si tenta di sovrascrivere il valore `item['endTime']`, ossia il timestamp che rappresenta il momento in cui l'utente ha smesso di ascoltare la canzone rappresentata da `item`. Nella fattispecie, si utilizza il modulo `strptime` della libreria `datetime` per convertire il timestamp di ascolto da un oggetto di tipo stringa a uno di tipo `datetime`.

- Per evitare ogni volta di dover calcolare il timestamp di inizio di ascolto di una canzone, assegniamo ad `item['endTime']` il risultato della sottrazione tra `item['endTime']` e il tempo di ascolto della canzone in millisecondi, ottenuto tramite il modulo `timedelta` di `datetime`.
- Se l'operazione precedente va a buon fine, si aggiunge `item` a `valid_items`, altrimenti si rimuove `item` da `data`.
- Successivamente, si ordina `valid_items` in senso decrescente di timestamp di ascolto tramite una funzione lambda e si memorizza il risultato in `recently_played_songs`.
- Si inizializza `unique_songs` ad una lista contentente soltanto il primo elemento di `recently_played_songs`. Da qui in poi la funzione si occupa di filtrare le canzoni presenti nella cronologia.
- Si itera per valori di `i` interi nell'intervallo da 1 (incluso) alla lunghezza di `recently_played_songs` (esclusa).
- Si memorizza, rispettivamente in `current_songs` e in `previous_songs`, l'elemento di indice `i` e di indice `i-1` all'interno di `recently_played_songs`.
- Si calcola la percentuale di ascolto `listen_percentage` della canzone corrente `current_song`. Per farlo, si divide `current_song['msPlayed']` per `current_song['msDuration']` (tempo di ascolto diviso per durata totale della canzone corrente espressi in millisecondi). Il risultato viene moltiplicato per 100 e troncato all'intero inferiore tramite il metodo `floor` della libreria `math`.
- Si verifica che l'ID della traccia corrente `current_song['TrackID']` sia diverso all'ID della traccia precedente `previous_song['TrackID']` e che la percentuale di ascolto della canzone corrente `listen_percentage` sia almeno pari a 50.
- Se entrambe le condizioni sono soddisfatte, si aggiunge a `unique_songs` la canzone corrente `current_song`.
- Si assegna `unique_songs` a `recently_played_songs`.
- Si restituisce `recently_played_songs`.

Se l'utente non dispone del file `StreamingHistory.json`, cerchiamo di recuperare gli ascolti recenti tramite la funzione `get_recently_played_songs()`. Questa funzione prende in input l'oggetto `spotify` precedentemente creato tramite `change_credentials` e lo utilizza per effettuare la chiamata `current_user_recently_played()`. Ecco il codice di `get_recently_played_songs()`:

```

1 def get_recently_played_songs(spotify):
2     try:
3         with open("data/recently_played_songs.bin", "rb") as
4             file:
5                 recently_played_songs = pickle.load(file)
6     except FileNotFoundError:
7         recently_played_songs = {'items': []}
8     existing_song_keys = set()
9     for item in recently_played_songs['items']:
10         timestamp = datetime.strptime(item['played_at'],
11                                         "%Y-%m-%dT%H:%M:%S.%fZ")
12         timestamp = pytz.utc.localize(timestamp)
13         timestamp = timestamp.astimezone(tzlocal.get_localzone())
14         item['played_at'] = timestamp
15         key = (item['track']['id'], item['played_at'])
16         existing_song_keys.add(key)
17     new_songs = [song for song in
18                  spotify.current_user_recently_played()['items'] if
19                  (song['track']['id'], song['played_at']) not in
20                  existing_song_keys]
21     recently_played_songs['items'].extend(new_songs)
22     with open("data/recently_played_songs.bin", "wb") as file:
23         pickle.dump(recently_played_songs, file)
24     recently_played_songs =
25         sorted(recently_played_songs['items'], key=lambda x:
26                x['played_at'], reverse=True)
27     unique_songs = [recently_played_songs[0]]
28     for i in range(1, len(recently_played_songs)):
29         current_song = recently_played_songs[i]
30         previous_song = recently_played_songs[i - 1]
31         current_time =
32             datetime.fromisoformat(current_song['played_at'])
33         previous_time =
34             datetime.fromisoformat(previous_song['played_at'])
35         time_difference = (previous_time -
36                             current_time).total_seconds()
37         if current_song['track']['id'] !=
38             previous_song['track']['id'] and time_difference >=
39             30:
40             unique_songs.append(current_song)
41     recently_played_songs = unique_songs
42     return recently_played_songs

```

Il metodo `current_user_recently_played()` restituisce soltanto le ultime 50 canzoni nella cronologia. Per questo motivo, le canzoni verranno memorizzate localmente e aggiornate a mano a mano che l'utente ascolta nuova musica, costruendo la cronologia di ascolto incrementalmente con il passare del tempo. Ecco il funzionamento di `get_recently_played_songs()` nel dettaglio:

1. Si importano le librerie `datetime`, `pickle`, `pytz` e `tzlocal` necessarie per l'esecuzione della funzione.
2. Si tenta di caricare, tramite il modulo `pickle`, la lista di brani musicali recentemente riprodotti dall'utente memorizzati all'interno del file binario "`recently_played_songs.bin`".
3. Se è la prima volta che l'utente utilizza il software, oppure se il file è stato cancellato, il blocco `except` catturerà l'eccezione `FileNotFoundException`. In questo caso, il codice analizza un dizionario vuoto, la cui unica chiave `items` è inizializzata ad una lista vuota. L'inizializzazione della variabile `recently_played_songs` effettuata in questo modo risulta necessaria: infatti, l'output della chiamata alla API di Spotify relativa alle canzoni recentemente ascoltate restituisce un dizionario. Le chiavi di questo dizionario sono "href", "limit", "next", "cursors", "total" e `items`. È proprio quest'ultima chiave quella di nostro interesse, in quanto contiene le informazioni che utilizzeremo per generare le playlist personalizzate.
4. Si cercano nuovi brani riprodotti di recente utilizzando l'API di Spotify e si aggiungono alla lista solo se non sono già presenti: per prima cosa, viene creato un insieme di chiavi `existing_song_keys`. Ciascun elemento di tale insieme è una coppia (`ID_canzone`, `orario_di_riproduzione`). L'utilizzo di coppie (`ID_canzone`, `orario_di_riproduzione`) è motivato dal fatto che le chiavi di un dizionario devono essere univoche, e un utente potrebbe aver ascoltato più volte la stessa canzone (anche in periodi diversi). In questo modo, nessuna canzone viene esclusa dagli ascolti recenti.
5. Si inizializza `existing_song_keys` ad un insieme vuoto.
6. Si itera su ciascun `item` presente in `recently_played_songs['items']`.
7. Si sovrascrive il timestamp di ascolto `item['played_at']` convertendolo in un oggetto `datetime` (tramite `strptime()`) e impostandone il fuso orario a quello UTC da cui è stata lanciata l'applicazione. Questa operazione è necessaria in quanto il timestamp di ascolto delle canzoni restituito dalla API di Spotify è di default in UTC+0 (GMT), ma potrebbe differire dal fuso orario effettivo nel quale si trovava l'utente quando ha ascoltato musica. Questa operazione non viene effettuata nel caso in cui l'utente dispone del file `StreamingHistory.json` perché i timestamp all'interno di tale file sono già provvisti del fuso orario corretto.

8. Si crea la lista `new_songs`. Tale lista conterrà soltanto quelle canzoni restituite dalla chiamata `currente_user_recently_played()` la cui coppia (`ID_canzone`, `orario_di_riproduzione`) non è già presente all'interno dell'insieme `existing_song_keys`.
9. Gli elementi di `new_songs` vengono aggiunti alla fine della lista `recently_played_songs['items']`.
10. Si sovrascrive il file binario "recently\_played\_songs.bin" con la lista aggiornata `recently_played_songs`.
11. Le canzoni in `recently_played_songs` vengono ordinate in senso decrescente di timestamp di ascolto. Questo viene effettuato tramite una list comprehension che combina l'utilizzo del metodo built-in `sorted` e una funzione lambda.
12. Si inizializza la lista `unique_songs` con un solo elemento (l'ultima canzone ascoltata `recently_played_songs[0]`). Questa lista conterrà tutte le canzoni non duplicate che sono state ascoltate per almeno 30 secondi e manterrà l'ordinamento delle canzoni.
13. Si itera, per valori interi crescenti di `i`, nell'intervallo che va da 1 al numero di canzoni presenti in `recently_played_songs`.
14. Si recuperano la canzone corrente (`recently_played_songs[i]`) e quella precedente (`recently_played_songs[i-1]`), memorizzandole rispettivamente nelle variabili `current_song` e `previous_song`.
15. Si recuperano i timestamp di ascolto `current_time` e `previous_time` relativi, rispettivamente, alle canzoni `current_song` e `previous_song`.
16. Si calcola il tempo `time_difference` sottraendo i due timestamp convertiti in `datetime`. Si recuperano i secondi `total_seconds()` da `time_difference`.
17. Se le canzoni `previous_song` e `current_song` sono state riprodotte a più di 30 secondi di distanza l'una dall'altra, e se il loro id non coincide, si aggiunge `current_song` alla lista `unique_songs`.
18. Alla fine del ciclo, si assegna `unique_songs` a `recently_played_songs`.
19. Si restituisce `recently_played_songs`.

### 3.4 Calcolo dei periodi

Una volta stabilita l'interazione con Spotify, recuperato e filtrato gli ascolti recenti, possiamo occuparci di calcolare i periodi, introdotti nella sezione 2.2. Si noti che, da questa sezione fino alla Sezione 3.8 inclusa, tutte le funzioni illustrate fanno parte del modulo `step1.py`. Per calcolare i periodi, utilizziamo la funzione `compute_periods()`:

```
1 def compute_periods(data, prefix_name, spotify):
2     periods = {}
3     periods_file_path = os.path.join(data_directory, prefix_name
4                                     + periods_suffix)
5     try:
6         with open(periods_file_path, "rb") as file:
7             periods = pickle.load(file)
8     except Exception as e:
9         pass
10    else:
11        return periods
12    if data:
13        songs = data
14        timestamp_key = 'endTime'
15    else:
16        songs = compute_recently_played_songs(spotify)
17        timestamp_key = 'played_at'
18    for track in songs:
19        timestamp = track[timestamp_key].replace(minute=0,
20                                                second=0, microsecond=0).replace(tzinfo=None)
21        if not periods.get(timestamp, None):
22            periods[timestamp] = []
23        periods[timestamp].append(track)
24    if not periods:
25        sys.exit()
26    else:
27        with open(periods_file_path, "wb") as file:
28            pickle.dump(periods, file)
29    return periods
```

La funzione prende in input la lista `data`, la stringa `prefix_name` e l'oggetto `spotify`. Se l'utente è in possesso del file `StreamingHistory.json`, la lista `data` coinciderà con l'output di `csv_to_dict()`, altrimenti sarà una lista vuota. La stringa `prefix_name`, che rappresenta un identificatore univoco per il file `StreamingHistory.json`, risulta particolarmente comoda in fase di sperimentazione, ed è calcolata dalla funzione `compute_prefix_name()`, della quale non forniremo l'implementazione.

1. Si inizializza `periods` a un dizionario vuoto.

2. Si definisce `periods_file_path` come la stringa contenente il percorso del file binario in cui sono memorizzati i periodi.
3. Si entra in un blocco try-catch nel quale si tenta di aprire in lettura il file relativo al percorso appena calcolato.
4. Si assegna a `periods` l'output del modulo `load` della libreria `pickle` sul file binario.
5. Se, durante il caricamento o l'apertura del file, viene sollevata un'eccezione, i periodi verranno calcolati, altrimenti, si restituirà il dizionario `periods`.
6. Se l'utente è in possesso del file `StreamingHistory.json`, si definisce `songs` come `data` e `timestamp_key` come `'endTime'`, altrimenti, `songs` costituirà l'output della funzione `get_recently_played_songs()` e `timestamp_key` sarà definito come `played_at`.
7. Si itera sulle tracce `track` all'interno della lista `songs`.
8. Si sostituiscono dal timestamp `track[timestamp_key]` i minuti, i secondi e i microsecondi con zeri, utilizzando il metodo `replace` di `datetime` e si rimuove l'informazione sul fuso orario. In pratica, questa operazione mantiene soltanto l'anno, il mese, il giorno e l'ora all'interno del `timestamp` di ascolto della traccia `track`.
9. Nel caso in cui il `timestamp` non faccia già parte del dizionario `periods`, si inizializza `periods[timestamp]` a una lista vuota.
10. Si aggiunge `track` alla lista `periods[timestamp]`.
11. Alla fine dell'esecuzione della funzione, se `periods` è vuoto, si esce forzatamente tramite il modulo `exit()` della libreria `sys`.
12. Altrimenti, si restituisce `periods` dopo averlo caricato, tramite `pickle`, sul file aperto in scrittura denotato dal percorso `periods_file_path`.

Concludendo, la funzione `compute_periods()` restituisce il dizionario `periods`. Tale dizionario ha come chiavi i timestamp di ascolto (fino all'ora) e come valori le liste di canzoni ascoltate nel relativo periodo. Si noti che il dizionario `periods` non rappresenta ancora la cronologia filtrata, ma risulta necessario per calcolarla (insieme alle abitudini di ascolto), in quanto contiene, tra le altre informazioni, gli ID delle canzoni che verranno utilizzati in un'altra chiamata API per recuperare le caratteristiche audio delle canzoni.

### 3.5 Calcolo della cronologia di ascolto

In questa sezione, discutiamo di come è stata calcolata la cronologia di ascolto. A questo proposito, abbiamo creato la funzione `compute_listening_history()`, che prende in input `periods` e restituisce il dizionario `history`, che rappresenta la cronologia filtrata.

```
1 def compute_listening_history(periods):
2     history = {}
3     history_file_path = os.path.join(data_directory, prefix_name
4                                     + listening_history_suffix)
5     try:
6         with open(history_file_path, "rb") as file:
7             history = pickle.load(file)
8     except Exception:
9         pass
10    else:
11        return history
12    for period, tracks in periods.items():
13        if isinstance(period, datetime):
14            hour = period.hour
15        else:
16            hour = period
17        if not history.get(hour, None):
18            history[hour] = []
19        if isinstance(period, datetime):
20            track_ids = [track['track']['id'] for track in
21                          tracks]
22        else:
23            track_ids = [track['id'] for track in tracks]
24        features = spotify.audio_features(tracks=track_ids)
25        for feature in features:
26            if feature:
27                track_features = feature.get('id', None)
28                if track_features:
29                    final_features = dict(filter(lambda item:
30                        item[0] not in feature_names_to_remove,
31                        feature.items()))
32                    history[hour].append(final_features)
33    if not history:
34        sys.exit()
35    else:
36        with open(history_file_path, "wb") as file:
37            pickle.dump(history, file)
38    return history
```

La funzione opera in questo modo:

1. Si inizializza `history` a un dizionario vuoto.
2. Si definisce `history_file_path` come la stringa contentente il percorso del file binario in cui è memorizzata la cronologia di ascolto.
3. Si entra in un blocco try-catch nel quale si tenta di aprire in lettura il file relativo al percorso appena calcolato.
4. Si assegna a `history` l'output del modulo `load` della libreria `pickle` sul file binario.
5. Se, durante il caricamento o l'apertura del file, viene sollevata un'eccezione, si calcolerà la cronologia di ascolto, altrimenti, si restituirà il dizionario `history`.
6. Si itera sulle coppie (periodo, canzoni) nel dizionario `periods`.
7. Si ottiene l'ora `hour` corrispondente al periodo `period` su cui si sta iterando. Se `period` è una istanza di `datetime`, per estrarre l'ora utilizzo `period.hour`. Altrimenti, se `period` non è una istanza di `datetime`, utilizzo il periodo stesso come ora. Questo aspetto è necessario in quanto la funzione `compute_listening_history_periods()` viene utilizzata anche nella seconda fase per calcolare i pattern di playlist partendo dal dizionario `song_sets` prodotto in output dalla prima fase. Tale dizionario ha come chiavi le singole ore, in formato intero, e non oggetti di tipo `datetime`. Se `hour` non è una chiave di `history`, la si crea e le si assegna una lista vuota.
8. Tramite una list comprehension, si estraggono gli ID `track_ids` delle tracce `tracks` corrispondenti al periodo `period` che si sta esaminando attualmente. Anche in questo caso, siccome `compute_listening_history_periods()` viene utilizzata anche nella seconda fase, l'estrazione degli ID viene effettuata in maniera differente in base a come è strutturato l'input `tracks`.
9. Gli ID `track_ids` vengono utilizzati come input per il metodo `audio_features()`, chiamato sull'oggetto `spotify` utilizzando la API di Spotify, in modo tale da ottenere le caratteristiche audio (`features`) dei brani. L'oggetto `features`, restituito dalla API di Spotify, è una lista di dizionari, uno per ogni ID fornito. Ciascun dizionario `feature` ha diverse chiavi: l'ID della canzone, i nominativi delle 12 caratteristiche audio e, infine, le stringhe "uri", "track\_href", "analysis\_url", "type" e "duration\_ms", che andremo ad eliminare successivamente.
10. Si itera sulla lista di dizionari `features`.

11. Si controlla se il dizionario `feature` esaminato attualmente è valido, ossia se è diverso da `None` e contiene la chiave `id`. Questo controllo è necessario in quanto la chiamata al metodo `audio_features()` può restituire dizionari nulli o con una struttura diversa da quella che ci aspettiamo nel caso in cui uno o più ID in `track_ids` siano relativi a podcast o a brani introduttivi di una playlist. Ad esempio, il primo brano della playlist "Il mio Daily" generata automaticamente da Spotify, è sempre un brano introduttivo, il cui titolo cambia in base al giorno della settimana (ad esempio "È Lunedì"). Gli ID di brani di questo tipo, se passati al metodo `audio_features()`, producono un dizionario nullo.
12. Come anticipato, si filtra il dizionario `features`, rimuovendo le chiavi che non ci servono, ottenendo il dizionario `final_features`, che viene aggiunto alla fine della lista `history['hour']`.
13. Alla fine dell'esecuzione della funzione, se `history` è vuoto, si esce forzatamente tramite il modulo `exit()` della libreria `sys`.
14. Altrimenti, si restituisce `history` dopo averlo caricato, tramite `pickle`, sul file aperto in scrittura denotato dal percorso `history_file_path`.

L'output di questa funzione consiste nel dizionario `history`, rappresentativo della cronologia di ascolti dell'utente, filtrata per periodo. Si noti che questo approccio mantiene invariato l'ordinamento delle canzoni, indipendentemente da se esiste un solo periodo per un determinato orario o più di uno. Inoltre, una volta creata, la cronologia di ascolto viene salvata sul file binario "`listening_histroy..bin`", in modo tale da renderla accessibile agli altri moduli.

### 3.6 Calcolo delle abitudini di ascolto

Come discusso nella sezione 2.2, per ogni periodo nella cronologia, andiamo a calcolare la percentuale di nuove canzoni riprodotte dall'utente, distinguendo tra artisti conosciuti (NTKA, New Track Known Artist) e sconosciuti (NTNA, New Track New Artist). Per comodità, ricordiamo le formule:

$$NTNA = 100 \times \left( \frac{\sum_{i \in Days} dNTNA_i}{h} \right)$$

$$NTKA = 100 \times \left( \frac{\sum_{i \in Days} dNTKA_i}{h} \right)$$

Dove, *Days* è l'insieme di giorni nella cronologia degli ascolti,  $dNTNA_i$  rappresenta il numero di nuove tracce di artisti sconosciuti riprodotte durante il giorno  $i$  nel periodo  $P$ , e  $dNTKA_i$  rappresenta il numero di nuove tracce di artisti conosciuti riprodotte durante

il giorno  $i$  nel periodo  $P$ . Nella nostra implementazione, ci sono due funzioni coinvolte nel calcolo della coppia  $P_h = (NTNA, NTKA)$ : la prima, `compute_dNTNA_dNTKA()`, prende in input un periodo `current_period` e il dizionario dei periodi `periods`, e restituisce la coppia  $(dNTNA, dNTKA)$  per quel periodo. La seconda, `compute_listening_habits()`, prende in input il dizionario dei periodi `periods` e restituisce, chiamando più volte `compute_dNTNA_dNTKA()`, il dizionario `Ph`, che contiene la coppia  $(NTNA, NTKA)$  per ogni ora del giorno presente in `periods`.

### 3.6.1 Calcolo di dNTNA e dNTKA per ciascun periodo

```

1 def compute_dNTNA_dNTKA(current_period, periods):
2     period_songs = periods.get(current_period)
3     new_song = new_artist = True
4     period_dNTNA = period_dNTKA = 0
5     history_file = check_listening_history_file(periods)
6     for song in period_songs:
7         for period, tracks in periods.items():
8             if period < current_period:
9                 for track in tracks:
10                     if history_file:
11                         track_id = track['TrackID']
12                         track_artist_id = track['artistName']
13                         period_track_id = song['TrackID']
14                         period_track_artist_id =
15                             song['artistName']
16                     else:
17                         track_id = track['track']['id']
18                         track_artist_id =
19                             track['track']['artists'][0]['id']
20                         period_track_id = song['track']['id']
21                         period_track_artist_id =
22                             song['track']['artists'][0]['id']
23                         if track_id == period_track_id:
24                             new_song = False
25                         if track_artist_id == period_track_artist_id:
26                             new_artist = False
27                         if new_song and new_artist:
28                             period_dNTNA += 1
29                         if new_song and not new_artist:
30                             period_dNTKA += 1
31     return period_dNTNA, period_dNTKA

```

La funzione `compute_dNTNA_dNTKA()` opera come segue:

1. Si recuperano le canzoni `period_songs` relative al periodo corrente `current_period`, si inizializzano i contatori `period_dNTNA` e `period_dNTKA` a 0 e i flag `new_song` e `new_artist` a `True`.
2. Si definisce `history_file` come il risultato alla chiamata della funzione `check_listening_history_file()`. Tale funzione, della quale non forniremo l'implementazione, restituisce `True` se le tracce all'interno del dizionario `periods` sono identificate dalla chiave `TrackID`. Questo controllo risulta necessario in quanto la struttura del dizionario `periods` varia leggermente in base a se l'utente ha fornito o meno il file `StreamingHistory.json`.
3. Si itera sulle canzoni `song` nella lista `period_songs`.
4. Si itera sulle coppie (periodo, canzoni) nel dizionario `periods`, tenendo in considerazione soltanto i periodi `period` precedenti a quello corrente `current_period`.
5. Si itera sulle canzoni `track` nella lista `tracks` di canzoni relative al periodo `period`.
6. Si definiscono le variabili `track_id`, `track_artist_id`, `period_track_id` e `period_track_artist_id` in base a se `history_file` è `True` o `False`.
7. Si verifica se l'id della traccia `track` coincide con l'id della traccia `song`, e se l'id dell'artista della traccia `track` coincide con l'id dell'artista della traccia `song`. Nel primo caso, si imposta a `False` il flag `new_song`, nel secondo caso, si imposta a `False` il flag `new_artist`.
8. Se i flag `new_song` e `new_artist` sono entrambi `True`, si incrementa di uno `period_dNTNA`, se il flag `new_song` è `True` e `new_artist` è `False`, si incrementa di uno `period_dNTKA`.
9. Si restituisce la coppia (`period_dNTNA`,`period_dNTKA`).

### 3.6.2 Calcolo di NTNA e NTKA

```
1 def compute_listening_habits(periods):
2     Ph = {}
3     days = list(set([datetime(day.year, day.month, day.day) for
4                      day in periods.keys()]))
5     for hour in range(24):::
6         dNTNA = 0
7         dNTKA = 0
8         h = 0
9         for day in days:
10            current_period = datetime(day.year, day.month,
11                                         day.day, hour)
12            if periods.get(current_period, None):
13                h += len(periods.get(current_period))
14                period_dNTNA, period_dNTKA =
15                    compute_dNTNA_dNTKA(current_period, periods)
16                dNTNA += period_dNTNA
17                dNTKA += period_dNTKA
18            NTNA = 100 * dNTNA / h
19            NTKA = 100 * dNTKA / h
20            Ph[hour] = (NTNA, NTKA)
21        if not Ph:
22            sys.exit()
23    return Ph
```

La funzione `compute_listening_habits()` opera come segue:

1. Si inizializza `Ph` a un dizionario vuoto.
2. Si recuperano tutti i giorni (anno/mese/giorno) presenti all'interno del dizionario `periods` e si memorizzano nella lista `days`.
3. Si itera sulle ore `hour` da 0 a 23.
4. Si inizializzano a 0 le variabili `dNTNA`, `dNTKA` e `h`.
5. Si itera sui giorni `day` presenti nell'insieme memorizzato globalmente `days` contenente i giorni presenti nel dizionario `periods`.
6. Si costruisce il periodo `current_period` creando un oggetto `datetime` a partire dal giorno `day` e dall'ora `hour`.
7. Se il periodo `current_period` è presente nel dizionario `periods`, incremento `h` di una quantità pari al numero di tracce presenti in `current_period`, chiamo la

funzione `compute_dNTNA_dNTKA()` passandogli `current_period` e `periods` in input, e utilizzo i valori `period_dNTNA` e `period_dNTKA` da essa restituiti per incrementare le variabili `dNTNA` e `dNTKA`.

8. Una volta finito di iterare sui giorni `days`, si calcola la coppia (`NTNA,NTKA`) relativa all'ora `hour`, e la si assegna a `Ph[hour]`.
9. Alla fine dell'esecuzione della funzione, se `Ph` è vuoto, si esce forzatamente tramite il modulo `exit()` della libreria `sys`.
10. Altrimenti, si restituisce `Ph`.

Il dizionario `Ph` verrà utilizzato per produrre song-set la cui coppia (`NTNA,NTKA`) è più simile possibile alle abitudini di ascolto dell'utente.

## 3.7 Clustering

In questa sezione ci occupiamo di applicare alla cronologia di ascolto dell'utente gli algoritmi di clustering discussi nella sezione 2.4. Alla fine di questo procedimento otterremmo due clustering, uno tramite l'applicazione dell'algoritmo K-means e l'altro tramite l'applicazione dell'algoritmo Furthers-Point-First (FPF). Questi due clustering verranno utilizzati successivamente per produrre i song-set, dai quali si sceglierà quello più simile possibile alle abitudini di ascolto dell'utente. Per implementare il clustering della cronologia di ascolto, abbiamo utilizzato tre funzioni: le funzioni `kmeans()` e `furthest_point_first()` implementano gli omonimi algoritmi di clustering. Queste due funzioni prendono in input i dizionari di caratteristiche audio relativi ad un periodo e restituiscono una tripla (`clusters`, `centroidi`, `indice_di_Davies_Bouldin`) come risultato del clustering. La terza funzione, `compute_clusterings`, prende in input la cronologia di ascolti e restituisce, chiamando `kmeans()` e `furthest_point_first()` con un numero di cluster crescenti, il dizionario `clusterings`. Tale dizionario contiene, per ogni periodo della cronologia, un dizionario le cui chiavi sono 'kmeans' e 'fpf' e i cui valori costituiscono la soluzione che ha minimizzato l'indice di Davies Bouldin per tale periodo e per tale algoritmo. In questo contesto, le soluzioni sono rappresentate sottoforma di triple (`clusters`, `centroidi`, `numero_cluster_utilizzati`) relativi all'output del rispettivo algoritmo (K-means o FPF) che ha minimizzato l'indice di Davies Bouldin. Vediamo nel dettaglio queste tre funzioni.

### 3.7.1 Algoritmo K-means

Per l'implementazione dell'algoritmo K-means, abbiamo utilizzato la classe `KMeansConstrained` della libreria `k-means-constrained`. Questa classe ci permette di specificare un numero minimo di punti per ciascun cluster (nel nostro caso 4). Nel caso in

cui la cardinalità di `features_vector` (ossia il numero di canzoni presenti all'interno della cronologia in un determinato periodo) sia minore di quella richiesta ( $4 * n_{clusters}$ ), il clustering fallirà con l'eccezione `ValueError`. Tale eccezione è catturata all'interno della funzione `compute_clusterings()`.

```

1 def kmeans(features_vector, n_clusters):
2     all_features = np.array([[value for value in
3         features.values() if not isinstance(value, str)] for
4         features in features_vector])
5     scaler = StandardScaler()
6     all_features_scaled = scaler.fit_transform(all_features)
7
8     kmeans = KMeansConstrained(n_clusters=n_clusters, size_min=4)
9     kmeans.fit(all_features_scaled)
10    davies_bouldin_index =
11        davies_bouldin_score(all_features_scaled, kmeans.labels_)
12
13    clusters = [[] for _ in range(n_clusters)]
14    for i, label in enumerate(kmeans.labels_):
15        clusters[label].append(features_vector[i])
16
17    centroids = scaler.inverse_transform(kmeans.cluster_centers_)
18    centroids_dict = {key: value for key, value in
19        zip(features_vector[0].keys(), centroid)} for centroid in
20        centroids]
21
22    for centroid in centroids_dict:
23        centroid['time_signature'] = centroid.pop('id')
24    return clusters, centroids_dict, davies_bouldin_index

```

Ecco una descrizione dettagliata del funzionamento di `kmeans()`:

1. Viene estratto un array numpy `all_features` che contiene solo le features numeriche da `features_vector` (in pratica rimuoviamo l'id della canzone da ciascun dizionario di features). Le features numeriche vengono standardizzate in modo che abbiano una media di 0 e una varianza di 1 utilizzando `StandardScaler()` dalla libreria `sklearn.preprocessing`.
2. Si inizializza l'algoritmo di clustering K-means con un numero specificato di cluster (`n_clusters`) utilizzando `KMeansConstrained`. Il modello K-means viene addestrato sui dati standardizzati utilizzando il metodo `fit()`.
3. Viene calcolato l'indice di Davies-Bouldin come misura di qualità del clustering utilizzando la funzione `davies_bouldin_score()` dalla libreria `sklearn.metrics`.

4. Viene creata una lista di liste vuote `clusters` per memorizzare i punti assegnati a ciascun cluster. Per ogni punto nel dataset, viene aggiunto il punto al cluster corrispondente basato sull'etichetta restituita dall'algoritmo K-means.
5. I centroidi dei cluster, rappresentati come vettori, vengono trasformati indietro nello spazio delle features originali utilizzando `scaler.inverse_transform()`. I centroidi vengono quindi convertiti in dizionari in modo che possano essere facilmente interpretati, con le chiavi che corrispondono alle features e i valori corrispondenti ai centroidi. La chiave 'id' viene rinominata in 'time\_signature' in ciascun dizionario dei centroidi.
6. Vengono restituiti tre oggetti: una lista `clusters` di liste dove ogni lista rappresenta i punti assegnati a un cluster, una lista di dizionari `centroids_dict` dove ogni dizionario rappresenta un centroide, con le chiavi corrispondenti alle features e i valori corrispondenti ai centroidi, e infine l'indice di Davies-Bouldin `davies_bouldin_index` calcolato come misura di qualità del clustering.

### 3.7.2 Algoritmo Furthers-Point-First (FPF)

L'implementazione del clustering tramite Furthers-Point-First non si avvale di librerie esterne, in quanto non esistono per questo algoritmo. Come per K-means, anche in questa implementazione facciamo in modo che ciascun cluster abbia almeno 4 punti e, nel caso in cui la cardinalità di `features_vector` sia minore di quella richiesta ( $4 * n\_clusters$ ), lanciamo l'eccezione `ValueError` con un messaggio di errore opportuno. Come anticipato, questa eccezione viene catturata all'interno della funzione `compute_clusterings()`. Per comodità, vista la lunghezza, spezziamo l'algoritmo in quattro macroblocchi:

```

1 def furthest_point_first(features_vector, n_clusters,
2     min_size=4):
3     num_samples = len(features_vector)
4     required_samples = min_size * n_clusters
5     if required_samples > num_samples:
6         raise ValueError("The product of min_size and n_clusters
7             is greater than the number of samples.")
8     all_features = np.array([[value for value in
9         features.values() if not isinstance(value, str)] for
          features in features_vector])
10    distances = squareform(pdist(all_features,
11        metric='euclidean'))
12    scaler = StandardScaler()
13    all_features_scaled = scaler.fit_transform(all_features)

```

Questa parte di algoritmo, all'inizio della funzione `furthest_point_first()`, esegue diverse operazioni per preparare i dati per l'algoritmo di clustering. Ecco una spiegazione dettagliata di ciò che fa:

1. Si calcola il numero totale di punti (ossia la cardinalità di `features_vector`) presenti e il numero totale di punti `num_samples` richiesti in modo che ciascun cluster abbia almeno `min_size` (4) punti.
2. Viene verificato se il numero richiesto di campioni supera il numero di campioni effettivi nel dataset e, in tal caso, viene sollevata un'eccezione `ValueError`.
3. Viene creato un array numpy `all_features` contenente solo le features numeriche in ciascun elemento di `features_vector` (si rimuove l'id della canzone da ciascun dizionario di features).
4. Viene calcolata una matrice delle distanze euclidee tra tutte le coppie di campioni utilizzando `pdist` dalla libreria `scipy.spatial.distance`.
5. La funzione `squareform` viene utilizzata per convertire la matrice delle distanze in una forma quadrata.
6. Viene istanziato un oggetto `StandardScaler()` per standardizzare le features in modo che abbiano una media di 0 e una deviazione standard di 1.
7. Le features vengono standardizzate utilizzando il metodo `fit_transform()` del `scaler`, e il risultato viene memorizzato in `all_features_scaled`.

```
1  centroids = [np.random.randint(num_samples)]
2  while len(centroids) < n_clusters:
3      max_distances = distances[:, centroids].min(axis=1)
4      new_centroid = np.argmax(max_distances)
5      if new_centroid not in centroids:
6          centroids.append(new_centroid)
7  labels = np.argmin(distances[:, centroids], axis=1)
```

Questa seconda parte di algoritmo si occupa del calcolo dei centroidi e dell'assegnamento delle etichette. Ecco come funziona:

1. Si inizializza una lista `centroids` con un singolo punto scelto casualmente.
2. Si continua ad aggiungere nuovi centroidi fino a quando il numero desiderato di centroidi `n_clusters` non è stato raggiunto.
3. Si calcolano le distanze massime `max_distances` di ciascun punto rispetto ai centroidi attuali.

4. Si trova l'indice `new_centroid` del punto che ha la massima distanza da uno qualsiasi dei centroidi esistenti.
5. Si verifica se il nuovo candidato `new_centroid` ad essere scelto come centroide non è già presente nella lista dei centroidi, in tal caso, viene aggiunto alla lista `centroids` dei centroidi.
6. Si costruisce la lista `labels` assegnando un'etichetta a ciascun punto basandosi sulla distanza minima tra esso e i centroidi selezionati.

```

1  while any(np.sum(labels == label) < min_size for label in
2      range(n_clusters)):
3      for label in range(n_clusters):
4          cluster_indices = np.where(labels == label)[0]
5          if len(cluster_indices) < min_size:
6              remaining_indices =
7                  np.setdiff1d(np.arange(num_samples),
8                  centroids)
9              max_distances = distances[remaining_indices][:, ,
10                  cluster_indices].min(axis=1)
11              new_point_index =
12                  remaining_indices[np.argmax(max_distances)]
13              centroids.append(new_point_index)
14              labels[new_point_index] = label

```

In questa terza parte di algoritmo ci si assicura che ciascun cluster abbia almeno `min_size` (4) punti:

1. Si itera fino a quando esiste almeno un cluster che ha meno punti del valore minimo desiderato `min_size`.
2. Si itera su ciascun cluster.
3. Si calcolano gli indici `cluster_indices` dei punti nel dataset che sono assegnati al cluster specificato da `label`.
4. Viene verificato se il numero di punti nel cluster (`len(cluster_indices)`) è inferiore al valore minimo desiderato e, in tal caso, si effettuano le seguenti operazioni:
  - (a) Si crea un array `remaining_indices` di indici dei punti nel dataset che non sono ancora stati selezionati come centroidi dei cluster. Questi indici corrispondono ai punti che potrebbero essere considerati come candidati per diventare nuovi centroidi.

- (b) Vengono calcolate le distanze `max_distances` dei punti in `remaining_indices` da ogni punto nel cluster.
- (c) Viene selezionato il nuovo punto come quello .
- (d) Il punto `new_point_index`, ossia quello con la massima distanza da un punto qualunque nel cluster, viene aggiunto ai centroidi e gli viene assegnato il label `label` corrispondente.

```

1 davies_bouldin_index =
2     davies_bouldin_score(all_features_scaled, labels)
3 clusters = [[] for _ in range(n_clusters)]
4 for i, label in enumerate(labels):
5     clusters[label].append(features_vector[i])
6 centroids =
7     scaler.inverse_transform(all_features_scaled[centroids])
8 centroids_dict = {key: value for key, value in
9         zip(features_vector[0].keys(), centroid)} for centroid in
10    centroids]
11 for centroid in centroids_dict:
12     centroid['time_signature'] = centroid.pop('id')
13 return clusters, centroids_dict, davies_bouldin_index

```

L'ultima parte dell'algoritmo si occupa di produrre l'output:

1. Si calcola l'indice di Davies-Bouldin `davies_bouldin_index`, una misura della qualità del clustering, utilizzando le features standardizzate `all_features_scaled` e le etichette `labels` dei cluster.
2. Si crea una lista di liste vuote, dove ogni lista vuota rappresenta un cluster.
3. Tramite un ciclo for, ggni punto `features_vector['i']` nel dataset viene aggiunto al cluster corrispondente sulla base dell'etichetta assegnata (`cluster['label']`).
4. Si Trasforma indietro i centroidi standardizzati nello spazio delle features originale, utilizzando l'operazione inversa della standardizzazione `scaler.inverse_transform()`.
5. Si crea un dizionario `centroids_dict` per ciascun centroide `centroid`, dove le chiavi corrispondono alle features originali e i valori corrispondono ai valori dei centroidi.
6. Per ogni centroide si rinomina la chiave 'id' in 'time\_signature'.
7. La funzione restituisce tre oggetti: i cluster, i centroidi e l'indice di Davies-Bouldin.

### 3.7.3 Calcolo dei clustering

Ora che abbiamo fornito l'implementazione di entrambi gli algoritmi di clustering, vediamo nel dettaglio come opera la funzione `compute_clusterings()`. Come anticipato, questa funzione prende in input il dizionario `periods_listening_history` che costituisce la cronologia di ascolto dell'utente e restituisce i clustering.

```
1 def compute_clusterings(periods_listening_history):
2     clusterings = {}
3     for period, features in periods_listening_history.items():
4         clusterings[period] = {}
5         for method in ['kmeans', 'fpf']:
6             best_solution = None
7             best_davies_bouldin_index = float('inf')
8             n_clusters = float('inf')
9             for k in range(2,
10                           math.floor(math.sqrt(len(features))) + 1):
11                 try:
12                     labels, centroids, davies_bouldin_index =
13                         kmeans(features, k) if method == 'kmeans'
14                         else furthest_point_first(features, k)
15                     if davies_bouldin_index <
16                         best_davies_bouldin_index:
17                         best_solution = labels
18                         best_centroids = centroids
19                         best_davies_bouldin_index =
20                             davies_bouldin_index
21                         n_clusters = k
22                     except ValueError:
23                         continue
24                     if best_davies_bouldin_index != float('inf'):
25                         clusterings[period][method] = (best_solution,
26                             best_centroids, n_clusters)
27                     if not clusterings:
28                         sys.exit()
29     return clusterings
```

La funzione opera in questo modo:

1. Si inizializza `clusterings` a un dizionario vuoto.
2. Si itera sulle coppie (periodo, vettori\_di\_caratteristiche) presenti nel dizionario `periods_listening_history`.
3. Si inizializza `clusterings[period]` a un dizionario vuoto.
4. Si itera sui metodi 'kmeans' e 'fpf'.
5. Si inizializza la miglior soluzione `best_solution` a `None`, il miglior indice di Davies Bouldin `best_davies_bouldin_index` a  $\infty$  e il numero di cluster `n_clusters` a  $\infty$ .
6. Si itera su valori di `k` che vanno da 2 a  $\lfloor \sqrt{\text{len}(\text{features})} \rfloor + 1$
7. Si invoca la funzione appropriata (`kmeans()` o `furthest_point_first()`) a seconda di `method` e si memorizza l'output nelle variabili `labels`, `centroids` e `davies_bouldin_index`.
8. Se l'indice `davies_bouldin_index` appena calcolato è inferiore all'indice migliore attuale `best_davies_bouldin_index`, si imposta `best_davies_bouldin_index` a `davies_bouldin_index`, si imposta `best_centroids` a `centroids`, si imposta `best_solution` a `labels` e si imposta `n_clusters` a `k`.
9. Se, durante questo procedimento, uno dei due algoritmi di clustering solleva l'eccezione `ValueError` ad indicare che non ci sono sufficienti punti per garantire che ogni cluster abbia 4 punti, si salta alla prossima iterazione (prossimo valore di `k`).
10. Se è stata trovata una soluzione, si otterrà un `best_davies_bouldin_index` diverso da  $\infty$ . In tal caso, si imposta `clusterings[period][method]` alla soluzione che ha minimizzato l'indice di Davies-Bouldin, espressa sottoforma di tripla (`best_solution`, `best_centroids`, `n_clusters`).
11. Alla fine dell'esecuzione della funzione, se `clusterings` è vuoto, si esce forzatamente tramite il modulo `exit()` della libreria `sys`.
12. Altrimenti, si restituisce `clusterings`.

Ricapitolando, la funzione itera su ciascun periodo e cerca di trovare il miglior clustering per quel periodo utilizzando due diversi metodi: K-means e Furthest Point First (FPF). Per entrambi i metodi, la funzione cerca il miglior numero di cluster e calcola l'indice di Davies-Bouldin per valutare la qualità del clustering. La ricerca del miglior clustering avviene iterando su diversi valori di `k` e calcolando l'indice di Davies-Bouldin per ogni valore di `k`. Se un valore di `k` produce un indice di Davies-Bouldin migliore

rispetto a quelli precedenti, quel valore di  $k$  diventa il nuovo candidato per il miglior clustering. Una volta trovati i migliori clustering per entrambi i metodi (K-means e FPF) per un dato periodo, i risultati vengono memorizzati in un dizionario annidato. La chiave esterna del dizionario rappresenta il periodo, mentre le chiavi interne rappresentano il metodo di clustering utilizzato (K-means o FPF). Ogni valore interno contiene le etichette dei cluster, i centroidi e il numero di cluster corrispondenti al miglior clustering trovato per il periodo e il metodo specifico.

### 3.8 Generazione dei song-set

In questa sezione ci occupiamo di spiegare dettagliatamente l'implementazione della generazione dei song-set. Ricordiamo che un song-set non costituisce la playlist finale, ma è bensì l'insieme di canzoni che rispecchia maggiormente le abitudini di ascolto di un utente (ossia la coppia (NTNA,NTKA)) per un dato periodo. Come anticipato nella Sezione 3.8, per ogni periodo che non ha generato errore nella fase di clustering, genereremo, tramite un'euristica lineare e una sferica, 4 song-set. Ciascuna di queste euristiche produrrà 2 song-set: una relativa al clustering ottenuto tramite l'algoritmo K-means e l'altra relativa al clustering ottenuto tramite l'algoritmo Furthest Point First. Per implementare la generazione dei song-set abbiamo utilizzato quattro funzioni:

1. `linear_heuristic()` e `spheric_heuristic()`: queste funzioni implementano le omonime euristiche. Prendono in input un cluster `cluster`, il relativo centroide `centroid`, un parametro `m` e il song-set parziale `song_set`. Ricordando che il song-set è generato cluster per cluster, `m` è definito come  $m = \frac{\ell}{n \times 4}$  dove  $\ell$  è il numero di brani che la playlist deve avere,  $n \in \{n_k, n_F\}$  è il numero di cluster, e 4 è il numero di richieste al sistema di raccomandazione musicale per ogni cluster.
2. `generate_clustering_song_sets()`: questa funzione prende in input il dizionario `clusterings` e restituisce il dizionario `song_sets`, che ha come chiavi i periodi per cui la generazione dei song-set è andata a buon fine e come valori i dizionari `clustering_song_sets`. Ciascun dizionario `clustering_song_sets` ha come chiavi i nomi degli algoritmi e come valori le relative song-set prodotte.
3. `compute_most_similar_song_sets()`: questa funzione prende in input il dizionario `song_sets`, il dizionario `periods` e il dizionario `listening_habits` e restituisce il dizionario `most_similar_song_set`, che contiene, per ogni periodo in `song_sets`, la song-set la cui coppia (NTNA,NTKA) è più simile possibile a quella dell'utente, cioè quella contenuta in `listening_habits` per un determinato periodo.

Vediamo nei dettagli ciascuna di queste funzioni.

### 3.8.1 Euristica lineare

La funzione `linear_heuristic` prende in input un cluster `cluster` di brani musicali, il suo centroide `centroid`, un parametro `m`, e un set di brani musicali `song-set` già selezionati. L'obiettivo della funzione è generare una lista di raccomandazioni `recommended_tracks` di brani musicali in base a una euristica lineare.

```
1 def linear_heuristic(cluster, centroid, m, song_set, spotify):
2     distances = [distance.euclidean([value for value in
3         song.values() if not isinstance(value, str)], [value for
4             value in centroid.values() if not isinstance(value, str)])
5             for song in cluster]
6     points_with_distances = list(zip(distances, cluster))
7     sorted_points = sorted(points_with_distances, key=lambda x:
8         x[0])
9     point_indexes = [0, math.floor(len(cluster)/3),
10                     2*math.floor(len(cluster)/3), len(cluster)-1]
11     recommended_tracks = []
12     for i, point in enumerate(sorted_points):
13         if i in point_indexes:
14             modified_song_data = {'target_'} + key: value for
15                 key, value in point[1].items() if key != 'id'}
16             track_id = point[1]['id']
17             recommendations =
18                 spotify.recommendations(seed_tracks=[track_id],
19                     limit=100, kwargs=modified_song_data).get('tracks')
20             count = 0
21             for track in recommendations:
22                 if track['id'] not in list(set(song['id'] for
23                     song in song_set)):
24                     recommended_tracks.append(track)
25                     count += 1
26                     if count == int(m):
27                         break
28     return recommended_tracks
```

Il codice funziona in questo modo:

1. Tramite list comprehension annidate, viene calcolata la distanza euclidea tra ciascun brano `song` all'interno del cluster `cluster` e il centroide `centroid` del cluster. Le distanze vengono memorizzate in una lista `distances`.
2. Viene creata una lista di coppie (distanza, brano) che viene ordinata, tramite una funzione lambda, in base alla distanza e memorizzata in `sorted_points`. Questo

ordine rappresenta la priorità dei brani in base alla loro vicinanza al centroide del cluster.

3. Si crea una lista `point_indexes` contenente le posizioni dei punti nell'ordinamento che verranno utilizzati per generare le canzoni. Inoltre, si inizializza `recommended_tracks` a una lista vuota.
4. Si itera sui punti `point` in `sorted_points` il cui indice è contenuto in `point_indexes`.
5. Si costruisce il dizionario `modified_song_data`, il quale va aggiungere il prefisso "target\_", alle chiavi del dizionario che contiene le caratteristiche audio del punto `point` su cui si sta iterando. Questo è necessario perché il sistema di raccomandazioni di Spotify si aspetta nella richiesta che le caratteristiche audio abbiano il prefisso "target\_". Si recupera l'id della canzone e lo si memorizza in `track_id`.
6. Utilizzando l'API di Spotify, vengono generate raccomandazioni di brani musicali basate su ciascun brano (la cui posizione fa parte di `recommended_tracks`) su cui si sta iterando. Nella fattispecie, al metodo `recommendations()` vengono passati l'id del brano `track_id`, le caratteristiche audio target `modified_song_data` e si richiede un limite di 50 brani simili.
7. Impostando il limite a 10, le raccomandazioni vengono filtrate per escludere i brani già presenti nel set di brani musicali `song_set` già selezionati. Il numero di raccomandazioni ottenute è limitato dal parametro `m`. Per ogni punto, le `m` raccomandazioni vengono aggiunte alla lista `recommended_tracks`.
8. La funzione restituisce la lista di tracce `recommended_tracks`.

### 3.8.2 Euristica sferica

La funzione `spheric_heuristic` prende in input un cluster `cluster` di brani musicali, il suo centroide `centroid`, un parametro `m`, e un set di brani musicali `song-set` già selezionati. Questa funzione implementa una strategia di raccomandazione basata su clustering e su una distribuzione sferica nello spazio delle caratteristiche. Questa funzione utilizza direzioni casuali per esplorare lo spazio delle caratteristiche intorno al centroide del cluster e genera una lista `recommended_tracks` di raccomandazioni di brani musicali in base ai punti trovati più vicini nel cluster.

La funzione `spheric_heuristic` è stata implementata in questo modo:

```
1 def spheric_heuristic(cluster, centroid, m, song_set):
2     recommended_tracks = []
3     random_points = []
4     constraints_list = list(constraints.keys())
5     distances = [distance.euclidean([value for value in
6         song.values() if not isinstance(value,str)], [value for
7             value in centroid.values() if not isinstance(value,str)])
8         for song in cluster]
9     radius = max(distances)
10    center = [centroid[key] for key in centroid.keys() if not
11        isinstance(centroid[key], str)]
12    unique_songs = len(list(set(track['id']) for track in
13        cluster)))
14    if unique_songs >= 4:
15        limit = 4
16    else:
17        limit = unique_songs
```

1. Si inizializzano `recommended_tracks` e `random_points` a una lista vuota.
2. Si ricava la lista di chiavi `constraints_list` dal dizionario `constraints` (Listato 1).
3. Tramite list comprehension annidata, viene calcolata la distanza euclidea tra ciascun brano `song` all'interno del cluster `cluster` e il centroide `centroid` del cluster. Le distanze vengono memorizzate in una lista `distances`.
4. Si calcola il raggio `radius` della ipersfera come il massimo delle distanze `distances`.
5. Si ricava il centro `center` dell'ipersfera, ossia il centroide `centroid` del cluster.
6. Si ricava la lunghezza `unique_songs` della lista degli ID univoci delle canzoni presenti all'interno del cluster.
7. Si imposta la variabile `limit` a 4, se `unique_songs` è maggiore o uguale a 4, altrimenti, `limit` viene impostata a `unique_songs`.

```

1  while len(random_points) < limit:
2      randomnessGenerator = np.random.default_rng()
3      X = randomnessGenerator.normal(size=len(center))
4      U = randomnessGenerator.random()
5      random_point = radius * (U**(1/len(center))) /
6          np.sqrt(np.sum(X**2)) * X
7      random_point += np.array(center) # Translate point to
8          center
9
10     modified_song_data = {}
11     for j, value in enumerate(point):
12         key = constraints_list[j]
13         modified_song_data[key] = value
14     for key,value in modified_song_data.items():
15         if value < constraints[key]['min']:
16             modified_song_data[key] = constraints[key]['min']
17         elif value > constraints[key]['max']:
18             modified_song_data[key] = constraints[key]['max']

```

8. Si entra in un ciclo while in cui si genereranno punti casuali all'interno della ipersfera fino a quando il numero di punti distinti all'interno della lista `random_points` non coincide con `limit`.
9. Si inizializza un generatore di numeri casuali utilizzando il modulo `random.default_rng()` della libreria `numpy`.
10. Si genera una vettore `X` di dimensione 12, ossia il numero di dimensioni della ipersfera, utilizzando campioni casuali da una distribuzione normale standard.
11. Si genera un singolo numero `U` casuale uniformemente distribuito nell'intervallo  $[0, 1]$ .
12. Si calcola `random_points` in questo modo:
  - `(U**(1/len(center)))`: Eleva `U` alla potenza di 1 diviso per il numero di dimensioni dello spazio (`len(center)`), in modo da ottenere una lunghezza casuale per il vettore.
  - `np.sqrt(np.sum(X**2))`: Calcola la norma Euclidea del vettore `X`, che rappresenta la lunghezza del vettore casuale nella dimensione.
  - `radius * (U**(1/len(center))) / np.sqrt(np.sum(X**2))` : Normalizza la lunghezza del vettore casuale rispetto al raggio della sfera iperdimensionale.

- $\mathbf{X}$ : Moltiplica il vettore casuale per il vettore di direzione  $\mathbf{X}$ , che determina la direzione del vettore casuale nello spazio.
13. Si trasla il punto casuale dal centro dell'ipersfera specificato da `center`. In questo modo, il punto casuale sarà distribuito casualmente all'interno della sfera iperdimensionale centrata in `center`.
  14. Si inizializza `modified_song_data` a un dizionario vuoto.
  15. Iteriamo su ogni coordinata all'interno del punto `point` generato casualmente.
  16. Ricaviamo la chiave `key` estraendo il nome del vincolo in posizione `j` da `constraints_list`.
  17. Assegniamo il valore `value` della coordinata al dizionario `modified_song_data`, utilizzando la chiave `key`.
  18. Il punto casuale viene quindi controllato per garantire che ciascuna caratteristica rispetti i vincoli specificati. Se una caratteristica supera il valore massimo o minimo consentito (specificato nel dizionario `constraints`, vedi Listato 1), viene regolato di conseguenza. Il punto modificato viene quindi memorizzato in `modified_song_data`, dove ogni caratteristica ha il prefisso "target\_" necessario per la chiamata al sistema di raccomandazioni di Spotify.

```

1  min_distance = float('inf')
2  nearest_song = None
3  for song in cluster:
4      distance_to_song = distance.euclidean([value for
5          value in song.values() if not
6          isinstance(value,str)], [value for value in
7          modified_song_data.values()])
8      if distance_to_song < min_distance:
9          min_distance = distance_to_song
          nearest_song = song
10     if nearest_song not in random_points:
11         random_points.append(nearest_song)

```

19. Viene inizializzata la variabile `min_distance` a  $\infty$  e `nearest_song` a `None`.
20. Si itera su ciascun brano `song` nel cluster `cluster`. Per ogni brano, viene calcolata la distanza euclidea tra le sue caratteristiche e quelle del punto modificato `modified_song_data`. Se la distanza calcolata è minore della distanza minima attuale, la distanza minima `min_distance` viene aggiornata e il brano più vicino viene memorizzato in `nearest_song`.

21. Si verifica se `nearest_song` non è già presente nella lista `random_points`. In tal caso, si aggiunge `nearest_song` a `random_points`.
22. Si esce dal ciclo while quando la cardinalità di `random_points` coincide con `limit`.

```

1 for i, nearest_song in enumerate(random_points):
2     modified_song_data = {'target_' + key: value for key,
3                           value in nearest_song.items() if key != 'id'}
4     track_id = nearest_song['id']
5     recommendations =
6         spotify.recommendations(seed_tracks=[track_id],
7                                   limit=100, kwargs=modified_song_data).get('tracks')
8     count = 0
9     for track in recommendations:
10        if track['id'] not in list(set(song['id'] for song
11                                      in song_set)):
12            recommended_tracks.append(track)
13            count += 1
14        if count == m:
15            break
16    return recommended_tracks

```

23. Una volta usciti dal ciclo while, si itera sulle tracce `nearest_song` all'interno della lista `random_points`.
24. Le caratteristiche del brano `nearest_song` più vicino vengono memorizzate in `modified_song_data`, aggiungendovi il prefisso "target\_".
25. Si ottiene l'identificatore del brano più vicino (`track_id`) e vengono richieste le raccomandazioni di brani musicali simili utilizzando l'API di Spotify, analogamente a ciò che è stato fatto all'interno della funzione `linear_heuristic()`. Le raccomandazioni sono filtrate per escludere i brani già presenti nel set di brani musicali selezionati. Il numero di raccomandazioni ottenute è limitato dal parametro `m`.
26. Le raccomandazioni valide vengono aggiunte alla lista delle raccomandazioni `recommended_tracks`.
27. Si restituisce `recommended_tracks`.

Di seguito vengono elencati i vincoli imposti dal sistema di raccomandazioni di Spotify, memorizzati nel dizionario `constraints`.

```

1 constraints = {
2     'target_danceability': {'min':0, 'max':1},
3     'target_energy': {'min':0, 'max':1},
4     'target_key': {'min':0, 'max':11},
5     'target_loudness': {'min':float('-inf'), 'max':float('inf')},
6     'target_mode': {'min':0, 'max':1},
7     'target_speechiness': {'min':0, 'max':1},
8     'target_acousticness': {'min':0, 'max':1},
9     'target_instrumentalness': {'min':0, 'max':1},
10    'target_liveness': {'min':0, 'max':1},
11    'target_valence': {'min':0, 'max':1},
12    'target_tempo': {'min':float('-inf'), 'max':float('inf')},
13    'target_time_signature': {'min':0, 'max':5}
14 }
```

Listato 1: Vincoli imposti dal sistema di raccomandazioni di Spotify

### 3.8.3 Applicazione delle euristiche

Questa funzione, `generate_clustering_song_sets()`, accetta un dizionario `clusterings` contenente i clustering prodotti in output dalla funzione `compute_clusterings()`. La funzione restituisce un altro dizionario `song_sets`. Il dizionario `song_sets` associa un dizionario a ciascun periodo per cui sono stati prodotti i clustering. Tale dizionario ha come valori i song-set e come chiavi i nomi degli algoritmi che li hanno prodotti.

- kmlh: euristica lineare applicata al clustering ottenuto tramite K-means;
- kmsh: euristica sferica applicata al clustering ottenuto tramite K-means;
- fpflh: euristica lineare applicata al clustering ottenuto tramite FPF;
- fpfsh: euristica sferica applicata al clustering ottenuto tramite FPF.

Ecco i dettagli implementativi della funzione `generate_clustering_song_sets()`:

```
1 def generate_clustering_song_sets(clusterings):
2     song_sets = {}
3     for period, clustering_item in clusterings.items():
4         if clustering_item['kmeans'] and clustering_item['fpf']:
5             K, F = clustering_item['kmeans'],
6                     clustering_item['fpf']
7             kmlh_song_set, kmsh_song_set = [], []
8             fpflh_song_set, fpfsh_song_set = [], []
9             for clustering, song_set, heuristic in [(K,
10                 kmlh_song_set, 'l'), (K, kmsh_song_set, 's'), (F,
11                 fpflh_song_set, 'l'), (F, fpfsh_song_set, 's')]:
12                 for i, cluster in enumerate(clustering[0]):
13                     m = playlist_length / (clustering[2] * 4)
14                     if heuristic == 'l':
15                         tracks = linear_heuristic(cluster,
16                                         clustering[1][i], m, song_set)
17                         song_set.extend(tracks)
18                     else:
19                         tracks = spheric_heuristic(cluster,
20                                         clustering[1][i], m, song_set)
21                         song_set.extend(tracks)
22             song_sets[period] = {'kmlh': kmlh_song_set, 'kmsh':
23                 kmsh_song_set, 'fpflh': fpflh_song_set, 'fpfsh':
24                 fpfsh_song_set}
25     return song_sets
```

Ecco come funziona il codice:

1. Si inizializza `song_sets` a un dizionario vuoto.
2. Si itera sulle coppie (`period, clustering_item`) all'interno del dizionario `clusterings`, soltanto se esiste sia il clustering prodotto da K-means che quello prodotto da FPF.
3. Si recuperano i clustering K, prodotto tramite l'algoritmo K-means e F, prodotto tramite l'algoritmo FPF. Inoltre, si inizializzano a liste vuote i 4 song-set da produrre per il periodo su cui si sta iterando.
4. Si itera sulle triple (`clustering, song_set, heuristic`), dove `heuristic` può essere il carattere 'l' o 's' in base a se si sta considerando l'euristica lineare o quella sferica.
5. Si itera sui cluster `cluster` di ciascun clustering (`clustering[0]` contiene i cluster).

6. Si calcola  $m = \frac{\text{playlist\_length}}{\text{clustering}[2] \times 4}$ , dove `playlist_length` è una variabile globale (nel nostro caso impostata a 48) e `clustering[2]` contiene il numero di cluster  $k$  utilizzati per produrre il clustering (K o F) su cui si sta iterando.
7. In base a se `heuristic` coincide con 'l' o 's', si chiama la funzione corrispondente (`linear_heuristic()` o `spheric heuristic`), e si memorizza il risultato nella lista `tracks`, che viene utilizzata per estendere `song_set`. Indipendentemente da quale sia l'euristica, la funzione che la implementa prende in input il parametro `m`, la song-set parziale `song_set`, il cluster `cluster` e l'oggetto `clustering[1][i]`, ossia il centroide del cluster.
8. Una volta terminati i due cicli interni, si costruisce un dizionario, le cui chiavi sono i nomi degli algoritmi che hanno prodotto il song-set e i cui valori sono i song-set stessi. Questo dizionario viene assegnato, per ogni periodo `period`, al dizionario `song_sets`.
9. Si restituisce il dizionario `song_sets`.

### 3.8.4 Calcolo dei migliori song-set

La funzione `compute_most_similar_song_sets()` prende in input tre elementi: il dizionario di song-set `periods_song_sets` prodotto dalla funzione `generate_clustering_song_sets`, il dizionario di periodi `periods` e il dizionario delle abitudini di ascolto `listening_habits`. Per prima cosa, questa funzione calcola la coppia (NTNA,NTKA). Questo viene effettuato per ogni periodo e per tutte e 4 le song-set di quel periodo, e il risultato viene memorizzato all'interno del dizionario Ph. Successivamente, per ogni periodo, si calcola la distanza euclidea tra la coppia (NTNA,NTKA) di ciascun song-set generato per quel periodo e la coppia (NTNA,NTKA) presente nel dizionario `listening_habits`, ossia quella che era stata calcolata, per ciascun periodo, nella sezione 3.6.1. Infine, per ogni periodo, si recupera il song-set la cui coppia (NTNA,NTKA) ha minimizzato la distanza euclidea con le abitudini di ascolto dell'utente. Questi song-set vengono memorizzati nel dizionario `most_similar_song_set`, che costituisce l'output di questa funzione. L'implementazione della funzione è suddivisa in tre blocchi: il primo blocco si occupa di calcolare la coppia (NTNA,NTKA) per tutti i song-set, il secondo calcola le distanze euclideanee e il terzo restituisce i migliori song-set per ciascun periodo. Nelle pagine seguenti mostriamo come abbiamo implementato `compute_most_similar_song_sets()`.

```

1 def compute_most_similar_song_sets(periods_song_sets, periods,
2   listening_habits):
3   Ph = {}
4   for period, songsets in periods_song_sets.items():
5     Ph_songsets = {}
6     for algorithm_name, song_set in songsets.items():
7       dNTNA = dNTKA = h = 0
8       for track in song_set:
9         h += 1
10        track_id = track_item_song_set['id']
11        track_artist_id =
12          track_item_song_set['artists'][0]['id']
13        if check_listening_history_file(periods):
14          history_ids = [song['TrackID'] for
15            period_history, tracks_history in
16            periods.items() for song in
17            tracks_history]
18          history_artists = [song['artistName'] for
19            period_history, tracks_history in
20            periods.items() for song in
21            tracks_history]
22        else:
23          history_ids = [song['track']['id'] for
24            period_history, tracks_history in
25            periods.items() for song in
26            tracks_history]
27          history_artists =
28            [song['track']['artists'][0]['id'] for
29              period_history, tracks_history in
30              periods.items() for song in
31              tracks_history]
32        if track_id not in history_ids and
33          track_artist_id not in history_artists:
34            ntna += 1
35          elif track_id not in history_ids and
36            track_artist_id in history_artists:
37              ntna += 1
38            Ph_songsets[algorithm_name] =
39              (100*dNTNA/h,100*dNTKA/h)
40            Ph[period] = Ph_songsets

```

1. Si inizializza Ph a un dizionario vuoto.

2. Si itera sulle coppie (`period`, `songsets`) all'interno del dizionario `periods_song_sets`, ricordando che `songsets` è a sua volta un dizionario che contiene come chiavi i nomi degli algoritmi e come valori i song-set corrispondenti per il periodo `period`.
3. Si inizializza `Ph.songsets` a un dizionario vuoto. Questo dizionario, calcolato per ogni periodo, conterrà la coppia (NTNA,NTKA) per ogni nome di algoritmo.
4. Si itera sulle coppie (`algorithm_name`, `song_set`) all'interno del dizionario `songsets`.
5. Si inizializzano `dNTNA`, `dNTKA` e `h` a 0.
6. Si itera sulle tracce `track` presenti nel song-set `song_set` su cui si sta iterando.
7. Si incrementa `h`, si estrae l'id `track_id` della traccia `track` e l'id dell'artista `track_artist_id`.
8. Si verifica, tramite la funzione `check_listening_history_file()`, se il dizionario `periods` è stato costruito a partire dal file `StreamingHistory.json` oppure se è stato costruito a partire dalla chiamata alla API di Spotify relativa alle ultime 50 canzoni ascoltate, e si recuperano da `periods` i dati di nostro interesse (id delle canzoni e degli artisti) per calcolare le abitudini di ascolto.
9. Si estraggono gli id delle canzoni presenti nel dizionario `periods` tramite una list comprehension. Lo stesso viene fatto per gli artisti. Queste due informazioni vengono memorizzate, rispettivamente, all'interno delle variabili `history_ids` e `history_artists`.
10. Se l'id `track_id` della traccia `track` non è presente in `history_ids` e l'id `track_artist_id` dell'artista non è presente in `history_artists_ids`, si incrementa `dNTNA` di uno.
11. Se l'id `track_id` della traccia `track` non è presente in `history_ids` e l'id `track_artist_id` dell'artista è presente in `history_artists_ids`, si incrementa `dNTKA` di uno.
12. Si calcola la coppia (NTNA,NTKA), la cui formula è consultabile all'interno della Sezione 2.2, e la si assegna a `Ph.songsets[algorithm_name]`.
13. Si assegna `Ph.songsets` a `Ph[period]`.

```

1 most_similar = {}
2 for period, Ph_songsets in Ph.items():
3     euclidean_distances = []
4     for algorithm_name, (NTNA, NTKA) in Ph_songsets.items():
5         euclidean_distance = np.sqrt((NTNA -
6             listening_habits[period][0])**2 + (NTKA -
7             listening_habits[period][1])**2)
8         euclidean_distances.append((algorithm_name,
9             euclidean_distance))
10    most_similar[period] = min(euclidean_distances,
11        key=lambda x: x[1])

```

14. Si inizializza `most_similar` a un dizionario vuoto.
15. Si itera sulle coppie (`period, Ph_songsets`) all'interno del dizionario `Ph`.
16. Si inizializza `euclidean_distances` a una lista vuota.
17. Si itera sulle coppie (`algorithm_name, (NTNA,NTKA)`) all'interno del dizionario `Ph_songsets`.
18. Si calcola la distanza euclidea `euclidean_distance` tra le abitudini di ascolto dell'utente e la coppia (`NTNA,NTKA`) relativa all'algoritmo `algorithm_name` e al periodo `period`. Le abitudini di ascolto dell'utente sono memorizzate in `listening_habits[period][0]`, per quanto riguarda NTNA e in `listening_habits[period][1]`, per quanto riguarda NTKA.
19. Si aggiunge `euclidean_distance` come ultimo elemento della lista `euclidean_distances`.
20. Una volta calcolate tutte le coppie (`NTNA,NTKA`) per il periodo `period` su cui si sta iterando, si calcola la distanza euclidea minima e la si assegna a `most_similar[period]`.

```

1 most_similar_song_set = {}
2 for period, (algorithm_name, _) in most_similar.items():
3     most_similar_song_set[period] =
4         song_sets[period][algorithm_name]
5     if not most_similar_song_set:
6         sys.exit()
7 return most_similar_song_set

```

21. Si inizializza `most_similar_song_set` a un dizionario vuoto.
22. Si itera sulle coppie (`period, (algorithm_name, (NTNA,NTKA))`) all'interno del dizionario `most_similar`, tralasciando il valore di (NTNA,NTKA) che non ci serve in quanto dobbiamo restituire il song-set.
23. Si assegna a `most_similar_song_set[period]` il song-set `song_sets[period][algorithm_name]`, ossia il song-set relativo al nome dell'algoritmo `algorithm_name` che ha minimizzato la distanza euclidea tra la relativa coppia (NTNA,NTKA) e le abitudini di ascolto dell'utente considerando il periodo `period`.
24. Se il dizionario `most_similar_song_set` non esiste, si esce forzatamente tramite il modulo `exit()` della libreria `sys`.
25. Altrimenti, si restituisce `most_similar_song_set`.

Dopo l'esecuzione di `compute_most_similar_song_sets()`, Il dizionario `most_similar_song_set` viene caricato sul file binario "`most_similar_song_sets.bin`". In questo modo, il modulo `step2.py`, la cui implementazione è discussa nelle successive tre sezioni, avrà accesso ai song-set generati dalla prima fase, memorizzati in "`most_similar_song_sets.bin`" e alla cronologia di ascolto, memorizzata in "`listening_history.bin`" (come descritto nella Sezione 3.5).

### 3.9 Calcolo dei pattern di cronologia

Una volta generato il miglior song-set per ciascun periodo, possiamo procedere nell’illustrare l’implementazione della seconda fase di SMART. Ricordiamo che l’obiettivo di questa fase consiste nel trovare il miglior ordinamento possibile di ciascun song-set prodotto in output dalla fase precedente. Per farlo, si cerca di ordinare le canzoni presenti in ciascun song-set in modo tale da ottenere il pattern di playlist più simile possibile, utilizzando la Playlist Pattern Distance come metrica, al pattern di cronologia. In questa sezione mostriamo, a livello implementativo, come vengono calcolati i pattern di cronologia. Le nozioni teoriche necessarie per comprendere il contenuto di questa sezione (e di quelle successive relative alla seconda fase) sono consultabili alla Sezione 2.6. Nella nostra implementazione, il compito di calcolare i pattern di cronologia è affidato alle funzioni `compute_history_patterns()` e `compute_history_patterns_day_hour()`. La prima funzione prende in input il dizionario di periodi `periods` e la stringa `timestamp_key` e restituisce il dizionario

`history_patterns`, che ha come chiavi le ore del giorno e come valori i dizionari `history_patterns_days`. Ciascun dizionario `history_patterns_days` associa ai vari giorni della settimana ('Monday', 'Tuesday', ...) una lista di pattern, ciascuno dei quali è rappresentato come una lista di canzoni. Per calcolare i pattern di playlist relativi a una determinata ora di un determinato giorno, la funzione `compute_history_patterns()` chiama più volte la funzione `compute_history_patterns_day_hour()`, che, dati in input il dizionario `periods`, il timestamp `timestamp` di un periodo e la stringa `timestamp_key`, restituisce la lista `patterns` di pattern di cronologia relativa al giorno e all’ora della giornata a cui si riferisce il timestamp del periodo. In pratica, alla fine dell’esecuzione di `compute_history_patterns()`, il dizionario `history_patterns` conterrà, per ogni ora e per ogni giorno, una lista di pattern di playlist. La stringa `timestamp_key` serve per accedere correttamente ai timestamp di ascolto delle canzoni, e varia in base a se l’utente ha fornito o meno il file di cronologia `StreamingHistory.json`.

```

1 def compute_history_patterns(periods, timestamp_key):
2     history_patterns = {}
3     sorted_periods = sorted(periods.items(), key=lambda x: x[0],
4                             reverse=True)
5     for hour in hours:
6         history_patterns_days = {}
7         for timestamp, tracks in sorted_periods:
8             if timestamp.hour == hour:
9                 day = timestamp.strftime("%A")
10                if not history_patterns_days.get(day, None):
11                    history_patterns_days[day] = []
12                patterns =
13                    compute_history_patterns_day_hour(periods,
14                        timestamp, timestamp_key)
15                history_patterns_days[day].extend(patterns)
16            history_patterns[hour] = history_patterns_days
17        return history_patterns

```

Ecco come funziona `compute_history_patterns()`:

1. Si inizializza `history_patterns` a un dizionario vuoto.
2. Tramite l'utilizzo della funzione built-in `sorted` combinata a una funzione lambda, si ordinano i periodi in senso decrescente di timestamp, e si memorizza il risultato in `sorted_periods`.
3. Si itera per ciascuna ora `hour` della giornata.
4. Si inizializza `history_patterns_days` a un dizionario vuoto.
5. Si itera sulle coppie (`timestamp, tracks`) all'interno di `sorted_periods` per le quali l'ora del timestamp coincide con l'ora `hour` su cui si sta iterando.
6. Si estrae il nome del giorno dal timestamp tramite il metodo `strftime` di `datetime`, e lo si memorizza in `day`.
7. Se la chiave `day` non è presente in `history_patterns_days`, la si aggiunge e le si assegna una lista vuota.
8. Si calcolano i pattern di cronologia `patterns` relativi all'ora `hour` del giorno `day` trovati, tramite la funzione `compute_history_patterns_day_hour()`, all'interno del periodo `period` su cui si sta iterando. Si aggiunge la lista di pattern `patterns` a `history_patterns_days[day]`.

9. Si assegna `history_patterns_days` al dizionario `history_patterns`, utilizzando la chiave `hour`.

10. Si restituisce il dizionario `history_patterns`.

Prima di iniziare a spiegare l'implementazione della funzione `compute_history_patterns_day_hour()`, è utile notare che una canzone presente in un periodo non è detto che faccia parte del pattern di cronologia relativo a tale periodo. Ad esempio, se un utente inizia ad ascoltare musica alle 8:30 e finisce alle 10:15, le canzoni ascoltate dalle 10:00 alle 10:15 non saranno contenute nel pattern di cronologia relativo all'ora 10:00, ma faranno parte nel pattern relativo all'ora 8:00. Questo aspetto implica che non ci sia una corrispondenza 1:1 tra le canzoni ascoltate in un periodo e le canzoni contenute in un pattern di cronologia. Per questo, per calcolare i pattern di cronologia, è necessario capire quando è terminato il pattern precedente e quando inizia il pattern successivo. Il criterio per stabilire se due canzoni presenti consecutivamente nella cronologia fanno parte dello stesso pattern consiste semplicemente nel calcolare la differenza tra i due timestamp di ascolto e verificare se questa è minore o maggiore di 15 minuti. Nel primo caso, le due canzoni fanno parte dello stesso pattern, altrimenti faranno parte di pattern diversi. Ecco come funziona `compute_history_patterns_day_hour()`:

```
1 def compute_history_patterns_day_hour(periods, timestamp,
2                                     timestamp_key):
3     patterns = []
4     current_pattern = []
5
6     previous_timestamp = timestamp - timedelta(hours=1)
7     tracks = sorted(periods[timestamp], key=lambda x:
8         x[timestamp_key])
9     previous_tracks = periods.get(previous_timestamp, None)
10    if previous_tracks:
11        previous_tracks = sorted(previous_tracks, key=lambda x:
12            x[timestamp_key])
13        if tracks[0][timestamp_key] -
14            previous_tracks[-1][timestamp_key] <=
15            timedelta(minutes=15):
16            first_track_index = next((i for i, track in
17                enumerate(tracks) if track[timestamp_key] -
18                tracks[i-1][timestamp_key] >
19                timedelta(minutes=15)), None)
20            if first_track_index is not None:
21                tracks = tracks[first_track_index:]
```

1. Si inizializzano a liste vuote `patterns` e `current_pattern`, ricordando che `patterns` è una lista di liste, ciascuna delle quali rappresentanti un pattern.

2. Si calcola il timestamp del periodo immediatamente precedente a quello corrente `previous_timestamp`, sottraendo un ora dal timestamp in input corrente tramite la funzione `timedelta()`, importata da `datetime`.
3. Si crea la lista `tracks` contenente le canzoni del periodo per cui si stanno generando i pattern ordinate in senso decrescente di timestamp di ascolto.
4. Si memorizzano in `previous_tracks` i timestamp, ordinati in senso decrescente di riproduzione, delle canzoni `periods[previous_timestamp]` presenti nel periodo precedente (se esiste) a quello corrente.
5. Si verifica se la differenza tra i timestamp di ascolto della prima canzone del pattern corrente e quello dell'ultima canzone del pattern precedente differiscono di meno di 15 minuti.
6. In tal caso, si cerca l'indice `first_track_index` della prima traccia all'interno della lista `tracks` in cui la differenza di tempo tra la traccia corrente e la traccia precedente è maggiore di 15 minuti.
7. Se `first_track_index` esiste, si fa partire `tracks` dall'elemento di indice `first_track_index`.

Questo primo blocco di codice serve ad identificare se è presente un pattern relativo a un periodo precedente che termina nel periodo corrente. In tal caso, la ricerca dei pattern del periodo corrente partirà dalla canzone successiva all'ultima canzone del pattern precedente che termina nel periodo corrente. Se tale canzone non esiste, ossia se tutte le canzoni del periodo corrente sono state ascoltate consecutivamente, la funzione non restituirà pattern per il periodo corrente.

```

1  for track in tracks:
2      if not current_pattern or track[timestamp_key] -
3          current_pattern[-1][timestamp_key] <=
4              timedelta(minutes=15):
5                  current_pattern.append(track)
6              else:
7                  patterns.append(current_pattern)
8                  current_pattern = [track]

```

8. Si itera sulle tracce `track` presenti in `tracks`.
9. Se `current_pattern` è una lista vuota oppure se il timestamp di ascolto della traccia corrente `track` differisce di meno di 15 minuti dal timestamp di ascolto di quella precedente, si aggiunge `track` a `current_pattern`.

- Altrimenti, si aggiunge `current_pattern` a `patterns` e si inizializza nuovamente `current_pattern` a una lista contenente soltanto la traccia `track`. In questo modo, si incomincia la ricerca di un nuovo pattern.

In pratica, il blocco di codice appena esaminato si occupa di calcolare i pattern del periodo corrente, tranne l'ultimo, facendo eventualmente incominciare la ricerca dalla canzone successiva a `last_pattern_end`. L'ultimo pattern calcolato, ossia la variabile `current_pattern` una volta usciti dal ciclo for, verrà appeso a `patterns` soltanto alla fine della funzione, in quanto non è detto che termini nel periodo corrente. Il successivo blocco di codice si occupa di esaminare i periodi successivi, aggiungendo eventualmente altre canzoni a `current_pattern` prima di aggiungere quest'ultimo a `patterns`.

```

1  end_of_period = timestamp + timedelta(hours=1)
2  last_timestamp =
3      current_pattern[-1][timestamp_key].replace(tzinfo=None)
4  while end_of_period - last_timestamp <=
5      timedelta(minutes=15):
6      end_of_period = timestamp + timedelta(hours=1)
7      if end_of_period in periods:
8          tracks = periods[end_of_period]
9          tracks = sorted(tracks, key=lambda x:
10             x[timestamp_key])
11         for track in tracks:
12             if track[timestamp_key] -
13                 current_pattern[-1][timestamp_key] >
14                 timedelta(minutes=15):
15                 break
16             current_pattern.append(track)
17             last_timestamp =
18                 current_pattern[-1][timestamp_key]
19             last_timestamp =
20                 last_timestamp.replace(tzinfo=None)
21             timestamp = end_of_period
22         else:
23             break
24     patterns.append(current_pattern)
25 return patterns

```

- Si aggiunge un ora al timestamp in input e si memorizza il risultato in `end_of_period`.
- Si calcola il timestamp dell'ultima canzone dell'ultimo pattern del periodo corrente e si memorizza il risultato in `last_timestamp`.

13. Si itera fino a quando la differenza dei due timestamp rimane al massimo di 15 minuti.
14. Si aggiunge un ora a `end_of_period`, che denota la fine del periodo esaminato ad ogni iterazione. Si verifica che `end_of_periods` sia effettivamente una chiave di `periods`.
15. Si ordinano le canzoni del periodo successivo `periods[end_of_period]` in senso decrescente di timestamp di ascolto e si memorizzano in `tracks`.
16. Si itera sulle tracce `track` presenti in `tracks`.
17. Se il timestamp di ascolto di `track` e l'ultima canzone del pattern è maggiore di 15 minuti, si esce dal ciclo.
18. Altrimenti, si aggiunge `track` al pattern corrente `current_pattern` e si aggiorna `last_timestamp`, impostandolo al timestamp dell'ultimo elemento di `current_pattern`.
19. Dopo il ciclo for interno, si assegna a `timestamp` l'ora `end_of_period`, che verrà utilizzata per esaminare eventualmente il periodo successivo.
20. Se `end_of_period` non è una chiave di `periods` si esce dal ciclo.
21. Si aggiunge l'ultimo pattern `current_pattern` a `patterns`.
22. Si restituisce `patterns`.

Al termine dell'esecuzione di `compute_history_patterns()`, avremo ottenuto il dizionario `history_patterns`, contenente, per ogni ora e per ogni giorno, una lista di pattern di cronologia.

### 3.10 Scelta dei migliori pattern di cronologia

In questa sezione ci occupiamo di mostrare l'implementazione delle funzioni che sono coinvolte nella scelta del pattern di cronologia che verrà passato in input all'algoritmo di programmazione dinamica. I pattern calcolati nella sezione precedente non hanno particolari vincoli. Ad esempio, `history_patterns` potrebbe contenere pattern con una sola canzone o pattern relativi a porzioni di cronologia molto vecchie. Per questo, abbiamo sviluppato delle funzioni che si occupano di scegliere, per un dato periodo di un giorno della settimana, il miglior pattern di cronologia possibile. Il pattern scelto sarà utilizzato come input per l'algoritmo di programmazione dinamica. Data un'ora di un giorno della settimana, in base a quanti pattern sono presenti e a quante canzoni sono presenti in ciascun pattern, sceglio il miglior pattern tramite uno di questi metodi:

- Metodo 1: Uno o più dei pattern presenti sono sufficientemente lunghi presi singolarmente. In questo caso si sceglie un pattern casualmente, ponendo una probabilità di scelta maggiore per i pattern più recenti. Questo metodo è implementato dalla funzione `choose_random_pattern()`.
- Metodo 2: I pattern presenti non sono sufficientemente lunghi presi singolarmente, ma lo sono se sovrapposti. In questo caso si ordinano i pattern presenti in senso di timestamp di riproduzione crescente, fino a quando la lunghezza del pattern risultante non è sufficiente. Questo metodo è implementato dalla funzione `overlap_patterns()`.
- Metodo 3: Non sono presenti pattern oppure i pattern presenti non sono sufficientemente lunghi, né se presi singolarmente e né se vengono sovrapposti. In questo caso, si tenta di scegliere il pattern valido (ottenuto tramite uno dei due precedenti metodi) cronologicamente più vicino al periodo di riferimento. Questo metodo è implementato dalla funzione `compute_closest_pattern()`.

La funzione principale, `compute_best_history_patterns()`, prende in input il dizionario `periods`, per calcolare i pattern di cronologia tramite la funzione `compute_history_patterns()`, la stringa `timestamp_key`, per accedere correttamente al timestamp di ascolto delle canzoni, il nome di un giorno `day_name` e la stringa `prefix_name`, che identifica univocamente i file binari associati ad una determinata cronologia di ascolto. La funzione restituisce il dizionario `best_history_patterns` che contiene, per tutte le ore del giorno `day_name`, il pattern di cronologia scelto.

Ecco come funziona `compute_best_history_patterns()`:

```

1 def compute_best_history_patterns(periods, day_name,
2     timestamp_key, prefix_name):
3     best_history_patterns = {}
4     best_history_patterns_file_path =
5         os.path.join(data_directory, prefix_name +
6             best_history_patterns_suffix)
7     try:
8         with open(best_history_patterns_file_path, "rb") as file:
9             best_history_patterns = pickle.load(file)
10            return best_history_patterns
11    except Exception:
12        pass

```

```

1 history_patterns = compute_history_patterns(periods,
2     timestamp_key)
3 for period, patterns in history_patterns.items():
4     playlist_length = 48
5     chosen_pattern = None
6     today_patterns = patterns.get(day_name, None)
7     if today_patterns:
8         valid_patterns = [pattern for pattern in
9             today_patterns if len(pattern) >= playlist_length]
10        if valid_patterns:
11            chosen_pattern =
12                choose_random_pattern(valid_patterns,
13                    timestamp_key)
14        else:
15            while playlist_length > 0 and not chosen_pattern:
16                if sum([len(pattern) for pattern in
17                    today_patterns]) >= playlist_length:
18                    chosen_pattern =
19                        overlap_patterns(today_patterns,
20                            timestamp_key)
21                    playlist_length -= 1
22            else:
23                chosen_pattern =
24                    compute_closest_pattern(period, history_patterns,
25                        day_name, timestamp_key)
26        if chosen_pattern:
27            best_history_patterns[period] = chosen_pattern
28    if not best_history_patterns:
29        sys.exit()
30    else:
31        with open(best_history_patterns_file_path, "wb") as file:
32            pickle.dump(best_history_patterns, file)
33 return best_history_patterns

```

1. Si inizializza `best_history_patterns` a un dizionario vuoto.
2. Si definisce `best_history_patterns_file_path` come la stringa contenente il percorso del file binario in cui sono memorizzati i periodi.
3. Si entra in un blocco try-catch nel quale si tenta di aprire in lettura il file relativo al percorso appena calcolato.

4. Si assegna a `best_history_patterns` l'output del modulo `load` della libreria `pickle` sul file binario.
5. Se, durante il caricamento o l'apertura del file, viene sollevata un'eccezione, la funzione proseguirà, altrimenti, si restituirà il dizionario `best_history_patterns`.
6. Si calcolano, tramite la funzione `compute_history_patterns()` tutti i pattern di cronologia e si assegna il risultato a `history_patterns`.
7. Si itera sulle coppie (`period, patterns`) all'interno di `history_patterns`.
8. Si inizializza a `None` la variabile `chosen_pattern`, che rappresenta il pattern scelto per il periodo `period`. Inoltre, si imposta `playlist_length` a 48.
9. Si verifica se il dizionario `patterns` su cui si sta iterando contiene la chiave relativa al giorno `day_name`.
10. Se la condizione è verificata, si assegna il valore ottenuto a `today_patterns` e si calcolano i pattern validi `valid_patterns` tramite una list comprehension, includendo soltanto i pattern in `today_patterns` la cui lunghezza supera il valore della variabile globale `playlist_length`, che rappresenta la lunghezza desiderata della playlist.
11. Se `valid_patterns` non è vuoto, si assegna a `chosen_pattern` il risultato della chiamata alla funzione `choose_random_pattern()`, che prende in ingresso la lista `valid_patterns`.
12. Se `valid_patterns` è vuoto, si entra in un ciclo `while` che decrementa, ad ogni iterazione, la variabile `playlist_length`, fino a quando questa non si trova un pattern valido. In altre parole, il ciclo termina quando `chosen_pattern` non è più `None`, oppure quando si azzera `playlist_length`.
13. Si verifica se la somma delle lunghezze dei pattern in `today_patterns` è maggiore o uguale a `playlist_length`. In tal caso, si assegna a `chosen_pattern` il risultato della chiamata alla funzione `overlap_patterns()`, che prende in ingresso la lista `today_patterns` e la stringa `timestamp_key`.
14. Si decrementa di uno `playlist_length`.
15. Se il dizionario `patterns` non contiene la chiave `day_name`, ossia se `today_patterns` è vuoto, si assegna a `chosen_pattern` il risultato della chiamata alla funzione `compute_closest_pattern()`, che prende in ingresso il periodo `period` su cui si sta

iterando, il nome del giorno `day_name`, il dizionario `history_patterns` e la stringa `timestamp_key`.

16. Terminati questi controlli, se `chosen_pattern` non è `None` verrà assegnato, tramite la chiave `period`, a `best_history_patterns`.
17. Se `best_history_patterns` non esiste, si esce forzatamente tramite il modulo `exit()` della libreria `sys`.
18. Altrimenti, si restituisce `best_history_patterns` dopo averlo caricato, tramite `pickle`, sul file aperto in scrittura denotato dal percorso `best_history_patterns_file_path`.

Nelle tre sottosezioni successive vediamo nei dettagli l'implementazione delle funzioni `choose_random_pattern()`, `overlap_patterns()` e `compute_closest_pattern()`.

### 3.10.1 Scelta pseudocasuale del pattern

```
1 def choose_random_pattern(patterns_with_enough_songs,
2     timestamp_key):
3     timestamps = [pattern[0][timestamp_key] for pattern in
4         patterns_with_enough_songs]
5     probabilities = [1 / (i + 1) for i in range(len(timestamps))]
6     chosen_pattern = random.choices(patterns_with_enough_songs,
7         weights=probabilities, k=1)[0]
8     if len(chosen_pattern) > playlist_length:
9         excess = len(chosen_pattern) - playlist_length
10        start_index = excess // 2
11        end_index = start_index + playlist_length
12        chosen_pattern = chosen_pattern[start_index:end_index]
13
14    return chosen_pattern
```

Questa funzione `choose_random_pattern()` seleziona casualmente uno dei pattern dalla lista `valid_patterns`, dove ciascun pattern ha un numero sufficiente (maggiore o uguale a `playlist_length`) di tracce. Ecco cosa fa passo per passo:

1. Si crea una lista di timestamp estratti dal primo elemento di ciascun pattern nella lista `valid_patterns`.
2. Si calcolano le probabilità basate sull'ordine decrescente dei timestamp. Le probabilità sono inverse alla posizione del timestamp nella lista, quindi i timestamp più recenti hanno una probabilità più alta di essere scelti.

3. Si utilizza la funzione `random.choices()` per selezionare casualmente uno dei pattern dalla lista `valid_patterns`, assegnando pesi alle probabilità calcolate in base all'ordine decrescente dei timestamp.
4. Se la lunghezza del pattern scelto `chosen_pattern` supera `playlist_length`, vengono selezionati soltanto i `playlist_length` elementi centrali di `chosen_pattern`.
5. Si restituisce il pattern `chosen_pattern[0]` scelto casualmente.

In sintesi, questa funzione seleziona casualmente un pattern tra quelli disponibili, con una probabilità più alta assegnata ai pattern con timestamp più recenti.

### 3.10.2 Sovrapposizione dei pattern

```

1 def overlap_patterns(today_patterns, timestamp_key):
2     chosen_pattern = []
3     for pattern in today_patterns:
4         chosen_pattern.extend(pattern)
5     chosen_pattern.sort(key=lambda x: x[timestamp_key].time())
6     if len(chosen_pattern) > playlist_length:
7         excess = len(chosen_pattern) - playlist_length
8         start_index = excess // 2
9         end_index = start_index + playlist_length
10        chosen_pattern = chosen_pattern[start_index:end_index]
11    return chosen_pattern

```

Questa funzione `overlap_patterns()` sovrappone i pattern dei periodi giornalieri attuali (contenuti nella lista `today_patterns`) per ottenere un pattern più lungo, in modo da raggiungere la lunghezza desiderata (`playlist_length`). Ecco cosa fa la funzione:

1. Si inizializza una lista vuota `chosen_pattern` che conterrà il pattern sovrapposto.
2. Si estende `chosen_pattern` con tutti i pattern `pattern` presenti in `today_patterns`.
3. Le tracce in `chosen_pattern` vengono ordinate in base all'orario di riproduzione, utilizzando la funzione `sort()` e una funzione lambda. Questo assicura che le tracce siano ordinate cronologicamente all'interno del pattern.
4. La lunghezza di `chosen_pattern` viene ridotta alla lunghezza desiderata `playlist_length` selezionando soltanto le tracce centrali.
5. Si restituisce `chosen_pattern`, che ora contiene un unico pattern con tracce ordinate cronologicamente ottenuto tramite la sovrapposizione di pattern più brevi.

### 3.10.3 Scelta del pattern più simile

```
1 def compute_closest_pattern(period, history_patterns, day_name,
2     timestamp_key):
3     ordered_patterns = sorted(history_patterns, key=lambda x:
4         abs(x - period), reverse=True)
5     chosen_pattern = None
6     while ordered_patterns:
7         playlist_length = 48
8         patterns = history_patterns.get(ordered_patterns.pop(), None)
9         today_patterns = patterns.get(day_name, None)
10        if today_patterns:
11            valid_patterns = [pattern for pattern in
12                today_patterns if len(pattern) > playlist_length]
13            if valid_patterns:
14                chosen_pattern =
15                    choose_random_pattern(valid_patterns,
16                        timestamp_key)
17            else:
18                while playlist_length > 0 and not chosen_pattern:
19                    if sum([len(pattern) for pattern in
20                        today_patterns]) >=
21                        playlist_length:
22                            chosen_pattern =
23                                overlap_patterns(today_patterns,
24                                    timestamp_key)
25                            playlist_length -= 1
26
27    return chosen_pattern
```

Questa funzione `compute_closest_pattern()` cerca il pattern valido cronologicamente più vicino al periodo specificato all'interno di `history_patterns`. Ecco cosa fa la funzione:

1. I pattern in `history_patterns` vengono ordinati in base alla distanza temporale dal periodo specificato `period`. Per farlo, viene utilizzata la funzione `sorted()` con un criterio personalizzato basato sulla differenza in valore assoluto tra il timestamp dei pattern e il timestamp di `period`. I pattern più vicini avranno la differenza temporale più piccola. Viene impostato `reverse` a `True` per ordinare i pattern in ordine decrescente di distanza. Il risultato viene memorizzato in `ordered_patterns`.

2. Si inizializza a `None` `chosen_pattern`, che conterrà (eventualmente) il pattern valido più vicino a `period`, e si itera fino a quando `ordered_patterns` non contiene più elementi.
3. `playlist_length` viene inizializzato a 48.
4. Ad ogni iterazione, si ottengono i pattern `patterns` relativi al periodo ottenuto tramite `ordered_patterns.pop()`. I pattern ottenuti vengono memorizzati in `patterns`.
5. Si cerca di estrarre da `patterns` i pattern relativi al giorno di nome `day_name`, e si memorizza il risultato in `today_patterns`.
6. Se `today_patterns` esiste, si calcolano i pattern validi `valid_patterns` tramite una list comprehension, includendo soltanto i pattern in `today_patterns` la cui lunghezza supera il valore della variabile globale `playlist_length`, che rappresenta la lunghezza desiderata della playlist.
7. Se `valid_patterns` non è vuoto, si assegna a `chosen_pattern` il risultato della chiamata alla funzione `choose_random_pattern()`, che prende in ingresso la lista `valid_patterns`.
8. Se `valid_patterns` è vuoto, si entra in un ciclo while nel quale si decrementa ad ogni iterazione `playlist_length` e che termina quando si trova un pattern valido `chosen_pattern` o quando `playlist_length` si azzera.
9. Si verifica se la somma delle lunghezze dei pattern in `today_patterns` è maggiore o uguale a `playlist_length`. In tal caso, si assegna a `chosen_pattern` il risultato della chiamata alla funzione `overlap_patterns()`, che prende in ingresso la lista `today_patterns`.
10. Si decrementa di uno `playlist_length`.
11. Si restituisce `chosen_pattern`.

In questo modo, ad ogni iterazione, si cercherà di ottenere un pattern valido a partire dal periodo più cronologicamente vicino a `period`. Se non ci si riesce, si passa ad esaminare quelli progressivamente più lontani, fino a quando non ci sono più periodi da esaminare o fino a quando non viene trovato il primo pattern valido.

### 3.11 Ordinamento dei song-set

Ora che abbiamo generato i song-set, possiamo applicare a ciascuno di essi l'algoritmo di programmazione dinamica, spiegato nella sezione 2.6). L'implementazione di tale algoritmo avviene in un'unica funzione `compute_optimal_solution_indexes()`, che prende

in input i pattern di cronologia `history_patterns`, i pattern di playlist `playlist_patterns`, l'oggetto `spotify` e restituisce il dizionario `optimal_solutions_indexes`. Tale dizionario ha come chiavi i periodi per cui si vuole generare la playlist e come valori le liste di indici che rappresentano l'ordinamento ottimale del pattern di playlist di ciascun periodo. Si noti che per calcolare i pattern di playlist `playlist_patterns` e i pattern di cronologia `history_patterns` abbiamo utilizzato la funzione `compute_listening_history()`. Per quanto riguarda i pattern di playlist, alla funzione `compute_listening_history()` è stato passato il dizionario `song_sets` prodotto in output dalla prima fase di SMART. Il codice funziona in questo modo:

```

1 def compute_optimal_solution_indexes(history_patterns,
2     playlist_patterns, spotify):
3     optimal_solutions_indexes = {}
4     for period, song_set in playlist_patterns.items():
5         if not history_patterns[period]:
6             continue
7         history_pattern_features =
8             get_features(history_patterns[period], spotify)
9         song_set_features = get_features(song_set, spotify)
10        m = len(playlist)
11        k = len(history_pattern)
12        if k > m:
13            history_pattern = history_pattern[0:m]
14            k = m

```

1. Si inizializza `optimal_solutions_indexes` a un dizionario vuoto.
2. Si itera sulle coppie (`period, song_set`) all'interno del dizionario `playlist_patterns`. Questo è il ciclo esterno. Andremo ad applicare l'algoritmo di programmazione dinamica alla song-set di ogni periodo presente nel dizionario `playlist_patterns`.
3. Si controlla se il dizionario `history_patterns` non contiene la chiave `period`. In tal caso iteriamo sul periodo successivo.
4. Si recuperano le caratteristiche audio del pattern di cronologia `history_patterns[period]` e del pattern di playlist `song_set`. Per farlo, si utilizza la funzione `get_features()`. Gli output di `get_features()` vengono assegnati a `history_pattern_features` e `song_set_features`.
5. Poniamo `m` pari alla lunghezza di `song_set_features` e `k` pari alla lunghezza di `history_pattern_features`.

6. Siccome  $k$  deve essere minore o uguale a  $m$ , operiamo in questo modo: se  $k$  è maggiore di  $m$ , prendiamo in considerazione soltanto le prime  $m$  canzoni di `history_pattern[period]`. In tutti gli altri casi, ossia quando  $k$  è minore o uguale a  $m$ , prendiamo in considerazione tutte le canzoni di `history_pattern[period]`. Alla fine di questo blocco di codice, la variabile `history_pattern` conterrà il pattern di cronologia relativo al periodo `period`, e la sua cardinalità  $k$  sarà al più uguale a  $m$ .

```

1  M = []
2  V = []
3  M_first_row = [euclidean([value for value in
4      history_pattern_features[0].values() if not
5      isinstance(value, str)], [value for value in
6      song_set_features[j].values() if not
7      isinstance(value, str)]) for j in range(m)]
8  V_first_row = [j for j in range(m)]
9  M.append(M_first_row)
10 V.append(V_first_row)

```

7. Si inizializzano  $M$  e  $V$  a una lista vuota.  $M$  conterrà i valori della soluzione ottima per ciascun sottoproblema,  $V$  conterrà gli indici che hanno prodotto tale soluzione ottima. Entrambe le variabili sono matrici che hanno  $k$  righe e  $m$  colonne.
8. La prima riga di  $M$ , `M.first_row`, è calcolata tramite una list comprehension, seguendo la formula  $M[1, j] = d(h_1, c_j)$ , for  $j = 1, \dots, m$ ; In questo contesto,  $d(h_1, c_j)$  è la distanza euclidea tra `history_pattern_features[0]` e `song_set_features[j]`.
9. La prima riga di  $V$ , `V.first_row`, non verrà utilizzata, ed è costituita da valori interi crescenti nel range da 0 a  $m-1$ .
10. Si aggiunge la prima riga, rispettivamente `M.first_row` e `V.first_row`, alle matrici  $M$  e  $V$ .

```

1  for i in range(1,k):
2      M_i_th_row = []
3      V_i_th_row = []
4      for j in range(m):

```

11. Si itera per valori di  $i$  interi crescenti nel range da 1 a  $k-1$ .
12. Si inizializzano `M.i_th_row` ( $i$ -esima riga di  $M$ ) e `V.i_th_row` ( $i$ -esima riga di  $V$ ) a una lista vuota.

13. Si itera per valori di  $j$  interi crescenti nel range da 0 a  $m-1$ .

```

1      distance = euclidean([value for value in
2          history_pattern_features[i].values() if not
3              isinstance(value, str)], [value for value in
4                  song_set_features[j].values() if not
5                      isinstance(value, str)])
6
7      playlist_pattern_distances = [M[i-1][t] +
8          abs((euclidean([value for value in
9              history_pattern_features[i-1].values() if not
10             isinstance(value, str)], [value for value in
11                 history_pattern_features[i].values() if not
12                     isinstance(value, str)]) - euclidean([value
13                         for value in song_set_features[t].values() if not
14                             isinstance(value, str)], [value for value
15                             in song_set_features[j].values() if not
16                                 isinstance(value, str)]))) for t in range(m)]
17
18      min_distance = min(playlist_pattern_distances)
19      min_index =
20          playlist_pattern_distances.index(min_distance)
21      M_i_th_row.append(distance + min_distance)
22      V_i_th_row.append(min_index)
23
24      M.append(M_i_th_row)
25      V.append(V_i_th_row)

```

Si procede calcolando tutti gli altri  $M[i, j]$ , applicando la formula:

$$M[i, j] = d(h_i, c_j) + \min_{1 \leq t \leq m} \{M[i - 1, t] + |d(h_{i-1}, h_i) - d(c_t, c_j)|\}.$$

14.  $d(h_i, c_j)$ , ossia la distanza euclidea tra `history_pattern_features[i]` e `song_set[j]`, è calcolato tramite una list comprehension e viene memorizzato in `distance`.
15. Memorizziamo le Playlist Pattern Distances all'interno della variabile `playlist_pattern_distances`. Questa variabile è ottenuta tramite una list comprehension, dove, per valori crescenti di  $t$  nel range da 0 a  $m-1$ :
- (a) Si calcola la distanza euclidea tra `history_pattern_features[i-1]` e `history_pattern_features[i]`.
  - (b) Si calcola la distanza euclidea tra `song_set_features[t]` e `song_set_features[j]`.
  - (c) Si sottraggono questi due valori.

- (d) Si calcola il valore assoluto della sottrazione.
- (e) Si somma  $M[i-1][t]$  al valore ottenuto.
16. Si calcola il valore `min_distance` che minimizza `playlist_pattern_distances` e l'indice `t` corrispondente `min_index`.
  17. Si aggiunge a `M_i_th_row` la somma `distance + min_distance`.
  18. Si aggiunge a `V_i_th_row` l'indice `min_index`.
  19. Dopo aver iterato per valori di `j` interi crescenti nel range da 0 a `m-1`, Si aggiunge `M_i_th_row` a `M` e si aggiunge `V_i_th_row` a `V`.

```

1     min_value = min(M[k-1])
2     min_index = M[k-1].index(min_value)
3     vertices_period = [min_index]
4     duplicate_indexes = [0]
5     for i in range(k-2, -1, -1):
6         t = vertices_period[0]
7         min_index = V[i+1][t]
8         if min_index in vertices_period:
9             duplicate_indexes.insert(0, 1)
10        else:
11            duplicate_indexes.insert(0, 0)
12            vertices_period.insert(0, min_index)
13    optimal_solutions_indexes[period] =
14        (vertices_period, duplicate_indexes)
15    if not optimal_solutions_indexes:
16        sys.exit()
16 return optimal_solutions_indexes

```

Infine, dobbiamo recuperare i vertici di ciascuna soluzione ottima, ricordando che:

- Il vertice  $p_k^*$  (l'ultimo) è il candidato  $c_r \in C$  tale che  $M[k, r] = \min_{1 \leq j \leq m} M[k, j]$ .
  - Per ogni  $i = k-1, \dots, 1$ , sia  $c_t$  il candidato selezionato per la posizione  $i+1$  nella soluzione, quindi  $r = V[i+1, t]$  è l'indice del candidato per la posizione  $i$ , cioè abbiamo che  $p_i^* = c_r$ .
20. Si recupera il valore `min_value` relativo all'ultima riga della matrice `M`, e il relativo indice `min_index`. L'indice `min_index`, che costituisce l'ultimo vertice della soluzione ottima, viene aggiunto alla lista `vertices_period`.

21. Si inizializza la lista `duplicate_indexes`. Tale lista ha una lunghezza pari a quella di `vertices_period`, e, in ciascuna posizione, conterrà uno 0 o un 1 in base a sé l'indice nella medesima posizione all'interno di `vertices_period` è duplicato o meno. Le canzoni corrispondenti agli indici in `vertices_period` per cui è presente un 1 all'interno di `duplicate_indexes` verranno sostituite con una canzone simile.
22. Per recuperare gli altri vertici, si itera per valori decrescenti di `i` nel range da `k-2` a 0.
23. Si recupera l'indice `t`, ossia il candidato in posizione `i+1` nella soluzione.
24. Si recupera il vertice `min_index`, ossia il candidato per la posizione `i` nella soluzione.
25. L'indice `min_index` viene aggiunto in prima posizione all'interno di `vertices_period`. Se `min_index` è duplicato, si aggiunge un 1 nella medesima posizione di `duplicate_indexes`, altrimenti si aggiunge uno 0.
26. Si assegna la coppia (`vertices_period,duplicate_indexes`) al dizionario `optimal_solutions_indexes` utilizzando la chiave `period`.
27. Una volta usciti dal ciclo più esterno, si restituisce il dizionario `optimal_solutions_indexes`

Rammentiamo che il dizionario `optimal_solutions_indexes`, il quale rappresenta l'output di questa funzione, associa a ogni periodo `period` per cui è stato generato un insieme di brani, l'ordinamento ottimale di quest'ultimo. Tale disposizione è progettata per garantire la massima personalizzazione.

### 3.12 Recupero delle canzoni ordinate

Una volta che è stato stabilito l'ordinamento ottimale di ciascun song-set, utilizziamo il dizionario `optimal_solutions_indexes` prodotto in output dalla funzione `compute_optimal_solution_indexes()`, per recuperare gli id delle canzoni ordinate e per sostituire quelle duplicate. Il recupero degli id delle canzoni è necessario in quanto i valori del dizionario

`optimal_solutions_indexes` sono liste di posizioni. A tal proposito, la funzione `retrieve_optimal_solution_songs()` prende in input i pattern di playlist `playlist_patterns`, il dizionario `optimal_solutions_indexes`, l'oggetto `spotify` e restituisce il dizionario `playlists`. Il dizionario `playlists` associa una lista di id di canzoni ad ogni periodo per cui esiste un ordinamento del relativo song-set. Ecco i dettagli implementativi della funzione `retrieve_optimal_solution_songs()`:

```

1 def retrieve_optimal_solution_songs(optimal_solutions_indexes,
2     playlist_patterns, spotify):
3     playlists = {}
4     for period, songs in playlist_patterns.items():
5         if optimal_solutions_indexes.get(period, None):
6             vertices_period =
7                 optimal_solutions_indexes[period][0]
8             duplicate_indexes =
9                 optimal_solutions_indexes[period][1]
10            n_duplicate_indexes = len([index for index in
11                duplicate_indexes if index == 1])
12            playlist_length = len(vertices_period)
13            limit = playlist_length + n_duplicate_indexes
14            playlist = [songs[index]['id'] for index in
15                vertices_period]
16            for i in range(playlist_length):
17                if i == 0 or i == 1:
18                    ids = playlist[0:min(5, len(playlist))]
19                elif i == len(playlist) - 1 or i ==
20                    len(playlist) - 2:
21                    ids = playlist[max(0, len(playlist) - 5):]
22                else:
23                    ids = [playlist[max(0, i-2)],
24                        playlist[max(0, i-1)], playlist[i],
25                        playlist[min(len(playlist) - 1, i+1)],
26                        playlist[min(len(playlist) - 1, i+2)]]
27                if duplicate_indexes[i]:
28                    recommendations =
29                        spotify.recommendations(seed_tracks=ids,
30                            limit=limit).get('tracks')
31                    for recommendation in recommendations:
32                        if recommendation['id'] not in playlist:
33                            playlist[i] = recommendation['id']
34                            break
35            playlists[period] = playlist
36    return playlists

```

1. Si inizializza `playlists` a un dizionario vuoto.
2. Si itera sulle coppie (`period, songs`) all'interno del dizionario `playlist_patterns` per cui esiste un ordinamento ottimale prodotto da `compute_optimal_solution_indexes()`.

3. Si recuperano le liste `vertices_period` e `duplicate_indexes` dal dizionario `optimal_solution_indexes` relative al periodo `period` su cui si sta iterando.
4. Si calcola il numero di indici duplicati `n_duplicate_indexes` e si somma a questo valore `playlist_length`, ottenuto calcolando la lunghezza di `vertices_period`. Il risultato di questa somma viene memorizzato nella variabile `limit`, che verrà utilizzata per effettuare le richieste di raccomandazioni alla API di Spotify.
5. Si crea, tramite una list comprehension, la lista `playlist`, che contiene gli id `song[index]['id']` per ogni indice `index` presenti nella lista `optimal_solution_indexes[period]`. Ricordando che tale lista rappresenta l'ordinamento ottimale delle canzoni presenti nel song-set relativo al periodo `period`.
6. Si itera per valori interi di `i` compresi tra 0 e `playlist_length - 1`.
7. Si calcolano gli id delle tracce `ids` da passare in input al sistema di raccomandazioni di Spotify relativamente a ciascuna canzone duplicata in `vertices_period`. Siccome il limite massimo di canzoni che possono essere passate al sistema di raccomandazioni di Spotify è 5, in `ids` vengono inseriti gli id delle due canzoni precedenti a `i`, gli id delle due successive e l'id di `i` stessa. Ovviamente, ciò è fattibile soltanto se `i` non è agli estremi del range che va da 0 a `playlist_length - 1`: in quei casi si prendono, rispettivamente, gli id delle prime 5 canzoni e gli id delle ultime 5 canzoni di `vertices_period`.
8. Se è presente un 1 nella posizione `i` di `duplicate_indexes`, si utilizza `spotify` per effettuare la chiamata API che restituisce la lista `recommendations`. Si itera su tale lista e si sostituisce l'id dell'indice duplicato in posizione `i` di `playlist` con l'id della prima canzone in `recommendations` che non è già presente in `playlist`. Si assegna la lista di id `playlist` al dizionario `playlists`, utilizzando la chiave `period`.
9. Si restituisce il dizionario `playlists`.

Una volta ottenuto il dizionario `playlists`, possiamo salvarlo localmente per comparare le playlist generate con quelle prodotte dai metodi sperimentali e possiamo utilizzarlo per caricare le playlist su Spotify.

### 3.13 Salvataggio delle playlist

Per prepararci alla fase di sperimentazione, abbiamo bisogno di identificare univocamente ciascuna playlist generata per ciascun file di cronologia `StreamingHistory.json`. Per farlo, utilizziamo la funzione `create_playlists_dict()`. La funzione prende in input il dizionario `playlists` restituito da `retrieve_optimal_solution_songs()`, il nome del

giorno `day_name` per cui sono state generate le playlist, il dizionario dei pattern di cronologia `history_patterns` e la stringa `prefix` utilizzata per identificare univocamente il file `StreamingHistory.json`.

```
1 def create_playlists_dict(playlists, day_name, history_patterns,
2     prefix):
3     playlists_dict = {}
4     for period, playlist in playlists.items():
5         playlists_dict[(prefix, day_name, period, "SMART")] =
6             (playlist, history_patterns[period])
7
8     with open("data/playlists.bin", "ab") as file:
9         pickle.dump(playlists, file)
10
11 return playlists_dict
```

1. Si inizializza `playlists_dict` a un dizionario vuoto.
2. Si itera sulle coppie (`period,playlist`) all'interno del dizionario `playlists_dict`.
3. Le chiavi di `playlists_dict` sono quadruple del tipo (`prefix, day_name, period, nome_methodo`). Nel caso delle playlist generate con SMART, la stringa identificativa del metodo sarà "SMART".
4. I valori di `playlists_dict` sono invece coppie (`playlist, history_pattern[period]`. `history_patterns[period]` costituisce il pattern di playlist utilizzato per calcolare l'ordinamento ottimale delle canzoni, esso viene incluso per permetterci, durante la fase di sperimentazione, di calcolare agevolmente la Playlist Pattern Distance tra la playlist generata e il pattern di playlist corrispondente.
5. Alla fine del ciclo, si apre in modalità append il file binario che conterrà tutte le playlist generate e si aggiungono quelle contenute in `playlists_dict`.
6. `playlists_dict` viene restituito per poter caricare le playlist appena generate su Spotify.

Una funzione analoga a questa è presente all'interno di `other_methods.py`. Tale funzione si occupa di creare dizionari strutturati allo stesso modo, ma con una stringa diversa che identifica il metodo con cui è stata generata la playlist. La funzione `create_playlists()` che si occupa di caricare le playlist su Spotify è molto semplice:

```

1 def create_playlists(playlists, spotify):
2     user_id = spotify.current_user()['id']
3     for data, tracks in playlists.items():
4         playlist_name =
5             f"{data[0]}_{data[1]}_{data[2]}:00_{data[3]}"
6             playlist = spotify.user_playlist_create(user_id,
7                 playlist_name, public=False)
8             spotify.playlist_add_items(playlist['id'], tracks[0])

```

1. La funzione prende in input il dizionario `playlists`, che associa a ciascun periodo la relativa lista di id di canzoni ordinate in maniera ottimale, e l'oggetto `spotify`.
2. Si memorizza in `user_id` l'id dell'utente che è attualmente autenticato, chiamando il metodo `current_user()` ed estraendone l'id.
3. Si itera sulle coppie (`data, tracks`) all'interno del dizionario `playlists`.
4. Si ottiene il nome della playlist `playlist_name` combinando i valori della quadrupla che identifica ciascuna playlist.
5. Si invoca sull'oggetto `spotify` il metodo `user_playlist_create()`, che carica una playlist privata su Spotify nominata opportunamente utilizzando l'id dell'utente corrente `user_id`. Il risultato di questa chiamata è memorizzato all'interno della variabile `playlist`.
6. Si utilizza il metodo `playlist_add_items()` per aggiungere alla playlist `playlist` appena creata gli id delle canzoni presenti in `tracks[0]`.

## 3.14 Metodi alternativi

Per valutare l'efficacia di SMART, abbiamo deciso di metterlo a confronto con i metodi REC-1, REC-2 e HYB-1, descritti nella Sezione 4.1.2. Nelle sezioni seguenti forniremo le implementazioni di questi metodi, contenuti nel modulo `other_methods`.

### 3.14.1 REC-1

La funzione che implementa il metodo REC-1, `get_rec_1_recommendations()`, prende in input due oggetti: la cronologia di ascolto dell'utente `listening_history` e la lunghezza `playlist_length` desiderata della playlist. La funzione restituisce un dizionario `playlists` che associa una playlist `playlist`, rappresentata da una lista di id di canzoni, a ciascun periodo `period`. Vediamo come funziona:

```
1 def get_rec_1_recommendations(listening_history, playlist_length):
2     playlists = {}
3     for period, songs in listening_history.items():
4         playlist = []
5         i = 1
6         for song in songs:
7             if i % 12 == 0 or i == 1:
8                 spotify = change_credentials()
9                 try:
10                     recommendation =
11                         get_recommendations(spotify=spotify,
12                                         seed_tracks=[song['id']], limit=100)
13                 except IndexError:
14                     continue
```

1. Si inizializza `playlists` a un dizionario vuoto.
2. Si itera sulle coppie (`period, songs`) all'interno della cronologia di ascolto `listening_history`.
3. Si inizializza `playlist` a una lista vuota.
4. Si itera su ciascuna canzone `song` all'interno della lista `songs` di canzoni relative al periodo `period` su cui si sta iterando. Per evitare di effettuare troppe chiamate consecutive alla API di Spotify, scegliamo casualmente delle nuove credenziali ogni 12 iterazioni, tramite la funzione `change_credentials`, importata dal modulo `listening_history_manager`.

5. Si effettua la chiamata API al sistema di raccomandazioni di Spotify, utilizzando la funzione `get_recommendations()`, importata dal modulo `listening_history_manager`. A questa funzione passiamo in input l'id `song['id']` della canzone `song`.
6. Qualore il blocco try-except catturasse l'eccezione `IndexError` nel tentare di recuperare le raccomandazioni, si passa alla prossima canzone `song` nella cronologia filtrata.
7. Memorizziamo l'output in `recommendation`.

```

1      if recommendation:
2          try:
3              recommendation =
4                  recommendation.get('tracks')[0]
5          except IndexError:
6              pass
7          else:
8              if recommendation['id'] not in
9                  list(set(playlist)):
10                 playlist.append(recommendation['id'])
11                 i += 1
12                 if playlist_length == len(playlist):
13                     playlists[(period[0], period[1])] =
14                         playlist
15                     break
16             playlists[(period[0], period[1])] = playlist
17             return playlists

```

8. Verifichiamo che l'oggetto `recomendation` esista. In tal caso, estraiamo il primo elemento della lista di canzoni ottenuta da `recommendation` accedendovi tramite la chiave `tracks`. Assegniamo l'oggetto ottenuto a `recommendation`.
9. Aggiungiamo alla playlist l'id della canzone ottenuta soltanto se questa non è già presente all'interno della playlist `playlist`.
10. Continuiamo fino ad ottenere un numero di canzoni `len(playlist)` pari alla dimensione desiderata `playlist_length` o fino a quando non si esauriscono le canzoni `songs` nella cronologia del periodo `period`.
11. Alla fine, la playlist `playlist` generata viene assegnata al dizionario `playlists` utilizzando come chiave la coppia `(period[0], period[1])`, dove `period[0]` è una stringa che identifica il giorno della settimana e `period[1]` è un intero che identifica l'ora del giorno.

12. Si restituisce il dizionario `playlists`.

### 3.14.2 REC-2

La funzione che implementa il metodo REC-2, `get_rec_2_recommendations()`, prende in input la cronologia di ascolto filtrata dell'utente `listening_history` e la lunghezza `playlist_length` desiderata della playlist. La funzione restituisce un dizionario `playlists` che associa una playlist `playlist`, rappresentata da una lista di id di canzoni, a ciascun periodo `period`. Vediamo come funziona:

```

1 def get_rec_2_recommendations(listening_history, playlist_length):
2     playlists = {}
3     for period, songs in listening_history.items():
4         playlist = []
5         j = 1
6         for i, song in enumerate(songs):
7             if i == 0:
8                 track_ids = [song, songs[i + 1]]
9             elif i == len(songs) - 1:
10                 track_ids = [songs[i - 1], song]
11             else:
12                 track_ids = [songs[i - 1], song, songs[i + 1]]
13             if j % 12 == 0 or j == 1:
14                 spotify = change_credentials()
15                 recommendation =
16                     get_recommendations(spotify=spotify,
17                     seed_tracks=[track['id'] for track in track_ids],
18                     limit=100)

```

1. Si inizializza `playlists` a un dizionario vuoto.
2. Si itera sulle coppie (`period, songs`) all'interno della cronologia di ascolto filtrata `listening_history`.
3. Si inizializza `playlist` a una lista vuota.
4. Si itera su ciascuna canzone `song` all'interno della lista `songs` di canzoni relative al periodo `period` su cui si sta iterando. Utilizziamo `track_ids` per memorizzare gli id delle tracce da utilizzare come input per il sistema di raccomandazioni di Spotify. Durante la prima iterazione, memorizziamo in `track_ids` gli id delle prime due canzoni della cronologia filtrata, durante l'ultima iterazione, memorizziamo le ultime due. Durante tutte le altre iterazioni, `track_ids` conterrà la canzone corrente (indice `i`), quella successiva (indice `i+1`) e quella precedente (indice `i-1`). Per evitare di

effettuare troppe chiamate consecutive alla API di Spotify, scegliamo casualmente delle nuove credenziali ogni 12 iterazioni, tramite la funzione `change_credentials`, importata dal modulo `listening_history_manager`.

5. Si effettua la chiamata API al sistema di raccomandazioni di Spotify, utilizzando la funzione `get_recommendations()`, importata dal modulo `listening_history_manager`. A questa funzione passiamo gli id `track_ids`. Memorizziamo l'output in `recommendation`.

```
1     if recommendation:
2         try:
3             recommendation =
4                 recommendation.get('tracks')[0]
5         except IndexError:
6             continue
7         else:
8             if recommendation['id'] not in
9                 list(set(playlist)):
10                playlist.append(recommendation['id'])
11                j += 1
12                if playlist_length == len(playlist):
13                    playlists[(period[0], period[1])] =
14                        playlist
15                    break
16
17            playlists[(period[0], period[1])] = playlist
18
19    return playlists
```

6. Verifichiamo che `recommendation` esista. In tal caso, estraiamo il primo elemento della lista di canzoni ottenuta da `recommendation` accedendovi tramite la chiave `tracks`. Asegniamo l'oggetto ottenuto a `recommendation`. Se questa istruzione solleva l'eccezione `IndexError`, passiamo al prossimo blocco di canzoni nella cronologia.
7. Altrimenti, aggiungiamo alla playlist l'id della canzone ottenuta soltanto se questa non è già presente all'interno della playlist `playlist`.
8. Continuiamo fino ad ottenere un numero di canzoni `len(playlist)` pari alla dimensione desiderata `playlist_length` o fino a quando non si esauriscono le canzoni `songs` nella cronologia del periodo `period`.
9. Alla fine, la playlist `playlist` generata viene assegnata al dizionario `playlists` utilizzando come chiave la coppia `(period[0], period[1])`.
10. Si restituisce il dizionario `playlists`.

### 3.14.3 HYB-1

La funzione che implementa il metodo HYB-1, `get_hyb_1_recommendations()`, prende in input tre oggetti: la cronologia di ascolto filtrata dell'utente `listening_history`, il dizionario dei pattern di cronologia `history_patterns` e la lunghezza desiderata della playlist `playlist_length`. La funzione restituisce il dizionario `ordered_playlists`, che associa una playlist `playlist`, rappresentata da una lista di id di canzoni, a ciascun periodo `period`.

Vediamo come funziona:

```
1 def get_hyb_1_recommendations(listening_history,
2     history_patterns, playlist_length):
3     playlists = {}
4     for period, songs in listening_history.items():
5         playlist = []
6         blocks = [songs[i:i+5] for i in range(0, len(songs), 5)
7                 if len(songs[i:i+5]) == 5]
8         if len(songs) % 5 != 0:
9             blocks.append(songs[(len(songs) // 5) * 5:])
10        rec_limit = playlist_length / len(blocks)
11        if int(rec_limit) != rec_limit:
12            rec_limit = math.ceil(rec_limit)
13        length = False
14        progress = 0
```

1. Si inizializza `playlists` a un dizionario vuoto.
2. Si itera attraverso le coppie `period, songs` all'interno del dizionario `listening_history`.
3. Si inizializza `playlist` ad una lista vuota.
4. Ad ogni iterazione, si calcolano gli id di 5 canzoni consecutive, e si memorizza il risultato all'interno della lista `blocks`. Se il numero di canzoni presenti nella cronologia non è un multiplo di cinque, l'ultimo blocco sarà costituito dalle canzoni rimanenti.
5. Si calcola il numero di canzoni simili da chiedere al sistema di raccomandazioni di Spotify. Per calcolare questo valore, arrotondiamo all'intero superiore il risultato della divisione tra la lunghezza della playlist `playlist_length` e il numero di blocchi `len(blocks)`. Memorizziamo il risultato ottenuto in `rec_limit`.
6. Inizializziamo il flag `length` a `False`. Questo flag diventerà `True` quando la lunghezza della playlist generata diventerà pari alla lunghezza desiderata, evitando di aggiungere più canzoni del dovuto.

```

1   for block in blocks:
2       spotify=change_credentials()
3       if length:
4           playlists[period] = playlist
5           break
6       recommendations =
7           get_recommendations(spotify=spotify,
8               seed_tracks=[track['id'] for track in block],
9               limit=100)

```

7. si itera su ciascun blocco di canzoni `block` all'interno della lista `blocks`.
8. Ad ogni blocco, si inizializza l'oggetto `spotify`, utilizzato per ottenere le raccomandazioni, tramite la funzione `change_credentials`, importata dal modulo `listening_history_manager`.
9. Si verifica se il flag `length` è True. In tal caso, si assegna a `playlists`, la playlist `playlist` generata per il periodo `period`. Si esce forzatamente dal ciclo. Altrimenti, si memorizza in `recommendations`, l'output della funzione `get_recommendations()`, importata dal modulo `listening_history_manager`. A questa funzione vengono passati gli id delle canzoni presenti nel blocco `block`.

```

1   if recommendations:
2       recommendations = recommendations.get('tracks')
3       count = 0
4       for recommendation in recommendations:
5           if recommendation['id'] not in
6               list(set(playlist)):
7                   playlist.append(recommendation['id'])
8                   count += 1
9                   if playlist_length == len(playlist):
10                      length = True
11                      break
12                      if count == rec_limit:
13                          break
14      playlists[period] = playlist

```

10. Si verifica che `recommendations` esista. In tal caso, si assegna a `recommendations` la lista di canzoni ottenuta accedendo a `recommendations` tramite la chiave `tracks`. Si inizializza a 0 il contatore `count` di canzoni aggiunte alla playlist `playlist` a partire dal blocco `block` su cui si sta iterando.

11. Si itera su ciascuna raccomandazione `recommendation` presente nella lista `recommendations`.
12. Aggiungiamo alla playlist l'id della canzone soltanto se questa non è già presente all'interno della playlist `playlist`. Incrementiamo il contatore `count`.
13. Se il numero di canzoni `len(playlist)` è pari alla dimensione desiderata `playlist_length`, si esce forzatamente dal ciclo e si imposta il flag `length` a True.
14. Se il contatore `count` è pari a `rec_limit`, usciamo forzatamente dal ciclo interno e passiamo al prossimo blocco.
15. Una volta terminato il ciclo for che itera sui blocchi, qualora la dimensione della playlist desiderata non fosse raggiunta, si assegna comunque a `playlists` la playlist `playlist` generata, utilizzando come chiave il periodo `period`.

```

1  playlist_patterns = {}
2  for period, playlist in playlists.items():
3      while True:
4          try:
5              features = get_features(playlist, spotify)
6          except Exception as e:
7              print(e)
8              spotify = change_credentials()
9          else:
10             break
11      playlist_pattern_all_features = list(filter(None,
12          features))
13      playlist_pattern = [{key:value for key,value in
14          song.items() if key not in feature_names_to_remove}
15          for song in playlist_pattern_all_features]
16      playlist_patterns[period] = playlist_pattern

```

16. Si inizializza `playlist_patterns` a un dizionario vuoto.
17. Si itera sulle coppie `period, playlist` all'interno del dizionario `playlists` appena creato.
18. Si entra in un ciclo While infinito.
19. Si entra in un blocco try-except. Si tenta di recuperare le caratteristiche audio dei brani della playlist `playlist` tramite la funzione `get_features` importata dal modulo `listening_history_manager`. Si assegna il risultato a `features`. Se questa istruzione solleva un'eccezione qualunque, si ritenta con delle nuove credenziali. Altrimenti, si esce forzatamente dal ciclo.

20. Si rimuovono eventuali elementi nulli da `features` e si memorizza il risultato in `playlist_pattern_all_features`.
21. Si ricava il pattern di playlist tramite una list comprehension che mira a rimuovere, per ogni canzone in `playlist_pattern_all_features`, le caratteristiche che non ci interessano, memorizzate nella lista `features_names_to_remove`. Si memorizza il risultato in `playlist_pattern`.
22. Si assegna a `playlist_patterns` il pattern `playlist_pattern`, utilizzando la chiave `period`.

```

1  playlists_indexes =
2      compute_optimal_solution_indexes(history_patterns ,
3          playlist_patterns)
4  ordered_playlists =
5      retrieve_optimal_solution_songs(playlists_indexes ,
6          playlist_patterns)
7  return ordered_playlists

```

23. Si chiama la funzione `compute_optimal_solution_indexes()`, importata dal modulo `step2`, passando in input i pattern di cronologia `history_patterns` e i pattern di playlist `playlist_patterns` appena ottenuti. Si memorizza il risultato in `playlists_indexes`.
24. Si chiama la funzione `retrieve_optimal_solution_songs()`, anch'essa importata dal modulo `step2`, passando in input `playlists_indexes` e `playlist_patterns`. Si memorizza il risultato in `ordered_playlists`.
25. Si restituisce il dizionario `ordered_playlists`, che conterrà i song-set memorizzati in `playlist_patterns` ordinati tramite il nostro algoritmo di programmazione dinamica.

### 3.15 Calcolo della Playlist Pattern Distance

In questa sezione mostriamo brevemente il codice utilizzato per calcolare la Playlist Pattern Distance tra le playlist generate e il relativo pattern di playlist utilizzato come input per l'algoritmo di programmazione dinamica. Questo codice verrà utilizzato nella fase sperimentale come metrica principale per comparare i metodi di generazione di playlist.

Per prima cosa, mostriamo le funzioni coinvolte nel calcolo della Segment Distance e della Vertex Distance.

```

1 def compute_segment_distance(generated_playlist_features ,
2     history_pattern_features):
3     sum = 0
4     for i in range(len(generated_playlist_features) - 1):
5         generated_playlist_current_track_features = [feature for
6             feature in generated_playlist_features[i].values() if
7                 not isinstance(feature, str)]
8         generated_playlist_next_track_features = [feature for
9             feature in generated_playlist_features[i+1].values()
10                if not isinstance(feature, str)]
11
12         euclidean_distance_1 =
13             euclidean(generated_playlist_current_track_features ,
14                     generated_playlist_next_track_features)
15
16         history_pattern_current_track_features = [feature for
17             feature in history_pattern_features[i].values() if
18                 not isinstance(feature, str)]
19         history_pattern_next_track_features = [feature for
20             feature in history_pattern_features[i+1].values() if
21                 not isinstance(feature, str)]
22
23         euclidean_distance_2 =
24             euclidean(history_pattern_current_track_features ,
25                     history_pattern_next_track_features)
26
27         sum += abs(euclidean_distance_1 - euclidean_distance_2)
28
29     return sum

```

Per calcolare la Segment Distance, utilizziamo la funzione `compute_segment_distance()`. Questa funzione prende in input le caratteristiche audio di una playlist (`generated_playlist_features`) e del relativo pattern di cronologia (`history_pattern_features`). La funzione restituisce l'oggetto `sum`, che avrà il seguente valore:

$$SD(P_1, P_2) = \sum_{i=1}^{t-1} |d(p_{1,i}, p_{1,i+1}) - d(p_{2,i}, p_{2,i+1})|;$$

dove  $d(\cdot, \cdot)$  è la distanza euclidea e  $t$  è la lunghezza della playlist  $P_1$  e del pattern di cronologia  $P_2$ .

La funzione `compute_segment_distance()` opera in questo modo:

1. Si inizializza `sum` a 0.
2. Si itera sugli indici `i` delle caratteristiche audio delle canzoni `generated_playlist_features` relative alla playlist generata. Come indicato dalla formula, ci fermiamo al penultimo indice.
3. Tramite list comprehension, recuperiamo la lista di valori numerici delle caratteristiche audio della traccia corrente (`generated_playlist_current_track_features`) e di quella successiva (`generated_playlist_next_track_features`).
4. Tramite la funzione `euclidean`, importata dal modulo `scipy.spatial.distance`, si calcola la distanza euclidea (`euclidean_distance_1`) tra le caratteristiche audio della canzone corrente e le caratteristiche audio della canzone successiva.
5. Si ripete il medesimo procedimento per il pattern di cronologia: si estrae la lista di valori numerici delle caratteristiche audio dalla traccia corrente e dalla traccia successiva, e si utilizzano questi valori per calcolare la seconda distanza euclidea `euclidean_distance_2`.
6. Si aggiunge a `sum` la differenza assoluta tra `euclidean_distance_1` e `euclidean_distance_2`.
7. Alla fine del ciclo for, si restituisce `sum`.

```

1 def compute_vertex_distance(playlist_features, pattern_features):
2     sum = 0
3     for i in range(len(generated_playlist_features)):
4         generated_playlist_features = [feature for feature in
5             playlist_features[i].values() if not
6             isinstance(feature, str)]
7         history_pattern_features = [feature for feature in
8             pattern_features[i].values() if not
9             isinstance(feature, str)]
10        sum += euclidean(generated_playlist_features,
11                         history_pattern_features)
12    return sum

```

Per calcolare la Vertex Distance, utilizziamo la funzione `compute_vertex_distance()`. Questa funzione prende in input le caratteristiche audio di una playlist (`playlist_features`) e del relativo pattern di cronologia (`pattern_features`). La funzione restituisce l'oggetto `sum`, che avrà il seguente valore:

$$VD(P_1, P_2) = \sum_{i=1}^t d(p_{1,i}, p_{2,i});$$

dove  $d(\cdot, \cdot)$  è la distanza euclidea e  $t$  è la lunghezza della playlist  $P_1$  e del pattern di cronologia  $P_2$ .

La funzione `compute_vertex_distance()` opera in questo modo:

1. Si inizializza `sum` a 0.
2. Si itera sugli indici `i` delle caratteristiche audio delle canzoni `playlist_features` relative alla playlist generata.
3. Tramite una list comprehension, si recupera la lista di valori numerici delle caratteristiche audio della canzone di indice `i` nella playlist (`generated_playlist_features`) e nel pattern di cronologia `history_pattern_features`.
4. Si aggiunge a `sum` la distanza euclidea tra `generated_playlist_features` e `history_pattern_features`, calcolata tramite `euclidean`.
5. Alla fine del ciclo for, si restituisce `sum`.

Il recupero del dizionario `playlists` avviene in questo modo:

```
1 def retrieve_playlists():
2     playlists = {}
3     try:
4         with open("data/playlists.bin", "rb") as file:
5             try:
6                 while True:
7                     playlists.update(pickle.load(file))
8             except (EOFError):
9                 pass
10        except FileNotFoundError:
11            pass
12    return playlists
```

1. Si inizializza `playlists` a un dizionario vuoto.
2. Si entra in un blocco try-except.
3. Si tenta di aprire in lettura il file binario `data/playlists.bin` che memorizza le playlist generate.
4. Si entra in un secondo blocco try-except che serve a catturare l'eccezione `FileNotFoundException` che verrebbe sollevata in caso il file `data/playlists.bin` non esistesse.
5. Si entra in un ciclo infinito.
6. Si legge riga per riga il file binario, fino a quando non viene sollevata l'eccezione `EOFError`, catturata dal blocco try-except più interno. Per leggere dal file binario, si utilizza il metodo `load` della libreria `pickle`.
7. Si restituisce il dizionario `playlists`

Arrivati a questo punto, siamo pronti a fornire l'implementazione della funzione `compute_playlist_pattern_distances`. Questa funzione prende in input il dizionario `playlists`, recuperato tramite la funzione `retrieve_playlists`, e restituisce il dizionario `results` contenente i valori della Playlist Pattern Distance tra ogni playlist generata e il relativo pattern di cronologia.

```

1 def compute_playlist_pattern_distances(playlists):
2     spotify = change_credentials()
3     results = {}
4     for playlist_data, (generated_playlist, history_pattern) in
5         playlists.items():
6         while True:
7             pattern_distance = None
8             try:
9                 if len(generated_playlist) != len(history_pattern):
10                     break
11                 playlist_features =
12                     get_features(generated_playlist, spotify)
13                 pattern_features = get_features(history_pattern,
14                     spotify)
15                 vertex_distance =
16                     compute_vertex_distance(playlist_features,
17                         pattern_features)
18                 segment_distance =
19                     compute_segment_distance(playlist_features,
20                         pattern_features)
21                 pattern_distance = vertex_distance +
22                     segment_distance
23                 results[playlist_data] = pattern_distance
24             break
25         except SpotifyException:
26             spotify = change_credentials()
27         except Exception as e:
28             break
29     return results

```

La funzione opera in questo modo:

1. Si inizializza l'oggetto `spotify` al valore restituito dalla funzione `change_credentials()`, importata dal modulo `listening_history_manager`.
2. Si inizializza `results` a un dizionario vuoto.
3. Si itera sugli elementi del dizionario `playlists`.
4. Si entra in un ciclo infinito.
5. Si inizializza a `None` la `pattern_distance` relativa alla playlist `generated_playlist` e al pattern `history_pattern`.

6. Si entra in un blocco try-except.
7. Si verifica che la lunghezza della playlist `generated_playlist` coincida con quella del pattern di cronologia `history_pattern`.
8. Se la condizione non viene verificata, si passa all'iterazione successiva, in quanto non è possibile calcolare la Playlist Pattern Distance.
9. Tramite la funzione `get_features`, importata da `listening_history_manager`, si recuperano le caratteristiche audio della playlist (`playlist_features`) e del pattern di cronologia (`pattern_features`).
10. Si calcola la Vertex Distance `vertex_distance`, tramite la funzione `compute_vertex_distance()`, tra la playlist e il pattern di cronologia.
11. Si calcola la Segment Distance `segment_distance`, tramite la funzione `compute_segment_distance()`, tra la playlist e il pattern di cronologia.
12. Si calcola la Pattern Distance `pattern_distance` sommando i due valori appena ottenuti.
13. Si assegna `pattern_distance` a `results`, utilizzando la stessa chiave `playlist_data` con cui viene identificata nel dizionario `playlists` la playlist `playlist` su cui si sta iterando.
14. Si esce dal ciclo infinito e si passa alla prossima playlist `playlist` del ciclo for esterno.
15. Se il primo except cattura l'eccezione `SpotifyException`, si assegnano all'oggetto `spotify` delle nuove credenziali tramite la funzione `change_credentials()`.
16. Se il secondo except cattura una qualunque altra eccezione, vuol dire che c'è stato un problema nel calcolo della Vertex Distance o della Segment Distance. In questo caso, si esce dal ciclo infinito e si passa alla prossima playlist.
17. Una volta terminate tutte le iterazioni del ciclo for esterno, si restituisce il dizionario `results`.

## 4 Sperimentazione

In questo capitolo si discuterà di come è stata condotta la fase sperimentale per la proposta di questa tesi. La sperimentazione è articolata in diverse fasi, ciascuna delle quali mirava a raccogliere, analizzare e confrontare i dati per valutare l'efficacia del nostro metodo di generazione di playlist rispetto a metodi alternativi. In questa sezione introduttiva, descriviamo brevemente ciascuna fase della sperimentazione.

La prima fase ha riguardato la raccolta dei dati. Gli studenti volontari hanno inviato in forma anonima i file contenenti la loro cronologia di ascolto su Spotify. Questo processo è stato eseguito seguendo un procedimento specifico per scaricare la cronologia direttamente dal proprio account Spotify.

Una volta raccolti i file di cronologia, è stata condotta un'analisi approfondita delle cronologie di ascolto. Questa fase ha coinvolto la determinazione di vari indicatori chiave, tra cui il numero totale di canzoni ascoltate, la distribuzione delle canzoni ascoltate per giorno della settimana e per ora del giorno, e la lunghezza media dei pattern di ascolto. Tale analisi ha fornito una comprensione dettagliata delle abitudini di ascolto degli utenti, evidenziando variazioni significative nelle loro preferenze.

Per confrontare l'efficacia di SMART con altri approcci, abbiamo implementato tre metodi alternativi: REC-1, REC-2 e HYB-1. Questi metodi, la cui complessità è inferiore rispetto al nostro, servono per valutare i benefici del nostro approccio, confrontando il nostro metodo con altri più semplici per capire se vale la pena di perseguire la nostra idea. Idealmente, sarebbe opportuno confrontare il nostro metodo con altri esistenti, ma questo va oltre lo scopo della tesi.

Le playlist sono state generate utilizzando quattro metodi distinti: SMART, REC-1, REC-2 e HYB-1. La generazione delle playlist è stata eseguita per tutti i periodi possibili, considerando ogni giorno della settimana (da lunedì a domenica) e ogni ora del giorno (da 0 a 23). È importante notare che, se la fase di clustering necessaria per la generazione della playlist con SMART non è andata a buon fine, le playlist non sono state generate neanche per gli altri metodi, garantendo così una base comparativa equa.

Dopo aver generato le playlist per ogni file di cronologia e per ogni periodo possibile, il passo successivo della nostra sperimentazione è stato calcolare la Playlist Pattern Distance tra ciascuna playlist e il pattern di cronologia utilizzato per generarla. Questo calcolo è fondamentale per valutare la coerenza e la somiglianza tra le playlist generate dai diversi metodi e i pattern di ascolto degli utenti.

L'ultima fase ha riguardato l'analisi dei risultati ottenuti dalla generazione delle playlist. Sono stati creati boxplot e grafici a linea (line chart) per confrontare i vari metodi. Inoltre, è stata effettuata un'analisi dettagliata delle differenze tra le playlist generate tra SMART e HYB-1, che si è dimostrato il metodo più competitivo tra quelli alternativi. Questo confronto è avvenuto prendendo in considerazione diverse metriche, quali la lunghezza della cronologia, la lunghezza delle playlist, le caratteristiche sonore, la

percentuale media di artisti e canzoni comuni tra i vari metodi, e il numero di generi musicali distinti presenti in ciascuna playlist.

Ora che abbiamo fornito una panoramica introduttiva su come verrà condotta la fase sperimentale, possiamo procedere nel descrivere dettagliatamente il dataset utilizzato.

## 4.1 Descrizione del dataset

In questa sezione ci occupiamo di descrivere il dataset utilizzato per condurre la fase di sperimentazione. Per prima cosa, spiegheremo come sono stati raccolti i dati, successivamente, li analizzeremo nel dettaglio.

### 4.1.1 Raccolta dei dati

La raccolta dei dati è stata condotta chiedendo a studenti volontari di inviare in forma anonima il file contenente la cronologia di ascolto Spotify, ottenuta tramite il seguente procedimento:

1. Accedere al proprio account Spotify attraverso un browser.
2. Entrare nella sezione **Account** utilizzando il menù a tendina in alto a destra.
3. Cliccare su "**Impostazioni privacy**".
4. Selezionare la voce "**Cronologia di ascolto estesa**" dalla sezione "**Scarica i tuoi dati**".
5. Premere il bottone "**Richiedi dati**".
6. Risolvere l'eventuale reCAPTCHA dimostrando di non essere un robot.
7. Premere sul bottone "**Conferma**" contenuto nella e-mail ricevuta da Spotify.
8. Dopo diversi giorni di attesa, si riceverà una e-mail che notificherà che i dati richiesti sono pronti per il download. All'interno di questa e-mail cliccare, entro 14 giorni, sul bottone "**Download**".
9. Confermare la propria identità inserendo la password del proprio account. Il file "**my\_spotify\_data.zip**" verrà scaricato automaticamente.
10. Recuperare da "**my\_spotify\_data.zip**" il file "**Streaming\_History\_Audio\_XXXX-YYYY\_Z.json**", dove XXXX-YYYY indica il periodo di riferimento della cronologia (ad esempio 2023-2024) e Z è un numero progressivo utilizzato per suddividere la cronologia in più file nel caso in cui questa contenga molte canzoni. Nel nostro caso, è stato chiesto di condividere il primo

file di cronologia (con Z pari a 0) relativo all'ultimo anno disponibile. Questo file contiene i dati più importanti, come id, caratteristiche audio, tempo di ascolto, relativi a ciascuna canzone ascoltata nel periodo di riferimento.

Abbiamo collezionato un totale di 26 file di cronologia.

#### 4.1.2 Analisi della cronologia

Per fornire il contesto più dettagliato possibile in merito ai dati raccolti, e per comprendere al meglio i risultati ottenuti, abbiamo deciso di raccogliere alcuni dei dati più rappresentativi per ciascun file di cronologia. I dati che sono stati raccolti sono i seguenti:

1. Numero totale di canzoni presenti nella cronologia.
2. Numero totale di canzoni presenti nella cronologia per ogni giorno della settimana.
3. Incidenza percentuale di ascolto delle canzoni per ogni ora, tenendo conto di tutti i giorni della settimana.
4. Incidenza percentuale di ascolto delle canzoni per ogni ora e per ogni giorno.
5. Lunghezza del pattern di cronologia più lungo per ogni ora, tenendo conto di tutti i giorni della settimana.
6. Lunghezza del pattern di cronologia più lungo per ogni ora e per ogni giorno.
7. Lunghezza media dei pattern di cronologia per ogni ora, tenendo conto di tutti i giorni della settimana.
8. Lunghezza media dei pattern di cronologia per ogni ora e per ogni giorno

Nella restante parte di questa sezione, mostreremo alcuni dei dati più significativi che sono stati raccolti durante questa fase iniziale della sperimentazione.

Per quanto riguarda i primi due punti, abbiamo preparato una tabella riassuntiva che mostra per ogni file di cronologia il numero totale di canzoni e il numero di canzoni ascoltate per ogni giorno della settimana.

Come si può osservare dalla Tabella 1, il nostro dataset contiene file di cronologia con un numero totale di canzoni molto variabile: alcuni file contengono poche centinaia di canzoni (come LH10 e LH12), altri ne contengono decine di migliaia (LH25 e LH26), e la maggior parte ne contiene da poche centinaia a poche migliaia (come LH2, LH3 e LH5). Inoltre, è possibile osservare come il numero di canzoni ascoltate vari notevolmente di giorno in giorno e da un file di cronologia a un altro. Ad esempio, confrontando i due file di cronologia LH25 e LH26, possiamo osservare come cambia il numero totale di canzoni ascoltate da un giorno all'altro: nel caso di LH25, le canzoni ascoltate nei weekend sono

File	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì	Sabato	Domenica	Totale
LH1	31	58	13	29	37	54	11	233
LH2	119	196	92	142	178	118	216	1055
LH3	31	32	79	94	79	28	30	473
LH4	13	23	15	49	63	28	35	226
LH5	32	92	85	212	181	79	79	765
LH6	142	304	235	248	60	128	132	1249
LH7	32	72	80	181	181	79	79	704
LH8	111	106	144	170	129	164	73	897
LH9	285	198	237	307	387	281	201	1896
LH10	14	19	1	11	60	36	4	145
LH11	19	14	0	82	5	64	36	220
LH12	27	5	5	9	62	8	6	122
LH13	135	242	354	253	243	663	438	2328
LH14	139	153	197	194	72	61	35	851
LH15	126	54	171	75	130	135	132	823
LH16	98	39	45	76	96	38	72	464
LH17	48	49	59	45	66	54	66	387
LH18	67	72	45	55	110	103	93	545
LH19	187	70	70	178	101	82	94	782
LH20	71	100	110	286	117	143	94	921
LH21	226	206	175	123	243	161	118	1252
LH22	400	344	231	539	245	302	204	2265
LH23	40	2	45	59	0	0	38	184
LH24	145	180	107	98	262	259	351	1402
LH25	10009	10954	11265	10674	10510	3782	2234	59434
LH26	3127	3325	3716	3571	3725	3750	2531	23745

Tabella 1: Numero totale di canzoni e numero di canzoni per ogni giorno della settimana per ogni file di cronologia.

molto meno rispetto a quelle ascoltate nei giorni lavorativi. Al contrario LH26 mostra una distribuzione equa delle canzoni lungo tutti i giorni della settimana. Queste variazioni, che si riscontrano da una cronologia a un'altra e da un giorno all'altro all'interno della stessa cronologia, contribuiscono a motivare l'utilità di sviluppare un metodo altamente personalizzato per la generazione di playlist.

La grande variabilità dei dati riscontrata analizzando semplicemente il numero totale di canzoni, ci ha portato ad effettuare delle analisi più dettagliate di ciascun file di cronologia. A tal proposito, abbiamo preparato dei grafici che mostrano, per ogni file di cronologia e per ogni giorno della settimana, l'incidenza percentuale di ascolto delle canzoni per ogni ora. Siccome i giorni della settimana sono 7 e il numero totale di file di cronologia è 26, abbiamo un totale di 182 grafici di questo tipo. In aggiunta ai grafici giornalieri, sono stati preparati altri 26 grafici, uno per ciascun file di cronologia, che mostrano l'incidenza percentuale di ascolto delle canzoni per ogni ora, tenendo conto di tutti i giorni della settimana assieme anziché i giorni presi singolarmente.

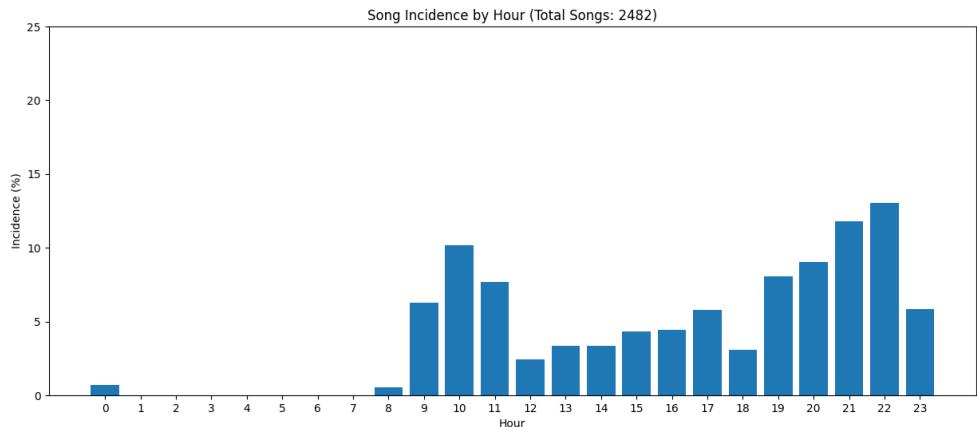


Figura 5: Incidenza percentuale di ascolto delle canzoni per ogni ora, tenendo conto di tutti i giorni della settimana. In questo caso, il file di cronologia di riferimento è LH22.

La Figura 5 e la Figura 6 costituiscono due esempi dei grafici appena descritti. Confrontando le due immagini, possiamo capire fin da subito come le abitudini di ascolto degli utenti possano variare notevolmente di giorno in giorno, perlomeno per quanto riguarda la quantità di musica ascoltata e gli orari di ascolto: guardando più da vicino la Figura 5, possiamo osservare come l'incidenza di ascolto delle canzoni, in media, tenda ad essere più alta nelle ore della tarda mattinata e inferiore nelle ore pomeridiane, per poi tornare ad essere alta nelle ore serali. Al contrario, se consideriamo la Figura 6, che mostra l'incidenza percentuale di ascolto delle canzoni tenendo in considerazione soltanto **Sabato**, riscontriamo un comportamento piuttosto differente, con la musica che tende ad essere ascoltata maggiormente nelle ore pomeridiane e molto meno alla sera. Que-

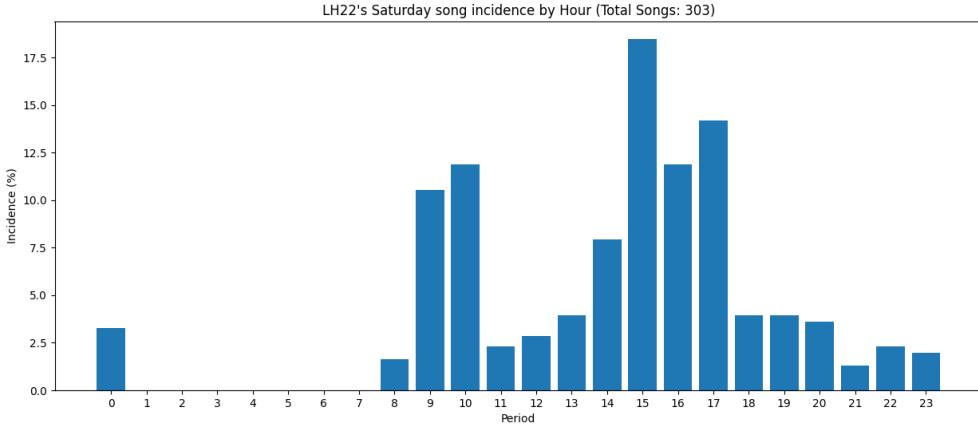


Figura 6: Incidenza percentuale di ascolto delle canzoni per ogni ora, tenendo conto soltanto del giorno **Sabato**. In questo caso, il file di cronologia di riferimento è LH22.

sto fenomeno si verifica anche in altri file di cronologia: ad esempio, non è raro che un utente abituato ad ascoltare musica nelle ore lavorative durante la settimana, abbia delle abitudini di ascolto "rovesciate" durante i weekend, con la musica che viene ascoltata maggiormente nelle ore intorno ai pasti.

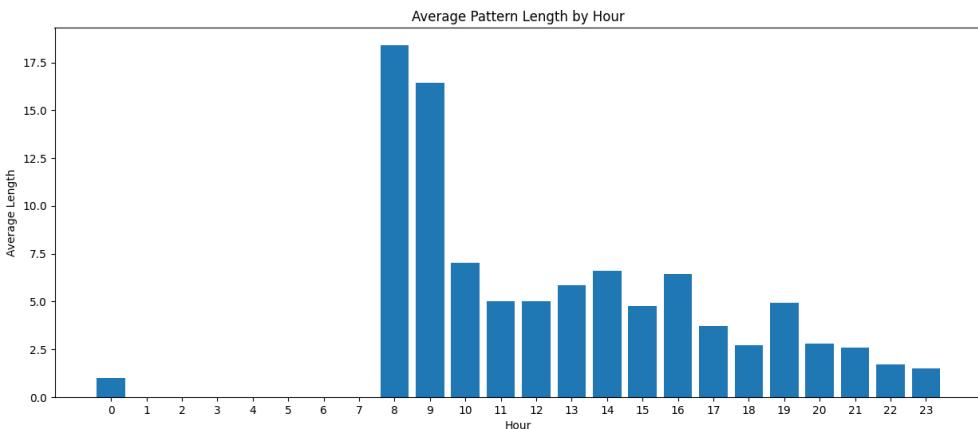


Figura 7: Lunghezza media del pattern di cronologia per ogni ora, tenendo conto di tutti i giorni della settimana. In questo caso, il file di cronologia di riferimento è LH20.

La Figura 7 e la Figura 8 costituiscono due esempi di grafici che analizzano i pattern di cronologia medi, dove la prima tiene conto di tutti i giorni della settimana e la seconda soltanto di **Sabato**. L'analisi dei pattern di cronologia è importante poiché ci indica

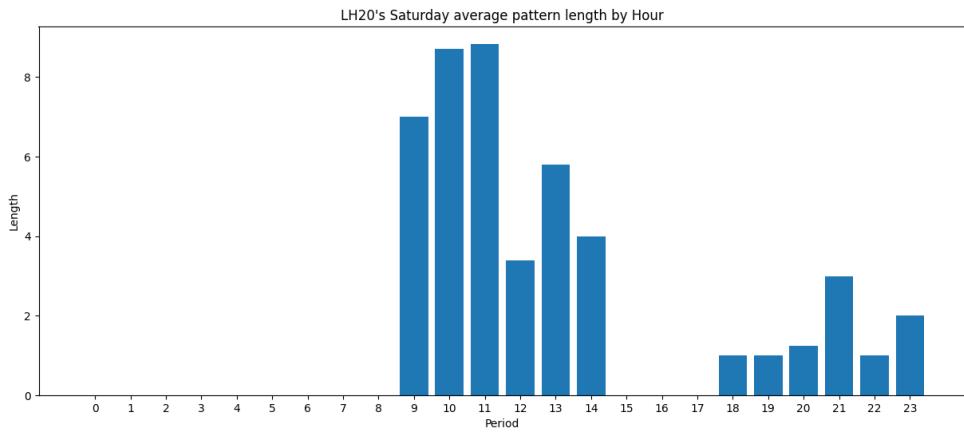


Figura 8: Lunghezza media del pattern di cronologia per ogni ora, tenendo in considerazione soltanto **Sabato**. In questo caso, il file di cronologia di riferimento è LH20

quante canzoni un determinato utente è abituato ad ascoltare consecutivamente nei vari periodi della settimana. Anche in questo caso, la media settimanale e quella giornaliera mostrano risultati differenti: la Figura 7 mostra come l’utente ascolti, in media, dalle 15 alle 20 canzoni dalle 8:00 alle 9:00. D’altra parte, se consideriamo la figura 8, che tiene conto soltanto di **Sabato**, notiamo che l’utente non ha mai ascoltato canzoni alle 8:00 e che alle 9:00 ascolta dalle 6 alle 8 canzoni, ossia meno della metà rispetto alla media settimanale.

Vista l’impossibilità, per ragioni di spazio, di includere tutti i grafici, abbiamo preparato, come nel caso dell’incidenza di ascolto giornaliera, due tabelle riassuntive: la Tabella 2 mostra i pattern di cronologia medi per ogni ora del giorno e per ogni file di cronologia, la Tabella 3 mostra i pattern di cronologia più lunghi per ogni giorno e per ogni file di cronologia. L’utilizzo delle ore anzichè i giorni nelle colonne della Tabella 2 è motivato dal fatto che, siccome stiamo mostrando una media, l’inclusione delle ore notturne penalizzerebbe quest’ultima, rendendo i risultati poco rilevanti. Analizzando meglio la Tabella 2, possiamo notare come le abitudini di ascolto degli utenti varino molto anche in base al numero medio di canzoni ascoltate consecutive: ad esempio, alcuni utenti, durante il lavoro, potrebbero ascoltare tanta musica a determinate ore del giorno in maniera consistente, rendendo la lunghezza media del pattern di cronologia alta. Al contrario, alcuni utenti potrebbero essere abituati ad ascoltare musica in macchina durante brevi tragitti che occorrono tutti i giorni alla stessa ora, rendendo il pattern di cronologia medio più corto.

Prefix Name	Hour	0:00	1:00	2:00	3:00	4:00	5:00	6:00	7:00	8:00	9:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00	17:00	18:00	19:00	20:00	21:00	22:00	23:00	
LH1		0	0	0	0	0	0	0	0	6.8	6.4	8.3	6	5.5	1.7	1.6	6	4	5.5	0	2	3	2	7.5	12	
LH2		5.2	3.3	0	0	0	0	0	15.3	16.9	11.6	9.8	5.8	5.4	7.2	9.9	18	13.8	8.1	6.5	11.8	15.8	10	5.6	4.4	
LH3		1	0	0	0	0	0	0	4.9	7.5	4.5	1.4	2.5	5.4	5.7	4.1	7.2	7.5	6.2	7	11.4	19.7	8.5	2.9	2	
LH4		0	0	0	0	0	0	1	2.8	5.2	3.8	3	3.2	6	1.5	4.5	8.2	7.2	3.4	1.4	1.3	2	0	0	0	
LH5		2	1	0	0	0	1	4	5.2	3.8	7.1	4.9	6.3	7.1	6.2	5.4	5.1	5.8	4.6	6.4	6	8.7	8.8	6.1	3.2	
LH6		0	0	0	0	0	0	0	37.5	36.9	25.4	16.6	10.7	39.3	32.5	26.9	16.4	14.9	7.8	5	0	0	0	0	0	
LH7		2	1	0	0	0	1	4	5.2	3.8	7.1	4.9	6.1	7.1	6.2	5.4	5.1	5.8	4.6	6.4	6	8.6	8.8	6.1	3.2	
LH8		1.5	0	0	0	0	0	0	6	9	35	25.5	9.6	5.7	8	7	7.4	9.9	10.1	9.8	7.9	6.9	4.8	7.5	3.3	
LH9		0	0	0	0	0	0	0	12.6	10.1	9.1	7.1	7.7	8	7.8	8	6.4	5	6.3	4.8	5.1	3.6	8.6	12	4	0
LH10		0	0	0	0	0	0	0	0	4	0	17	5.8	8.4	5.2	2	6	5	0	14.5	5.4	2.2	1	0	0	
LH11		5	0	0	0	0	0	0	0	0	0	0	4	0	14	18.7	10.6	12.4	7.6	11.5	7	5	0	3	9	13
LH12		7	0	0	0	0	0	0	0	0	1	1	2	1	1	7.2	5	5.2	1.8	1	2.5	1.7	10.5	4	15	
LH13		3.9	3.1	14.5	13	8	4.2	3.4	4.3	6.1	7.2	4.7	5	6.3	7	7.6	6.1	4.8	3.6	5.1	4.9	4.6	4.3	4.4	5.2	
LH14		2.4	2.1	0	0	0	0	5	4	1.8	3.7	1.2	4.9	6.6	6.3	6.3	5.8	6.1	5.2	3.9	3.3	6.3	6.8	5.5	4.2	
LH15		10.8	8.5	3	0	0	0	0	0	0	0	4.7	6.1	5.9	6.9	9.7	8.9	3.8	8.4	4.6	5.2	4.8	2.8	5.9	9.1	
LH16		1.5	0	0	0	0	0	0	0	0	0	0	1	3.2	2.4	3.8	2.2	2.6	1.7	2.8	2	5	2.2	1.8	1.6	
LH17		0	0	0	0	1	1	2	1.6	3.1	2	2.1	2.9	3.3	3.2	2.1	2.2	1.8	2	2.4	2	3	1.6	2.7	2	
LH18		0	0	0	0	0	0	0	7	3.7	3.5	3.3	2	1.9	2.9	2.5	2.9	2.5	2.2	4.1	3.1	4.4	2.7	2.8	2	
LH19		0	0	0	0	0	0	0	12	5.5	10.4	8.8	6.6	7.5	6.4	7.4	7.3	7.9	8.8	9.4	6	2.8	1	0	0	
LH20		1	0	0	0	0	0	0	0	18.4	16.4	7	5	5	5.8	6.6	4.8	6.4	3.7	2.7	4.9	2.8	2.6	1.7	1.5	
LH21		2	0	0	0	0	0	0	0	3	6.8	5.7	4.1	4.7	6.7	6.3	6.4	7.3	6	4.7	3.4	2.5	1	1	3	
LH22		2.8	0	0	0	0	0	0	0	12.3	8.6	7	7.1	12.1	13.8	11.8	10.9	6.9	5.2	7.4	11.6	14.3	12	8.6	5.1	
LH23		0	0	0	0	0	0	0	0	0	21	10.3	11	11	15	13	9	9	5	5.2	0	0	4	9	3.5	
LH24		4.2	4.4	4	0	0	0	0	0	0	0	6.2	7.6	7.1	8	8.4	7.9	11.4	9.2	5.6	5.6	4.5	5.1	3.7	3.2	
LH25		1.6	4.3	14	15	1	0	0	36.3	52.1	39.2	27.4	15.1	3.9	8.6	47.6	44.6	34.8	23.2	10.8	4	7	3.3	5	3.7	
LH26		1.5	5.4	8.1	7.8	4.5	3	4.9	5.3	6.4	4.4	5.5	6.6	5.5	6.7	5.2	7.5	7.9	7.6	6.4	2.9	3.1	3.7	2.6	2.1	

Tabella 2: Lunghezza media del pattern di cronologia per ogni ora del giorno e per ogni file di cronologia.

Un ulteriore aspetto utile da tenere in considerazione riguarda l'influenza che ha la lunghezza dei pattern di cronologia sulla generazione di playlist tramite SMART: infatti, come descritto nella Sezione 3.10, l'assenza di pattern di cronologia sufficientemente lunghi ci costringe a sovrapporre più pattern tra loro (quindi relativi a periodi di ascolto differenti), o peggio, a scegliere il pattern "valido" più vicino in ordine cronologico al periodo per cui si sta generando una playlist. Inoltre, per definizione, la lunghezza del pattern di cronologia scelto come input per l'algoritmo di programmazione dinamica stabilisce il numero totale di canzoni che avrà la playlist generata. Questo aspetto deve essere tenuto in considerazione, in quanto il nostro algoritmo e HYB-1 (REC-1 e REC-2 non operano sui pattern di cronologia) possono comportarsi in maniera differente in base alla lunghezza del pattern.

In sintesi, l'analisi della cronologia di ascolto degli studenti ha rivelato una grande variabilità nelle abitudini di ascolto, sia tra diversi utenti che all'interno della stessa settimana per ciascun utente. La variabilità osservata nel numero di canzoni ascoltate e nei pattern di ascolto per ora e per giorno evidenzia l'importanza di un approccio personalizzato nella generazione di playlist.

I grafici e le tabelle riassuntive mostrano chiaramente come le abitudini di ascolto possano cambiare in base al giorno della settimana e all'ora del giorno, influenzando sia la quantità di musica ascoltata sia la lunghezza dei pattern di ascolto. Ad esempio, alcuni utenti mostrano picchi di ascolto durante le ore lavorative nei giorni feriali, mentre altri tendono ad ascoltare più musica durante i weekend.

Queste osservazioni sottolineano la necessità di considerare le specifiche preferenze e routine di ciascun utente per offrire un'esperienza di ascolto più mirata e soddisfacente. Infine, l'analisi dei pattern di cronologia è cruciale per comprendere la frequenza e la

File	Lunedì	Martedì	Mercoledì	Giovedì	Venerdì	Sabato	Domenica
LH1	15	14	5	11	18	26	3
LH2	20	37	19	24	44	22	31
LH3	12	7	54	21	15	23	9
LH4	8	14	6	14	6	27	8
LH5	8	31	23	25	30	21	23
LH6	49	107	100	47	62	33	16
LH7	8	31	23	24	30	21	23
LH8	25	32	36	39	16	38	19
LH9	24	23	25	40	36	31	20
LH10	5	17	1	7	22	17	3
LH11	8	10	0	29	6	5	18
LH12	11	4	4	4	2	6	15
LH13	9	16	26	14	16	41	27
LH14	20	16	29	31	20	26	20
LH15	18	19	24	33	34	35	21
LH16	14	10	4	8	22	7	7
LH17	5	11	4	3	4	5	12
LH18	15	14	11	3	10	13	12
LH19	66	15	27	36	21	13	15
LH20	26	23	16	51	22	23	31
LH21	21	34	24	14	24	24	27
LH22	42	60	30	46	32	25	37
LH23	21	1	8	24	0	19	0
LH24	15	13	26	39	58	47	47
LH25	113	94	120	93	102	84	59
LH26	40	67	55	39	43	54	67

Tabella 3: Lunghezza massima del pattern di cronologia per ogni giorno della settimana e per ogni file di cronologia.

durata degli ascolti consecutivi, aspetti fondamentali per migliorare l'algoritmo di generazione delle playlist, assicurando che le playlist risultanti siano non solo personalizzate ma anche aderenti alle reali abitudini di ascolto degli utenti.

## 4.2 Metodi alternativi

Come anticipato nella Sezione 4, SMART verrà confrontato con tre metodi di complessità crescente: REC-1, REC-2 e HYB-1. L'implementazione di questi metodi è consultabile all'interno della Sezione 3.14. In questa sezione descriviamo il funzionamento di ciascuno di questi metodi sperimentali.

**REC-1** Tramite questo metodo, ogni canzone della cronologia di ascolto filtrata dell'utente viene passata in input al sistema di raccomandazione di Spotify per ottenere una canzone simile.

**REC-2** Questo metodo costituisce una "evoluzione" di REC-1: ciascuna chiamata al sistema di raccomandazioni di Spotify conterrà blocchi di tre canzoni consecutive nella cronologia filtrata. Il sistema di raccomandazioni, avendo a disposizione più canzoni come contesto, dovrebbe essere in grado di restituire canzoni simili in maniera più precisa rispetto a REC-1.

**HYB-1** Questo metodo ibrido genera un song-set utilizzando come input al sistema di raccomandazioni di Spotify blocchi di 5 canzoni consecutive all'interno della cronologia di ascolto filtrata. A questo song-set viene applicato il nostro algoritmo di programmazione dinamica per trovare l'ordinamento ottimale delle canzoni rispetto al pattern di cronologia dell'utente per un determinato periodo della giornata. In pratica, SMART e il metodo HYB-1 differiscono per il modo in cui viene generato l'insieme di canzoni: HYB-1 utilizza blocchi di 5 canzoni consecutive, mentre SMART utilizza l'approccio tramite clustering. Per ogni blocco di 5 canzoni, non chiederemo al sistema di raccomandazioni di Spotify una sola canzone simile (come nei metodi REC-1 e REC-2), bensì andremo a chiedere un numero di canzoni che renderà il song-set delle stesse dimensioni del pattern di cronologia utilizzato come input per l'algoritmo di programmazione dinamica.

## 4.3 Generazione delle playlist

La fase sperimentale vera e propria, come anticipato nella Sezione 4, è avvenuta generando playlist per ogni file di cronologia utilizzando il nostro metodo e i metodi sperimentali appena descritti. Questa fase della sperimentazione è stata eseguita per ogni cronologia di ascolto disponibile, considerando ogni giorno della settimana (da lunedì a domenica) e ogni ora del giorno (da 0 a 23). È importante ricordare che, se la fase di clustering necessaria per la generazione della playlist con il nostro metodo non è andata a buon fine, le playlist non sono state generate neanche per gli altri metodi, garantendo così una base comparativa equa. Di seguito descriviamo in dettaglio il processo di generazione delle playlist.

#### 4.3.1 Preparazione dei dati

- **Filtraggio della cronologia:** In questa fase, la cronologia di ascolto di ciascun utente viene filtrata per eliminare eventuali incongruenze e ridurre il rumore nei dati: vengono eliminate dalla cronologia le canzoni ascoltate per troppo poco tempo e le canzoni ascoltate più volte consecutivamente.
- **Suddivisione della cronologia in periodi:** La cronologia filtrata viene suddivisa in periodi, ciascuno dei quali rappresenta una specifica combinazione di giorno della settimana e ora del giorno.
- **Calcolo dei pattern di cronologia per ogni periodo:** Per ogni periodo identificato, vengono calcolati i pattern di cronologia. Questi pattern rappresentano le sequenze di brani ascoltati consecutivamente in ciascun periodo e costituiscono la base per la generazione delle playlist tramite SMART (e HYB-1).

#### 4.3.2 Generazione della playlist tramite SMART

- **Clustering:** La cronologia di ascolto filtrata per ciascun periodo è stata sottoposta a clustering, come descritto nella Sezione 2.4.
- **Generazione dei song-set:** I due clustering ottenuti (tramite K-Means e FPF) sono stati combinati alle due tecniche euristiche (lineare e sferica) per generare quattro song-set.
- **Scelta del miglior song-set:** Tra i quattro song-set generati, è stato selezionato il migliore, ossia quello più simile possibile alla coppia (NTNA,NTKA) dell'utente per il periodo considerato.
- **Ordinamento ottimale del miglior song-set:** Il song-set scelto è stato ordinato in modo ottimale tramite il nostro algoritmo di programmazione dinamica, permettendo di minimizzare la Playlist Pattern Distance con il pattern di cronologia scelto per il periodo considerato.
- **Sostituzione di eventuali canzoni duplicate con canzoni simili:** Infine, eventuali duplicati nel song-set sono stati sostituiti con canzoni simili, garantendo così una maggiore varietà nella playlist generata.

#### 4.3.3 Generazione delle playlist tramite i metodi alternativi

Questa fase è stata eseguita solo se la fase di clustering di SMART è andata a buon fine. In caso contrario, le playlist non sono state generate per nessuno dei metodi alternativi.

- **Generazione della playlist tramite REC-1 relativa al periodo considerato:** È stata generata una playlist utilizzando il metodo REC-1, che si basa su raccomandazioni singole.
- **Generazione della playlist tramite REC-2 relativa al periodo considerato:** È stata generata una playlist utilizzando il metodo REC-2, che utilizza blocchi di tre canzoni consecutive per generare le raccomandazioni.
- **Generazione della playlist tramite HYB-1 relativa al periodo considerato:** È stata generata una playlist utilizzando il metodo HYB-1, che opera combinando un approccio di raccomandazione basato su blocchi di 5 canzoni con il nostro algoritmo di programmazione dinamica.

## 4.4 Calcolo della Playlist Pattern Distance

Dopo avere generato le playlist per ogni file di cronologia e per ogni periodo possibile, il passo successivo della nostra sperimentazione è stato calcolare le Playlist Pattern Distance tra ciascuna playlist generata e il relativo pattern di cronologia. Per farlo, abbiamo seguito questo procedimento:

1. **Recupero di ogni playlist e del relativo pattern di cronologia**
2. **Recupero delle caratteristiche audio delle canzoni:** vengono recuperati i vettori di caratteristiche utilizzati per rappresentare le canzoni contenute nelle playlist e nei pattern di cronologia.
3. **Calcolo della Vertex Distance e della Segmment Distance:** le caratteristiche audio di ciascuna playlist e del relativo pattern di cronologia vengono utilizzate per calcolare la Vertex Distance e la Segment Distance.
4. **Calcolo della Playlist Pattern Distance:** la Vertex Distance e la Segment Distance vengono sommate per ogni playlist generata, ottenendo in questo modo la Playlist Pattern Distance.

## 4.5 Analisi dei risultati

In questa sezione discuteremo dei risultati ottenuti durante la sperimentazione, analizzando le performance dei vari metodi e confrontando le playlist generate.

### 4.5.1 Confronto basato sulla Playlist Pattern Distance

Dopo aver calcolato la Playlist Pattern Distance per tutte le playlist, abbiamo calcolato:

- Pattern Distance media di ogni metodo per ogni file di cronologia.

- Pattern Distance media giornaliera di ogni metodo per ogni file di cronologia.
- Pattern Distance media di ogni metodo per ogni file di cronologia per ogni ora del giorno.
- Pattern Distance media giornaliera di ogni metodo per ogni file di cronologia per ogni ora del giorno.
- Playlist Pattern Distance media per ogni metodo in funzione dei diversi file di cronologia.

I valori così calcolati verranno poi visualizzati utilizzando istrogrammi, box plot o grafici a linee. Prima di presentare alcuni dei grafici più interessanti che abbiamo ottenuto durante questa fase di sperimentazione, vogliamo mostrare un resoconto globale, fornendo due tabelle: la Tabella 5 mostra le differenze interquartili della Playlist Pattern Distance media per ogni file di cronologia, la Tabella 4 mostra la Playlist Pattern Distance media per ogni file di cronologia e per ogni metodo.

La tabella 5 presenta le differenze interquartili (Interquartile Range, IQR) della Playlist Pattern Distance media per ciascun file di cronologia, indicando la variabilità della distanza di pattern tra le playlist generate dai diversi metodi. L'IQR è una misura della dispersione che rappresenta la differenza tra il terzo quartile (Q3) e il primo quartile (Q1) dei dati. In altre parole, ci mostra quanto variabili sono le distanze di pattern all'interno del 50% centrale dei dati. I valori sono ordinati in senso decrescente di variabilità. Osservando la tabella 5, si osserva che la variabilità tende ad essere più alta nei file di cronologia che contengono un numero maggiore di canzoni. Per capire meglio come il numero totale di canzoni influisce sulla Playlist Pattern Distance abbiamo condotto un esperimento apposito all'interno della Sezione 4.5.2.

File	hyb-1	SMART	rec-1	rec-2
LH1	517.53	318.00	699.50	761.66
LH2	520.21	557.19	996.40	954.40
LH3	548.60	506.75	882.58	782.37
LH4	225.28	320.21	548.27	484.23
LH5	468.51	509.79	876.52	847.15
LH6	615.90	786.49	1015.45	1021.28
LH7	460.74	534.54	832.05	865.94
LH8	554.90	554.87	877.62	869.96
LH9	737.37	692.11	1041.30	1035.57
LH10	529.67	189.44	675.72	704.89
LH11	870.61	674.89	677.82	769.82
LH12	295.33	147.42	459.75	492.85
LH13	896.35	710.11	1130.41	1161.25
LH14	596.78	469.27	748.10	722.68
LH15	653.53	442.37	735.65	736.90
LH16	475.24	282.71	683.07	686.27
LH17	496.01	165.76	642.03	607.15
LH18	481.64	361.34	784.22	764.93
LH19	894.11	769.94	1038.74	1033.50
LH20	829.92	611.30	964.82	972.56
LH21	953.74	702.53	1318.58	1272.25
LH22	830.01	609.45	1013.47	978.15
LH23	1036.83	834.83	1180.62	1181.15
LH24	1161.48	1102.13	1605.99	1624.26
LH25	1872.89	1409.50	2287.24	2347.99
LH26	1076.11	920.13	1378.79	1396.97

Tabella 4: Playlist Pattern Distance media per ogni file di cronologia e per ogni metodo

<b>LH</b>	<b>Variabilità (Differenza Interquartile)</b>	Numero totale di canzoni
LH25	1466.164333	59441
LH26	1399.174142	23565
LH22	1240.991078	2482
LH21	946.009965	1158
LH24	941.150125	1402
LH19	868.836538	915
LH6	848.363930	1159
LH13	846.053663	2328
LH20	715.082696	959
LH9	623.924105	1896
LH2	616.153976	1055
LH11	589.702128	220
LH3	554.465351	473
LH14	527.698413	851
LH8	522.348401	902
LH18	506.780439	431
LH15	505.532414	823
LH5	486.031917	765
LH7	476.967631	763
LH10	436.815477	145
LH16	375.703747	464
LH17	372.168393	299
LH1	358.158886	233
LH23	348.613292	181
LH4	333.946465	214
LH12	198.592222	122

Tabella 5: Differenze Interquartili della Playlist Pattern Distance media per ciascun file di cronologia

Per quanto riguarda la Tabella 4, guardando ai totali complessivi, SMART ha una media inferiore (843,53) rispetto agli altri metodi, suggerendo che, in generale, potrebbe essere più efficiente nel fornire playlist di qualità superiore. SMART è seguito da HYB-1 (933,48), REC-1 (1310,88), e REC-2 (1308,28). Dalla tabella, si osserva che i metodi REC-1 e REC-2 sono meno efficaci nel mantenere una bassa Playlist Pattern Distance rispetto a SMART e a HYB-1. Inoltre, la complessità maggiore di REC-2 nel generare le raccomandazioni non è sufficiente a produrre dei benefici significativi rispetto all'utilizzo del metodo più semplice REC-1.

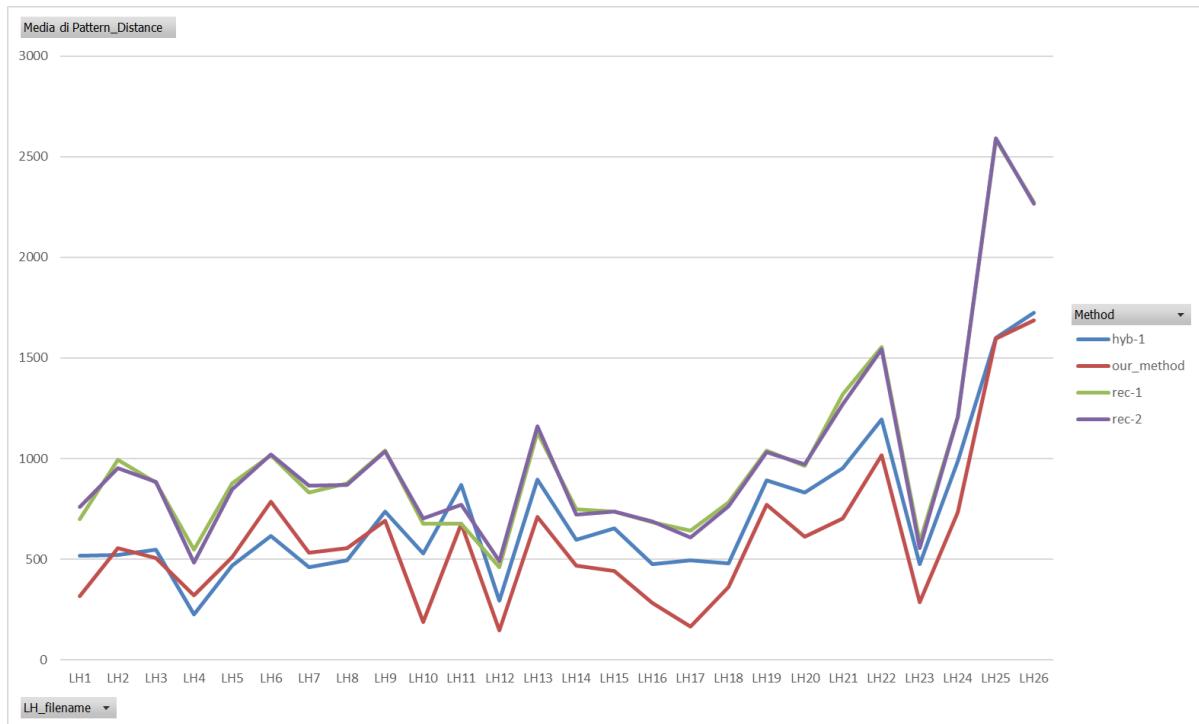


Figura 9: Grafico a linee globale che mostra come varia la Playlist Pattern Distance media per ogni metodo in funzione dei diversi file di cronologia.

La Figura 9 rappresenta il grafico a linee relativo ai dati contenuti nella Tabella 4. Sull'asse delle ascisse sono presenti tutti i file di cronologia, mentre sulle ordinate abbiamo la Playlist Pattern Distance media. Come si può notare, le linee relative ai metodi REC-1 e REC-2 sono sostanzialmente sovrapposte, indicando che le differenze tra i due metodi in termini di Playlist Pattern Distance sono trascurabili. Il metodo SMART, rappresentato dalla linea rossa, produce playlist che, per la maggior parte dei file di cronologia, hanno una Pattern Distance media inferiore a tutti gli altri metodi. Il metodo HYB-1, rappresentato dalla linea blu, mostra prestazioni migliori rispetto a REC-1 e REC-2 per tutti i file di cronologia eccetto LH11. Inoltre, HYB-1 è leggermente più efficace di SMART per i file di cronologia da LH2 a LH8.

Di seguito mostreremo, per ogni tipo di grafico che abbiamo prodotto, un esempio in cui HYB-1 mostra una performance migliore di SMART (LH6), e uno in cui SMART mostra una performance migliore di HYB-1 (LH24).

La Figura 10 mostra come HYB-1 produca una Playlist Pattern Distance media minore rispetto a tutti gli altri metodi per il file di cronologia LH6. Tuttavia, il metodo SMART produce una Playlist Pattern Distance media minore rispetto a tutti gli altri metodi per il file di cronologia LH24. I metodi REC-1 e REC-2 si equivalgono in entrambi i casi.

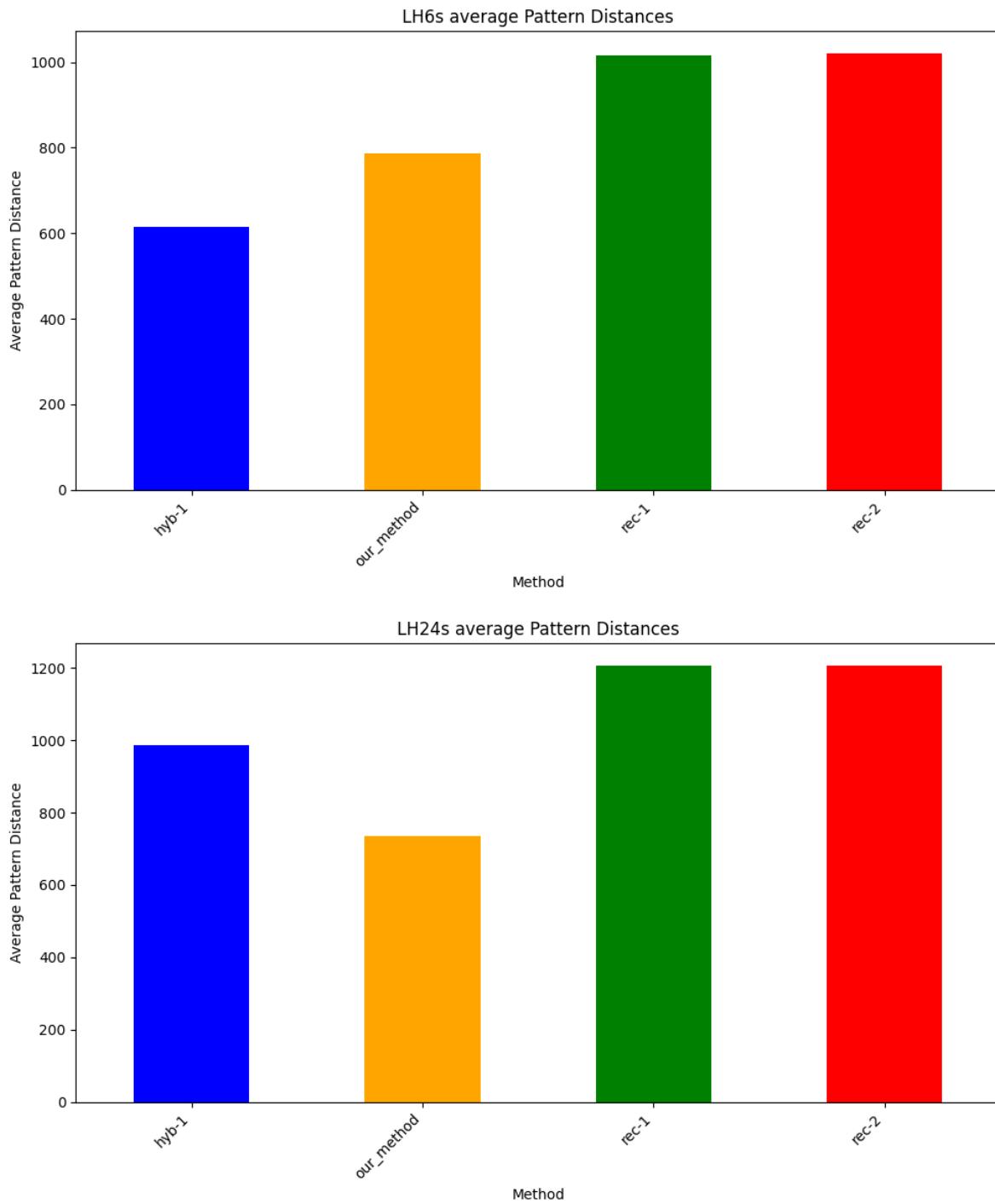


Figura 10: Istogramma della Pattern Distance media per ogni metodo considerando i file di cronologia LH6 e LH24.

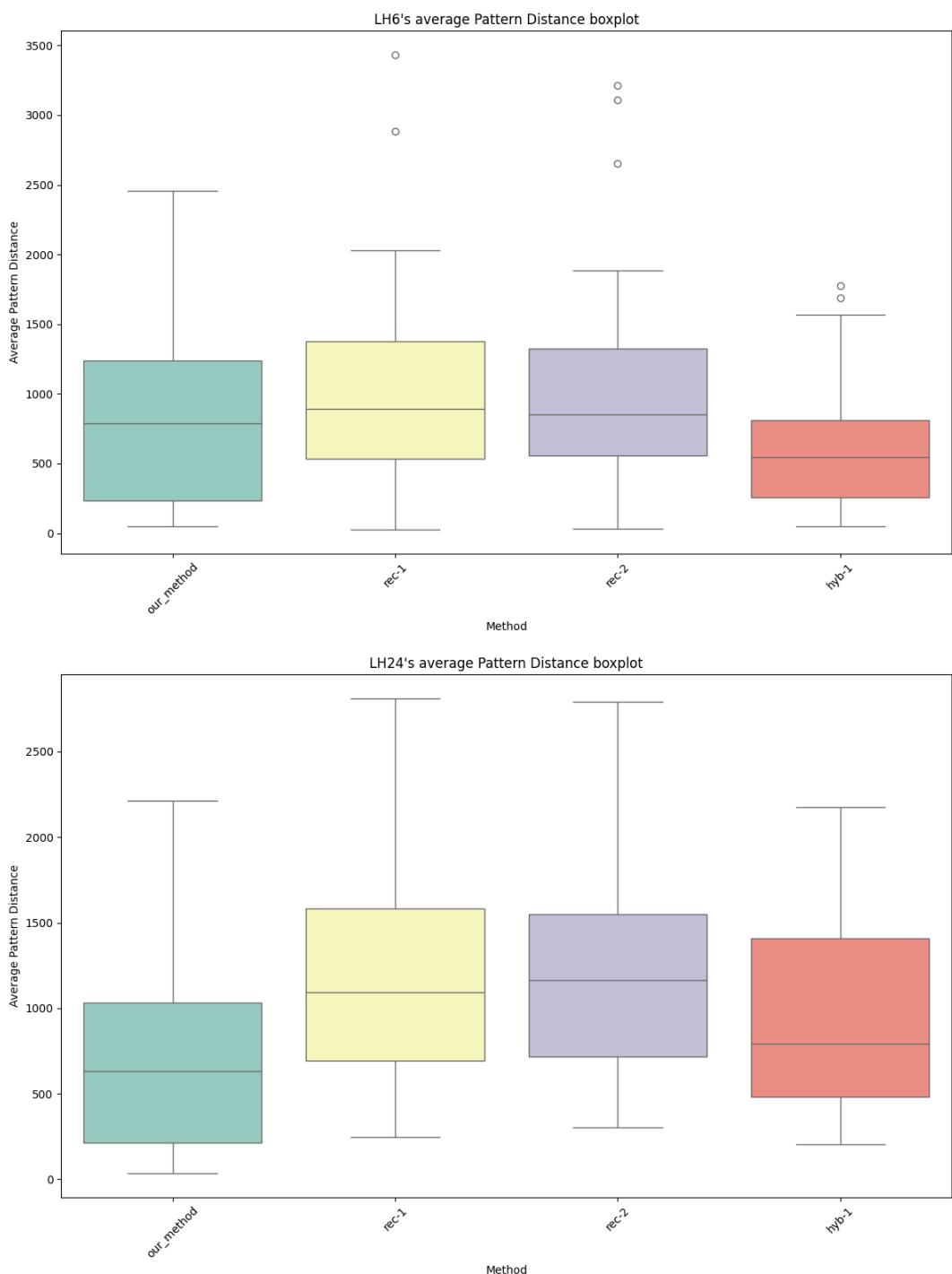


Figura 11: Boxplot della Pattern Distance media per ogni metodo considerando il file di cronologia LH6 e LH24.

La Figura 11 mostra come il metodo HYB-1 abbia una mediana e una media inferiori rispetto agli altri metodi, suggerendo che sia il più efficace sul file di cronologia LH6. REC-1 e REC-2 hanno distribuzioni e mediane simili, ma REC-1 ha meno outliers rispetto a REC-2. Il metodo SMART mostra una media e una mediana inferiori a REC-1 e REC-2, ma è comunque meno efficace di HYB-1 se consideriamo LH6.

Considerando invece LH24, SMART presenta una mediana minore, suggerendo che sia il più efficace considerando. REC-1 e REC-2 hanno distribuzioni simili, ma REC-2 ha una mediana leggermente superiore rispetto a REC-1. HYB-1 ha una mediana (e media) leggermente più alta rispetto al metodo SMART.

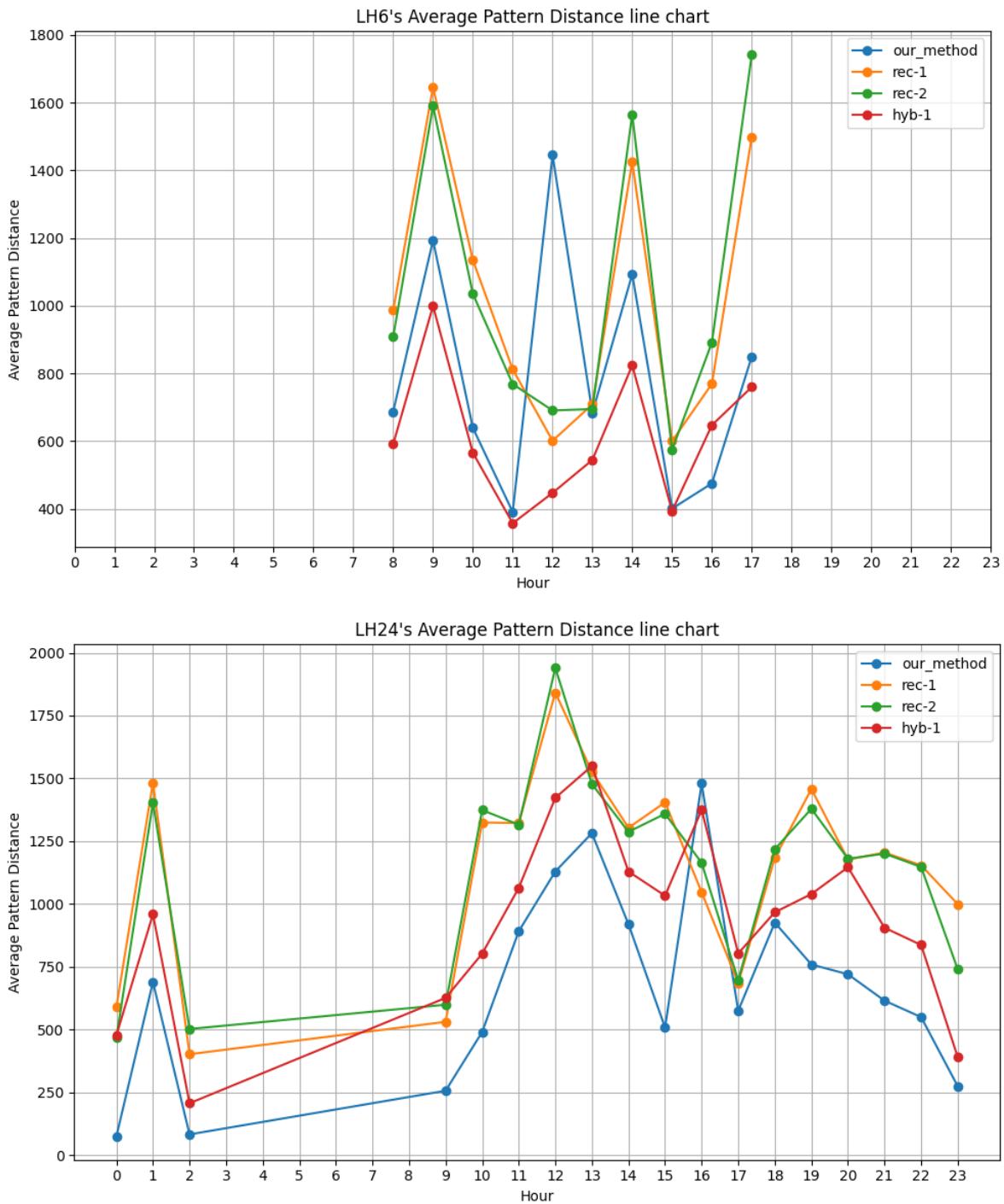


Figura 12: Grafico a linee della Pattern Distance media per ogni metodo considerando i file di cronologia LH6 e LH24.

Il grafico a linee presente in Figura 12 mostra come HYB-1 applicato al file di cronologia LH6 sia generalmente il metodo più efficace. In alcune ore del giorno, come alle 11 e alle 15, la differenza tra SMART e HYB-1 è minima. Al contrario, in altre ore del giorno, come alle 12 e alle 14, HYB-1 è riuscito a mantenere una Playlist Pattern Distance bassa nonostante valori elevati degli altri metodi. Il grafico a linee relativo al file di cronologia LH24 mostra invece come SMART si comporti in maniera notevolmente migliore rispetto a tutti gli altri metodi per tutte le ore del giorno eccetto le 16.

Gli esperimenti condotti fino a questo punto dimostrano come i metodi REC-1 e REC-2 non siano sufficientemente competitivi. Pertanto, nelle fasi successive della sperimentazione, proseguiremo considerando soltanto SMART e HYB-1.

#### 4.5.2 Confronto basato sulla lunghezza della cronologia

Proseguiamo quindi con la nostra sperimentazione conducendo un’analisi delle differenze tra le playlist generate da SMART e da HYB-1, basandoci sulla lunghezza della cronologia di ascolto. Questa scelta è motivata dall’osservazione che le abitudini di ascolto degli utenti possono variare notevolmente da un giorno all’altro e ancora di più da un mese all’altro. Inoltre, siccome abbiamo riscontrato una correlazione tra variabilità della Playlist Pattern Distance e numero totale di canzoni presenti nella cronologia, abbiamo ipotizzato che la presenza di canzoni piuttosto vecchie all’interno della cronologia potesse influenzare in qualche modo i risultati.

Abbiamo quindi testato i metodi utilizzando come input i file di cronologia filtrata per periodi incrementali (1 mese, 3 mesi, 6 mesi, 12 mesi). Questa fase della sperimentazione è stata condotta esclusivamente sui file LH25 e LH26, poiché erano gli unici a contenere un numero sufficiente di canzoni e a coprire un intervallo di tempo abbastanza lungo da garantire un’analisi significativa.

I risultati della nostra analisi sono stati presentati tramite boxplot e grafici a linee per mostrare la distribuzione e l’andamento della Pattern Distance in funzione dell’intervallo di tempo e dell’ora del giorno. Mostreremo un totale di 10 grafici a linee: i primi 8 grafici (4 per file di cronologia) si riferiscono a singoli intervalli di tempo (1, 3, 6 e 12 mesi), mentre gli ultimi due (uno per file di cronologia) sovrappongono i precedenti, mostrando contemporaneamente tutti gli intervalli di tempo.

Mostreremo inoltre due boxplot, uno per file di cronologia, per fornire una visione più pulita dei risultati ottenuti. I boxplot saranno accompagnati da una tabella riassuntiva che mostra la media e la varianza della Playlist Pattern Distance per ogni metodo, file di cronologia e intervallo temporale.

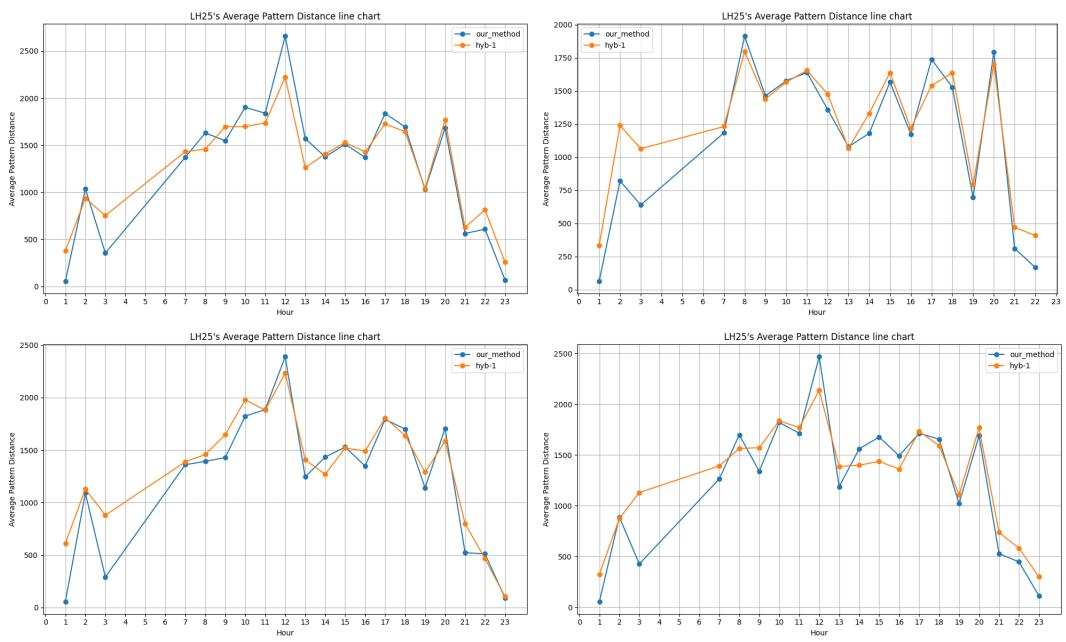


Figura 13: Andamento della Pattern Distance media di SMART e HYB-1 relativo al file di cronologia LH25 in base alla lunghezza della cronologia (1,3,6,12 mesi).

La Figura 13 mostra come la differenza in termini di Pattern Distance media tra SMART e HYB-1, considerando l'ultimo mese della cronologia LH25, sia poco significativa, eccetto nelle ore notturne, dove SMART dimostra di essere leggermente migliore. Considerando un periodo di ascolto di tre mesi, il comportamento dei due metodi è analogo a quello in cui la cronologia è lunga un mese. Considerando un periodo di ascolto di 6 mesi, i metodi SMART e HYB-1 si alternano nel produrre una Pattern Distance media inferiore. Pertanto, anche in questo caso, è difficile stabilire con certezza quale dei due metodi vinca. Infine, se consideriamo il file di cronologia LH25 su un periodo di ascolto di un anno, otteniamo un comportamento analogo a quello ottenuto su sei mesi.

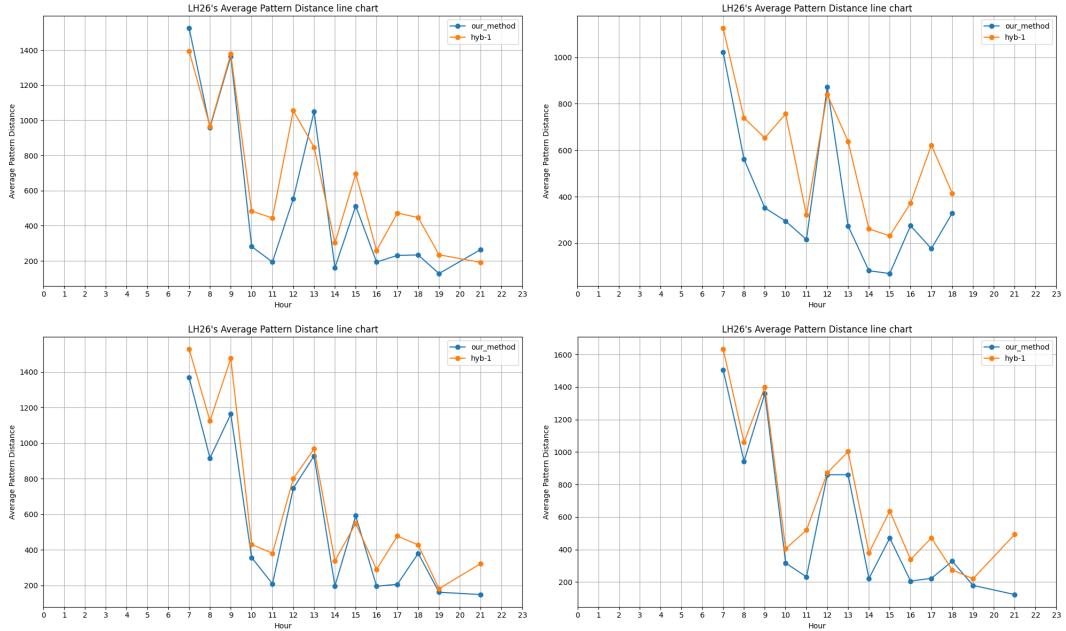


Figura 14: Andamento della Pattern Distance media di SMART e HYB-1 relativo al file di cronologia LH26 in base alla lunghezza della cronologia (1,3,6,12 mesi).

La Figura 14 mostra come SMART prevalga su HYB-1 considerando il file di cronologia LH26 e un intervallo di tempo di un mese. In questo caso, HYB-1 prevale soltanto alle 21 e alle 13 e si comporta analogamente a SMART alle 8 e alle 9. Considerando tre mesi, il metodo SMART prevale abbastanza nettamente su HYB-1 in tutte le ore del giorno eccetto alle 12, dove i due metodi sostanzialmente si equivalgono. Considerando un intervallo di tempo di sei mesi, SMART prevale su HYB-1, ma in maniera meno evidente rispetto a quanto riscontrato considerando tre mesi: i due metodi si equivalgono nelle ore dalle 7 alle 10 e alle 12. HYB-1 prevale alle 18. In tutte le altre ore del giorno prevale SMART. Infine, se consideriamo un anno di ascolto, i due metodi si equivalgono nella maggior parte delle ore, soprattutto quelle pomeridiane. Il metodo SMART prevale nelle prime ore del mattino e a metà pomeriggio. HYB-1 prevale di poco alle 15.

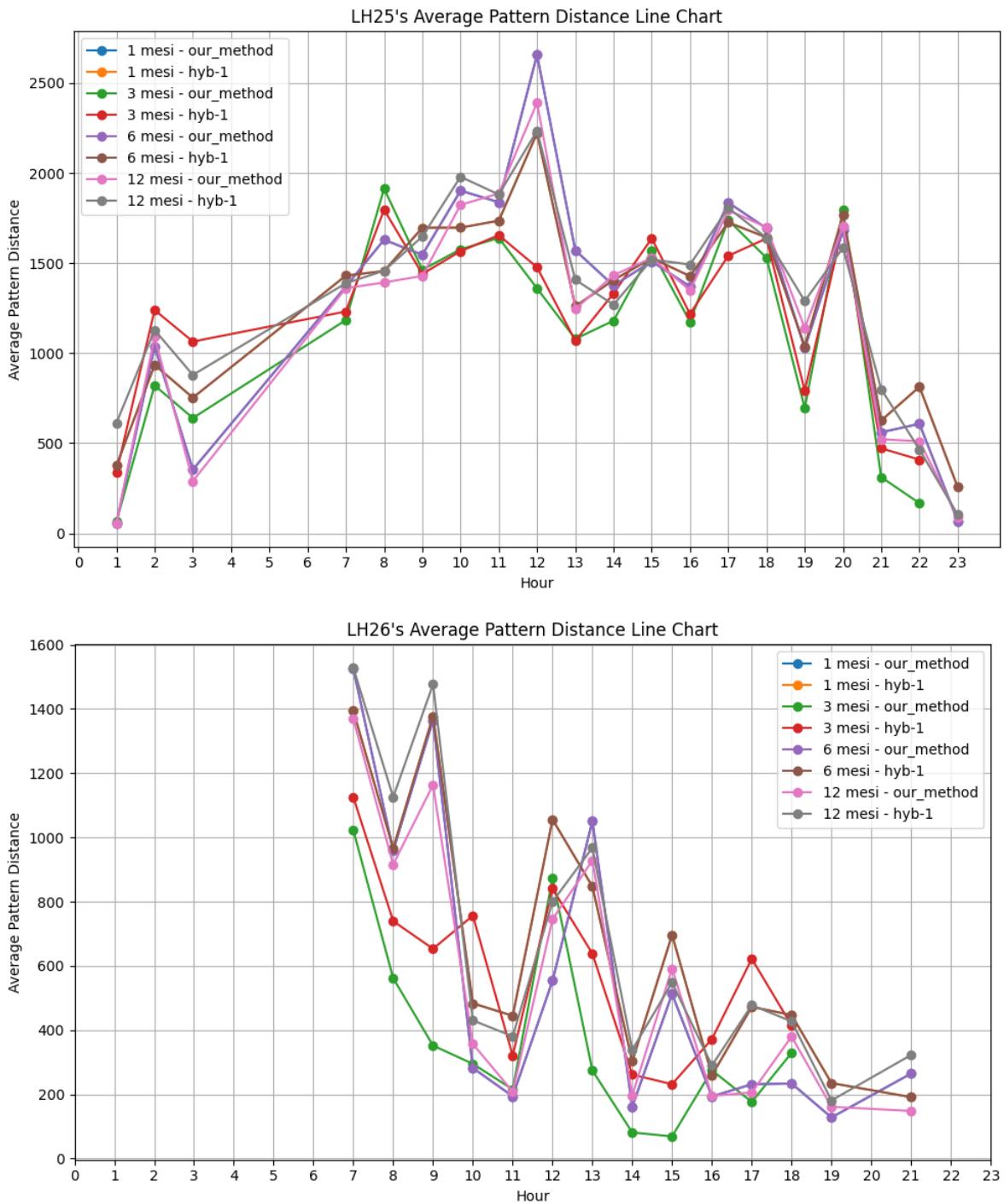


Figura 15: Andamento della Pattern Distance media di SMART e HYB-1 relativo ai file di cronologia LH25 e LH26 considerando tutti i periodi di ascolto (1,3,6,12 mesi).

I grafici a linee relativi al file di cronologia LH25, sovrapposti nella Figura 15, non forniscono informazioni esaustive su quale periodo di tempo sia migliore per generare le playlist, indipendentemente dal metodo utilizzato. In questo caso, non è quindi possibile stabilire una combinazione di metodo e intervallo di tempo ottimale per minimizzare la Playlist Pattern Distance.

Per quanto riguarda LH26, i grafici a linee, sovrapposti nella Figura 15, mostrano che le playlist generate tramite SMART su un intervallo di tempo di tre mesi abbiano una Pattern Distance media più bassa rispetto alle altre combinazioni. Infatti, si può notare come la linea verde (che rappresenta il metodo SMART applicato agli ultimi 3 mesi di ascolto alla cronologia LH26) rimanga per la maggior parte delle ore sotto tutte le altre linee.

Per verificare se le nostre osservazioni sono attendibili, mostreremo ora i boxplot corrispondenti a questi grafici a linee globali, accompagnati da una tabella riassuntiva.

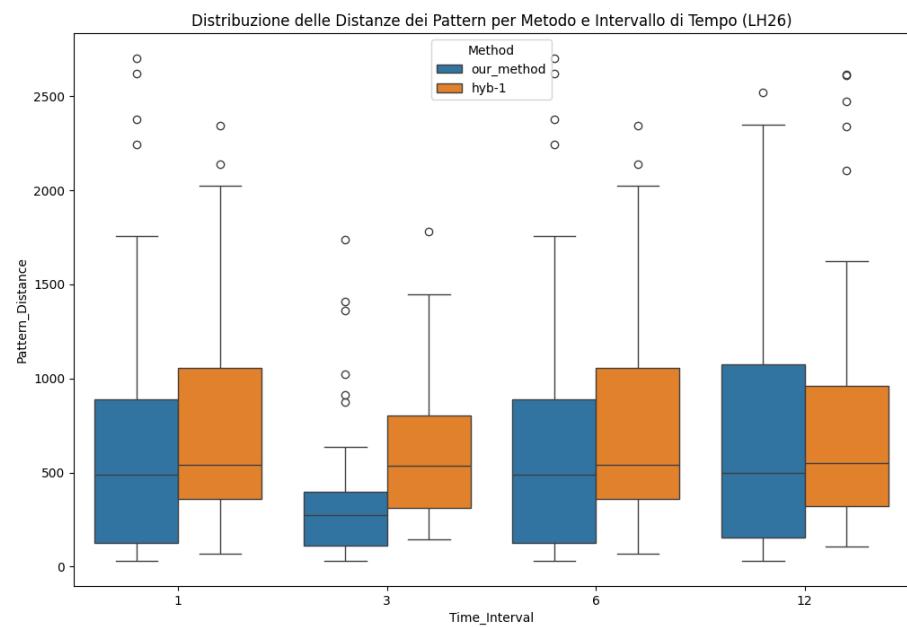
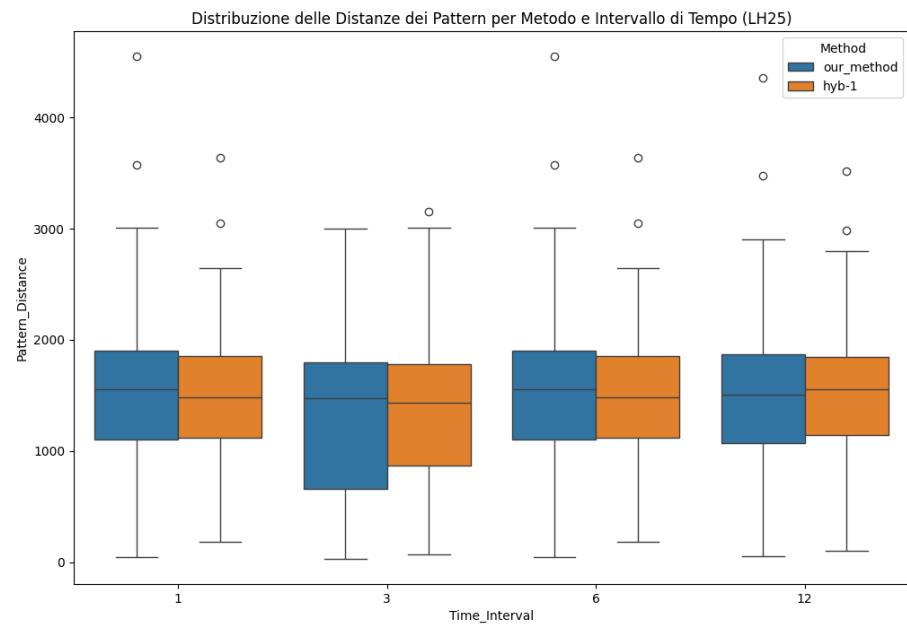


Figura 16: Boxplot della Pattern Distance media per metodo e intervallo di tempo relativo ai file di cronologia LH25 e LH26.

I boxplot appena mostrati confermano le nostre osservazioni iniziali: per quanto riguarda LH25, non c'è una differenza significativa tra i due metodi.

La tabella riepilogativa delle medie e delle varianze (Tabella 6) mostra come la differenza di Pattern Distance media tra SMART e HYB-1 per l'intervallo di 3 mesi non sia statisticamente significativa ( $p$ -value: 0.6675). Inoltre, le mediane sono molto vicine tra loro per tutti e quattro gli intervalli di tempo, indicando che non ci sono prove sufficienti per affermare che un metodo sia migliore dell'altro per LH25.

Al contrario, per LH26, è evidente sia graficamente che statisticamente ( $p$ -value: 0.0329) come la differenza in termini di Pattern Distance media tra SMART e HYB-1 sia significativa tenendo conto di un periodo di tempo di tre mesi. Questo suggerisce che, per LH26, il metodo SMART sia effettivamente migliore di HYB-1 nel ridurre la Playlist Pattern Distance quando si considera un periodo di 3 mesi.

<b>File</b>	<b>Metodo</b>	<b>Intervallo (mesi)</b>	<b>Media</b>	<b>Varianza</b>
LH25	HYB-1	1	1469	391466
LH25	HYB-1	3	1352	404251
LH25	HYB-1	6	1469	391466
LH25	HYB-1	12	1512	385856
LH25	SMART	1	1511	566983
LH25	SMART	3	1312	483605
LH25	SMART	6	1511	566983
LH25	SMART	12	1459	553787
LH26	HYB-1	1	767	355269
LH26	HYB-1	3	634	165655
LH26	HYB-1	6	767	355269
LH26	HYB-1	12	810	487825
LH26	SMART	1	689	542482
LH26	SMART	3	408	189075
LH26	SMART	6	689	542482
LH26	SMART	12	654	394268

Tabella 6: Tabella riassuntiva dei risultati ottenuti analizzando le playlist generate tramite SMART e HYB-1 in funzione della lunghezza temporale del file di cronologia. I valori sono stati arrotondati al numero intero più vicino.

#### 4.5.3 Confronto basato sulla lunghezza della playlist

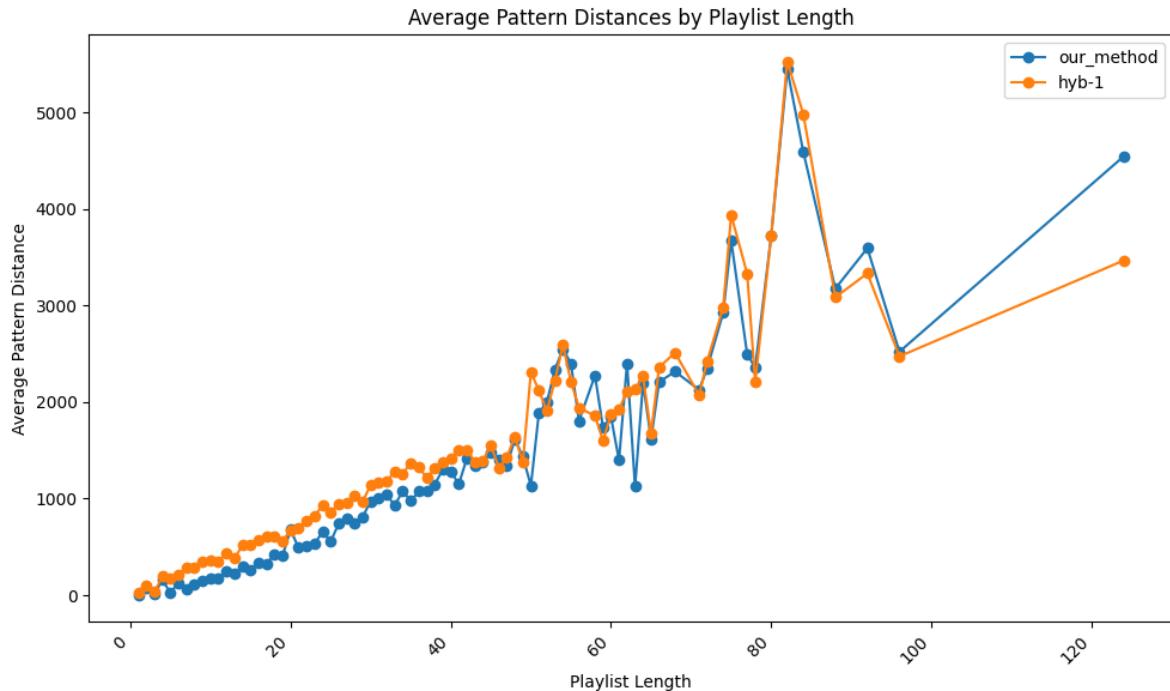


Figura 17: Grafico a linee che confronta l’andamento della Playlist Pattern Distance media dei metodi SMART e HYB-1 in base alla lunghezza della playlist generata.

Il grafico a linee in Figura 17 mostra l’andamento della Playlist Pattern Distance media per SMART e HYB-1 in funzione della lunghezza della playlist generata. Si osserva che entrambi i metodi presentano un aumento della Playlist Pattern Distance con l’augmentare della lunghezza della playlist. Tuttavia, il metodo SMART tende a mantenere una Playlist Pattern Distance più bassa rispetto a HYB-1 per la maggior parte delle lunghezze della playlist, suggerendo una maggiore efficienza nella creazione di playlist di qualità anche al crescere della loro lunghezza. Il comportamento appena descritto si verifica fino a una lunghezza di circa 50. Superata questa soglia i due metodi iniziano a mostrare performance simili. Tuttavia, è importante notare che questo risultato può essere influenzato dal fatto che la quantità di dati è inferiore per playlist molto lunghe, il che potrebbe alterare le osservazioni.

#### 4.5.4 Altre differenze tra le playlist

Fino a questo punto, abbiamo analizzato le differenze tra i vari metodi utilizzando come metrica di confronto solamente la *Playlist Pattern Distance*. In questa sezione conclusiva

della nostra fase sperimentale utilizzeremo altri metodi di confronto, in modo tale da catturare ulteriori differenze tra le playlist generate con il nostro metodo e quelle generate tramite HYB-1.

Di seguito elenchiamo uno ad uno i metodi utilizzati per confrontare le playlist generate con i due metodi.

**Metodo 1: Distanza Euclidea Normalizzata** Per ogni coppia di playlist, abbiamo calcolato:

- $V1$ : vettore medio di caratteristiche della playlist generata dal metodo SMART.
- $V2$ : vettore medio di caratteristiche della playlist generata dal metodo HYB-1.
- $V3$ : vettore medio di caratteristiche del pattern di cronologia utilizzato come input per l'algoritmo di programmazione dinamica.

Abbiamo poi calcolato:

- $D1 = 1 - \text{Distanza euclidea normalizzata tra } V1 \text{ e } V3$
- $D2 = 1 - \text{Distanza euclidea normalizzata tra } V2 \text{ e } V3$

Risultati:

- Media  $D1$  (SMART) = 0.964
- Media  $D2$  (HYB-1) = 0.950

**Metodo 2: Similarità Coseno** Per ogni coppia di playlist, abbiamo calcolato:

- $P1$ : coppia ( $NTNA$ ,  $NTKA$ ) della playlist generata con SMART
- $P2$ : coppia ( $NTNA$ ,  $NTKA$ ) della playlist generata con HYB-1
- $P3$ : coppia ( $NTNA$ ,  $NTKA$ ) della cronologia di ascolto

Abbiamo poi calcolato:

- $C1$ : Similarità coseno tra  $P1$  e  $P3$
- $C2$ : Similarità coseno tra  $P2$  e  $P3$

Risultati:

- Media  $C1$  (SMART) = 0.718
- Media  $C2$  (HYB-1) = 0.675

**Metodo 3: Percentuale di Artisti Distinti** Abbiamo calcolato la percentuale media di artisti distinti per playlist:

- SMART: 93.69%
- HYB-1: 90.37%

**Metodo 4: Numero di Generi Musicali Distinti** Abbiamo calcolato il numero medio di generi musicali distinti per playlist (ad ogni canzone Spotify assegna più generi):

- SMART: 29.08
- HYB-1: 24.75

**Metodo 5: Popolarità Media delle Canzoni** Abbiamo calcolato la popolarità media delle canzoni per playlist:

- SMART: 38.33%
- HYB-1: 39.84%

**Metodo 6: Popolarità Media degli Artisti** Abbiamo calcolato la popolarità media degli artisti per playlist:

- SMART: 45.98%
- HYB-1: 49.65%

**Metodo 7: Percentuale di Canzoni Uguali** Abbiamo calcolato la percentuale media di canzoni uguali per playlist tra il metodo SMART e il metodo HYB-1:

- Percentuale media di canzoni uguali: 0.9%

**Metodo 8: Percentuale di Artisti Uguali** Abbiamo calcolato la percentuale media di artisti uguali per playlist tra il metodo SMART e il metodo HYB-1:

- Percentuale media di artisti uguali: 7.77%

Ecco una tabella riassuntiva contenente i risultati di ciascun esperimento:

Tabella 7: Risultati dei Metodi di Confronto tra le Playlist generate con HYB-1 e con SMART

<b>Metodo</b>	<b>SMART</b>	<b>HYB-1</b>
media della Distanza Euclidea normalizzata dei vettori di caratteristiche medi tra playlist e pattern di cronologia	0.964	0.950
Similarità coseno media delle coppie (NTNA,NTKA) tra playlist e pattern di cronologia	0.718	0.675
Percentuale media di artisti distinti per playlist	93.69%	90.37%
Numero medio di generi musicali distinti per playlist	29.08	24.75
Popolarità media delle canzoni per playlist	38.33%	39.84%
Popolarità media degli artisti per playlist	45.98%	49.65%
Percentuale media di canzoni uguali per playlist	0.9%	
Percentuale media di artisti uguali per playlist	7.77%	

## 4.6 Altri grafici

In questa sezione mostriamo i grafici globali che non sono stati mostrati fino a questo momento. I grafici giornalieri, invece, possono essere consultati nell'Appendice A. I grafici mostrati sono suddivisi in due sezioni: analisi della cronologia di ascolto e analisi della Playlist Pattern Distance.

### 4.6.1 Analisi della cronologia di ascolto

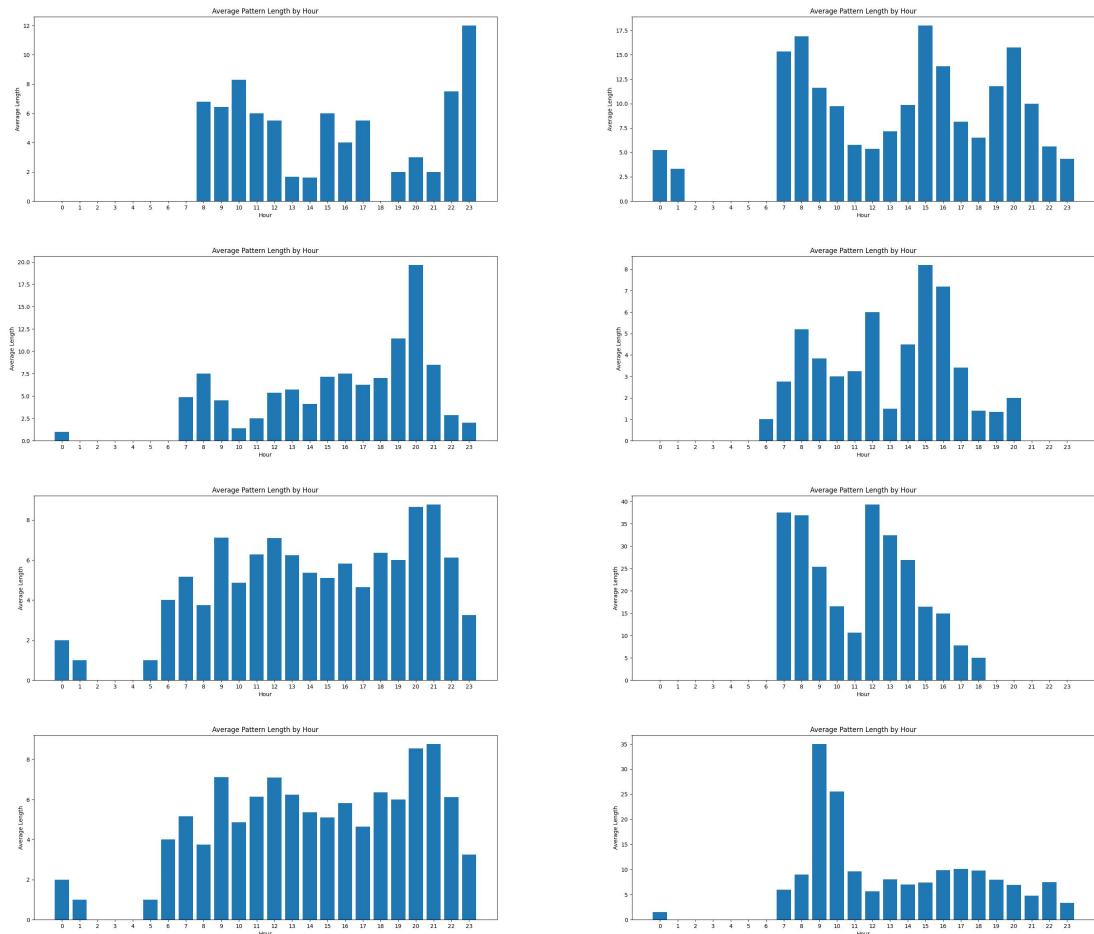


Figura 18: Lunghezza media oraria del pattern di cronologia di ascolto per i file di cronologia da LH1 a LH8.

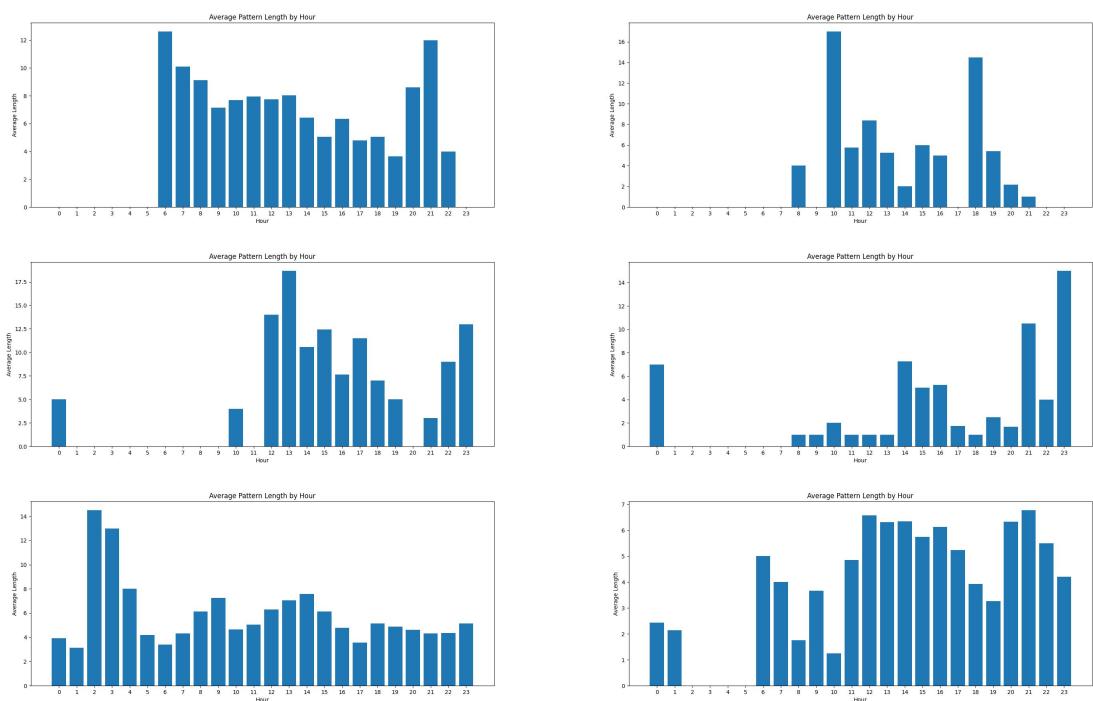


Figura 19: Lunghezza media oraria del pattern di cronologia di ascolto per i file di cronologia da LH9 a LH14.

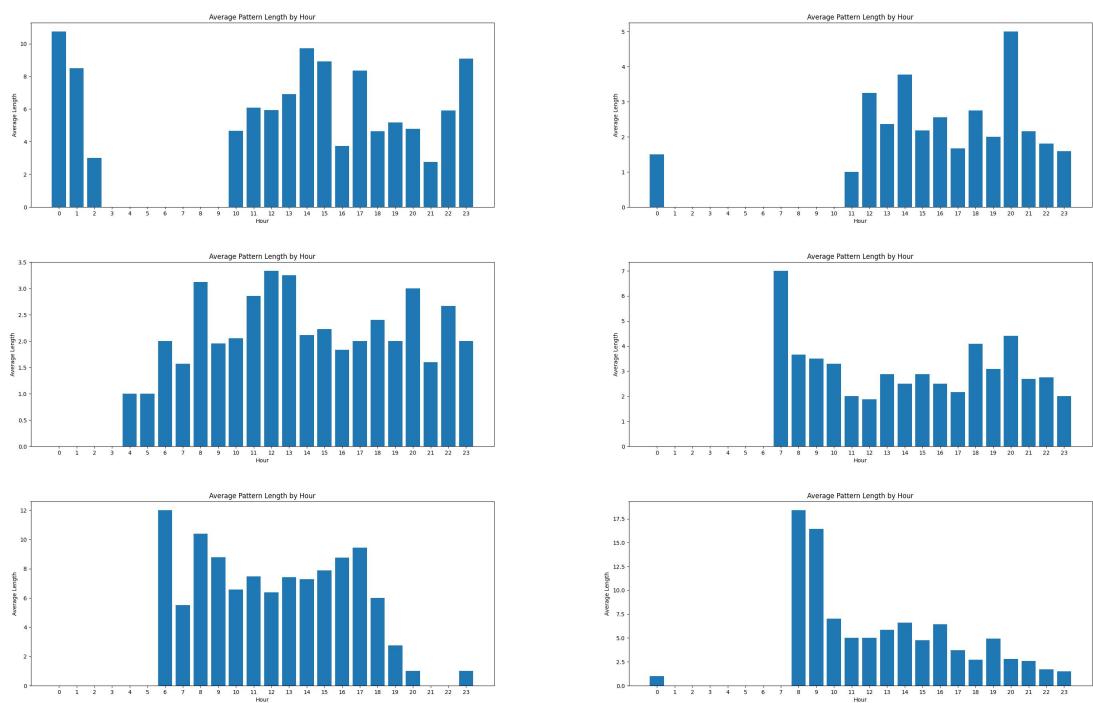


Figura 20: Lunghezza media oraria del pattern di cronologia di ascolto per i file di cronologia da LH15 a LH20.

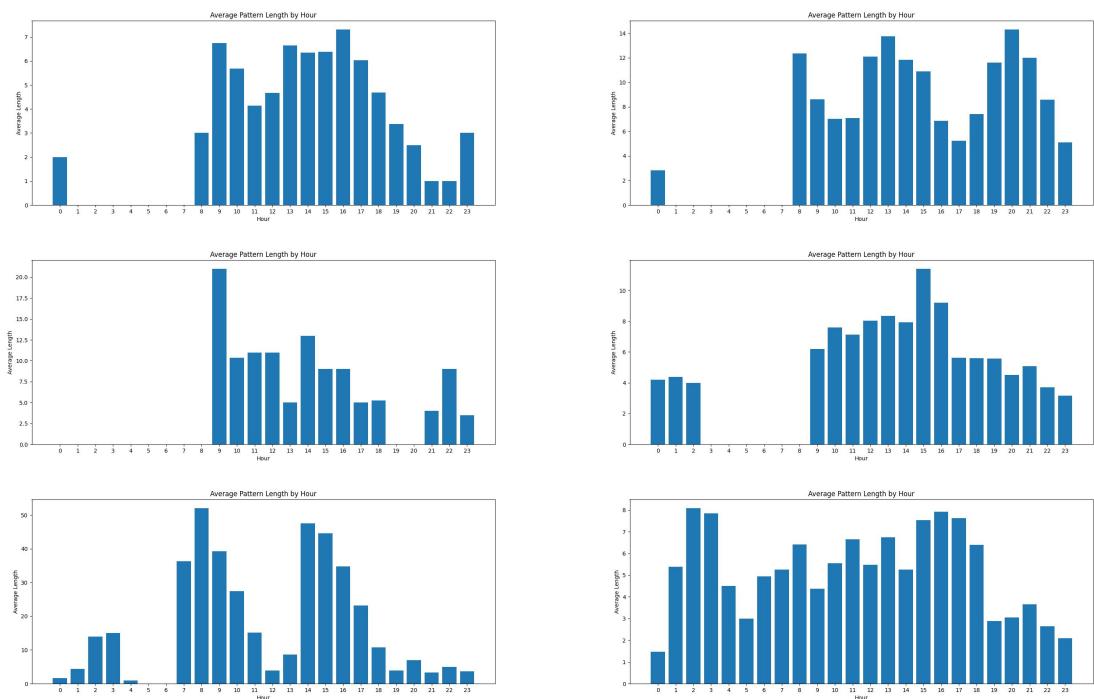


Figura 21: Lunghezza media oraria del pattern di cronologia di ascolto per i file di cronologia da LH21 a LH26.

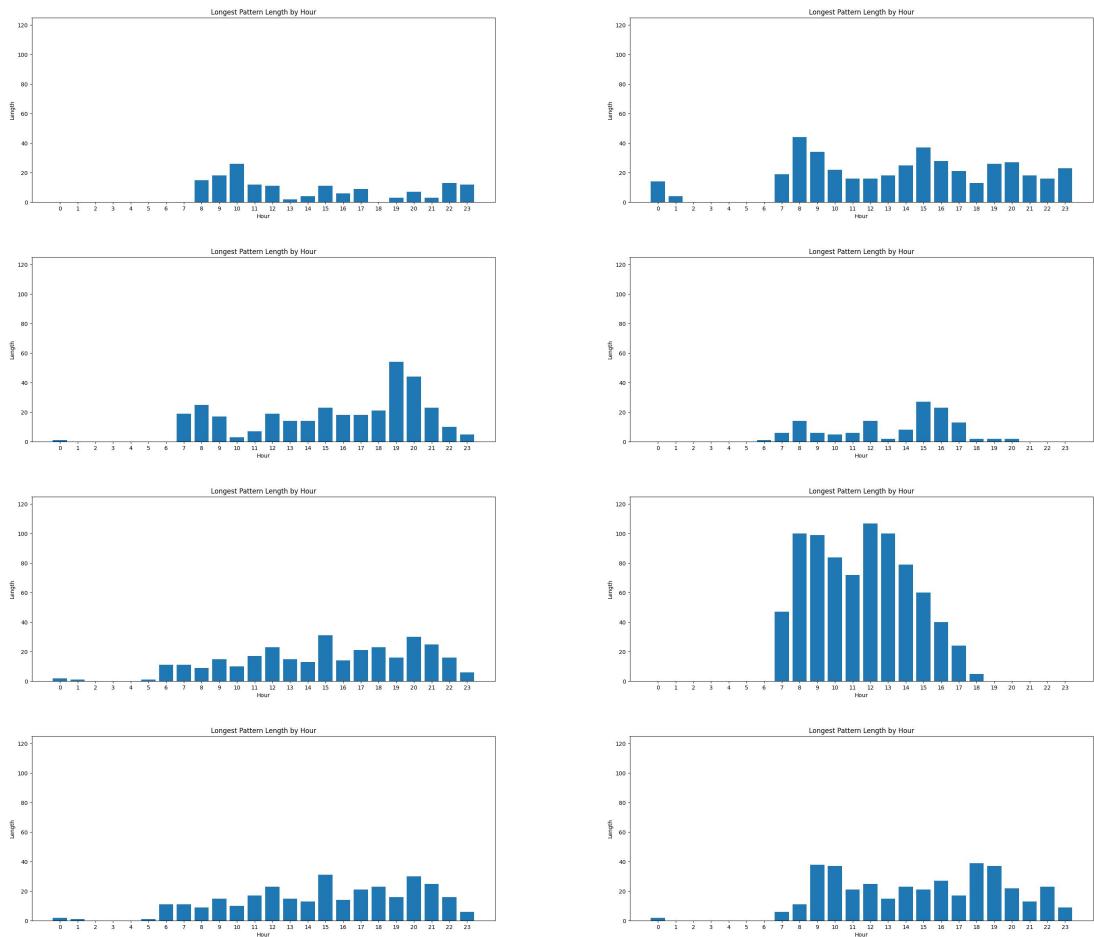


Figura 22: Massima lunghezza oraria del pattern di cronologia di ascolto per i file di cronologia da LH1 a LH8.

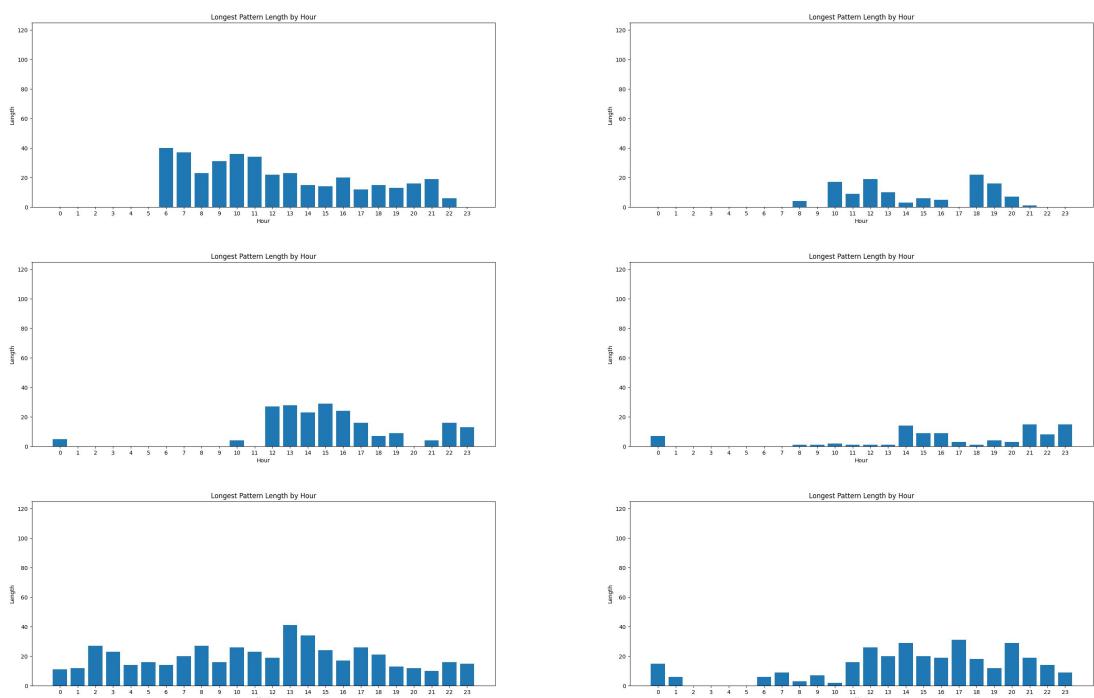


Figura 23: Massima lunghezza oraria del pattern di cronologia di ascolto per i file di cronologia da LH9 a LH14.

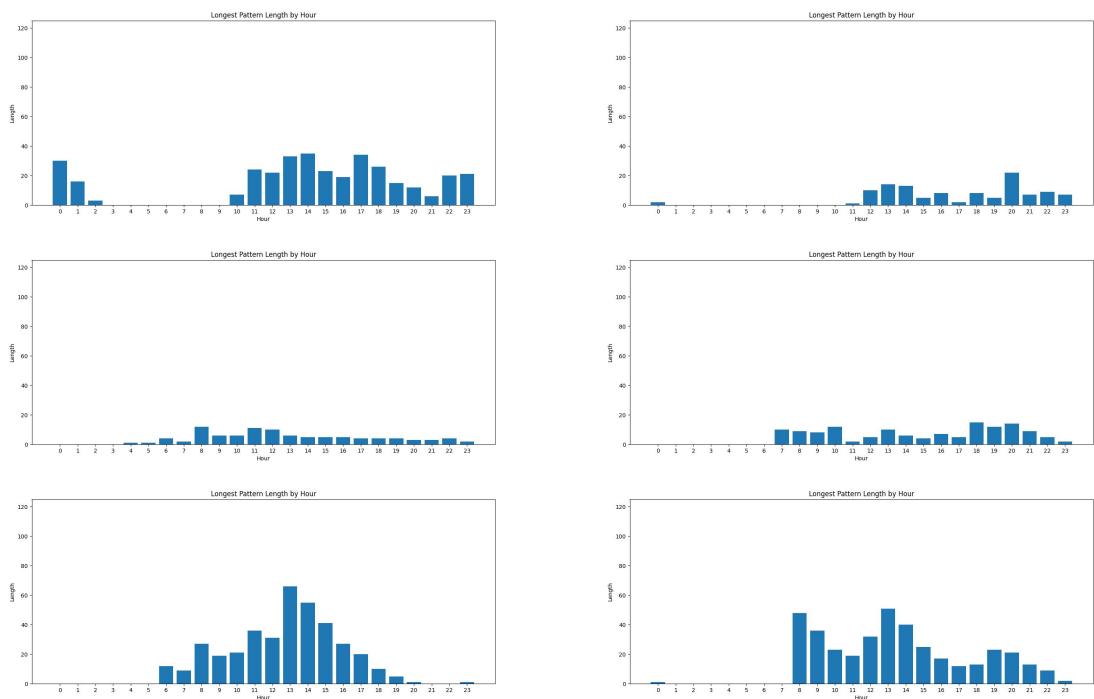


Figura 24: Massima lunghezza oraria del pattern di cronologia di ascolto per i file di cronologia da LH15 a LH20.

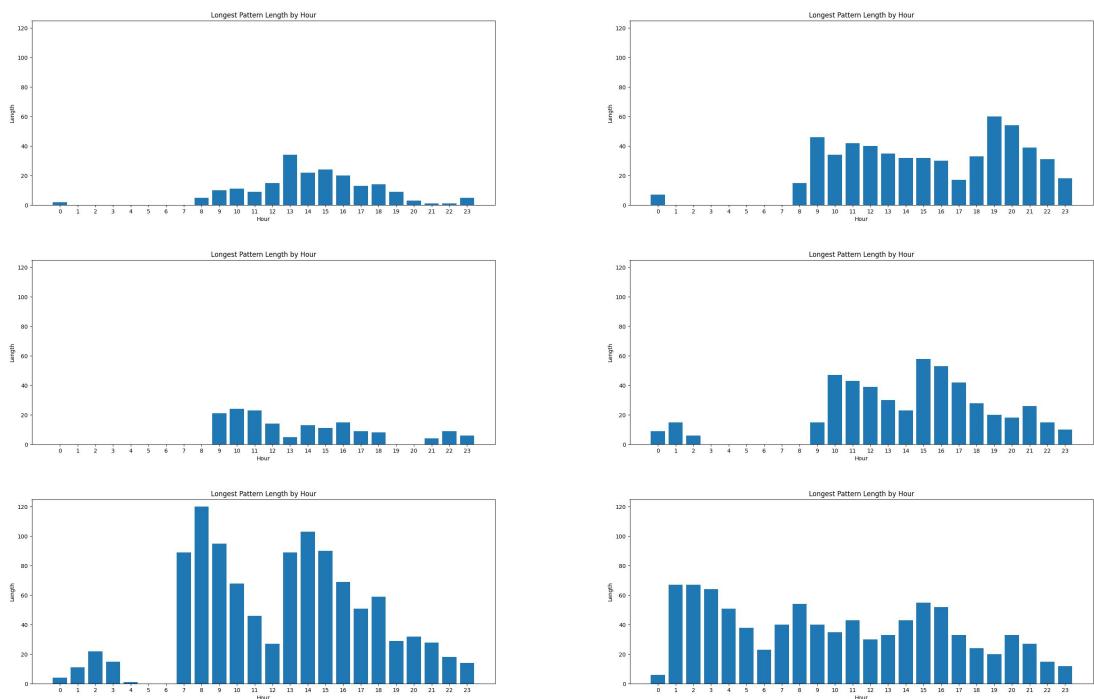


Figura 25: Massima lunghezza oraria del pattern di cronologia di ascolto per i file di cronologia da LH21 a LH26.

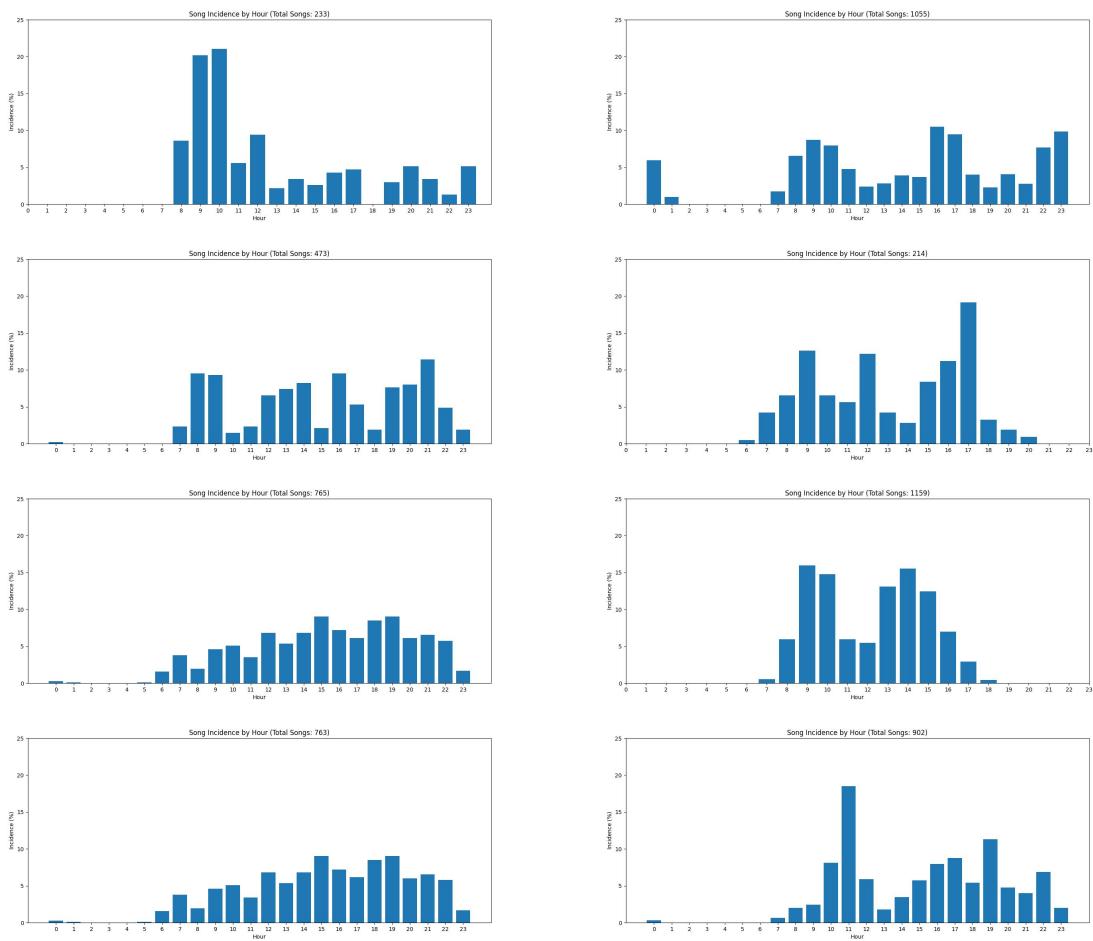


Figura 26: Incidenza percentuale oraria delle canzoni ascoltate sul totale per i file di cronologia da LH1 a LH8.

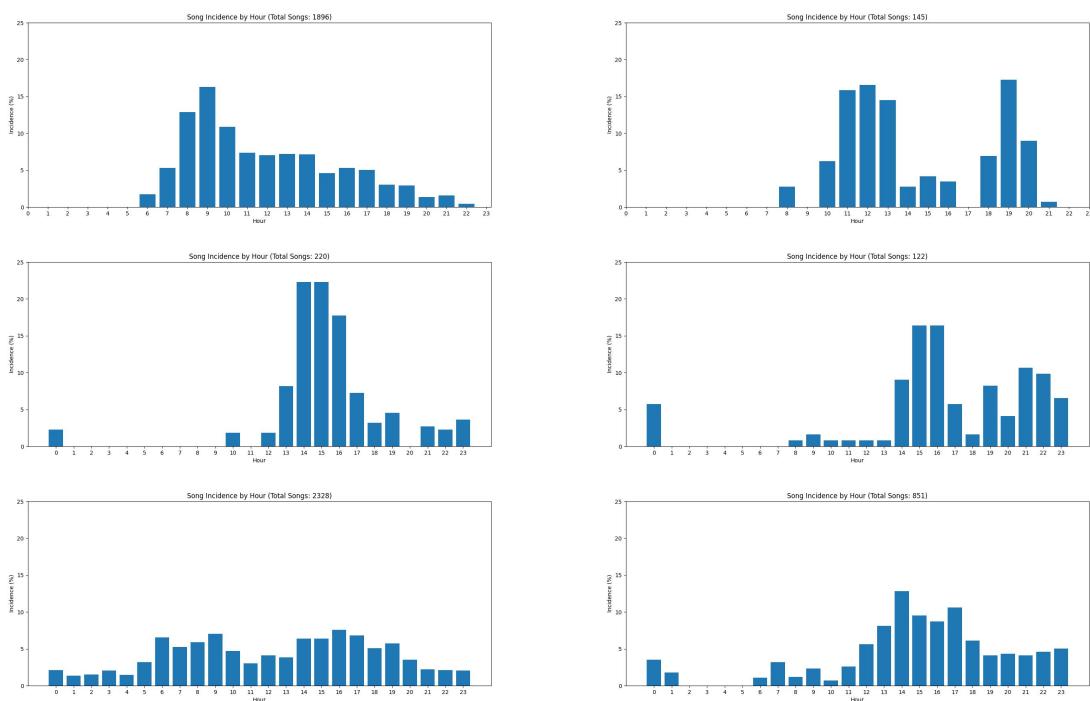


Figura 27: Incidenza percentuale oraria delle canzoni ascoltate sul totale per i file di cronologia da LH9 a LH14.

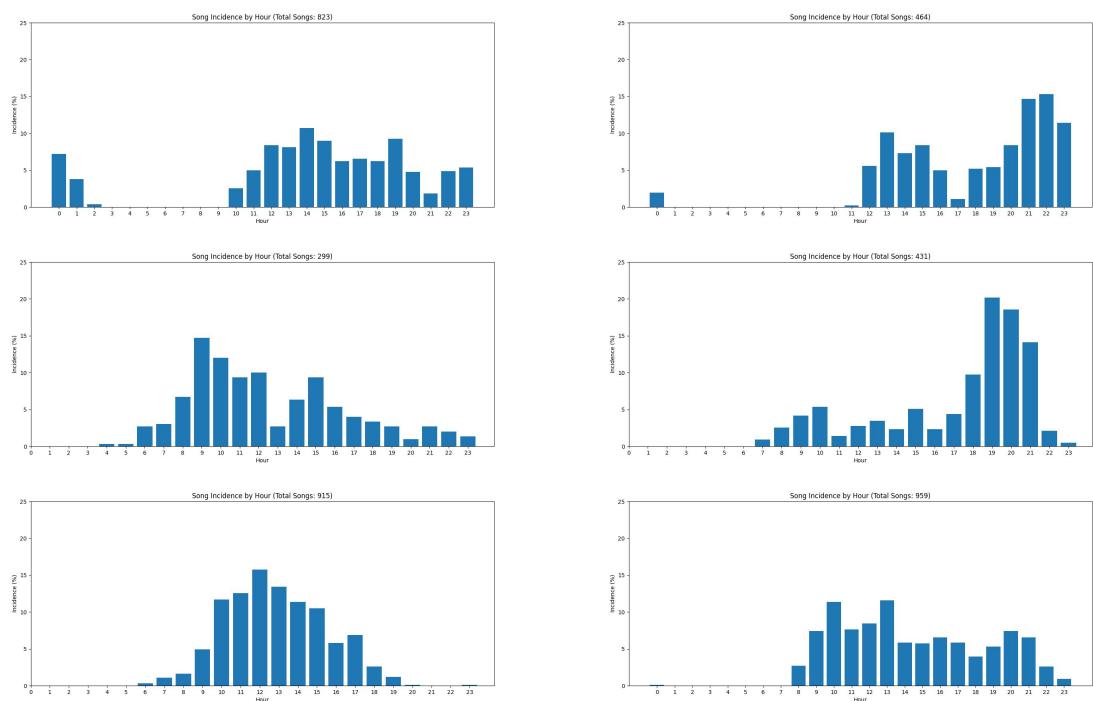


Figura 28: Incidenza percentuale oraria delle canzoni ascoltate sul totale per i file di cronologia da LH15 a LH20.

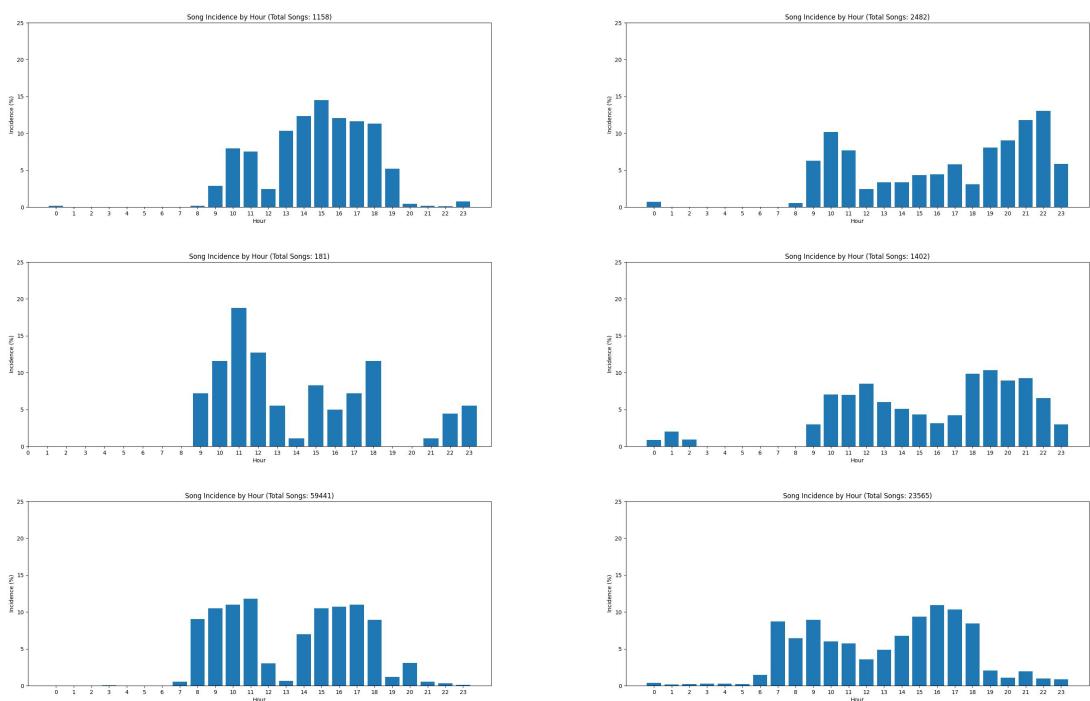


Figura 29: Incidenza percentuale oraria delle canzoni ascoltate sul totale per i file di cronologia da LH21 a LH26.

#### 4.6.2 Playlist Pattern Distance

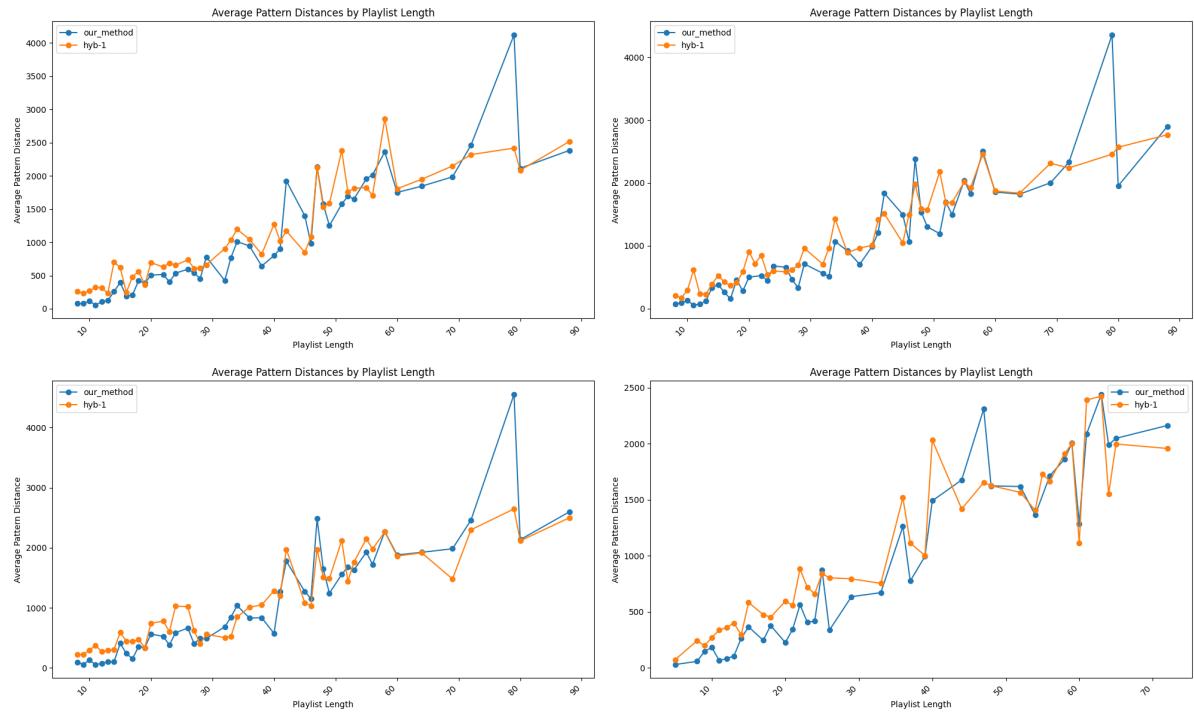


Figura 30: Grafici a linee che mostrano l'andamento della Playlist Pattern Distance media dei metodi SMART e HYB-1 in base alla lunghezza della playlist generata (6 mesi, 1 anno, 1 mese, 3 mesi).

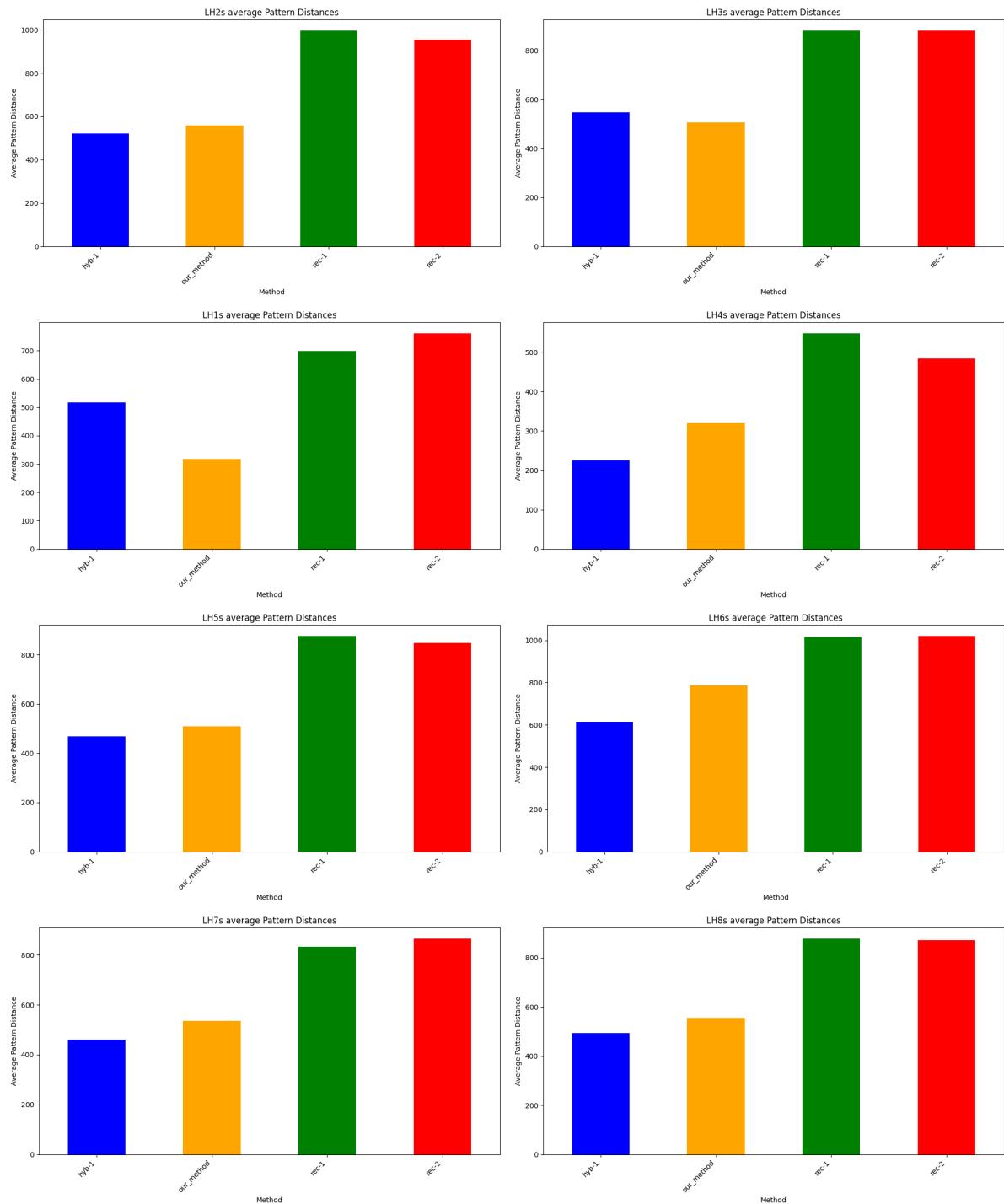


Figura 31: Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH1 a LH8.

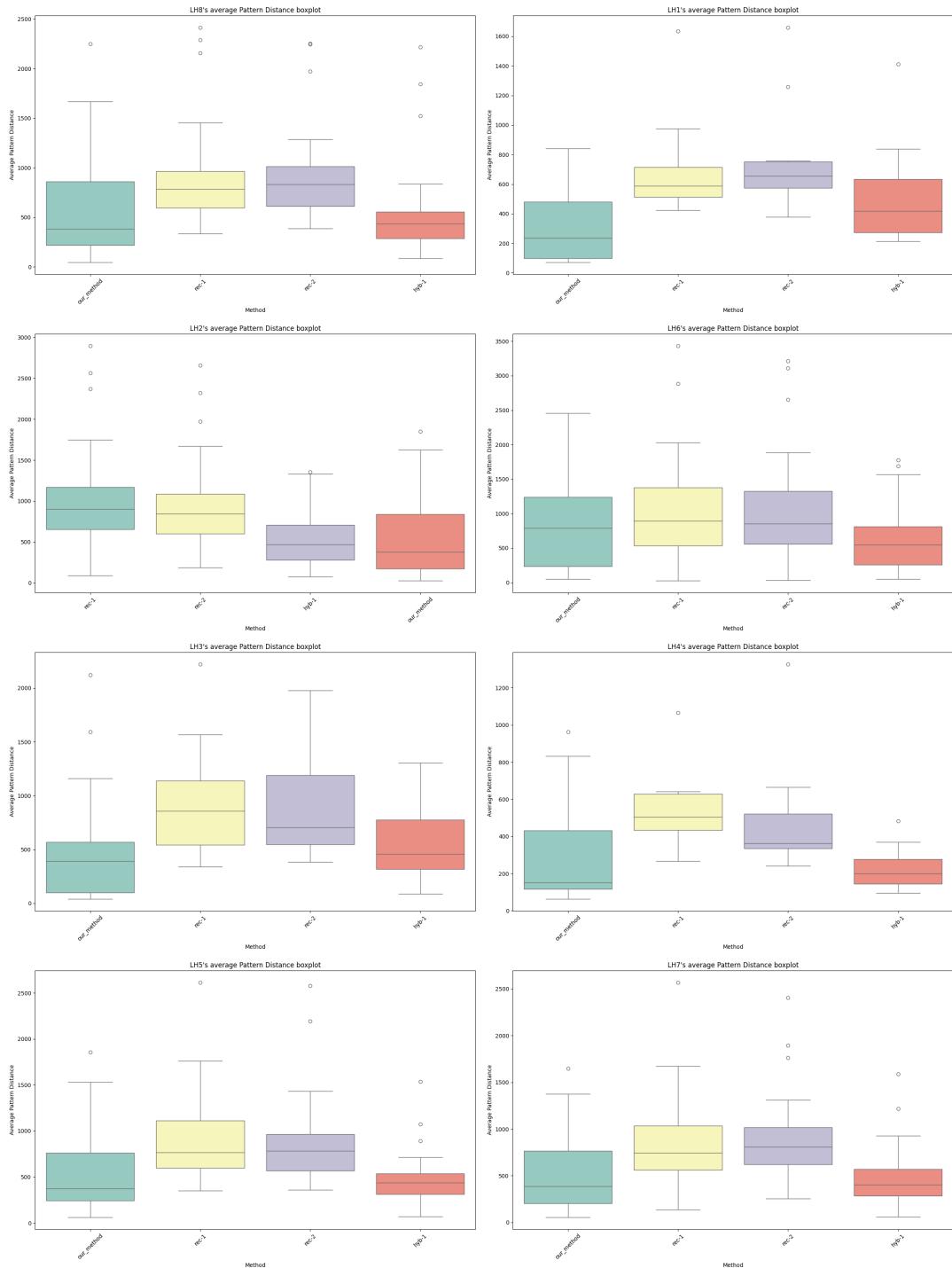


Figura 32: Boxplot della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH1 a LH8.

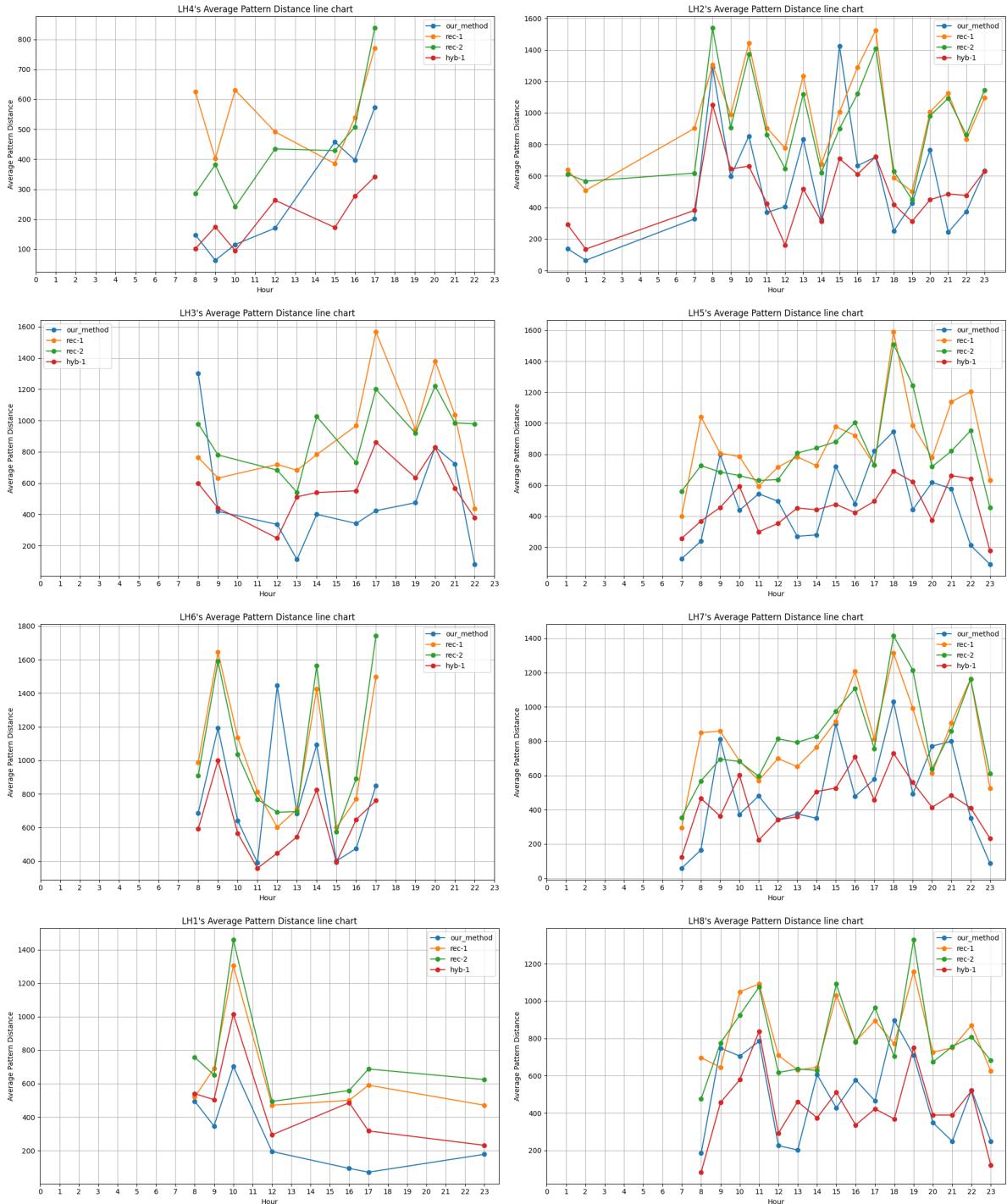


Figura 33: Andamento della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH1 a LH8.

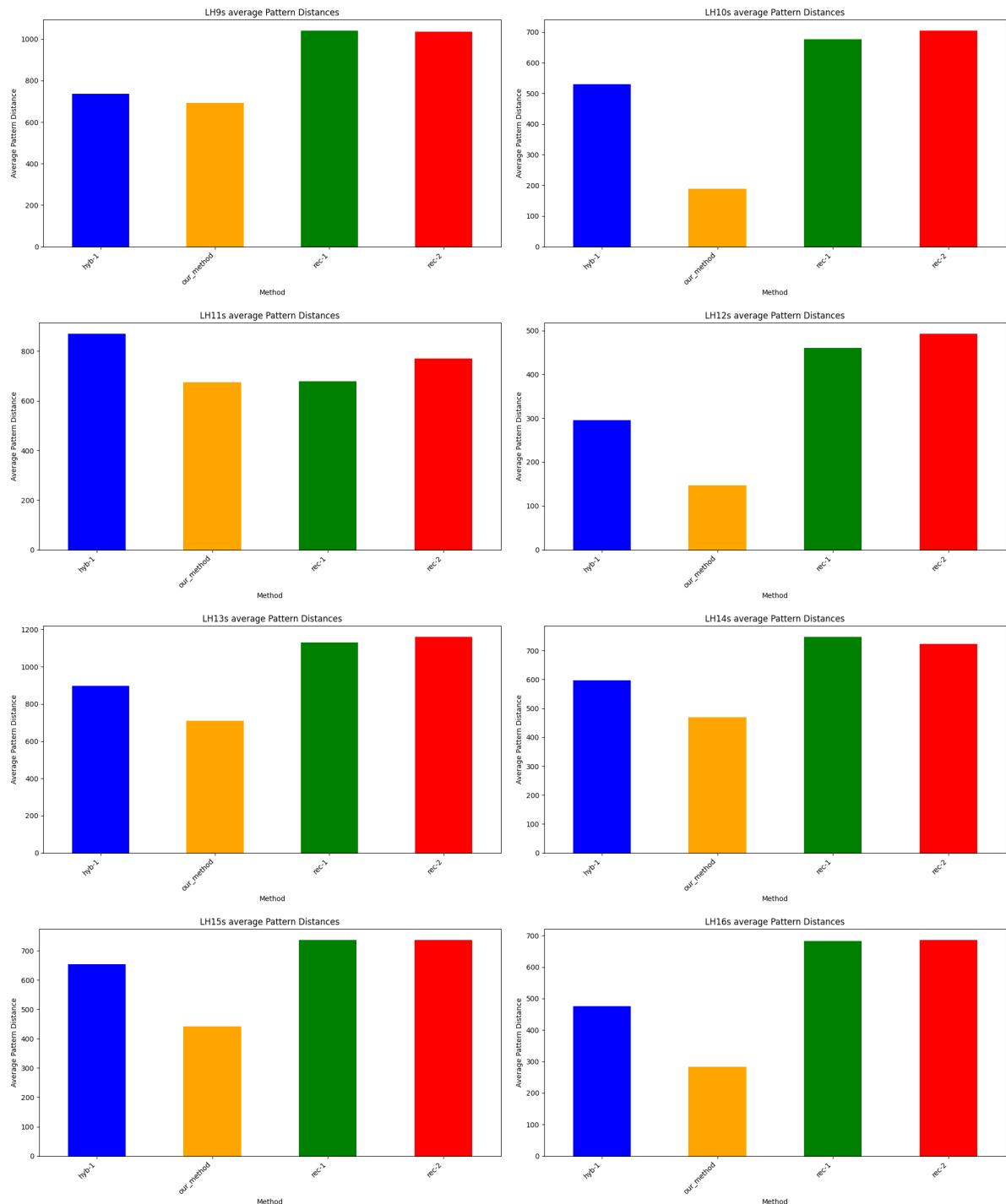


Figura 34: Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH9 a LH16.

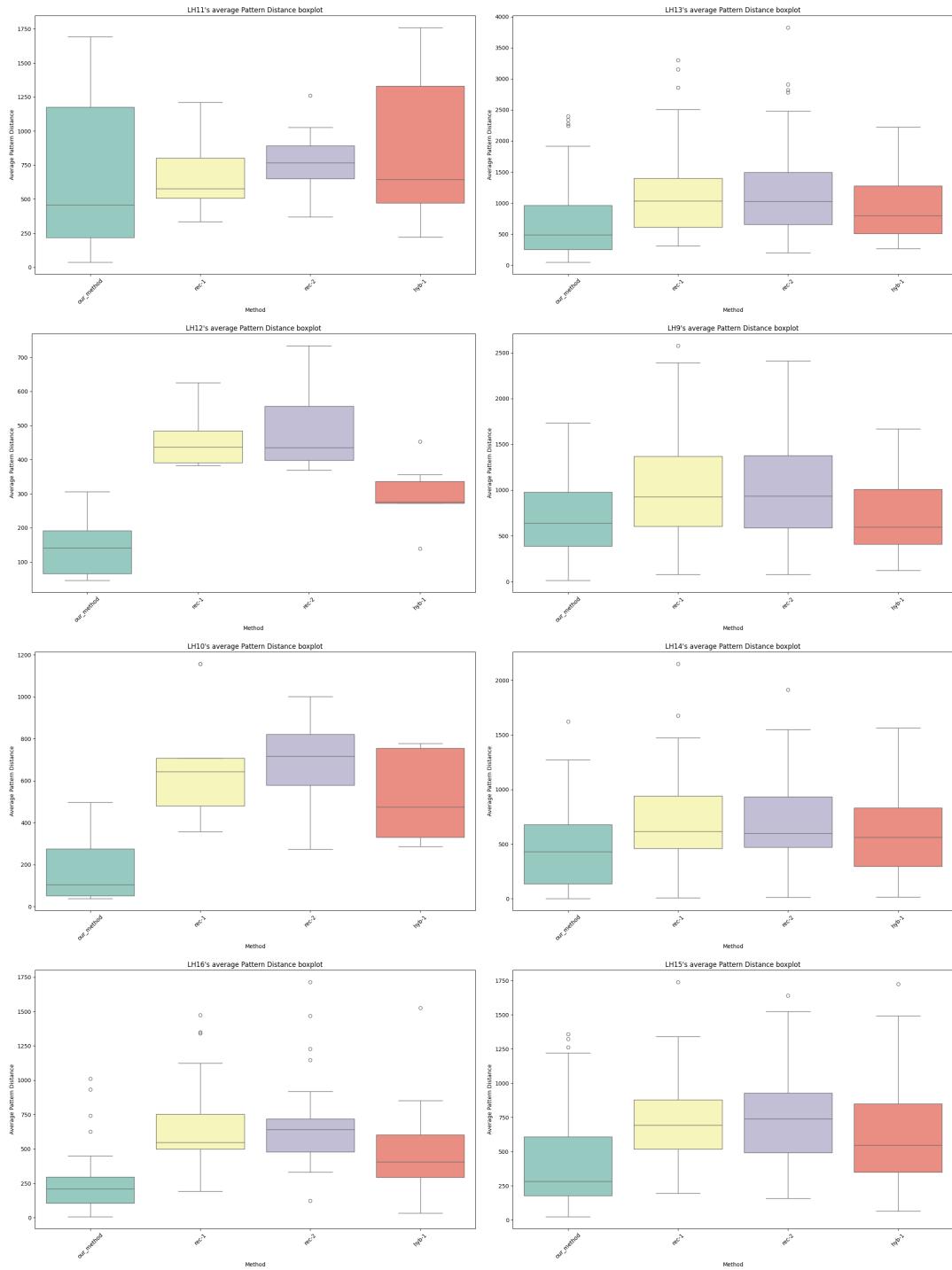


Figura 35: Boxplot della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH9 a LH16.

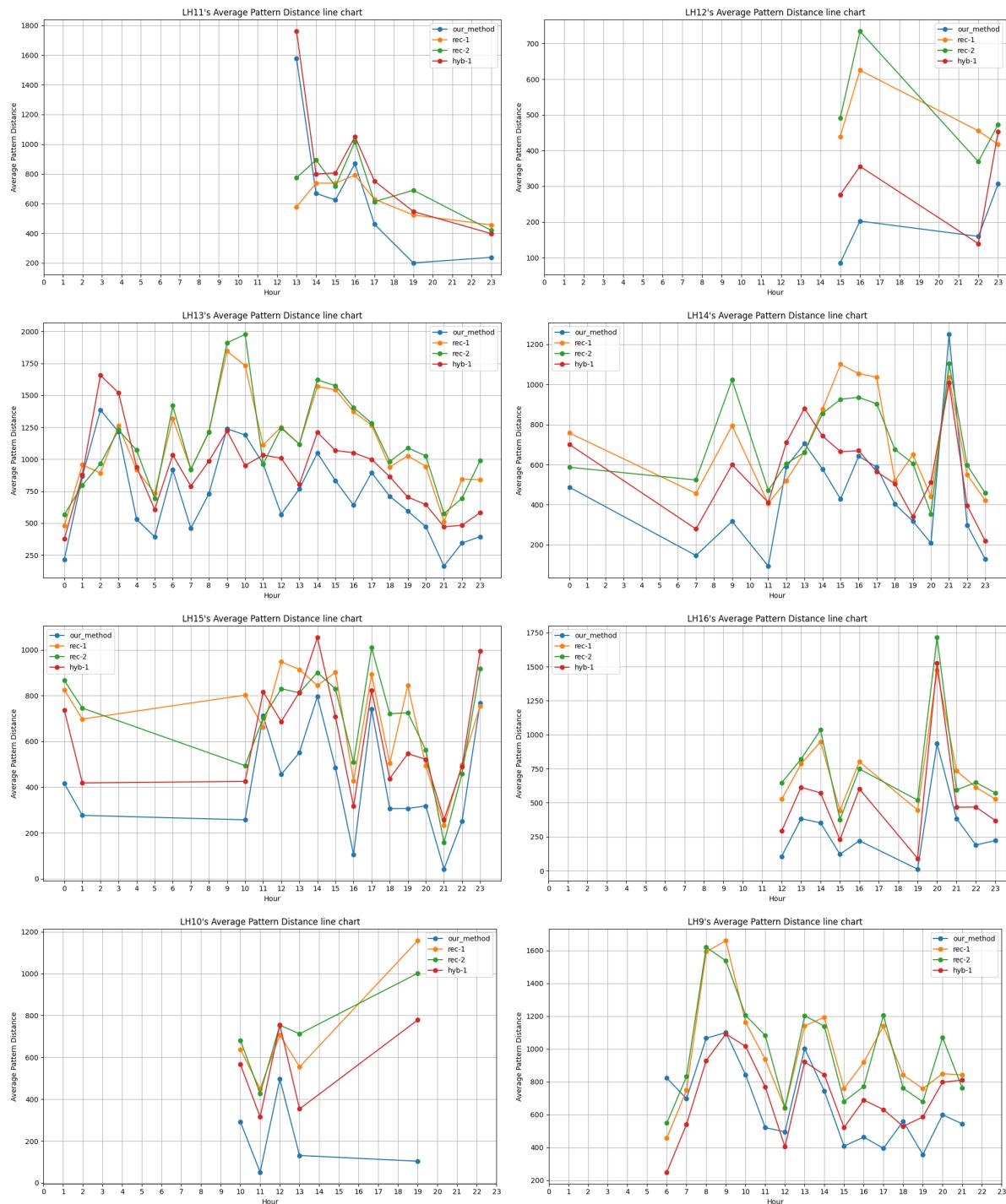


Figura 36: Andamento della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH9 a LH16.

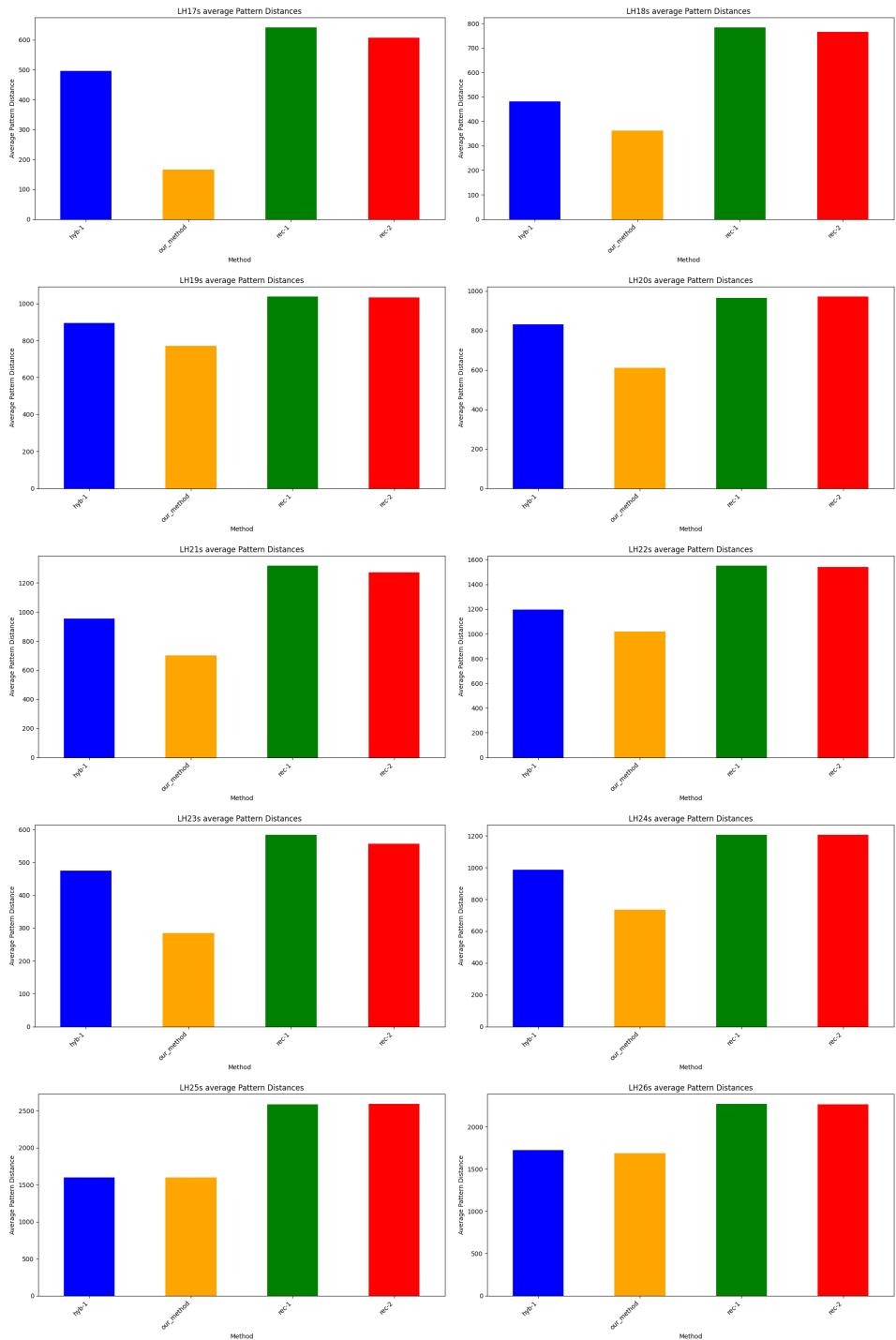


Figura 37: Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH17 a LH26.

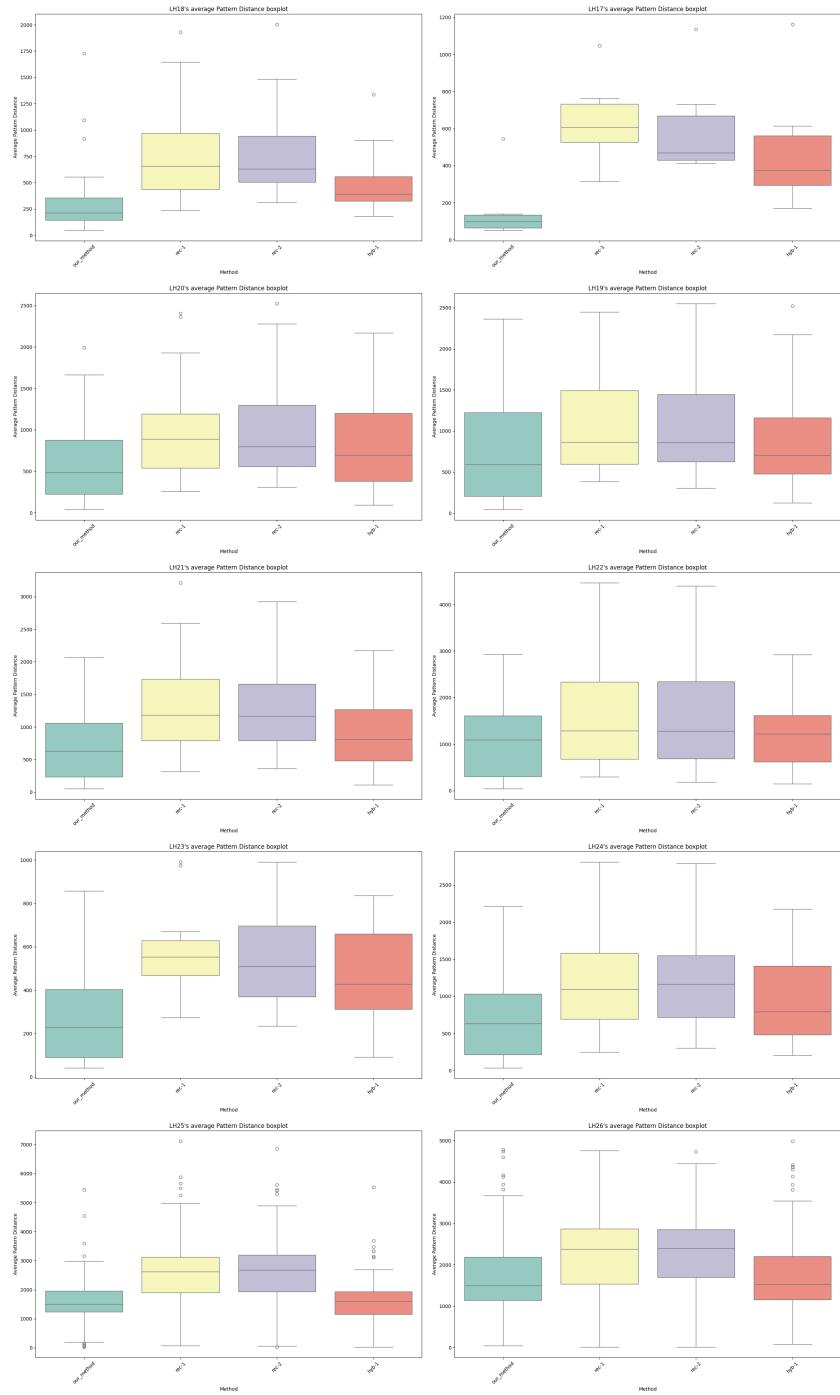


Figura 38: Boxplot della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH17 a LH26.

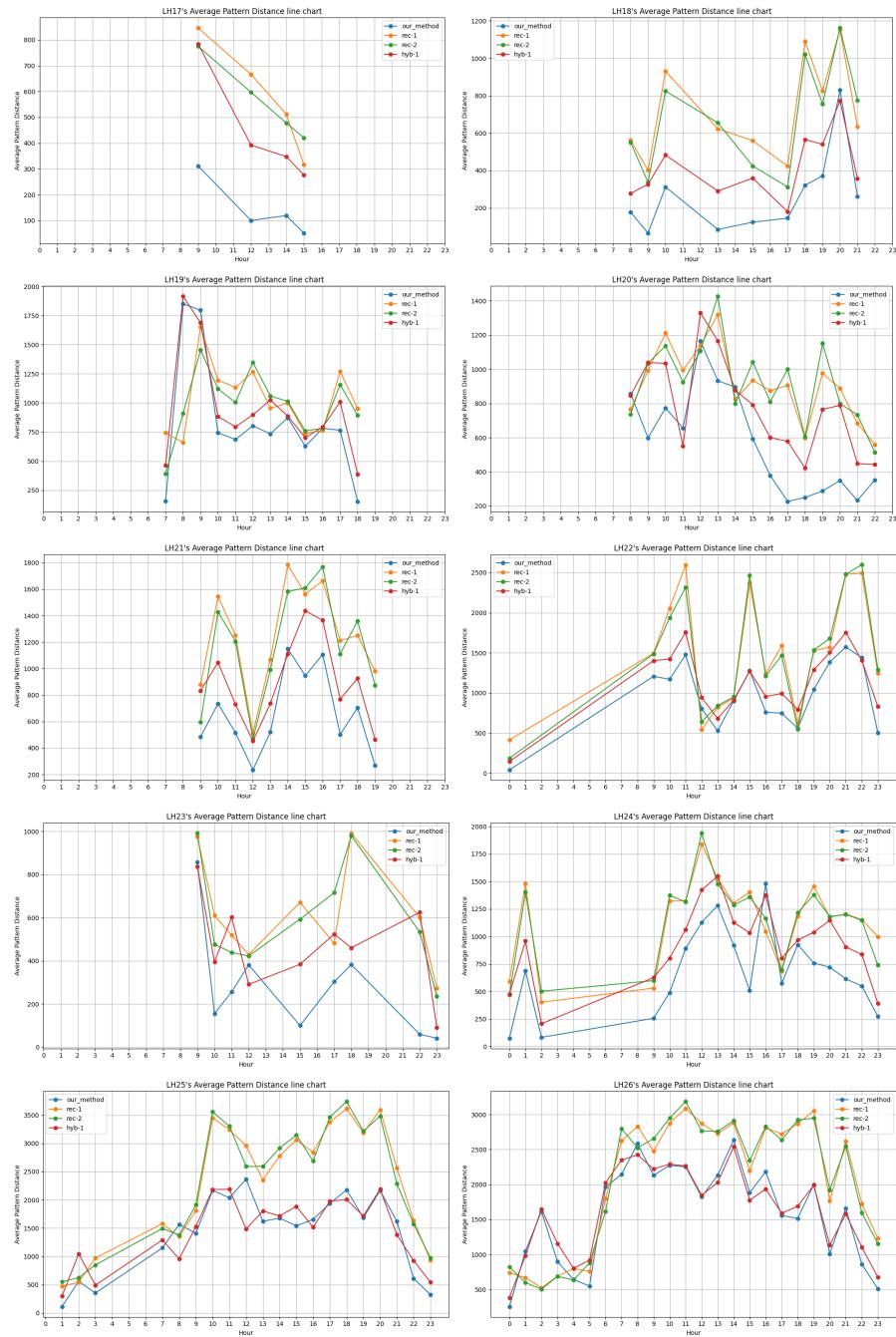


Figura 39: Andamento della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH17 a LH26.

## 4.7 Conclusioni

### 4.7.1 Playlist Pattern Distance

L’analisi della Playlist Pattern Distance per i vari metodi di generazione delle playlist ha rivelato alcune osservazioni interessanti. In generale, il metodo SMART si è dimostrato il più efficiente, con una Playlist Pattern Distance media inferiore (843,53) rispetto agli altri metodi, suggerendo una maggiore qualità delle playlist generate. Seguono HYB-1 (933,49), REC-1 (1310,88) e REC-2 (1308,29).

È emerso che gli utenti con un numero maggiore di canzoni nella loro cronologia tendono ad avere una maggiore variabilità nella Playlist Pattern Distance. Questa osservazione suggerisce che la quantità di dati storici influisca significativamente sulla variabilità dei risultati delle playlist generate. I file di cronologia con un numero elevato di canzoni, come LH25 e LH26, presentano anche una maggiore variabilità interquartile.

La Figura 9 illustra come SMART produca, per la maggior parte dei file di cronologia, playlist con una Pattern Distance inferiore rispetto agli altri metodi. HYB-1 mostra una performance migliore rispetto a REC-1 e REC-2, ma è generalmente meno efficace di SMART. Le linee relative ai metodi REC-1 e REC-2 sono sostanzialmente sovrapposte, indicando differenze trascurabili tra questi due metodi.

Le analisi dei singoli file di cronologia, come LH6 e LH24, evidenziano scenari specifici in cui HYB-1 è più efficace del metodo SMART e viceversa. Ad esempio, HYB-1 è più efficace per il file di cronologia LH6, mentre il nostro metodo supera tutti gli altri per il file di cronologia LH24.

In sintesi, sebbene HYB-1 sia un metodo competitivo, soprattutto per alcuni file di cronologia, il nostro metodo risulta essere generalmente più efficace.

### 4.7.2 Lunghezza della cronologia

Motivati dall’osservazione che la variabilità della Playlist Pattern Distance tende ad aumentare con l’aumentare del numero di canzoni presenti nella cronologia, abbiamo deciso di analizzare più nel dettaglio le differenze tra le playlist generate dal metodo SMART e da HYB-1 per periodi di tempo incrementali (1, 3, 6, 12 mesi).

I risultati di questa fase della sperimentazione suggeriscono che le prestazioni dei metodi variano a seconda del file di cronologia e dell’intervallo di tempo considerato. Mentre per LH25 non ci sono differenze significative tra i metodi, per LH26 il metodo SMART mostra prestazioni significativamente superiori per l’intervallo di 3 mesi.

Questi risultati preliminari, sebbene evidenzino la potenzialità del nostro approccio, indicano anche la necessità di ulteriori approfondimenti. Per giungere a conclusioni definitive, sarà necessario condurre ulteriori esperimenti che coinvolgano un numero maggiore di file di cronologia.

#### **4.7.3 Lunghezza delle playlist**

Analizzando l'andamento della Playlist Pattern Distance media in funzione delle lunghezze delle playlist, abbiamo osservato come, in media, le playlist generate tramite il metodo SMART abbiano una Playlist Pattern Distance minore rispetto a HYB-1 se la loro lunghezza non supera le 50 canzoni.

#### **4.7.4 Altre metriche di confronto**

In questa fase della sperimentazione, abbiamo deciso di confrontare le playlist generate con SMART e con HYB-1 basandoci su metriche differenti dalla sola Playlist Pattern Distance utilizzata nelle fasi precedenti della sperimentazione.

Tra i risultati ottenuti attraverso queste metriche di confronto, quello più significativo ci mostra che SMART tende a produrre playlist più efficaci nel rappresentare le abitudini di ascolto degli utenti rispetto a HYB-1. Infatti, il calcolo della Similarità coseno media delle coppie (NTNA, NTKA) tra playlist e pattern di cronologia risulta significativamente maggiore nel metodo SMART (0.718) rispetto a HYB-1 (0.675). Questo risultato giustifica la nostra scelta di generare quattro song-set e la conseguente scelta di quello migliore in base alle abitudini di ascolto dell'utente.

Per quanto riguarda il calcolo della distanza euclidea normalizzata tra i vettori di caratteristiche medi, il metodo SMART totalizza un punteggio maggiore (0.964 contro 0.95), ma la differenza non è significativa, indicando che i due metodi producono playlist con caratteristiche sonore globalmente simili.

Proseguendo con la nostra analisi, la percentuale media di artisti distinti per playlist (93.69% contro 90.37%) e il numero medio di generi musicali distinti per playlist (29.08 contro 24.75) evidenziano come il metodo SMART generi playlist più varie rispetto a HYB-1.

Un risultato interessante riguarda la popolarità media delle canzoni (38.33% contro 39.84%) e degli artisti (45.98% contro 49.65%) per playlist. In questo caso, HYB-1 genera playlist contenenti canzoni e artisti leggermente più noti rispetto a quelle generate tramite SMART.

Infine, la bassa percentuale media di canzoni uguali (0.9%) e di artisti uguali (7.77%) tra le playlist generate dai due metodi indica che entrambe le tecniche producono playlist con contenuti significativamente diversi.

## 5 Conclusioni

### 5.1 Riepilogo delle motivazioni e degli scopi della tesi

La presente tesi è stata motivata dalla crescente necessità di migliorare i metodi di generazione automatica delle playlist musicali, affinché queste possano meglio riflettere le abitudini di ascolto degli utenti e offrire un'esperienza di ascolto più personalizzata e soddisfacente. La letteratura scientifica preesistente è limitata nell'ambito dei metodi di generazione di playlist altamente personalizzate come quello discusso in questa tesi. L'obiettivo principale del progetto è stato quello di sviluppare e valutare un nuovo metodo di generazione delle playlist, confrontandolo con metodi di complessità inferiore.

### 5.2 Risultati ottenuti

La nostra analisi ha mostrato che il nostro metodo (SMART) di generazione delle playlist è senza alcun dubbio più efficace rispetto ai metodi REC-1 e REC-2. Per quanto riguarda HYB-1, sebbene ci siano scenari specifici in cui quest'ultimo può essere più performante, come nel caso di playlist molto lunghe, il metodo SMART ha dimostrato di essere generalmente più efficace.

La variabilità della Playlist Pattern Distance aumenta con l'aumentare del numero di canzoni nella cronologia, suggerendo che la quantità di dati storici influisce significativamente sulla variabilità dei risultati. Inoltre, le analisi delle metriche di confronto alternative mostrano che SMART tende a produrre playlist più rappresentative delle abitudini di ascolto degli utenti, più varie e con una percentuale maggiore di artisti e generi distinti.

### 5.3 Sviluppi futuri

Nonostante i risultati promettenti, ulteriori esperimenti sono necessari per confermare le nostre conclusioni e per esplorare ulteriormente le potenzialità di SMART. Per il futuro, si propongono i seguenti sviluppi:

- Ampliamento dei test: Conduzione di ulteriori esperimenti con un maggior numero di file di cronologia per validare i risultati ottenuti.
- Ottimizzazione del metodo: Effettuare ulteriori esperimenti per determinare il numero ottimale di canzoni da includere nelle playlist e stabilire la durata ideale della cronologia di ascolto, valutando come queste variabili influenzano la qualità e la personalizzazione delle playlist generate.
- Valutazione soggettiva: considerare il grado di soddisfazione degli utenti per valutare l'efficacia del metodo.

In conclusione, il lavoro svolto ha mostrato la validità del metodo proposto per la generazione di playlist personalizzate, evidenziando al contempo aree di miglioramento e spunti per future ricerche.

## Bibliografia

- [1] P. Knees et al. «Combining audio-based similarity with web-based data to accelerate automatic music playlist generation». In: *Proceedings of the ACM International Conference on Multimedia*. 2006. URL: <https://doi.org/10.1145/1178677.1178699>.
- [2] Steffen Pauws, Wim Verhaegh e Mark Vossen. «Fast Generation of Optimal Music Playlists using Local Search.» In: gen. 2006, pp. 138–143.
- [3] J. Hsu e S. Chung. «Constraint-based playlist generation by applying genetic algorithm». In: *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*. 2011. URL: <https://doi.org/10.1109/icsmc.2011.6083868>.
- [4] S. Chen et al. «Playlist prediction via metric embedding». In: *Proceedings of the ACM Conference on Recommender Systems*. 2012. URL: <https://doi.org/10.1145/2339530.2339643>.
- [5] X. Wang et al. «Exploration in interactive personalized music recommendation». In: *ACM Transactions on Multimedia Computing Communications and Applications* 11.1 (2014), pp. 1–22. URL: <https://doi.org/10.1145/2623372>.
- [6] Dietmar Jannach, Lukas Lerche e Iman Kamehkhosh. «Beyond "Hitting the Hits": Generating Coherent Music Playlist Continuations with the Right Tracks». In: *Proceedings of the 9th ACM Conference on Recommender Systems*. RecSys '15. Vienna, Austria: Association for Computing Machinery, 2015, pp. 187–194. ISBN: 9781450336925. DOI: 10.1145/2792838.2800182. URL: <https://doi.org/10.1145/2792838.2800182>.
- [7] Rachel M. Bittner et al. «Automatic Playlist Sequencing and Transitions». In: *International Society for Music Information Retrieval Conference*. 2017. URL: <https://api.semanticscholar.org/CorpusID:11244249>.
- [8] Massimo Quadrana. «Algorithms for sequence-aware recommender systems». In: (2017).
- [9] Dietmar Jannach, Iman Kamehkhosh e Geoffray Bonnin. «Music Recommendations». In: *Collaborative Recommendations*. HAL Science, 2018. URL: <https://hal.science/hal-02476928>.
- [10] Shun-Yao Shih e Heng-Yu Chi. *Automatic, Personalized, and Flexible Playlist Generation using Reinforcement Learning*. 2018. arXiv: 1809.04214 [cs.CL]. URL: <https://arxiv.org/abs/1809.04214>.
- [11] I. Kamehkhosh, G. Bonnin e D. Jannach. «Effects of recommendations on the playlist creation behavior of users». In: *User Modeling and User-Adapted Interaction* 30.2 (2019), pp. 285–322. URL: <https://doi.org/10.1007/s11257-019-09237-4>.

- [12] Elad Liebman, Maytal Saar-Tsechansky e Peter Stone. «The Right Music at the Right Time: Adaptive Personalized Playlists Based on Sequence Modeling». In: *MIS Quarterly* 43 (gen. 2019), pp. 765–786. DOI: 10.25300/MISQ/2019/14750.
- [13] Markus Schedl. «Deep learning in music recommendation systems». In: *Frontiers in Applied Mathematics and Statistics* 5 (2019), p. 457883. URL: <https://doi.org/10.3389/fams.2019.00044>.
- [14] Dip Paul e Subhradeep Kundu. «A survey of music recommendation systems with a proposed music recommendation system». In: *Emerging Technology in Modelling and Graphics: Proceedings of IEM Graph 2018*. Springer. 2020, pp. 279–285. URL: [http://dx.doi.org/10.1007/978-981-13-7403-6\\_26](http://dx.doi.org/10.1007/978-981-13-7403-6_26).
- [15] Amrita Nair et al. «Emotion Based Music Playlist Recommendation System using Interactive Chatbot». In: *2021 6th International Conference on Communication and Electronics Systems (ICCES)*. 2021, pp. 1767–1772. DOI: 10.1109/ICCES51350.2021.9489138.
- [16] Kevin Patel e Rajeev Kumar Gupta. «Song playlist generator system based on facial expression and song mood». In: *2021 international conference on artificial intelligence and machine vision (AIMV)*. IEEE. 2021, pp. 1–6.
- [17] Marco Furini e Manuela Montangero. «Automatic and Personalized Sequencing of Music Playlists». In: *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2022, p. 4. DOI: 10.1109/ICDCSW56584.2022.00046.
- [18] Adria Mallol-Ragolta et al. «sustAGE 1.0 – First Prototype, Use Cases, and Usability Evaluation». In: gen. 2022.
- [19] K. Sakurai et al. «Controllable music playlist generation based on knowledge graph and reinforcement learning». In: *Sensors* 22.10 (2022), p. 3722. URL: <https://doi.org/10.3390/s22103722>.
- [20] Keigo Sakurai et al. «Controllable Music Playlist Generation Based on Knowledge Graph and Reinforcement Learning». In: *Sensors* 22.10 (2022). ISSN: 1424-8220. DOI: 10.3390/s22103722. URL: <https://www.mdpi.com/1424-8220/22/10/3722>.
- [21] PJ Upadhyay et al. «Mood based music playlist generator using convolutional neural network». In: *Int. J. Res. Appl. Sci. Eng. Technol.* 10.3 (2022), pp. 531–540.
- [22] W. Bendada et al. «A scalable framework for automatic playlist continuation on music streaming services». In: (2023). URL: <https://doi.org/10.1145/3539618.3591628>.
- [23] Shaurya Gaur. «Mood-dynamic playlist: interpolating a path of emotions using a KNN algorithm». In: (2023).

- [24] M. Rawat et al. «Moodsic - a mood based music player». In: (2023). URL: <https://doi.org/10.21203/rs.3.rs-1507602/v1>.
- [25] Federico Tomasi et al. «Automatic Music Playlist Generation via Simulation-based Reinforcement Learning». In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2023, pp. 4948–4957.
- [26] Makarand Velankar e Parag Kulkarni. «Music Recommendation Systems: Overview and Challenges». In: Cham: Springer International Publishing, 2023, pp. 51–69. ISBN: 978-3-031-18444-4. DOI: 10.1007/978-3-031-18444-4\_3. URL: [https://doi.org/10.1007/978-3-031-18444-4\\_3](https://doi.org/10.1007/978-3-031-18444-4_3).
- [27] Yashar Deldjoo, Markus Schedl e Peter Knees. «Content-driven music recommendation: Evolution, state of the art, and challenges». In: *Computer Science Review* 51 (2024), p. 100618. URL: <https://doi.org/10.48550/arXiv.2107.11803>.
- [28] Elias Mann. «Context Aware Music Recommendation and Playlist Generation». In: *SMU Journal of Undergraduate Research* 8.2 (2024), p. 2. DOI: 10.25172/jour.8.2.1.
- [29] Spotify. *Spotify Web API*. 2024. URL: <https://developer.spotify.com/documentation/web-api/>.

## Elenco delle figure

1	Architettura ad alto livello di SMART . . . . .	20
2	Caratteristiche audio di basso livello utilizzate per rappresentare le canzoni . . . . .	23
3	Euristica lineare . . . . .	27
4	Euristica a sfera . . . . .	28
5	Incidenza percentuale di ascolto delle canzoni per ogni ora, tenendo conto di tutti i giorni della settimana. In questo caso, il file di cronologia di riferimento è LH22. . . . .	122
6	Incidenza percentuale di ascolto delle canzoni per ogni ora, tenendo conto soltanto del giorno <b>Sabato</b> . In questo caso, il file di cronologia di riferimento è LH22. . . . .	123
7	Lunghezza media del pattern di cronologia per ogni ora, tenendo conto di tutti i giorni della settimana. In questo caso, il file di cronologia di riferimento è LH20. . . . .	123
8	Lunghezza media del pattern di cronologia per ogni ora, tenendo in considerazione soltanto <b>Sabato</b> . In questo caso, il file di cronologia di riferimento è LH20 . . . . .	124
9	Grafico a linee globale che mostra come varia la Playlist Pattern Distance media per ogni metodo in funzione dei diversi file di cronologia. . . . .	134
10	Iistogramma della Pattern Distance media per ogni metodo considerando i file di cronologia LH6 e LH24. . . . .	136
11	Boxplot della Pattern Distance media per ogni metodo considerando il file di cronologia LH6 e LH24. . . . .	137
12	Grafico a linee della Pattern Distance media per ogni metodo considerando i file di cronologia LH6 e LH24. . . . .	139
13	Andamento della Pattern Distance media di SMART e HYB-1 relativo al file di cronologia LH25 in base alla lunghezza della cronologia (1,3,6,12 mesi). . . . .	141
14	Andamento della Pattern Distance media di SMART e HYB-1 relativo al file di cronologia LH26 in base alla lunghezza della cronologia (1,3,6,12 mesi). . . . .	142
15	Andamento della Pattern Distance media di SMART e HYB-1 relativo ai file di cronologia LH25 e LH26 considerando tutti i periodi di ascolto (1,3,6,12 mesi). . . . .	143
16	Boxplot della Pattern Distance media per metodo e intervallo di tempo relativo ai file di cronologia LH25 e LH26. . . . .	145
17	Grafico a linee che confronta l'andamento della Playlist Pattern Distance media dei metodi SMART e HYB-1 in base alla lunghezza della playlist generata. . . . .	147
18	Lunghezza media oraria del pattern di cronologia di ascolto per i file di cronologia da LH1 a LH8. . . . .	151

19	Lunghezza media oraria del pattern di cronologia di ascolto per i file di cronologia da LH9 a LH14. . . . .	152
20	Lunghezza media oraria del pattern di cronologia di ascolto per i file di cronologia da LH15 a LH20. . . . .	153
21	Lunghezza media oraria del pattern di cronologia di ascolto per i file di cronologia da LH21 a LH26. . . . .	154
22	Massima lunghezza oraria del pattern di cronologia di ascolto per i file di cronologia da LH1 a LH8. . . . .	155
23	Massima lunghezza oraria del pattern di cronologia di ascolto per i file di cronologia da LH9 a LH14. . . . .	156
24	Massima lunghezza oraria del pattern di cronologia di ascolto per i file di cronologia da LH15 a LH20. . . . .	157
25	Massima lunghezza oraria del pattern di cronologia di ascolto per i file di cronologia da LH21 a LH26. . . . .	158
26	Incidenza percentuale oraria delle canzoni ascoltate sul totale per i file di cronologia da LH1 a LH8. . . . .	159
27	Incidenza percentuale oraria delle canzoni ascoltate sul totale per i file di cronologia da LH9 a LH14. . . . .	160
28	Incidenza percentuale oraria delle canzoni ascoltate sul totale per i file di cronologia da LH15 a LH20. . . . .	161
29	Incidenza percentuale oraria delle canzoni ascoltate sul totale per i file di cronologia da LH21 a LH26. . . . .	162
30	Grafici a linee che mostrano l'andamento della Playlist Pattern Distance media dei metodi SMART e HYB-1 in base alla lunghezza della playlist generata (6 mesi, 1 anno, 1 mese, 3 mesi). . . . .	163
31	Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH1 a LH8. . . . .	164
32	Boxplot della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH1 a LH8. . . . .	165
33	Andamento della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH1 a LH8. . . . .	166
34	Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH9 a LH16. . . . .	167
35	Boxplot della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH9 a LH16. . . . .	168
36	Andamento della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH9 a LH16. . . . .	169
37	Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH17 a LH26. . . . .	170

38	Boxplot della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH17 a LH26. . . . .	171
39	Andamento della Playlist Pattern Distance media di ogni metodo per i file di cronologia da LH17 a LH26. . . . .	172

## Elenco delle tabelle

1	Numero totale di canzoni e numero di canzoni per ogni giorno della settimana per ogni file di cronologia. . . . .	121
2	Lunghezza media del pattern di cronologia per ogni ora del giorno e per ogni file di cronologia. . . . .	125
3	Lunghezza massima del pattern di cronologia per ogni giorno della settimana e per ogni file di cronologia. . . . .	126
4	Playlist Pattern Distance media per ogni file di cronologia e per ogni metodo	132
5	Differenze Interquartili della Playlist Pattern Distance media per ciascun file di cronologia . . . . .	133
6	Tabella riassuntiva dei risultati ottenuti analizzando le playlist generate tramite SMART e HYB-1 in funzione della lunghezza temporale del file di cronologia. I valori sono stati arrotondati al numero intero più vicino.	146
7	Risultati dei Metodi di Confronto tra le Playlist generate con HYB-1 e con SMART . . . . .	150

## Elenco dei listati

1	Vincoli imposti dal sistema di raccomandazioni di Spotify . . . . .	74
---	---	----

# A Appendix

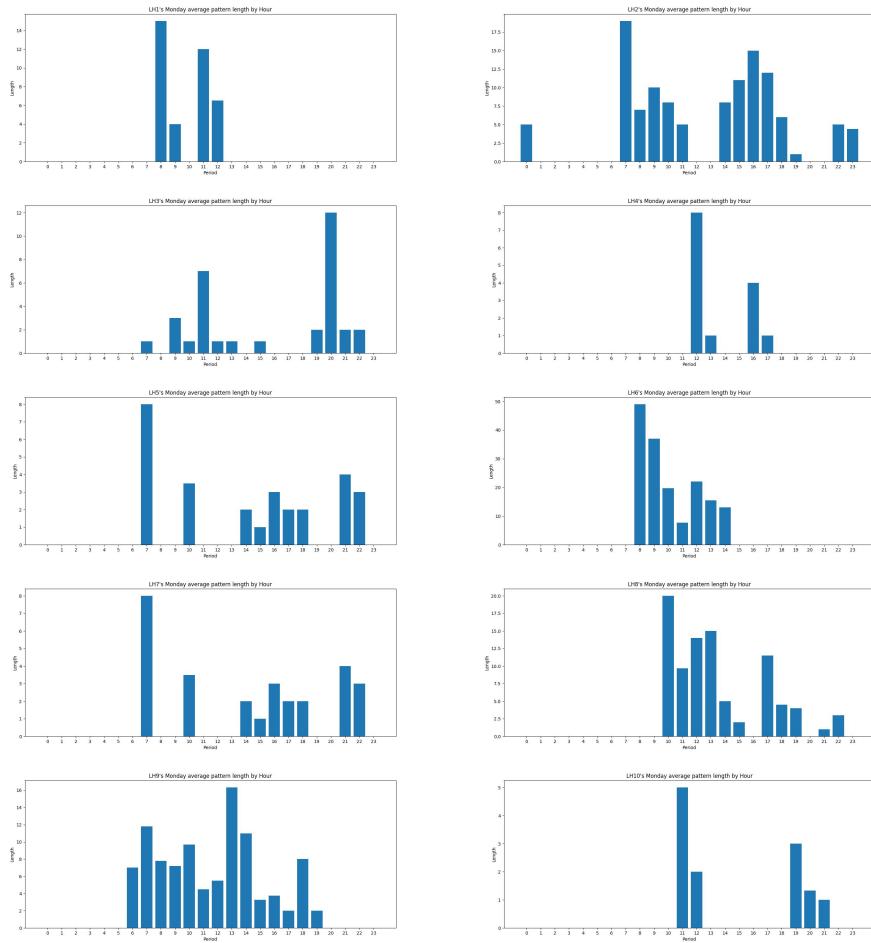


Figura 40: Lunghezza media oraria dei pattern di cronologia di Lunedì per i file di cronologia da LH1 a LH10.

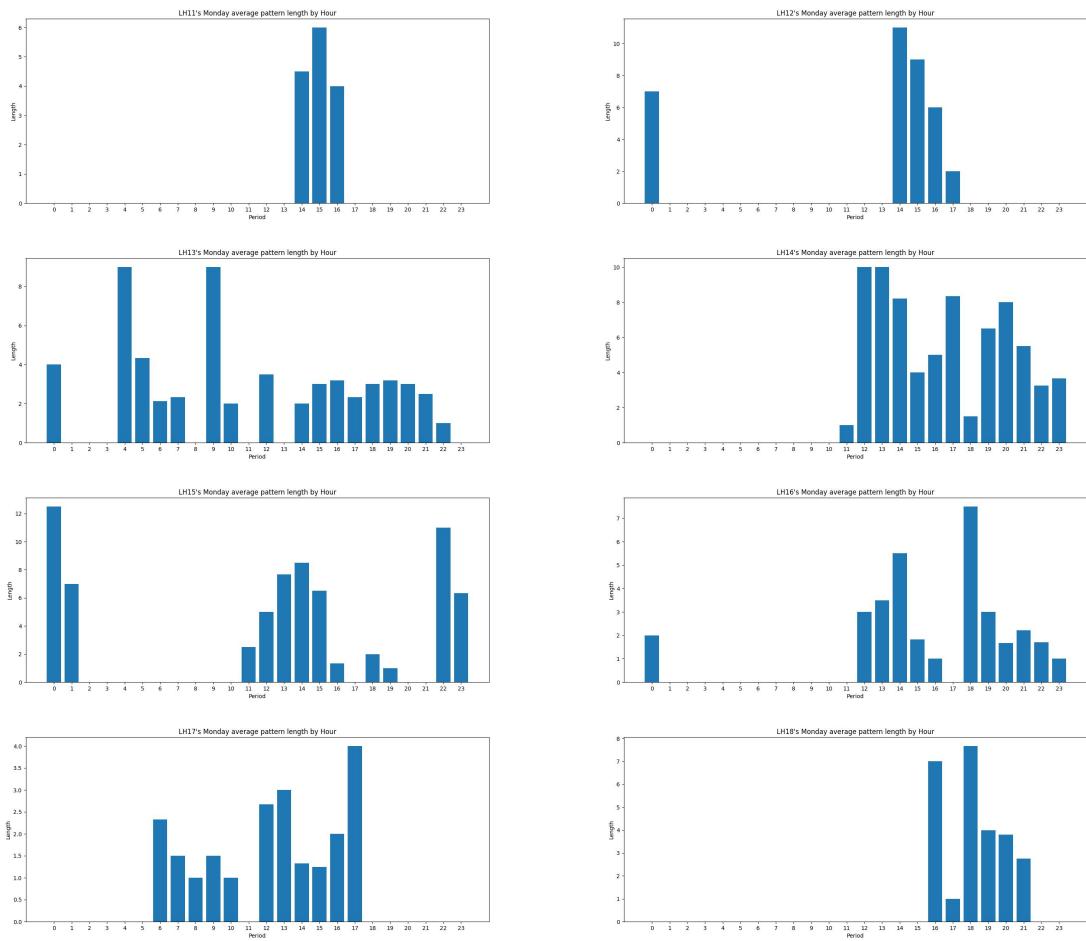


Figura 41: Lunghezza media oraria dei pattern di cronologia di Lunedì per i file di cronologia da LH11 a LH18.

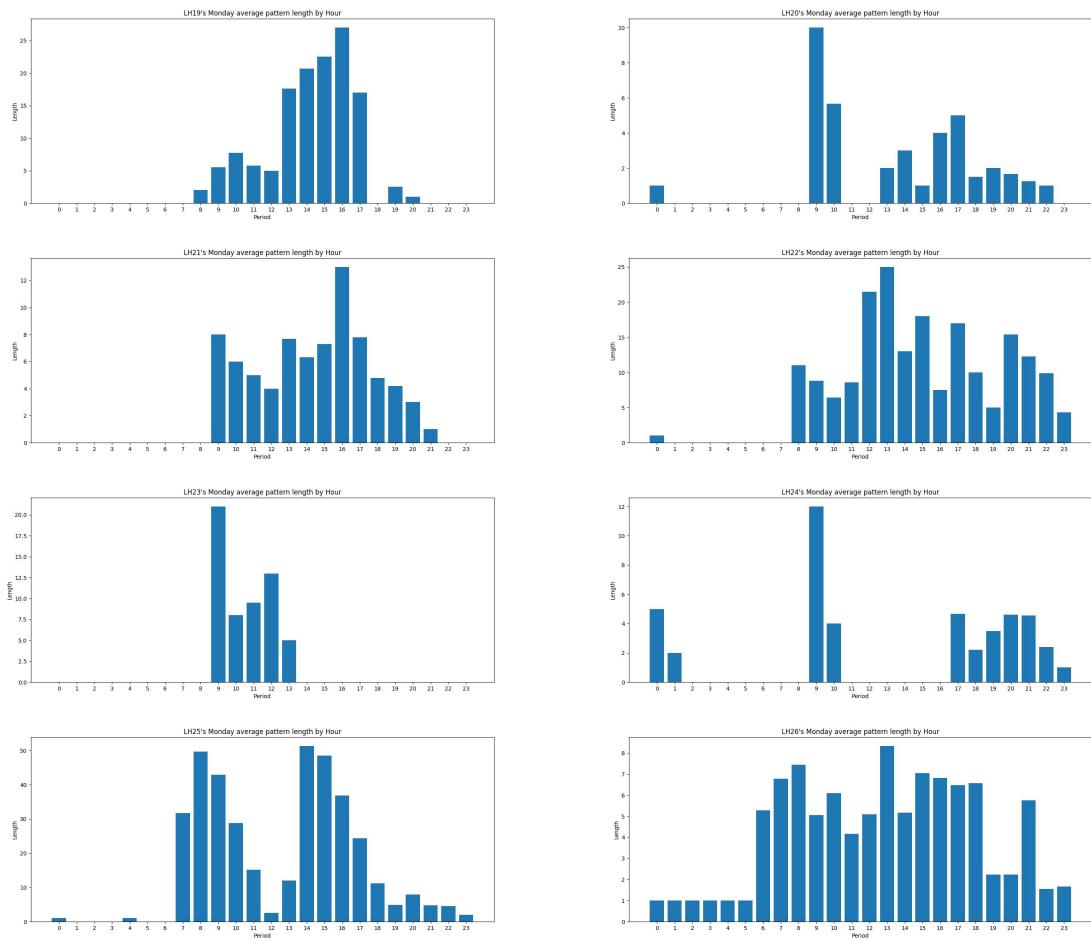


Figura 42: Lunghezza media oraria dei pattern di cronologia di Lunedì per i file di cronologia da LH19 a LH26.

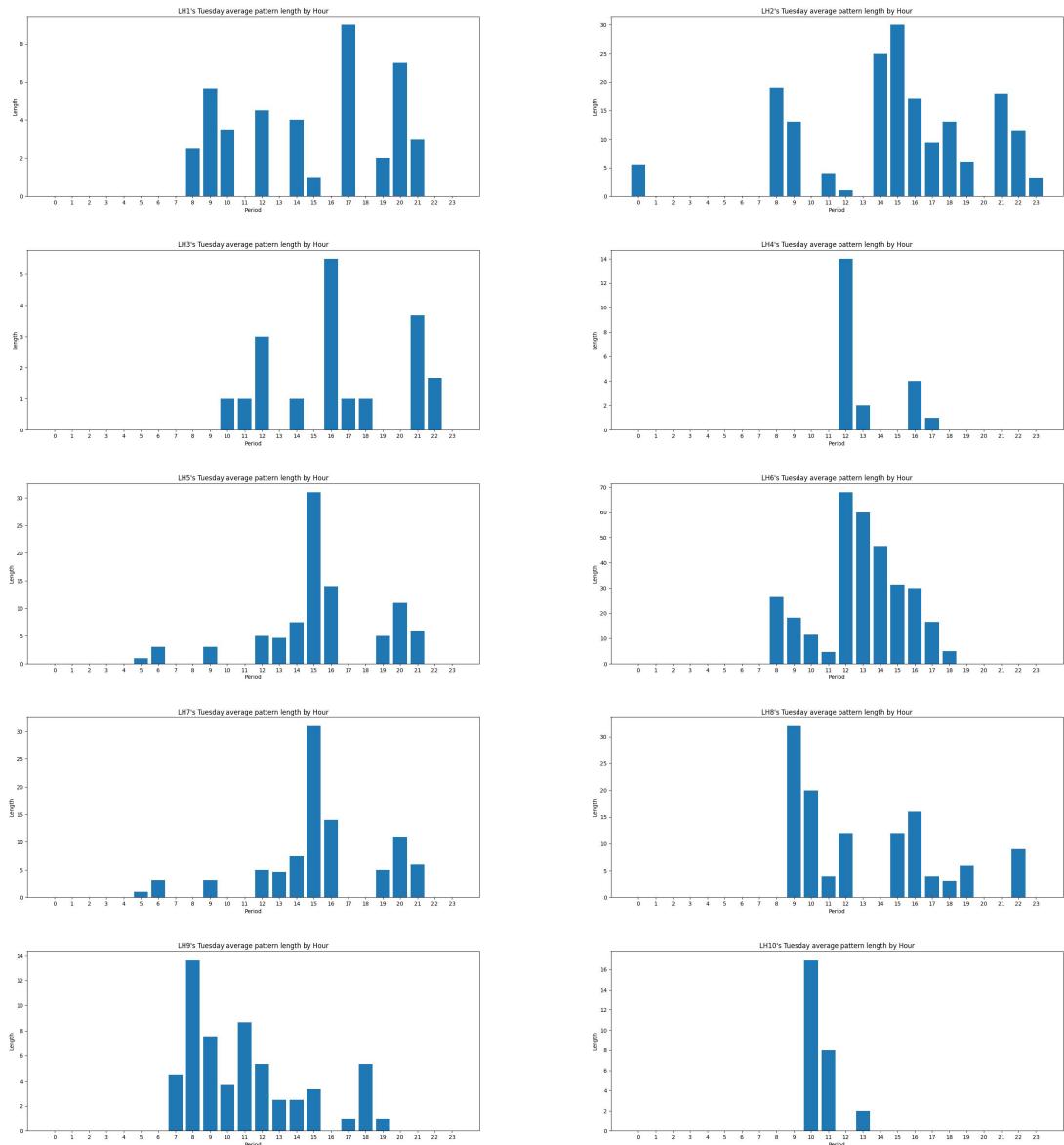


Figura 43: Lunghezza media oraria dei pattern di cronologia di Martedì per i file di cronologia da LH1 a LH10.

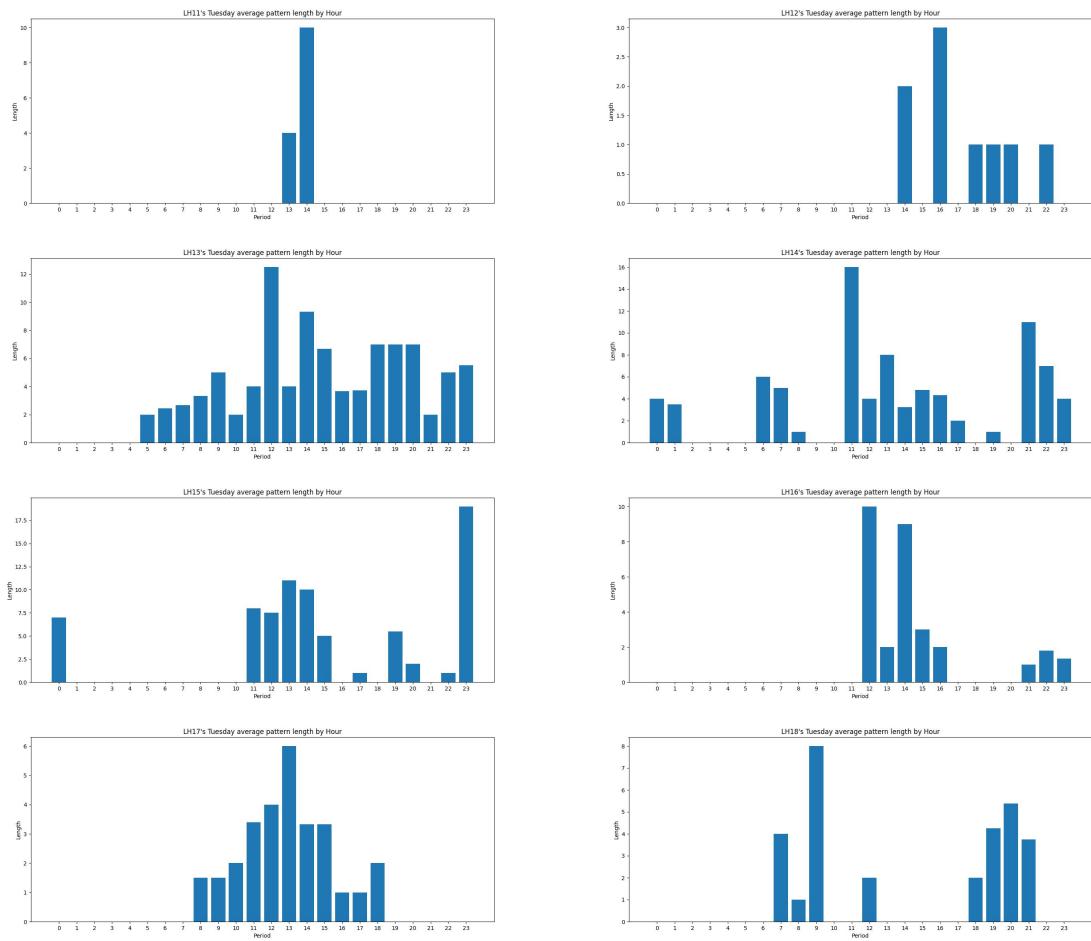


Figura 44: Lunghezza media oraria dei pattern di cronologia di Martedì per i file di cronologia da LH11 a LH18.

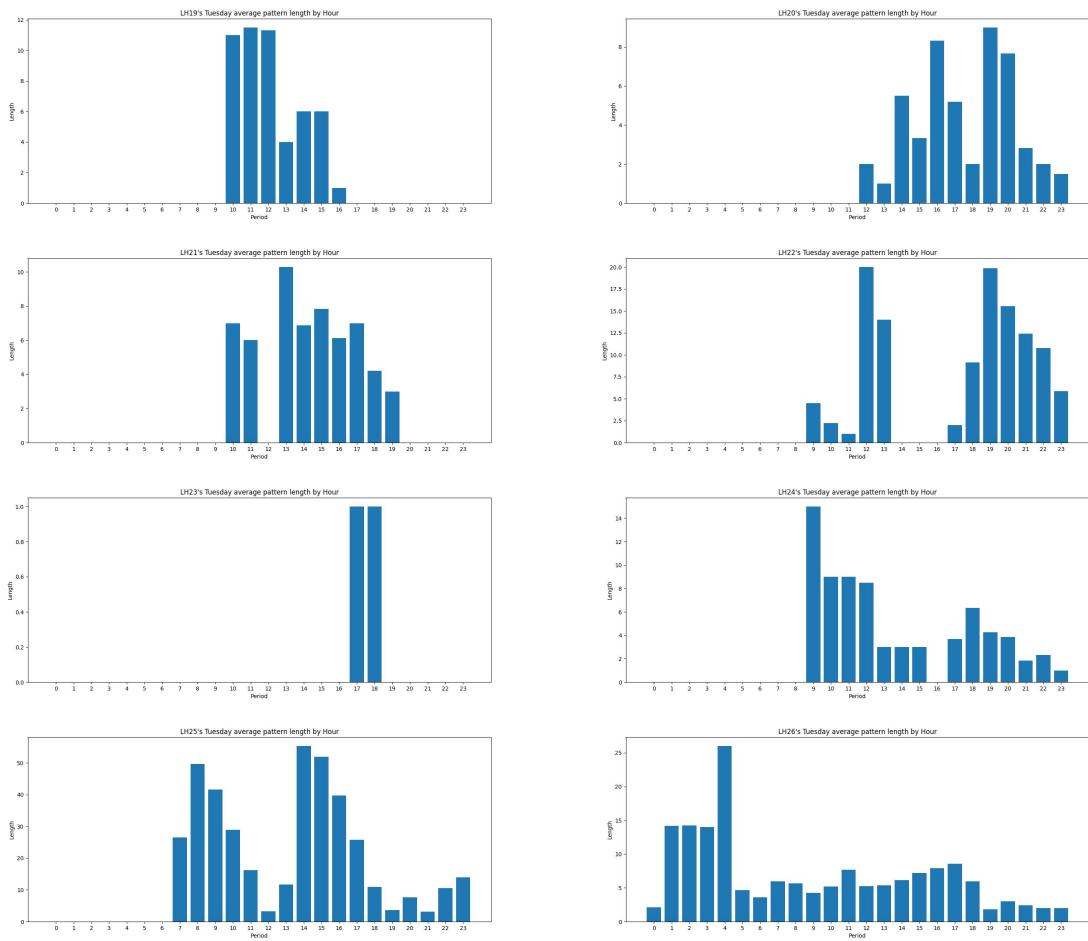


Figura 45: Lunghezza media oraria dei pattern di cronologia di Martedì per i file di cronologia da LH19 a LH26.

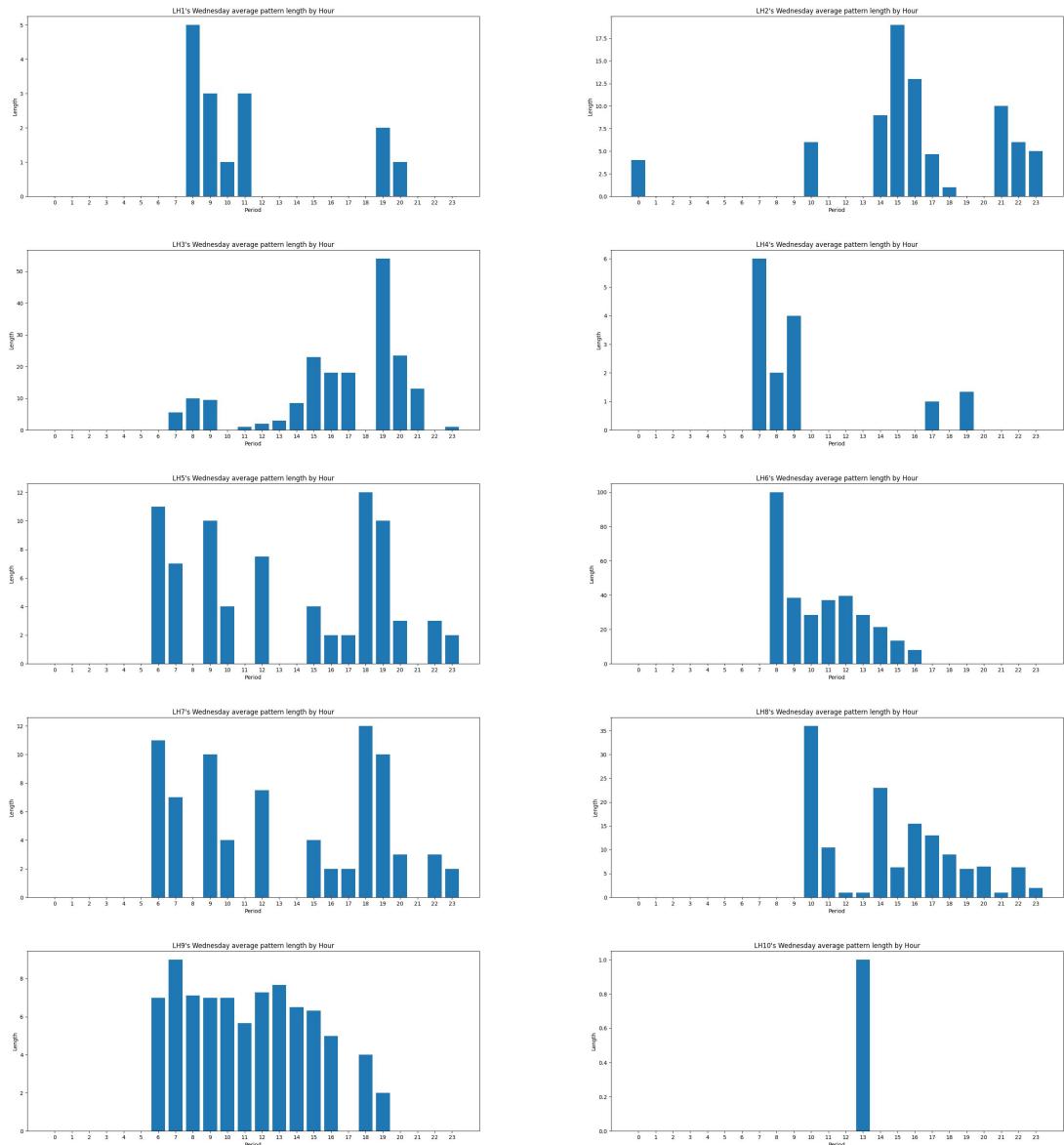


Figura 46: Lunghezza media oraria dei pattern di cronologia di Mercoledì per i file di cronologia da LH1 a LH10.

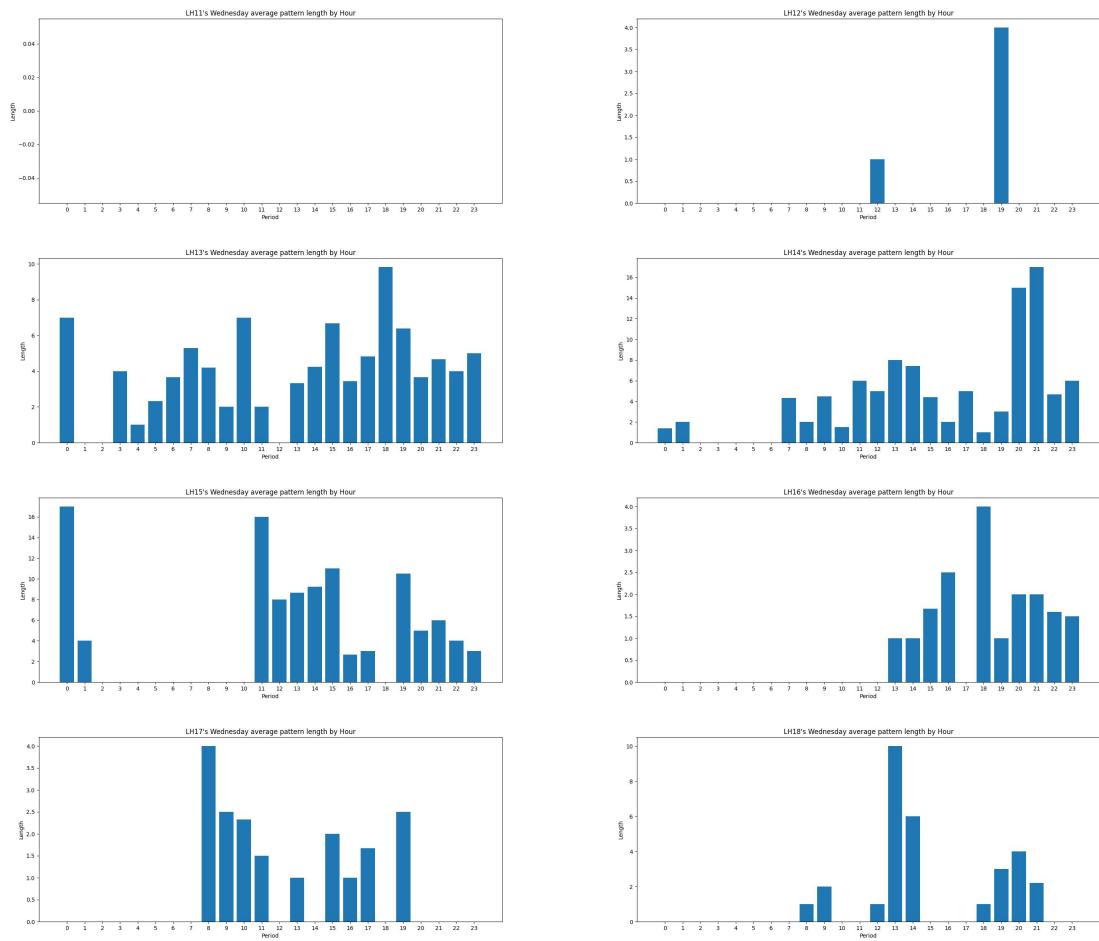


Figura 47: Lunghezza media oraria dei pattern di cronologia di Mercoledì per i file di cronologia da LH11 a LH18.

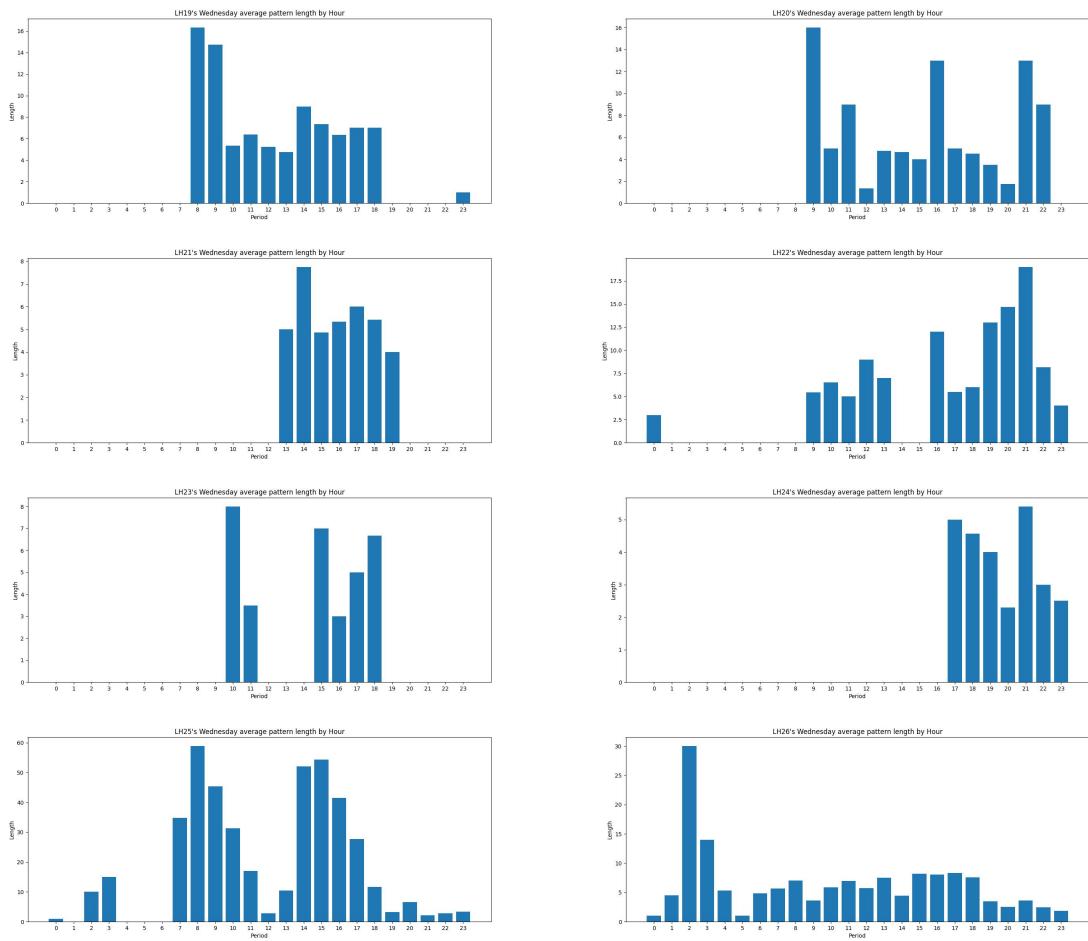


Figura 48: Lunghezza media oraria dei pattern di cronologia di Mercoledì per i file di cronologia da LH19 a LH26.

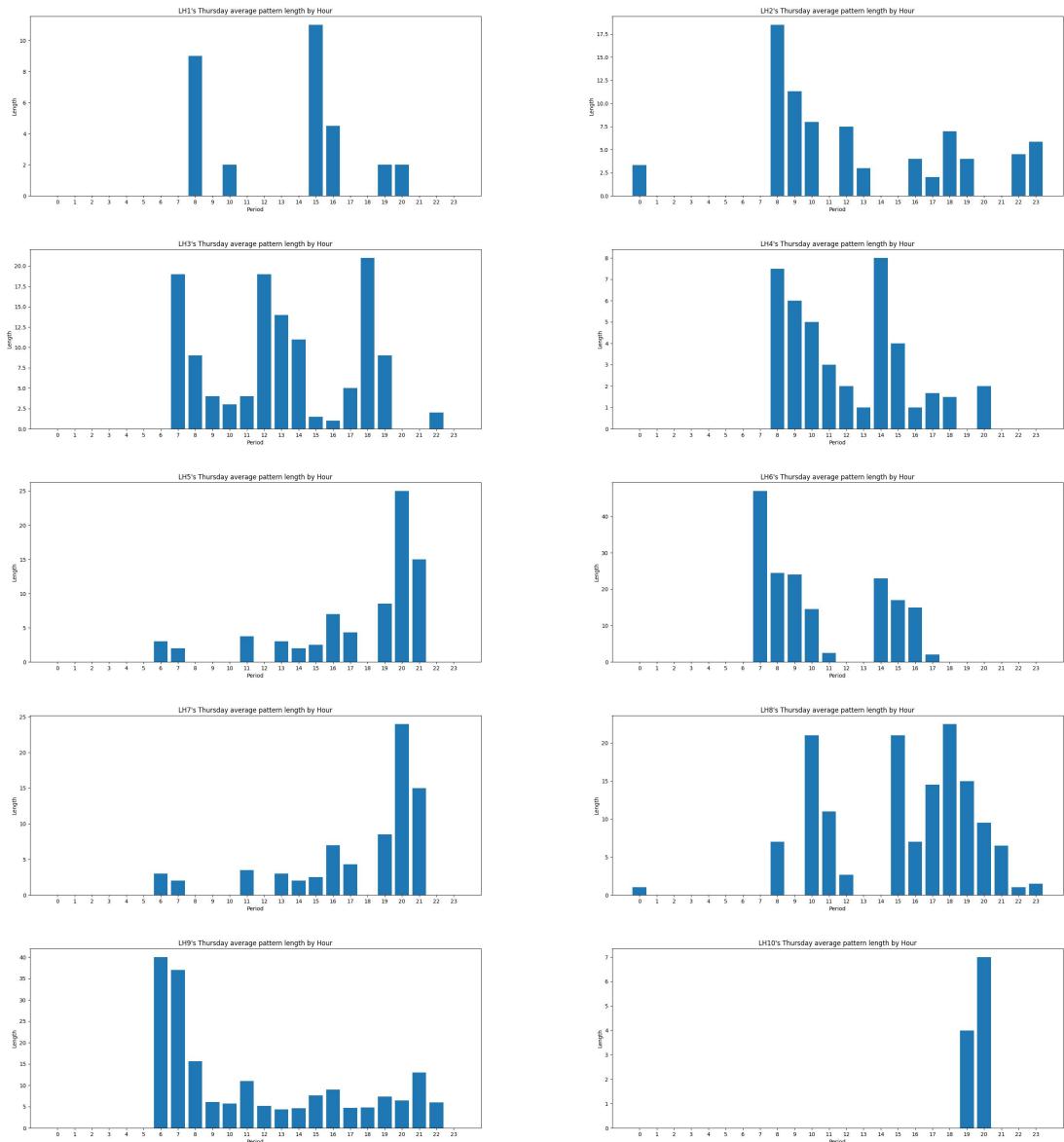


Figura 49: Lunghezza media oraria dei pattern di cronologia di Giovedì per i file di cronologia da LH1 a LH10.

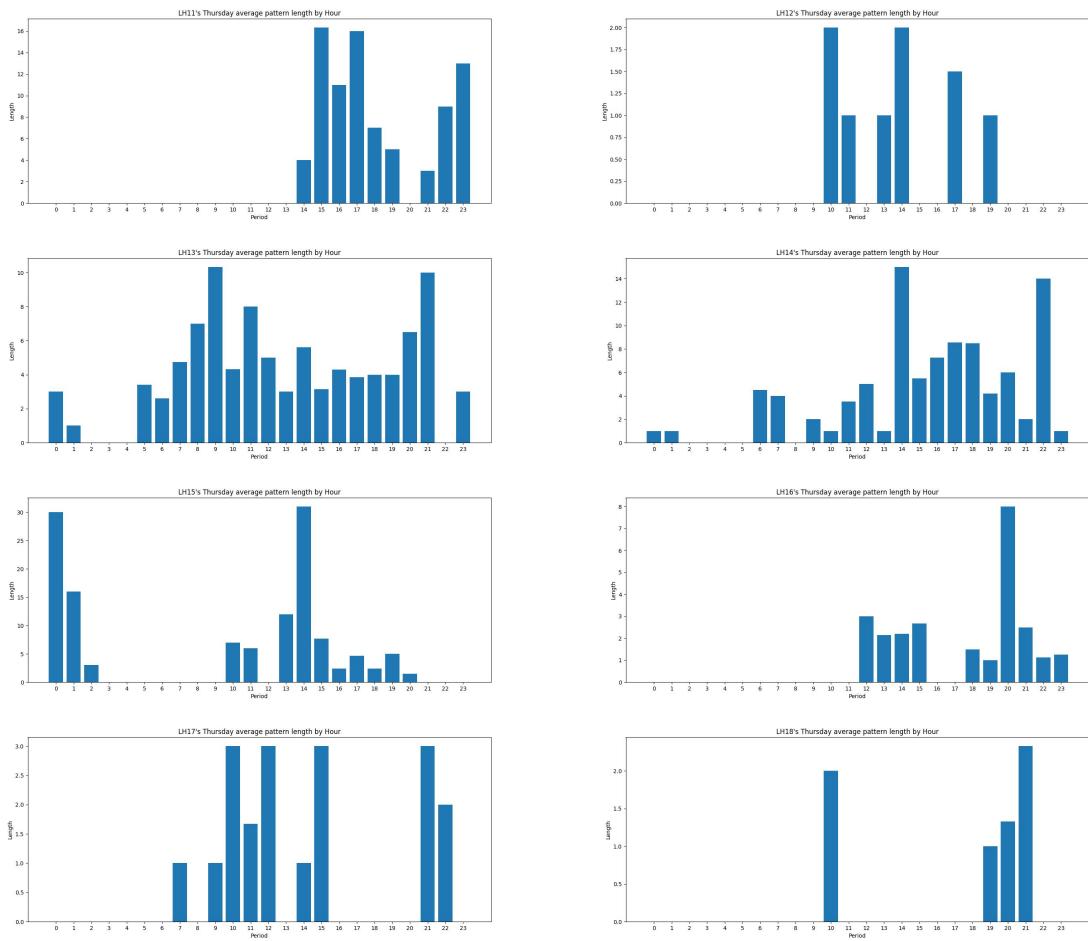


Figura 50: Lunghezza media oraria dei pattern di cronologia di Giovedì per i file di cronologia da LH11 a LH18.

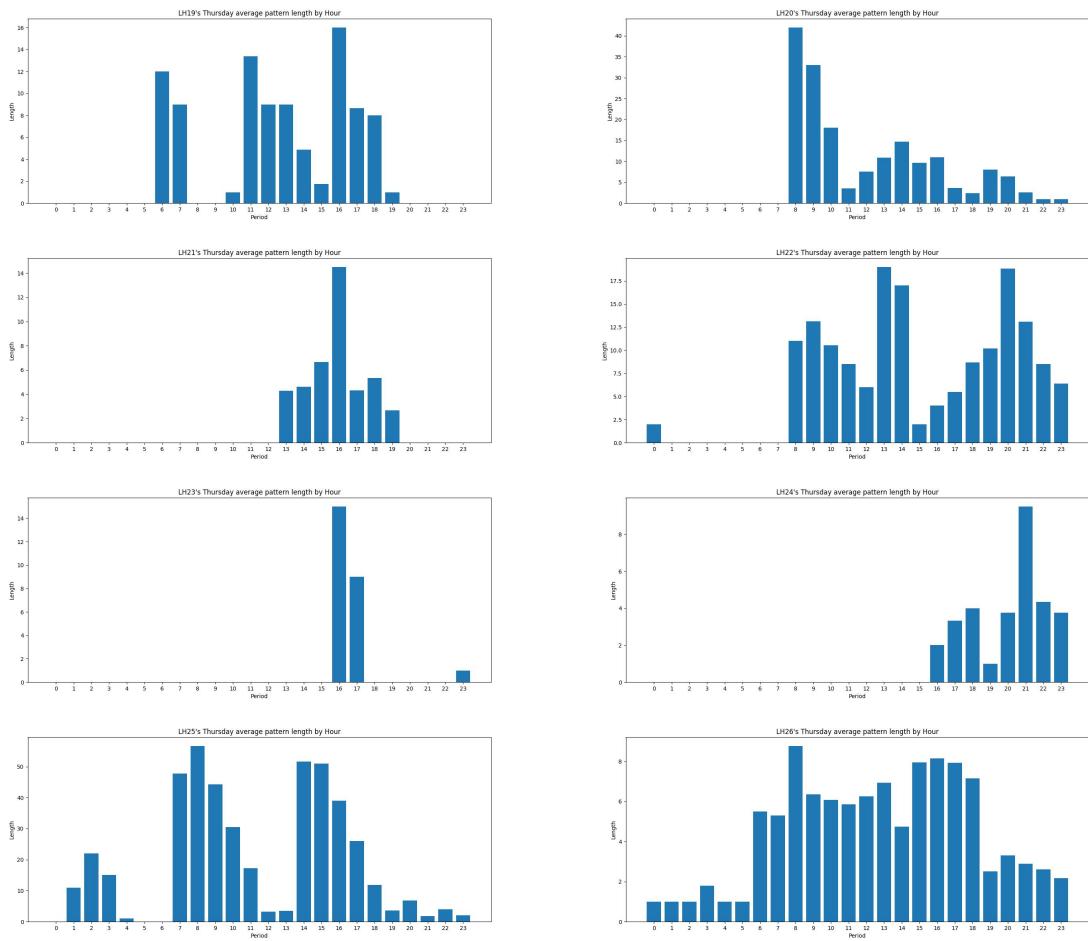


Figura 51: Lunghezza media oraria dei pattern di cronologia di Giovedì per i file di cronologia da LH19 a LH26.



Figura 52: Lunghezza media oraria dei pattern di cronologia di Venerdì per i file di cronologia da LH1 a LH10.

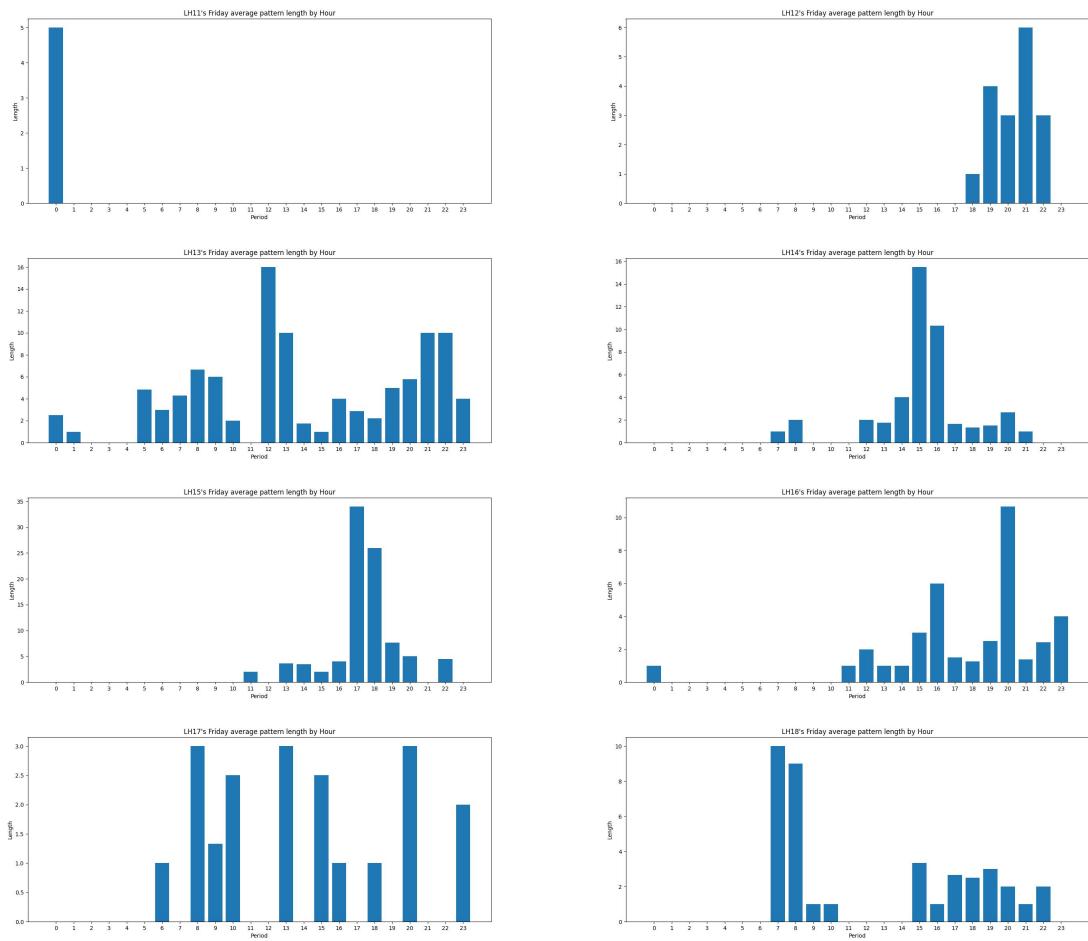


Figura 53: Lunghezza media oraria dei pattern di cronologia di Venerdì per i file di cronologia da LH11 a LH18.

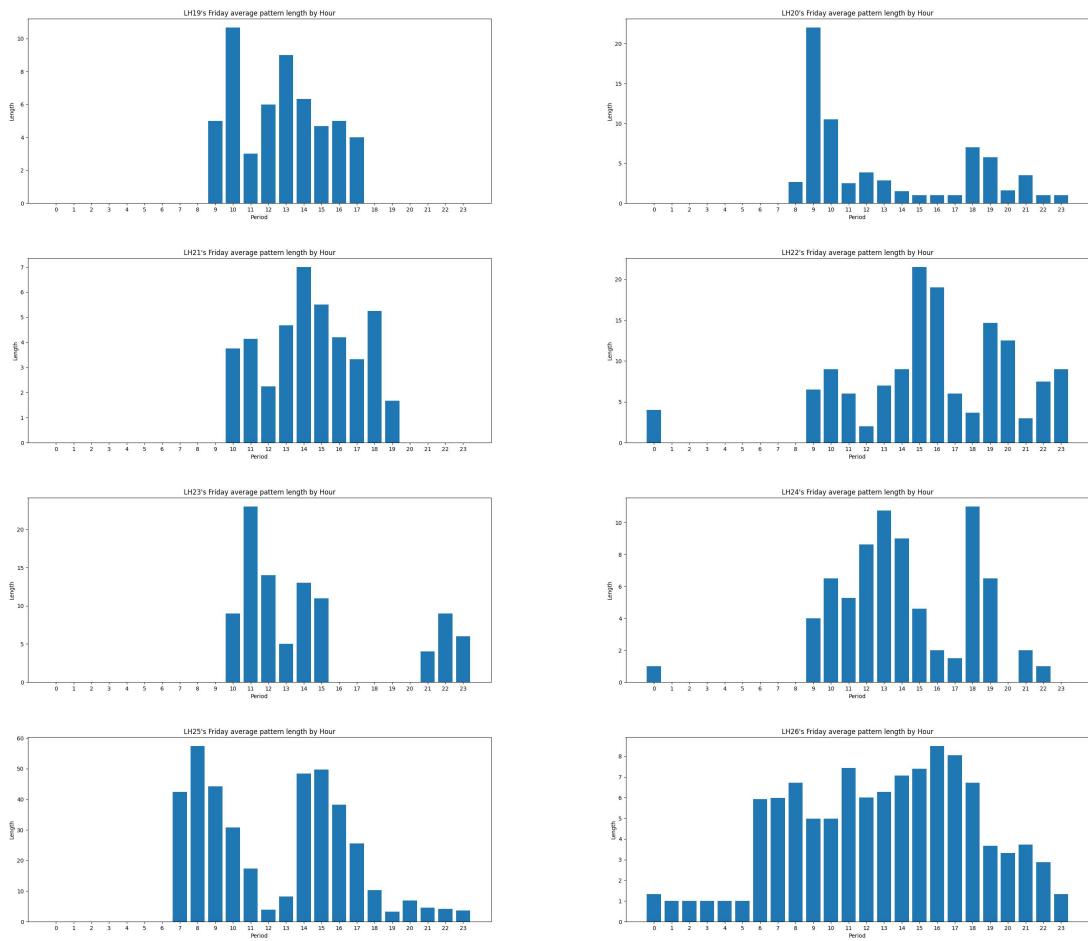


Figura 54: Lunghezza media oraria dei pattern di cronologia di Venerdì per i file di cronologia da LH19 a LH26.

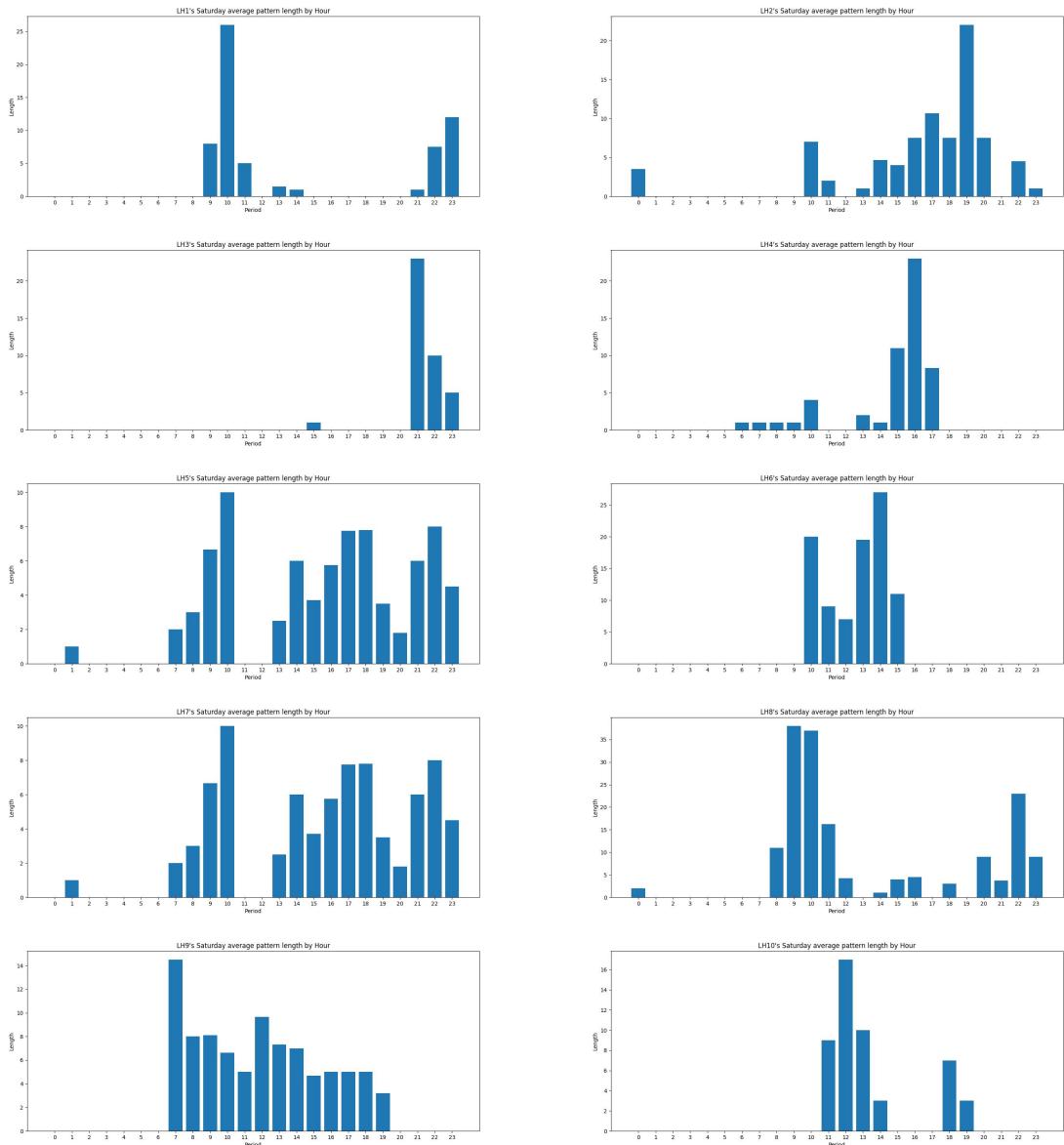


Figura 55: Lunghezza media oraria dei pattern di cronologia di Sabato per i file di cronologia da LH1 a LH10.

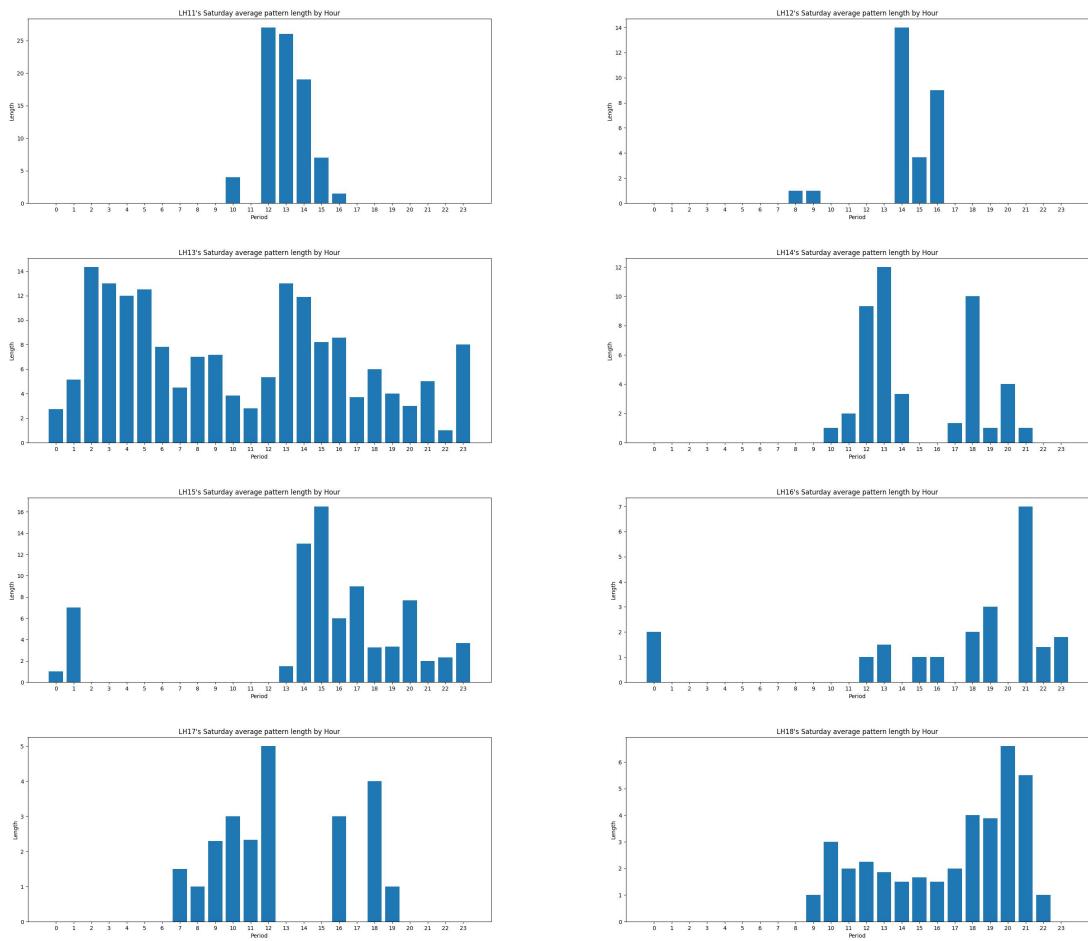


Figura 56: Lunghezza media oraria dei pattern di cronologia di Sabato per i file di cronologia da LH11 a LH18.

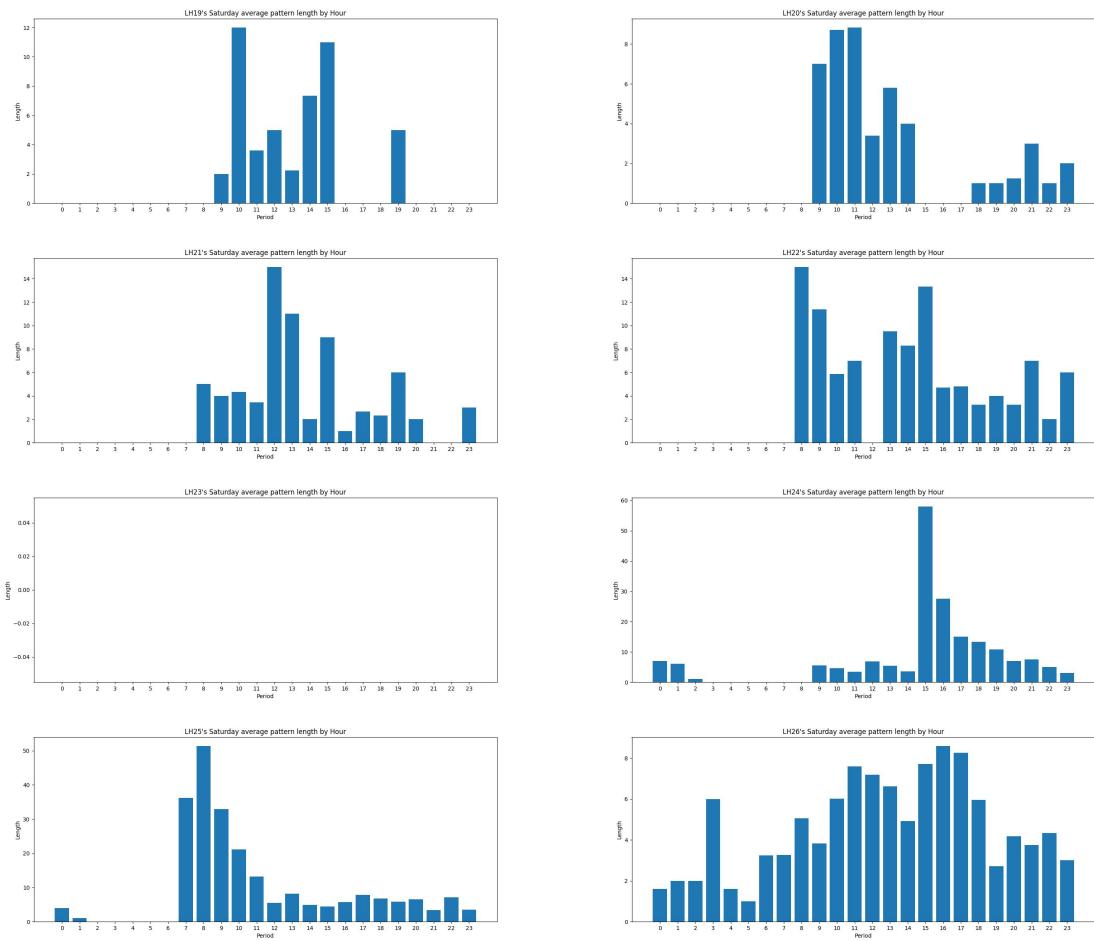


Figura 57: Lunghezza media oraria dei pattern di cronologia di Sabato per i file di cronologia da LH19 a LH26.

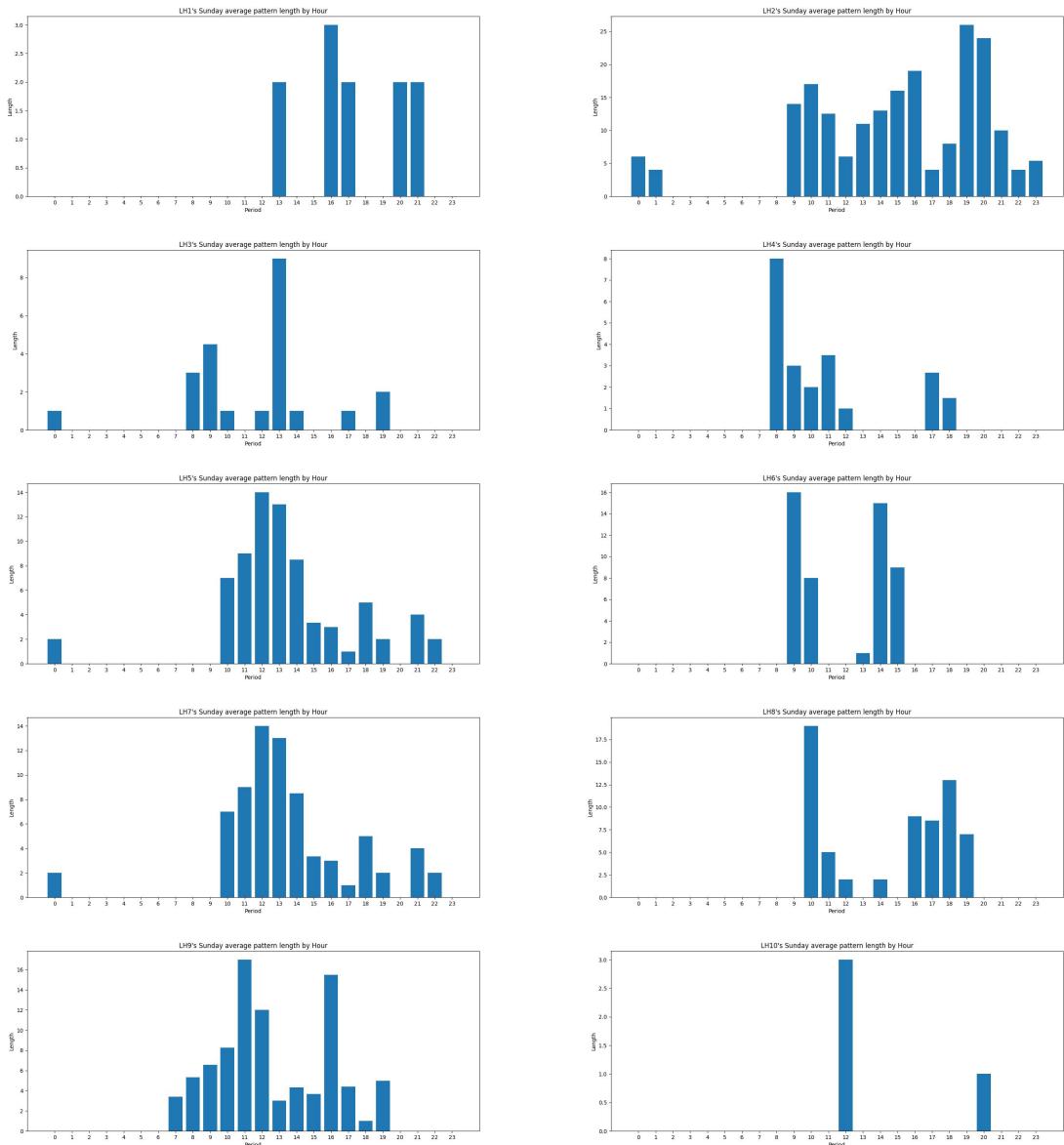


Figura 58: Lunghezza media oraria dei pattern di cronologia di Domenica per i file di cronologia da LH1 a LH10.

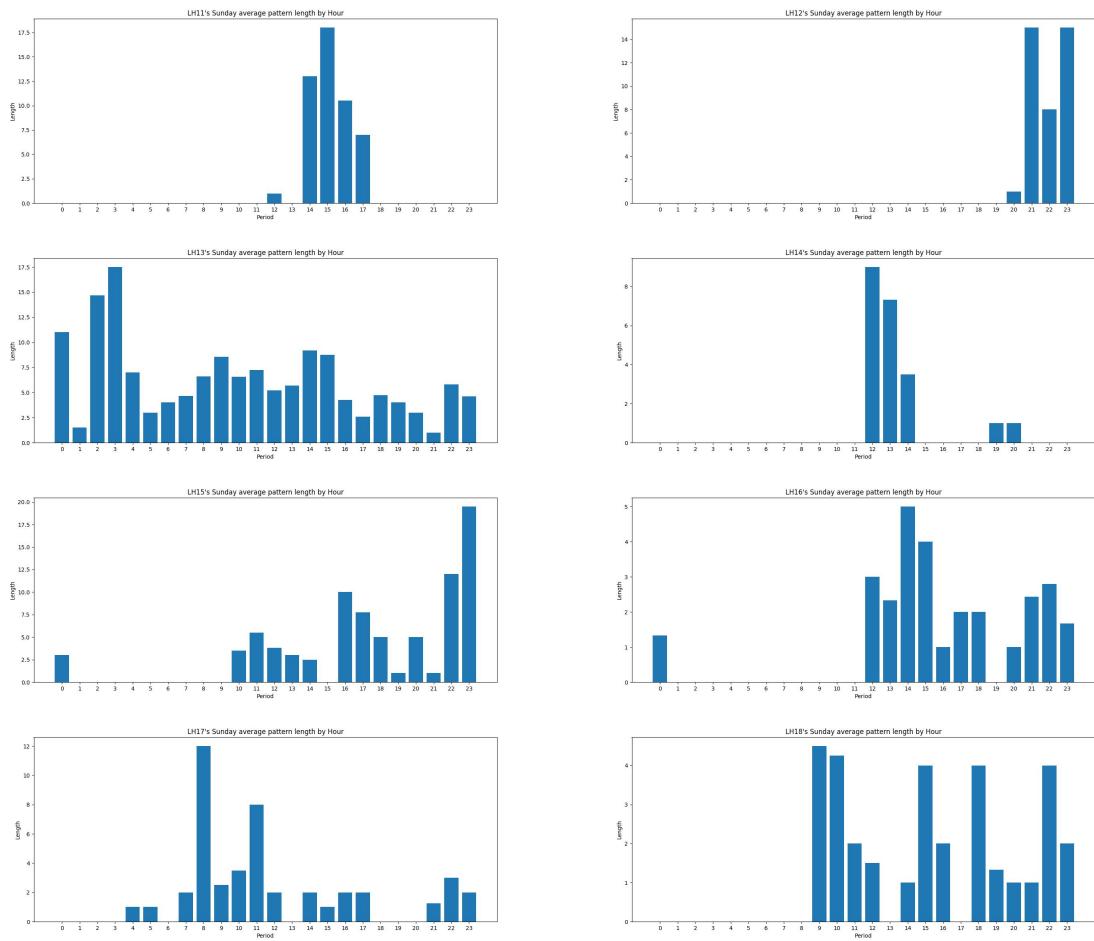


Figura 59: Lunghezza media oraria dei pattern di cronologia di Domenica per i file di cronologia da LH11 a LH18.

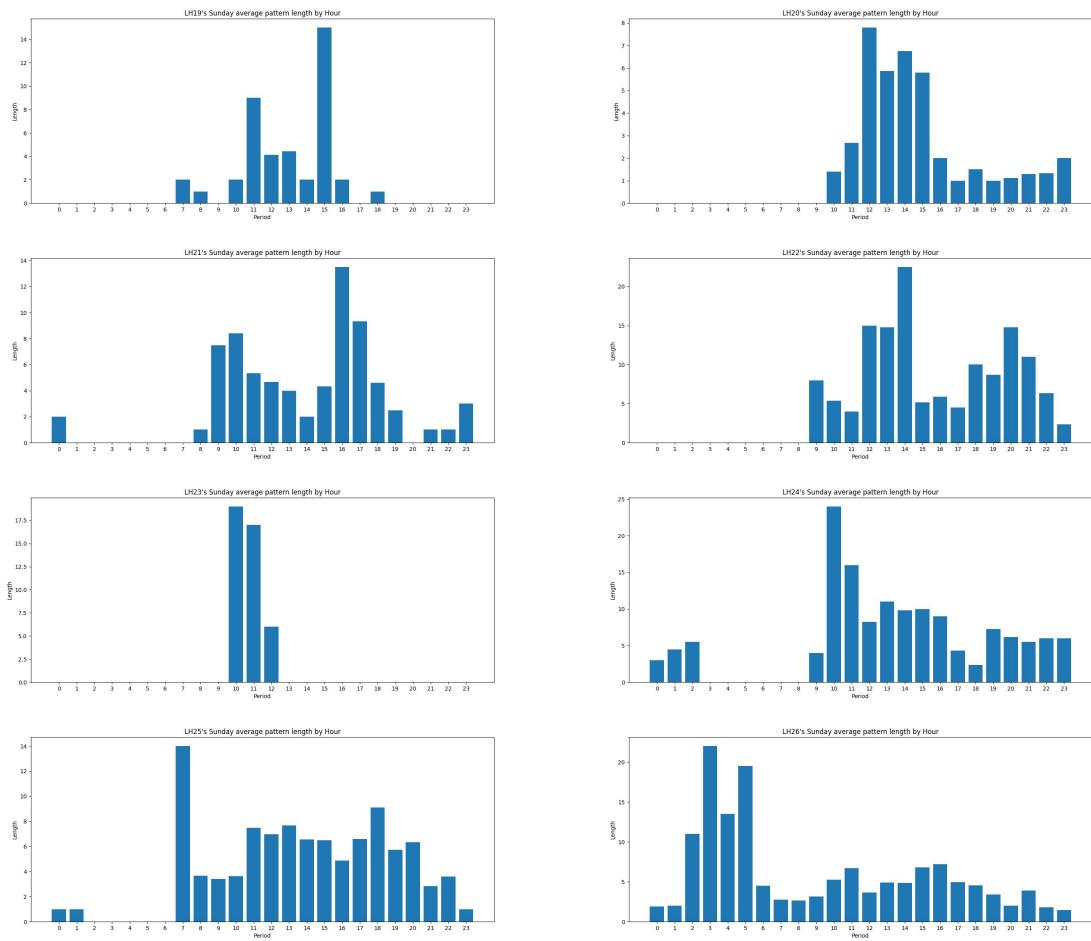


Figura 60: Lunghezza media oraria dei pattern di cronologia di Domenica per i file di cronologia da LH19 a LH26.

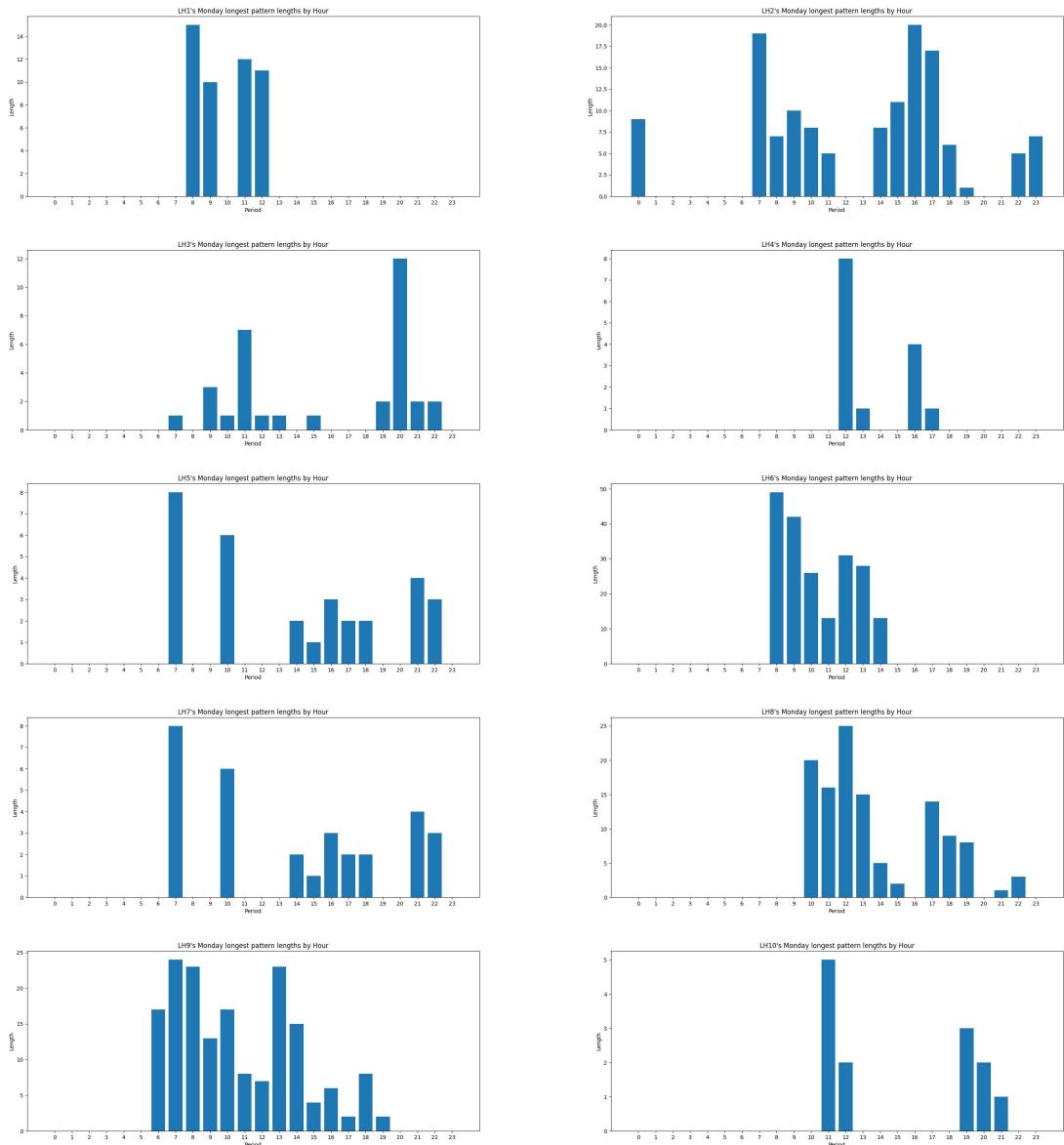


Figura 61: Lunghezza massima oraria dei pattern di cronologia di Lunedì per i file di cronologia da LH1 a LH10.

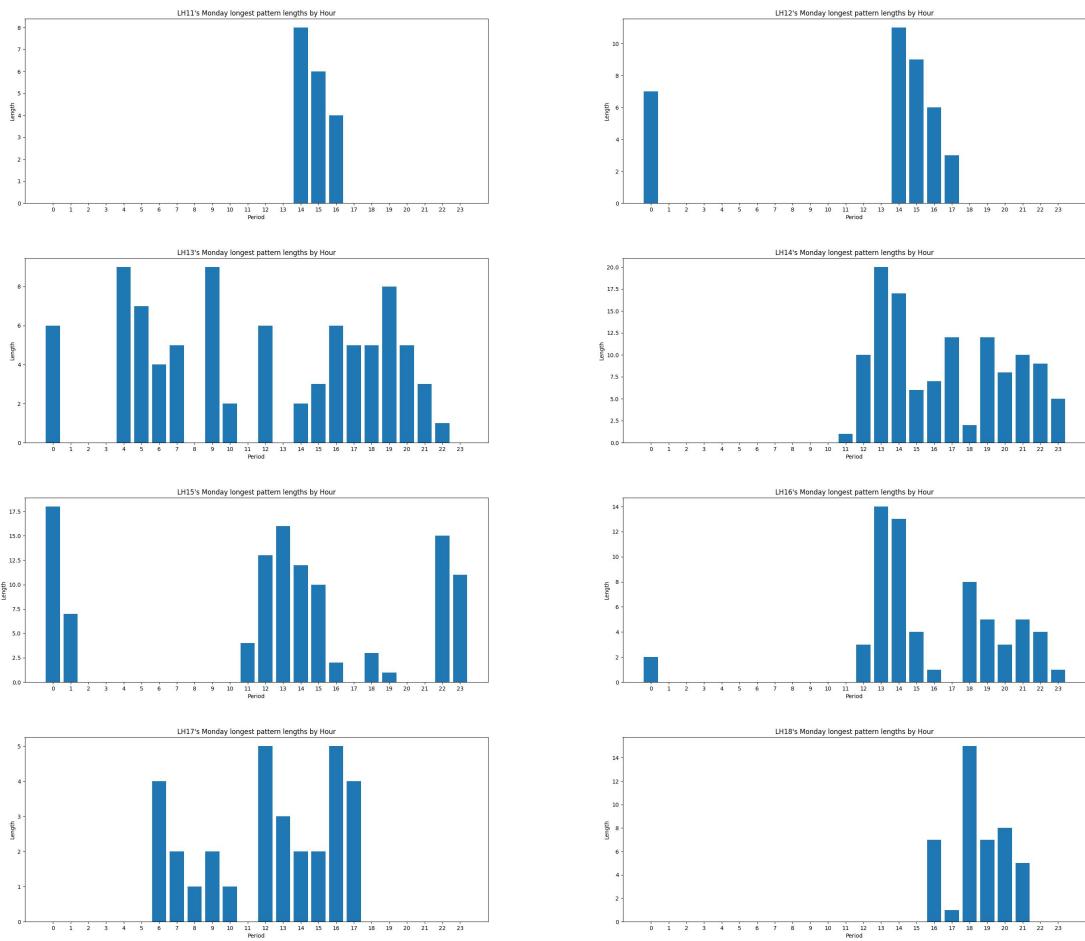


Figura 62: Lunghezza massima oraria dei pattern di cronologia di Lunedì per i file di cronologia da LH11 a LH18.

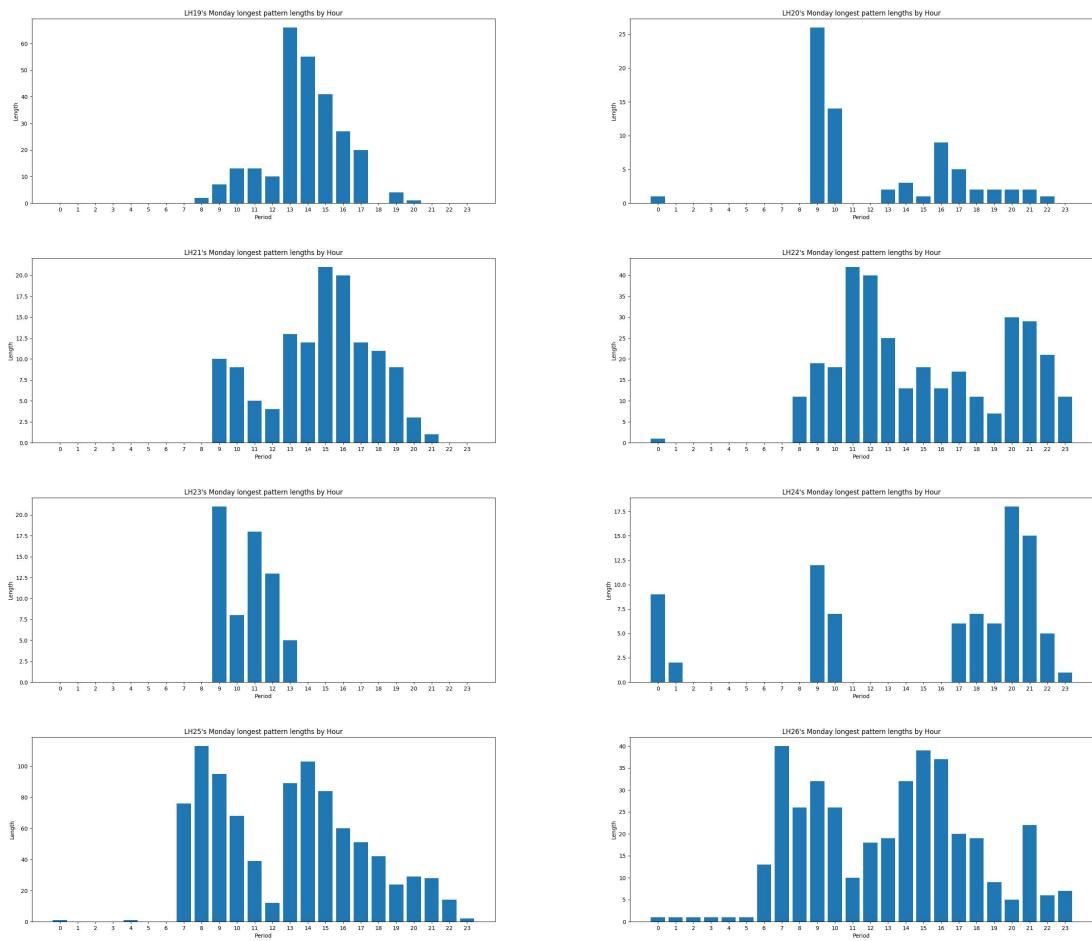


Figura 63: Lunghezza massima oraria dei pattern di cronologia di Lunedì per i file di cronologia da LH19 a LH26.

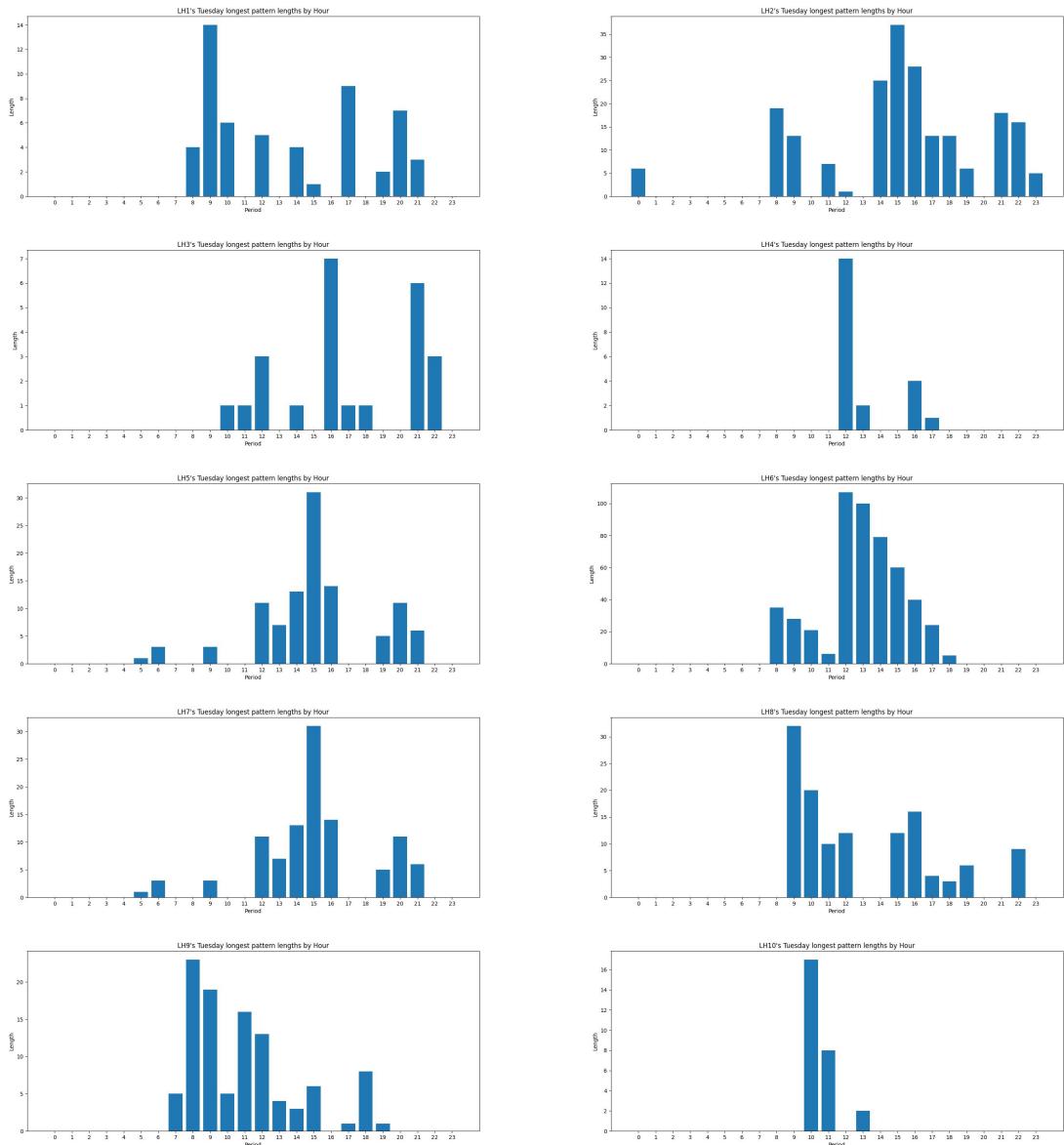


Figura 64: Lunghezza massima oraria dei pattern di cronologia di Martedì per i file di cronologia da LH1 a LH10.

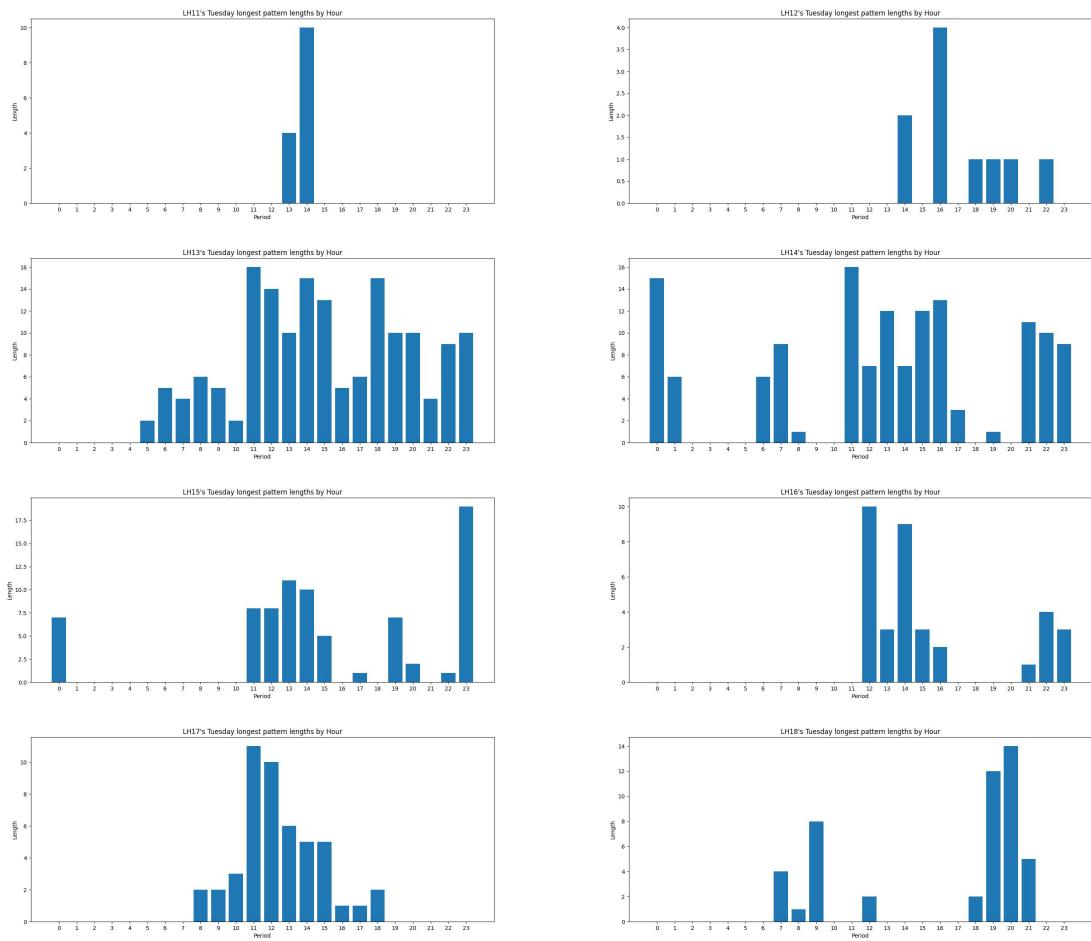


Figura 65: Lunghezza massima oraria dei pattern di cronologia di Martedì per i file di cronologia da LH11 a LH18.

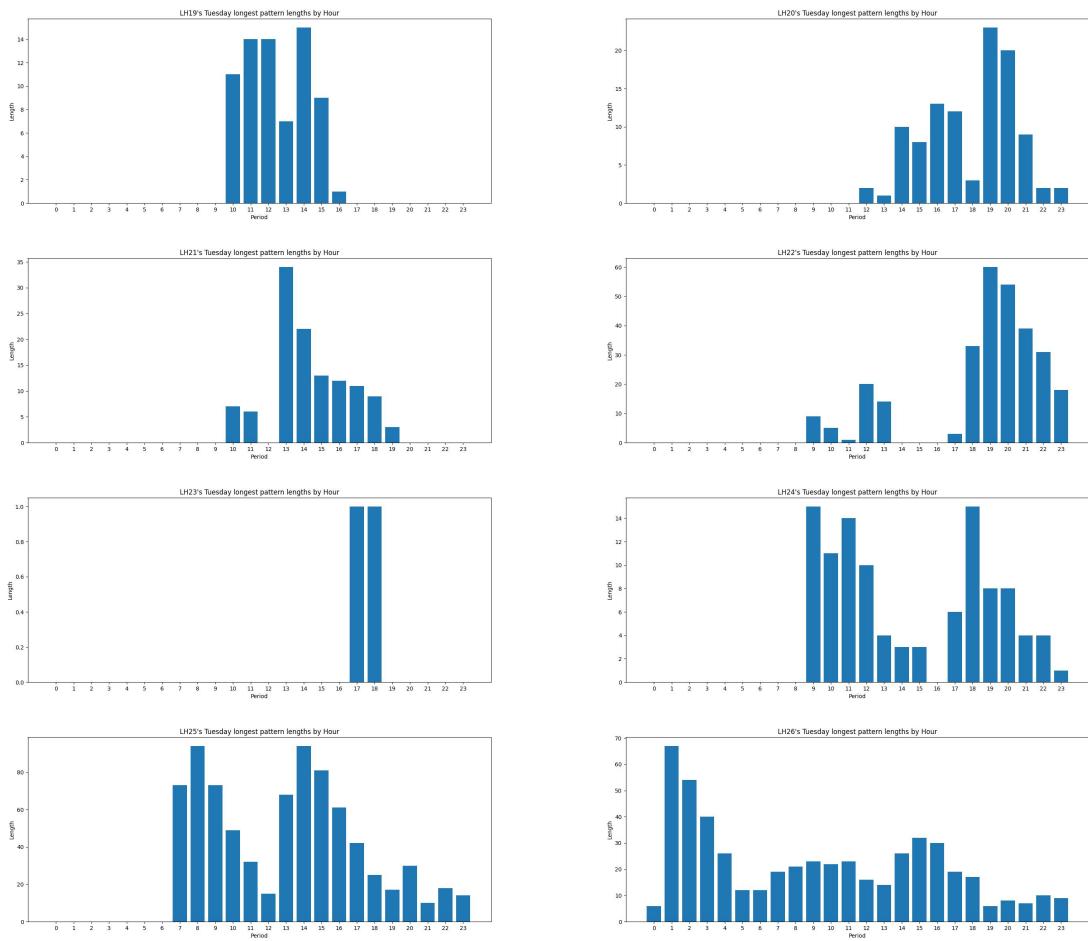


Figura 66: Lunghezza massima oraria dei pattern di cronologia di Martedì per i file di cronologia da LH19 a LH26.

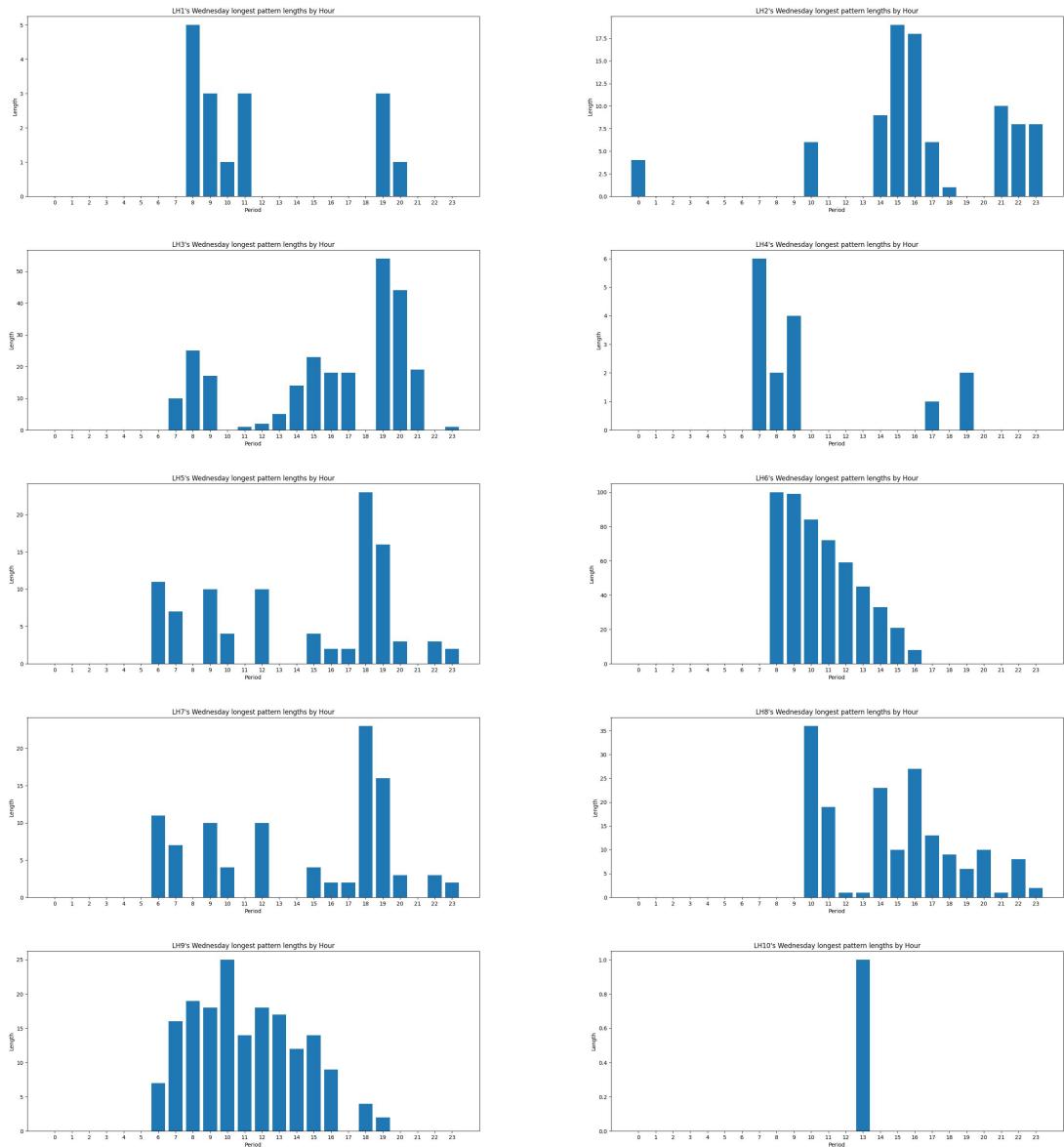


Figura 67: Lunghezza massima oraria dei pattern di cronologia di Mercoledì per i file di cronologia da LH1 a LH10.

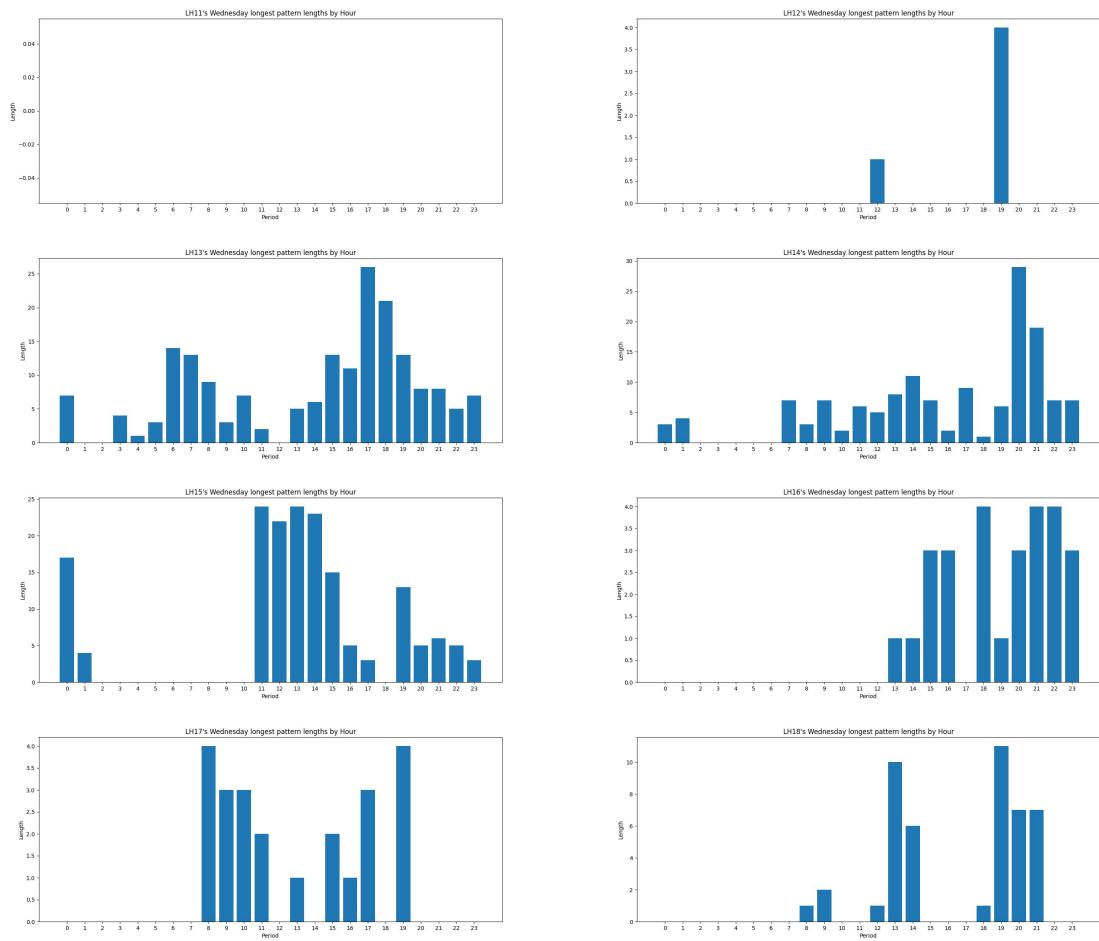


Figura 68: Lunghezza massima oraria dei pattern di cronologia di Mercoledì per i file di cronologia da LH11 a LH18.

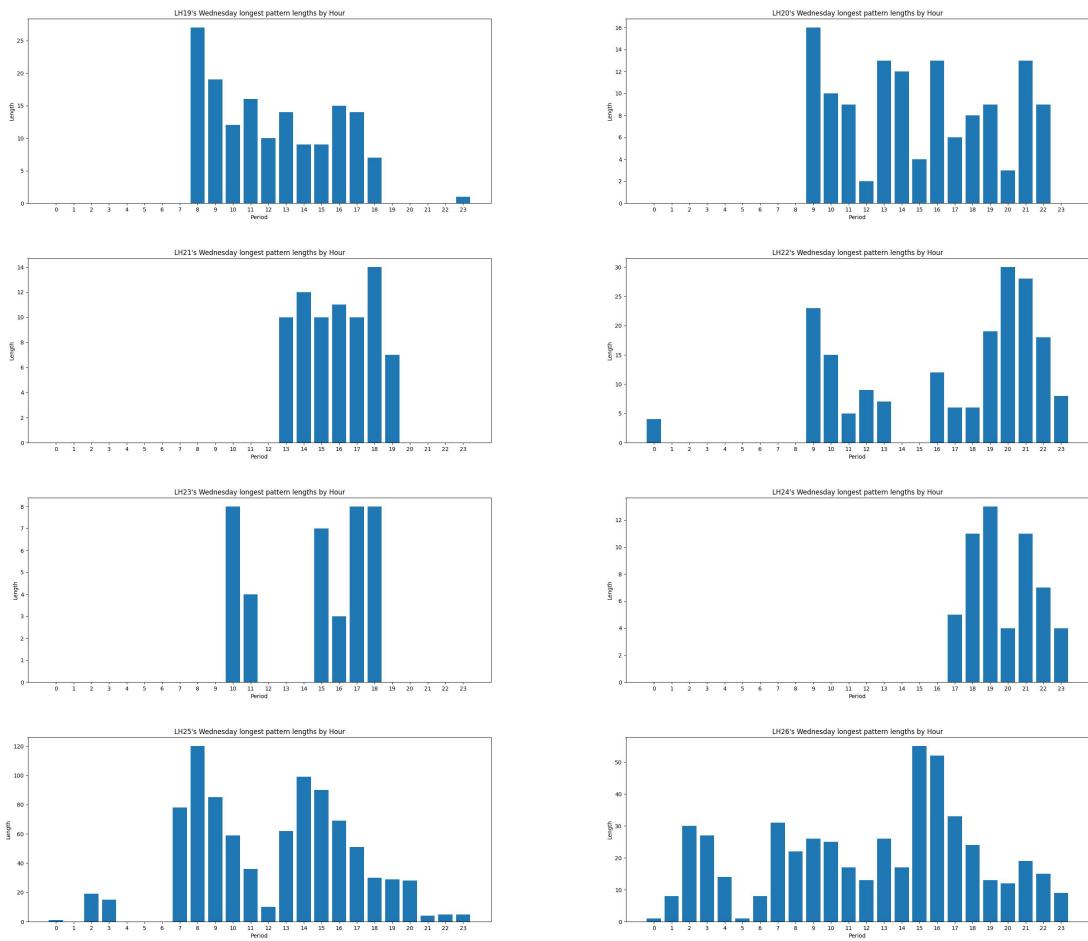


Figura 69: Lunghezza massima oraria dei pattern di cronologia di Mercoledì per i file di cronologia da LH19 a LH26.

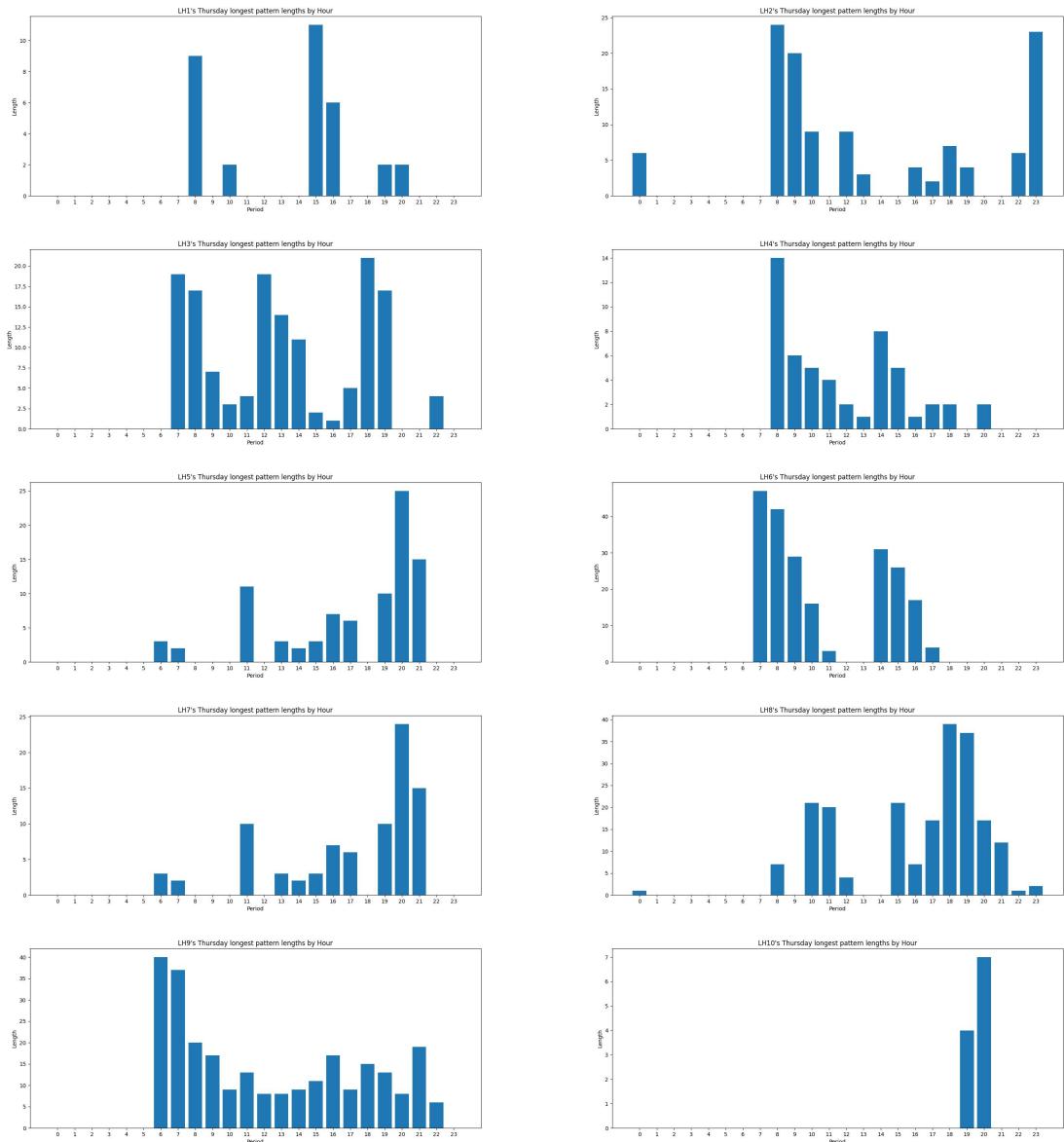


Figura 70: Lunghezza massima oraria dei pattern di cronologia di Giovedì per i file di cronologia da LH1 a LH10.

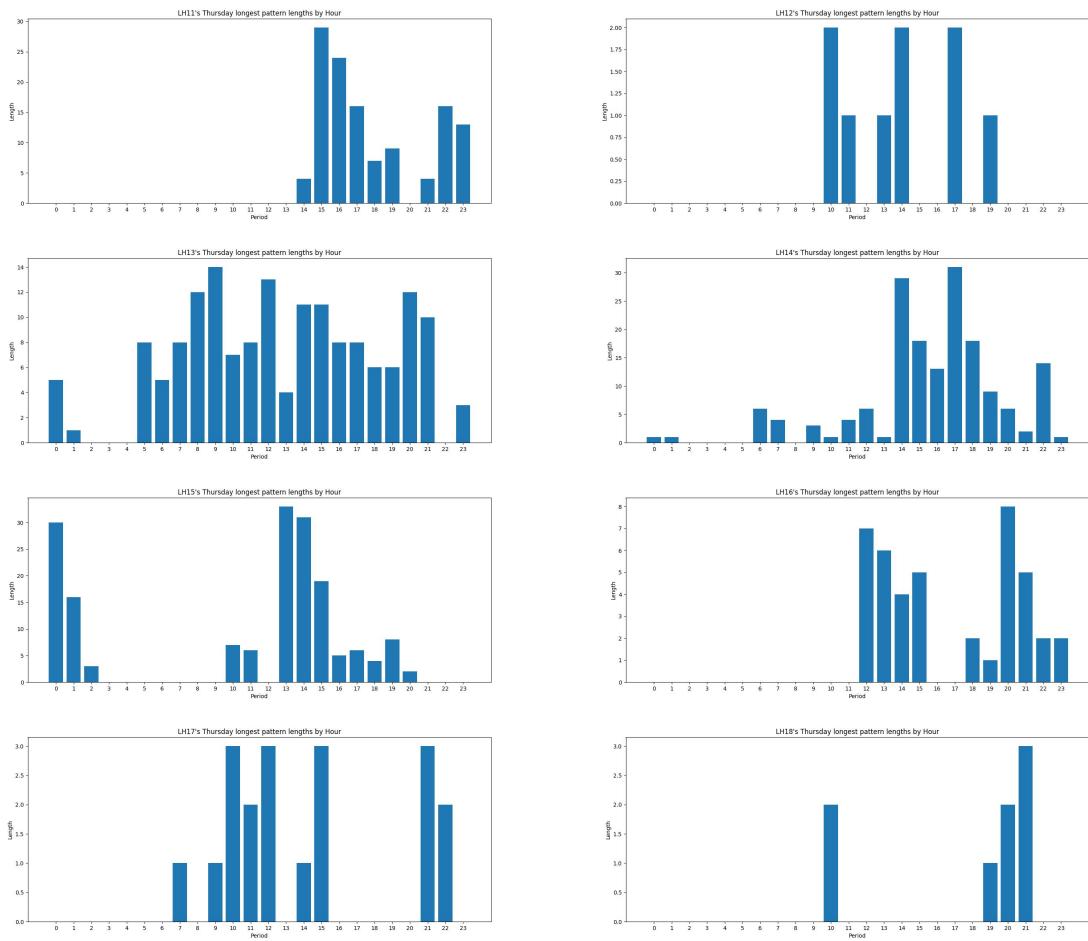


Figura 71: Lunghezza massima oraria dei pattern di cronologia di Giovedì per i file di cronologia da LH11 a LH18.

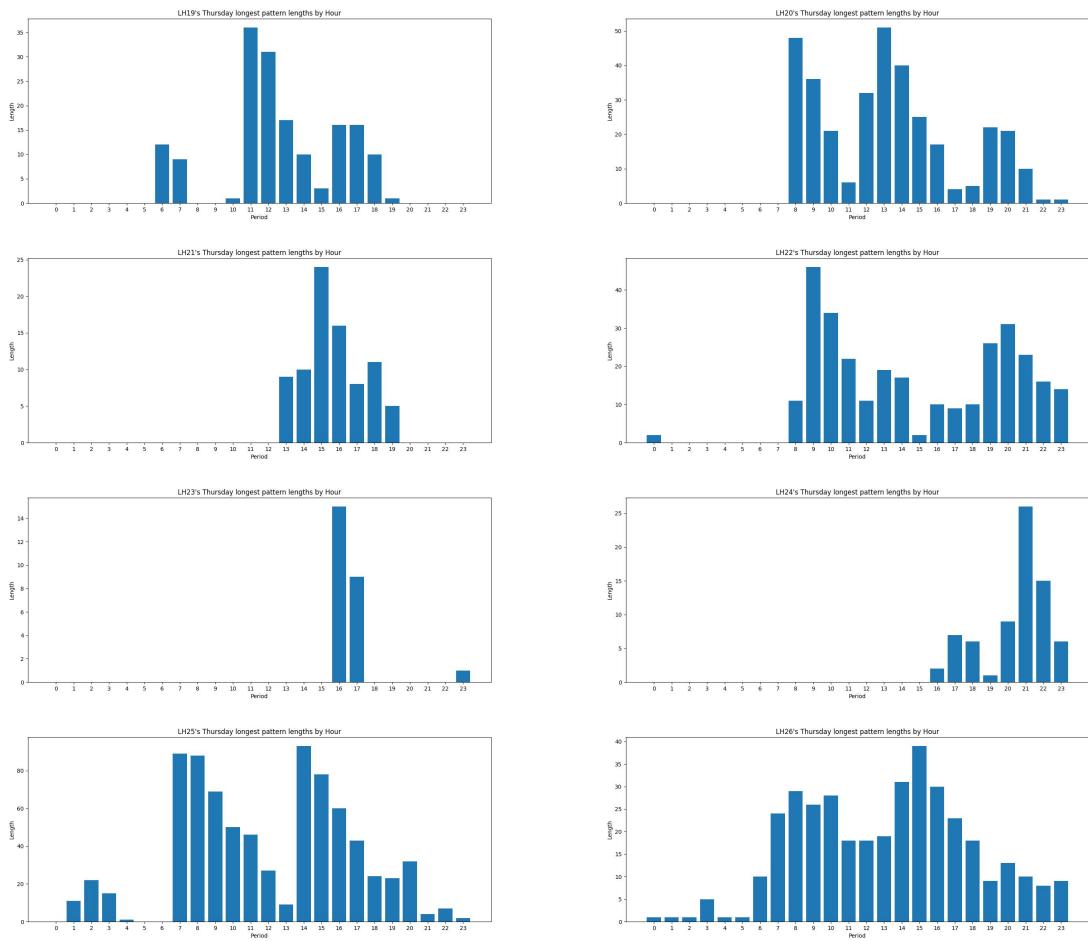


Figura 72: Lunghezza massima oraria dei pattern di cronologia di Giovedì per i file di cronologia da LH19 a LH26.



Figura 73: Lunghezza massima oraria dei pattern di cronologia di Venerdì per i file di cronologia da LH1 a LH10.

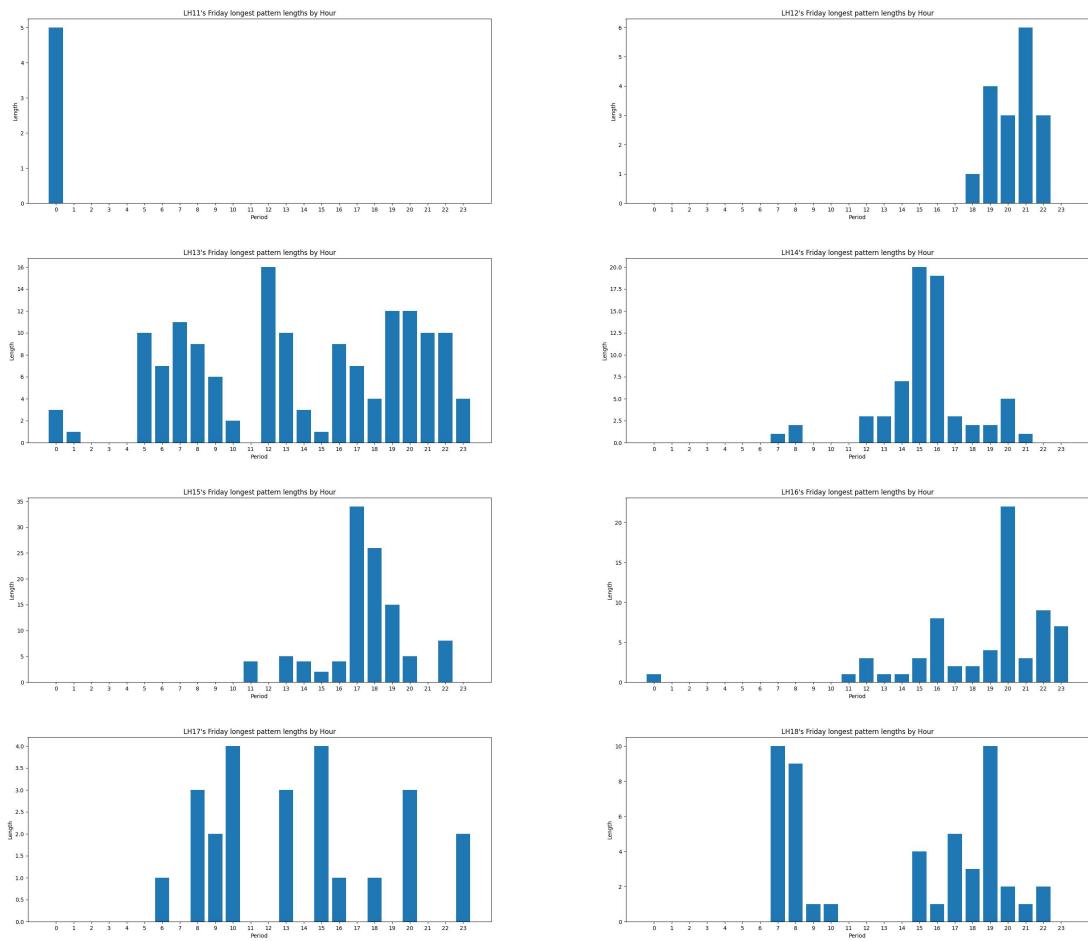


Figura 74: Lunghezza massima oraria dei pattern di cronologia di Venerdì per i file di cronologia da LH11 a LH18.

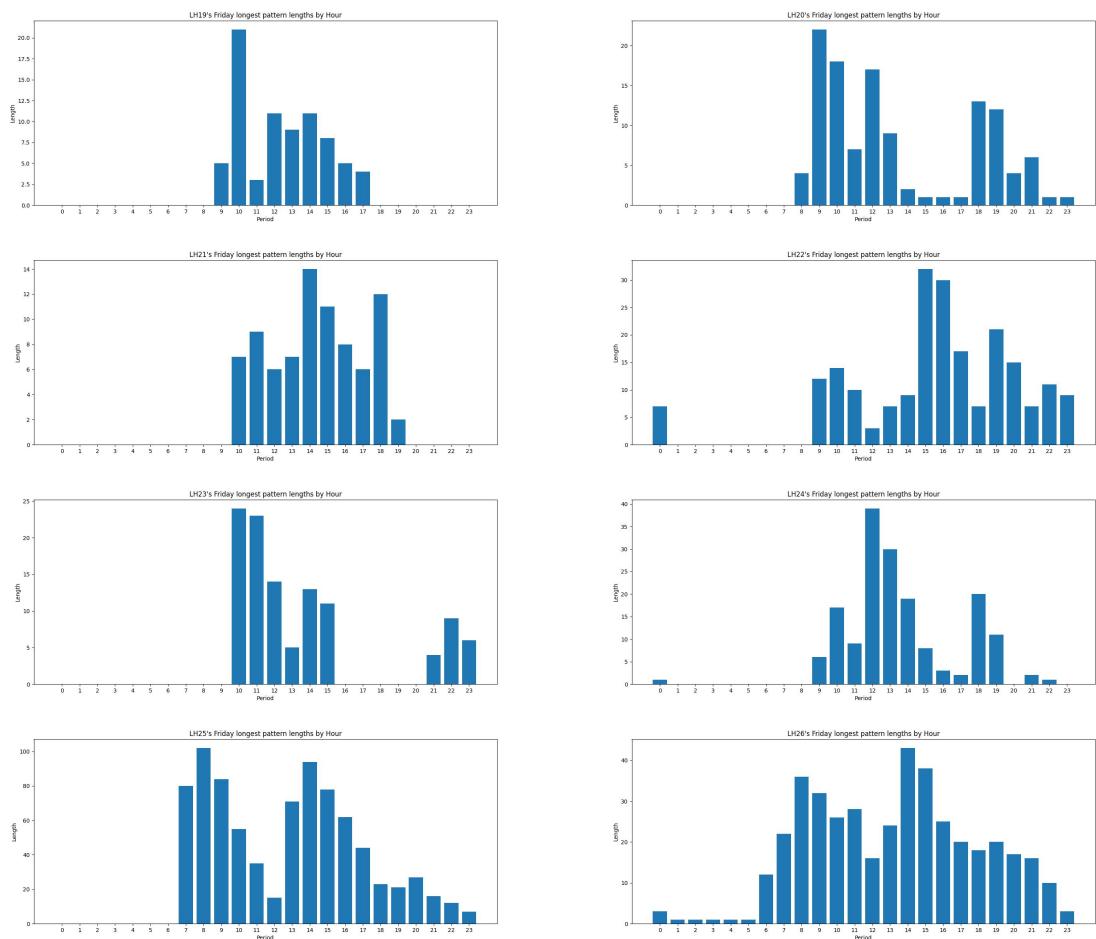


Figura 75: Lunghezza massima oraria dei pattern di cronologia di Venerdì per i file di cronologia da LH19 a LH26.

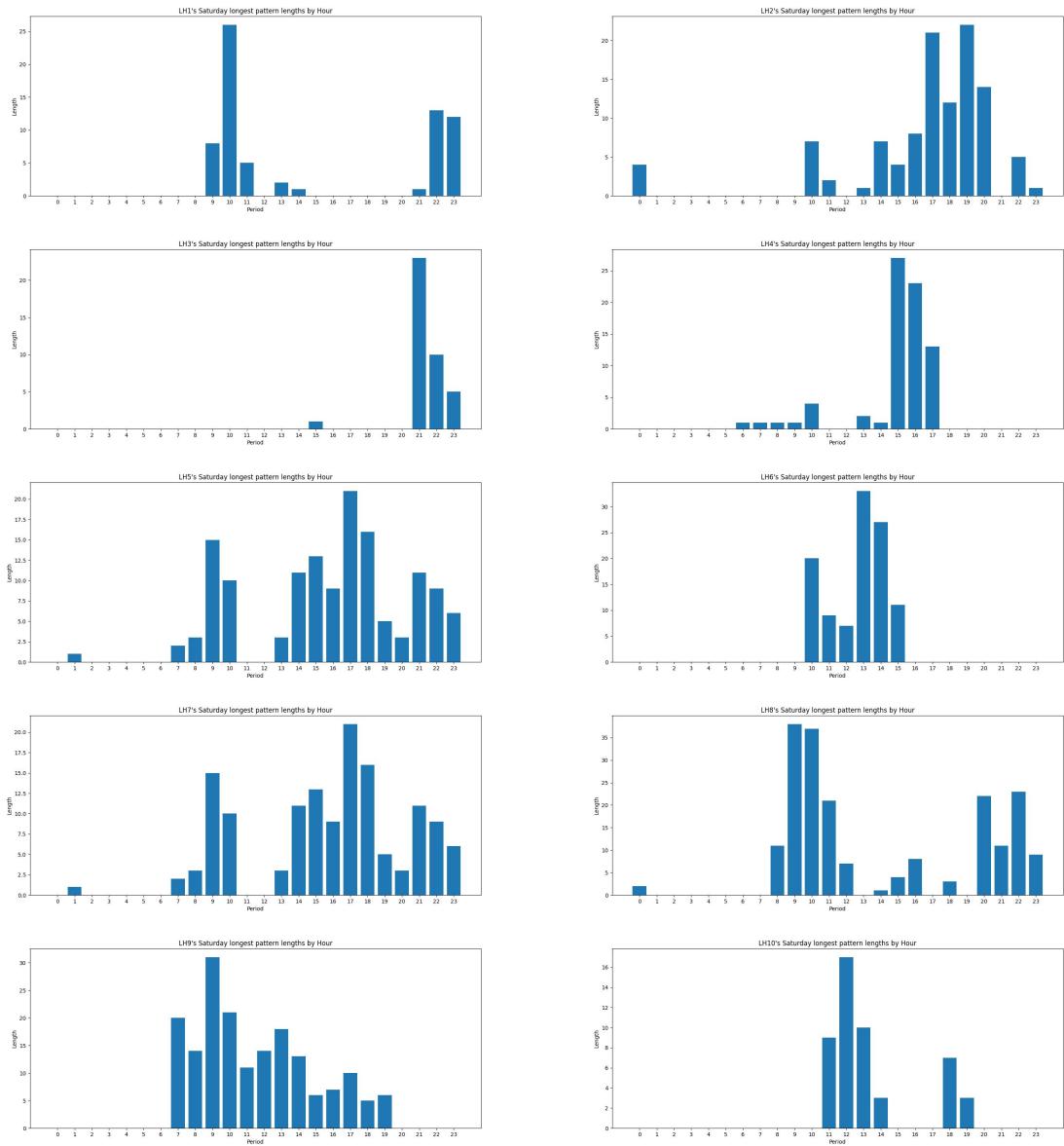


Figura 76: Lunghezza massima oraria dei pattern di cronologia di Sabato per i file di cronologia da LH1 a LH10.

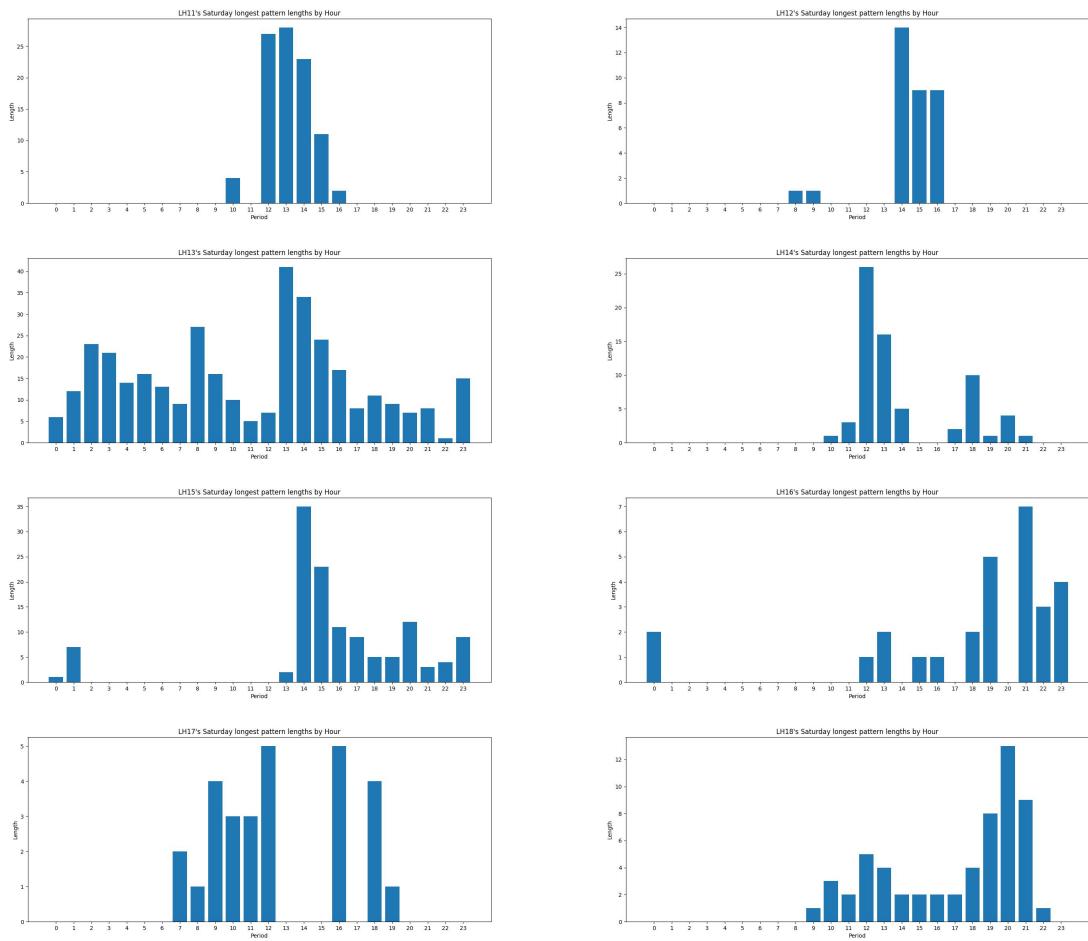


Figura 77: Lunghezza massima oraria dei pattern di cronologia di Sabato per i file di cronologia da LH11 a LH18.

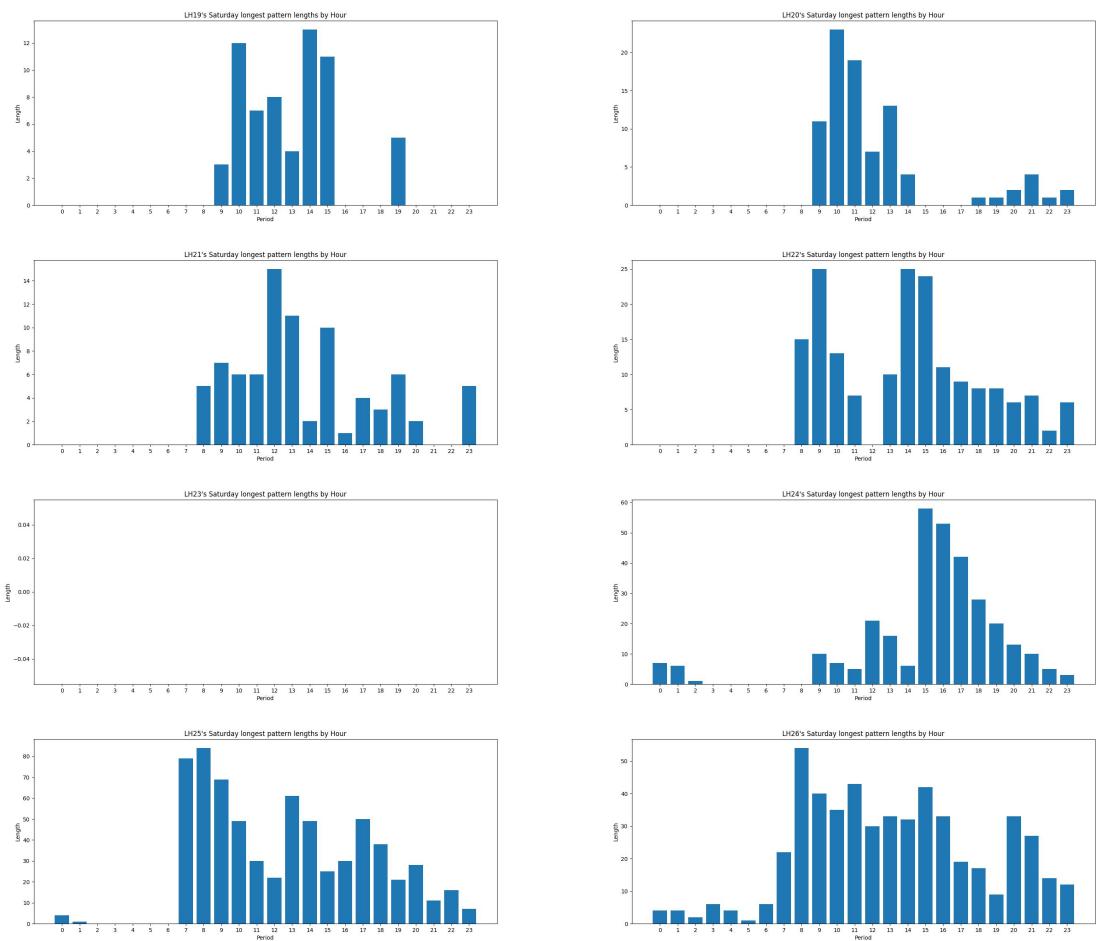


Figura 78: Lunghezza massima oraria dei pattern di cronologia di Sabato per i file di cronologia da LH19 a LH26.

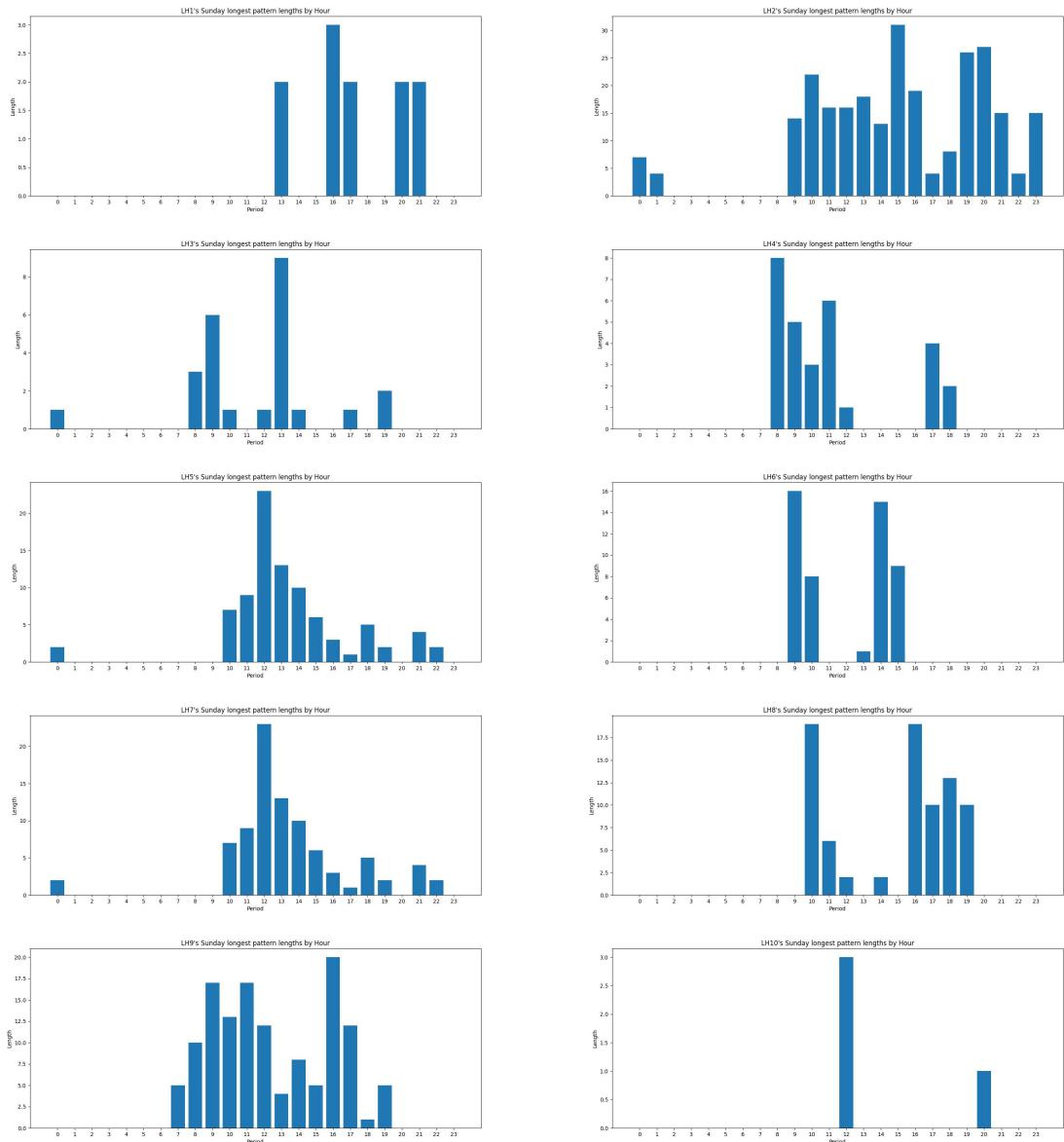


Figura 79: Lunghezza massima oraria dei pattern di cronologia di Domenica per i file di cronologia da LH1 a LH10.

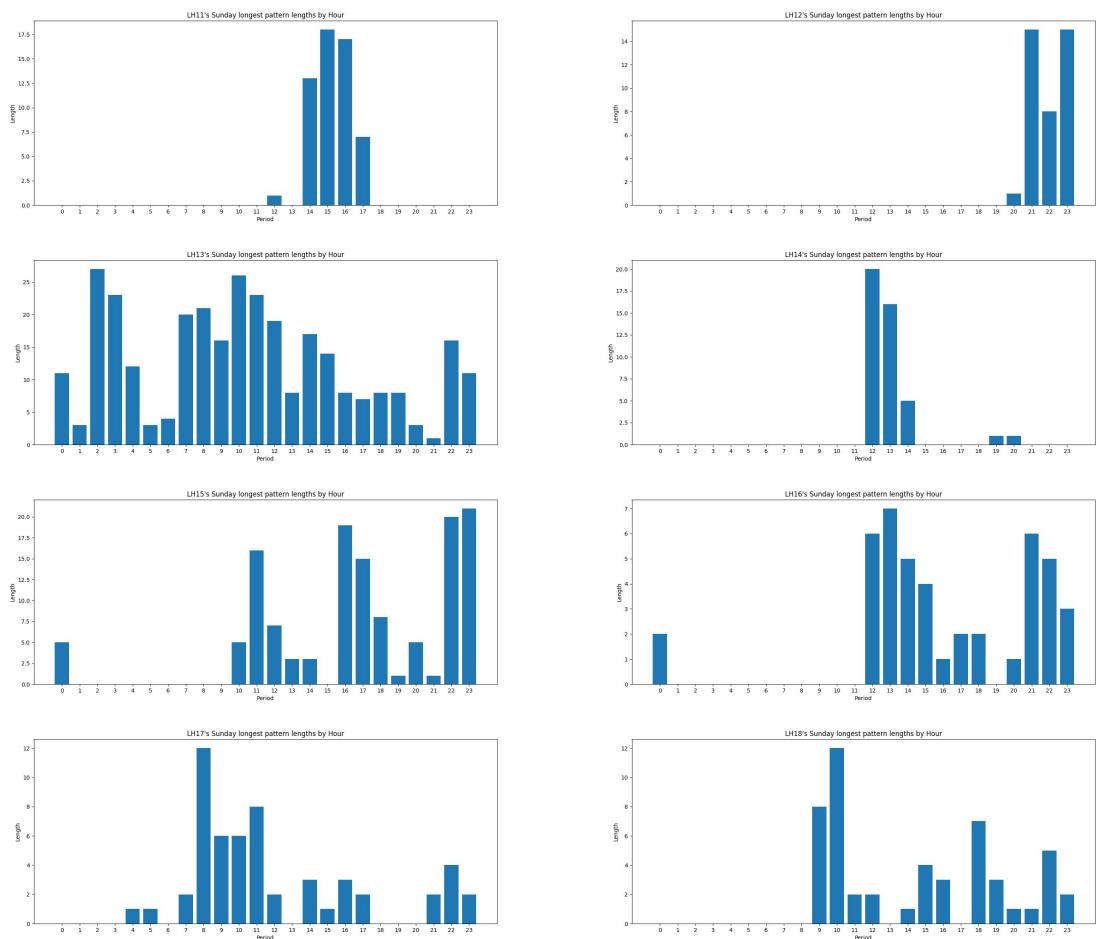


Figura 80: Lunghezza massima oraria dei pattern di cronologia di Domenica per i file di cronologia da LH11 a LH18.

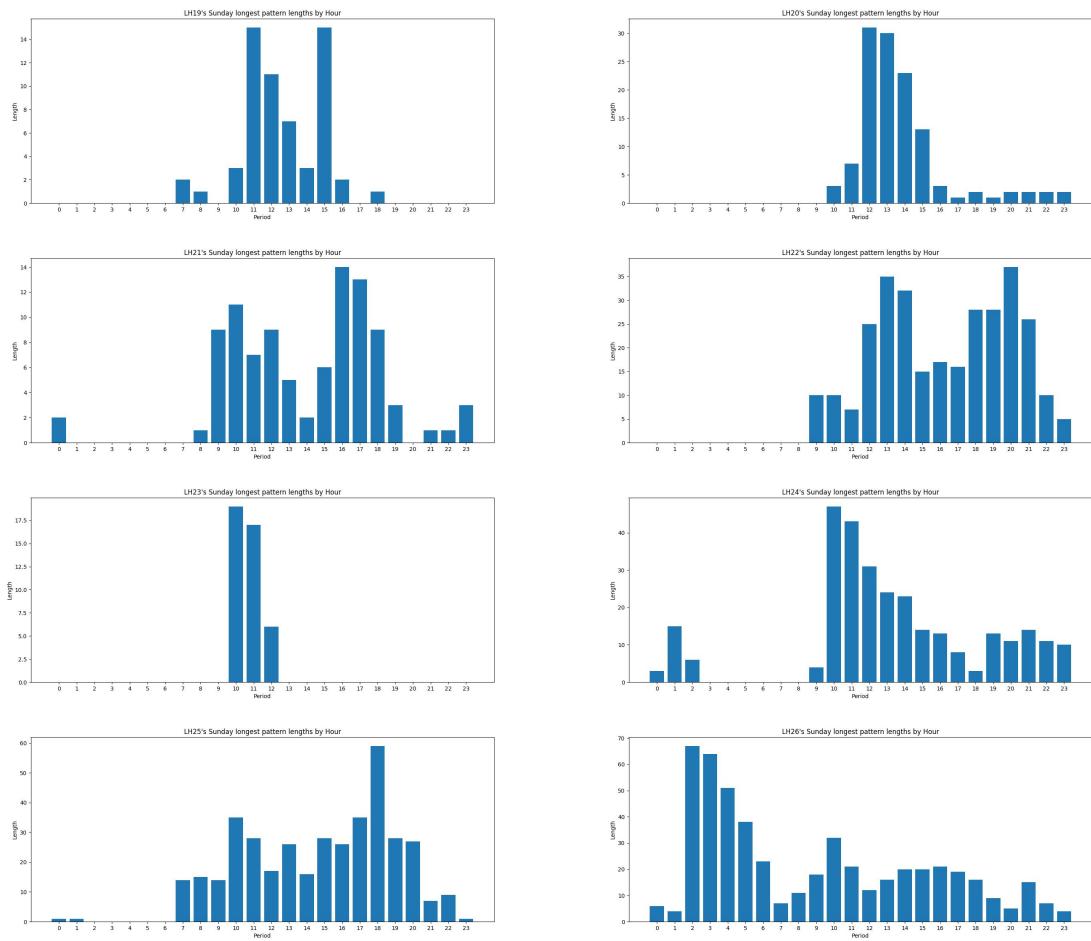


Figura 81: Lunghezza massima oraria dei pattern di cronologia di Domenica per i file di cronologia da LH19 a LH26.

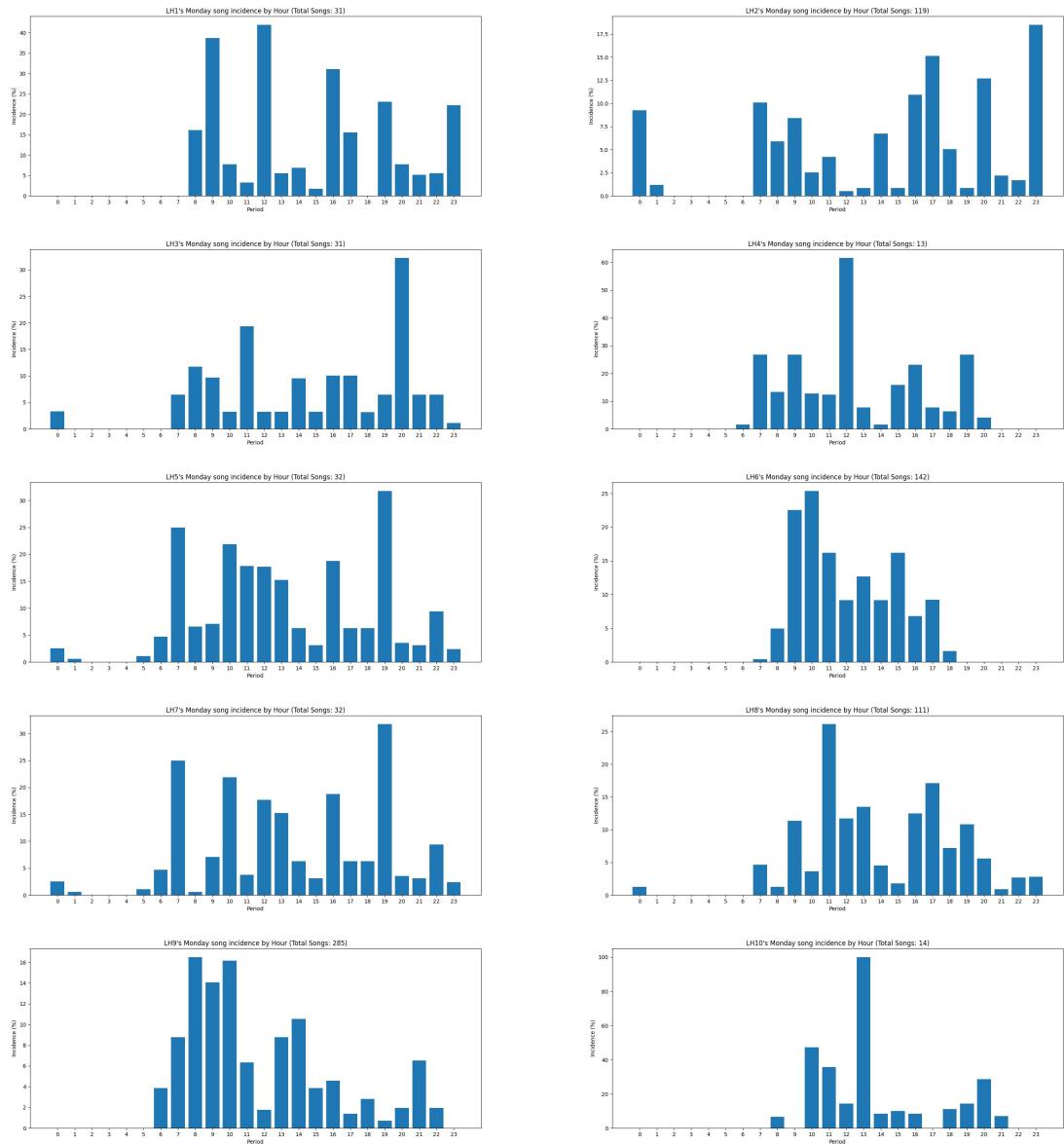


Figura 82: Incidenza percentuale oraria di ascolto durante il giorno Lunedì per i file di cronologia da LH1 a LH10.

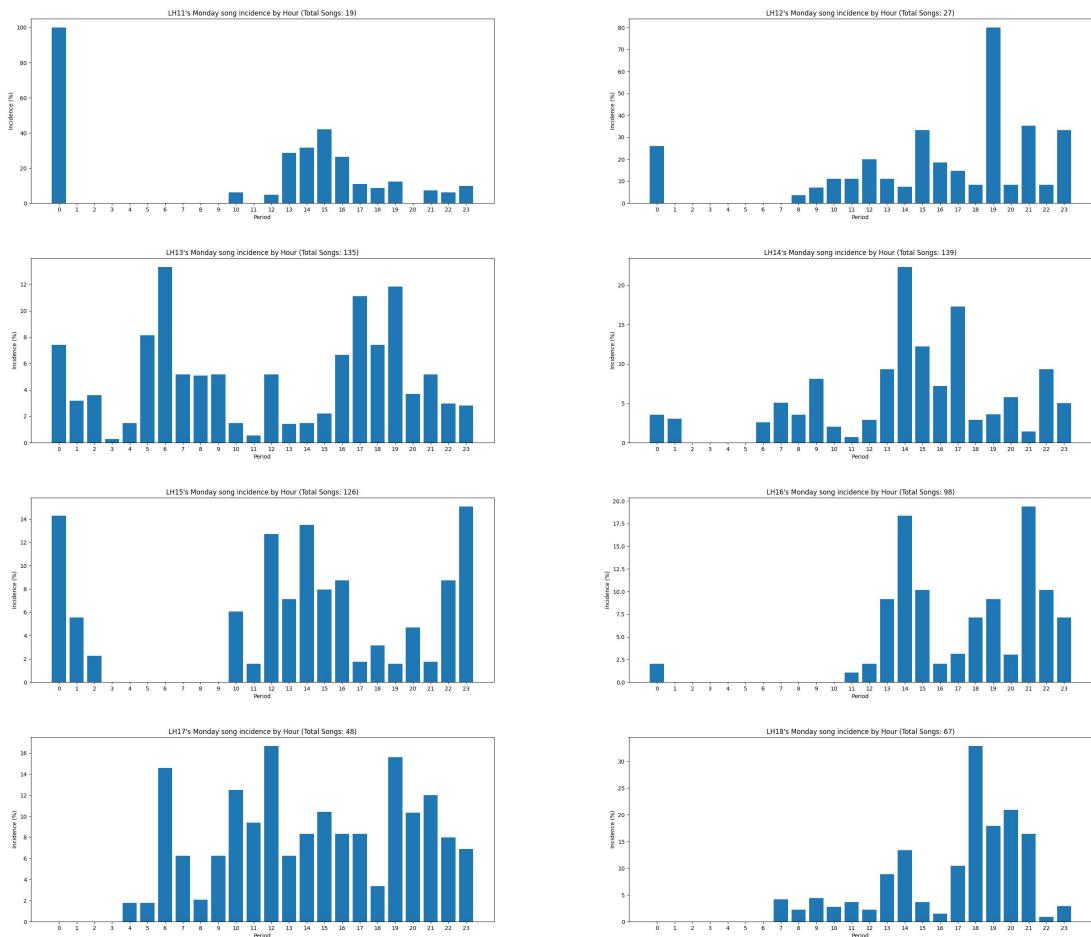


Figura 83: Incidenza percentuale oraria di ascolto durante il giorno Lunedì per i file di cronologia da LH11 a LH18.

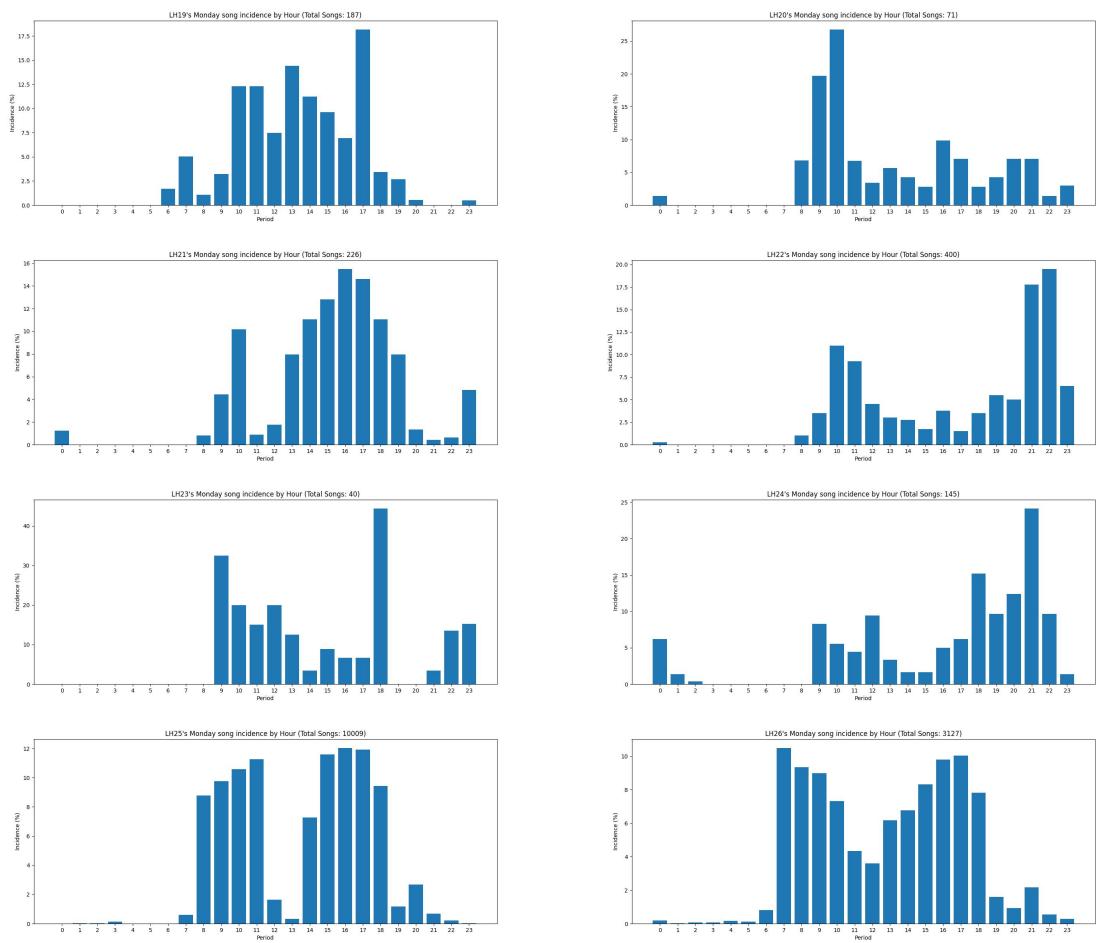


Figura 84: Incidenza percentuale oraria di ascolto durante il giorno Lunedì per i file di cronologia da LH19 a LH26.

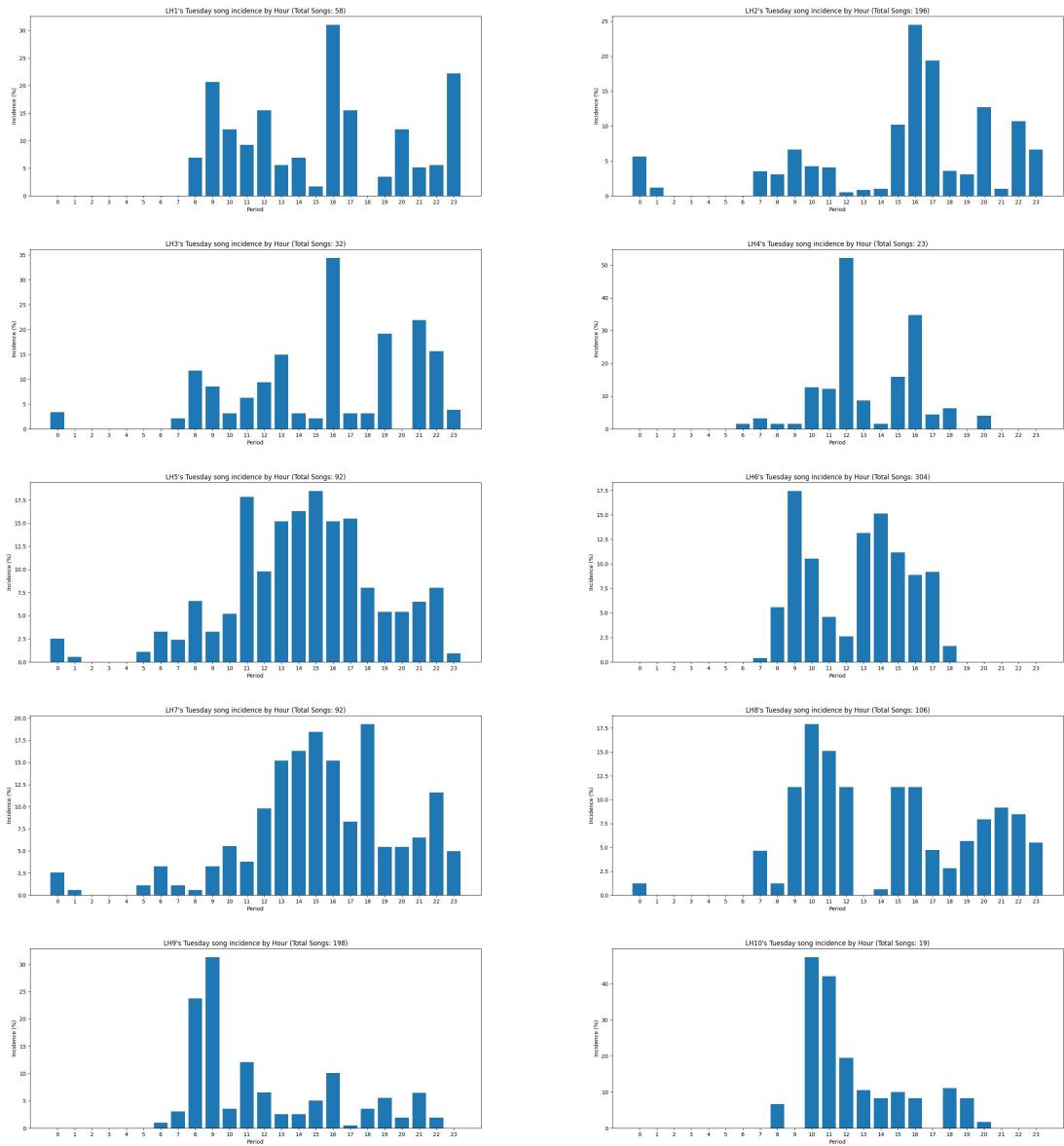


Figura 85: Incidenza percentuale oraria di ascolto durante il giorno Martedì per i file di cronologia da LH1 a LH10.

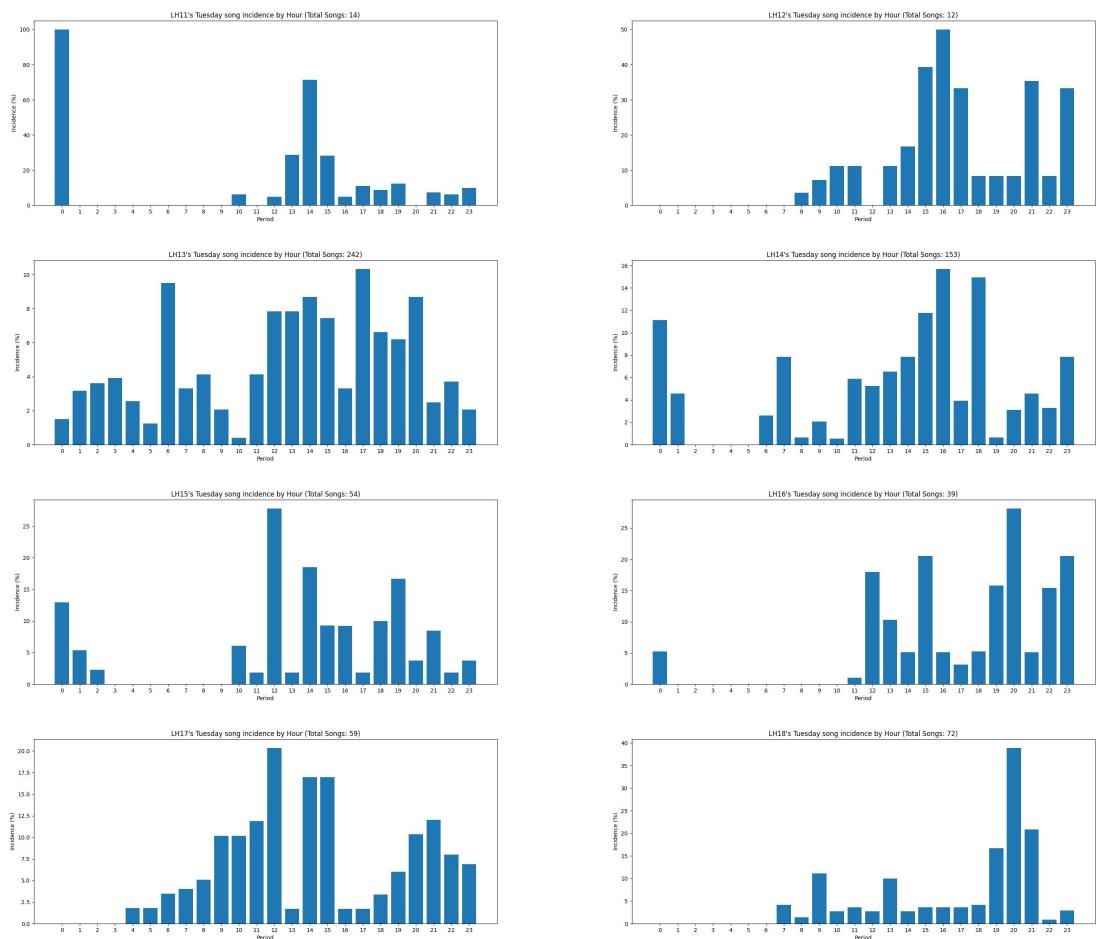


Figura 86: Incidenza percentuale oraria di ascolto durante il giorno Martedì per i file di cronologia da LH11 a LH18.

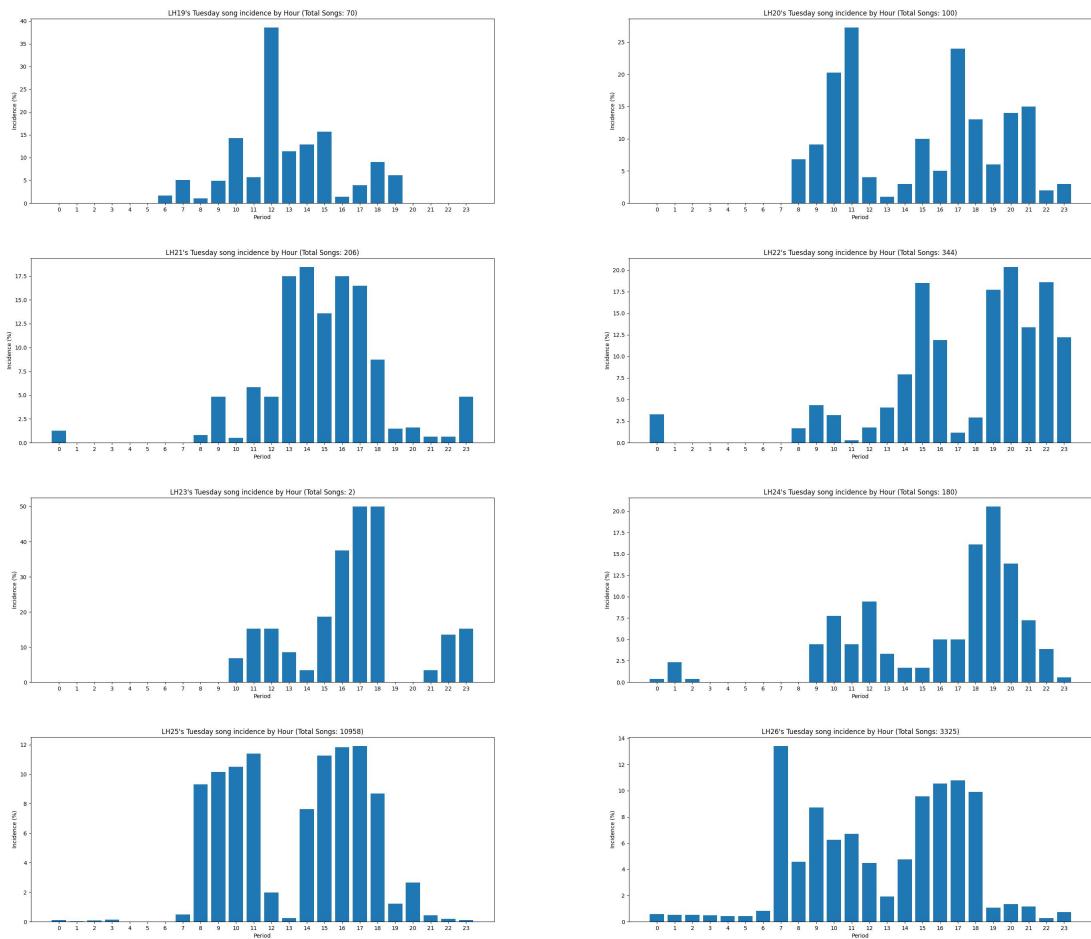


Figura 87: Incidenza percentuale oraria di ascolto durante il giorno Martedì per i file di cronologia da LH19 a LH26.

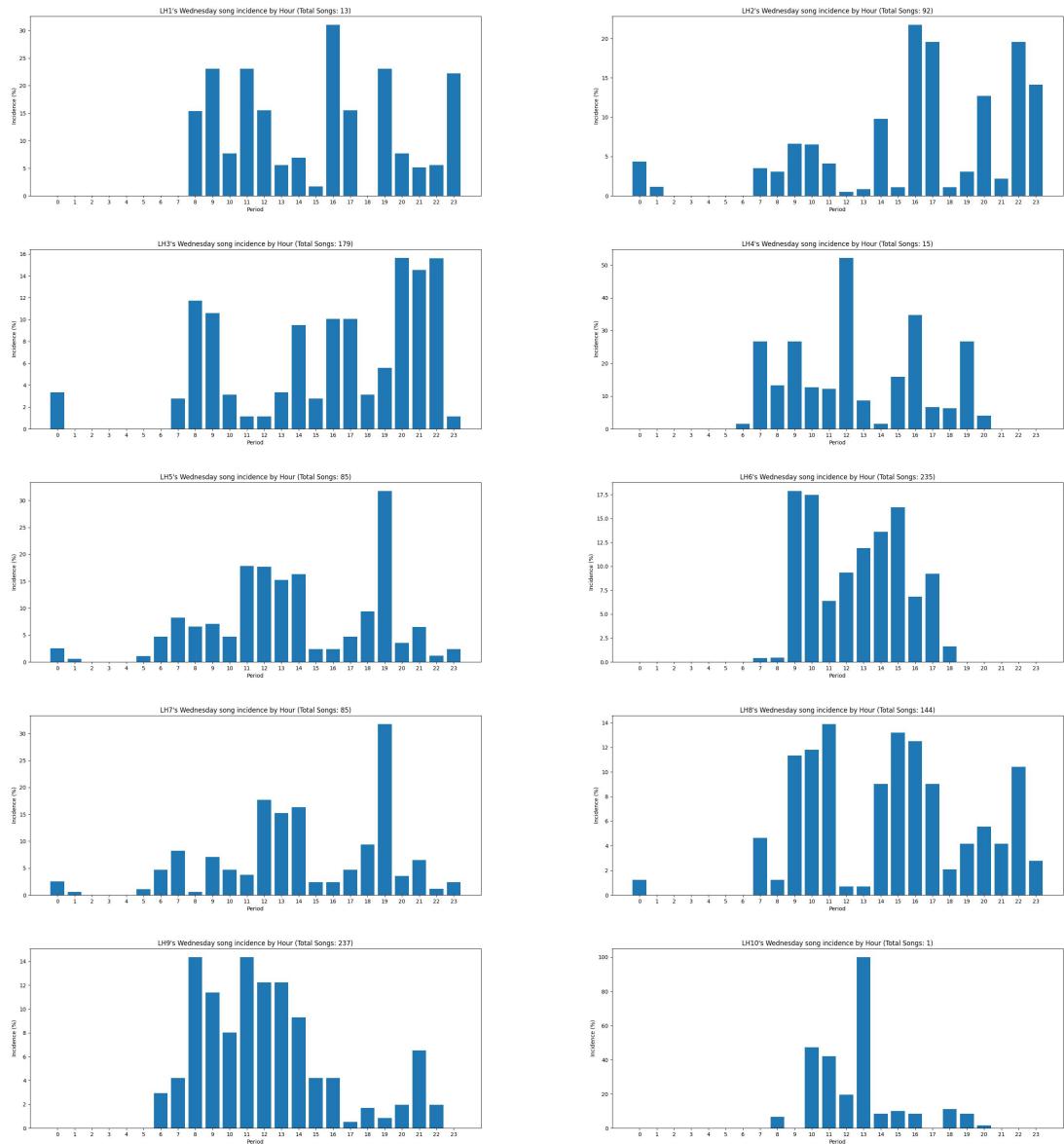


Figura 88: Incidenza percentuale oraria di ascolto durante il giorno Mercoledì per i file di cronologia da LH1 a LH10.

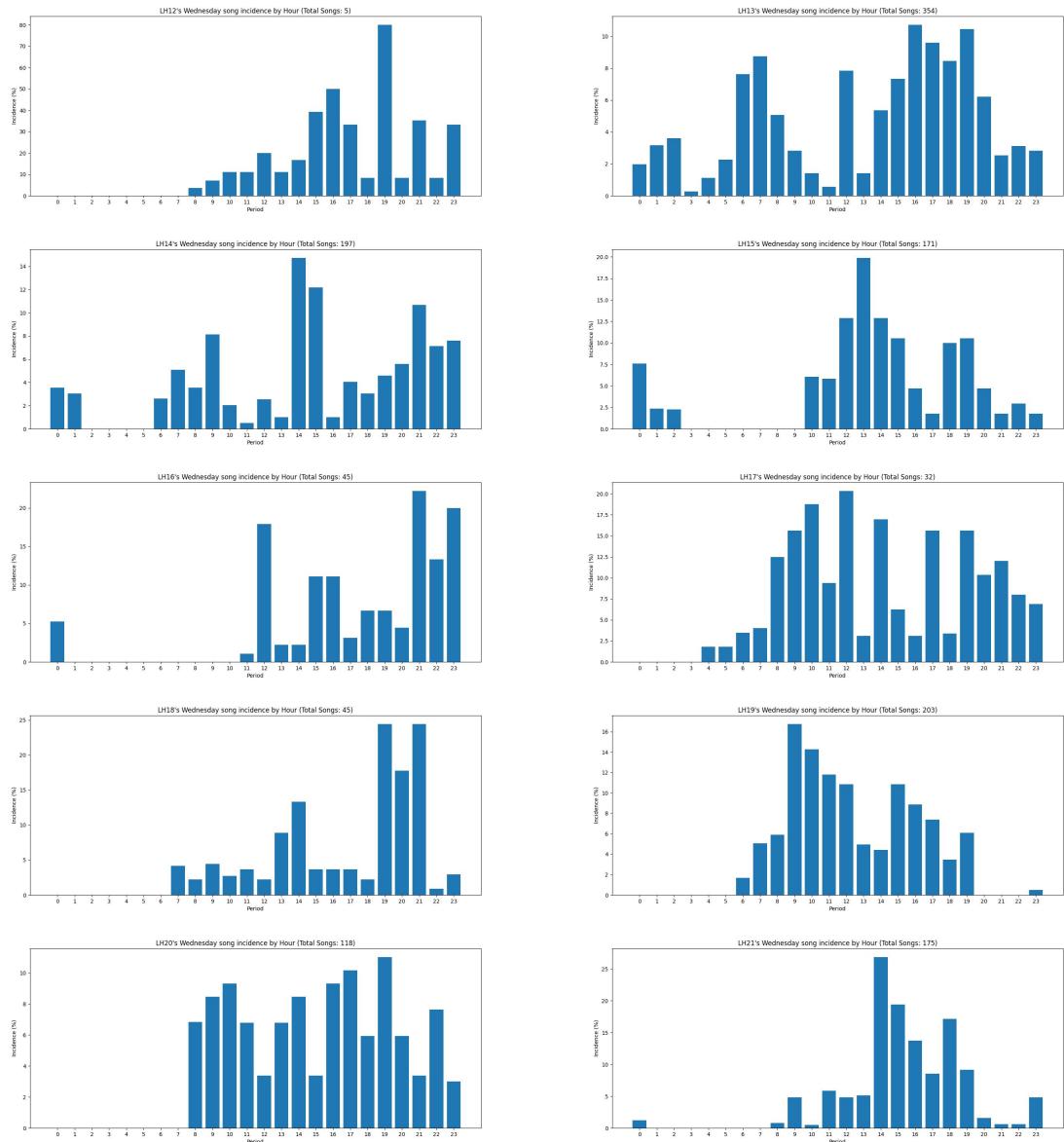


Figura 89: Incidenza percentuale oraria di ascolto durante il giorno Mercoledì per i file di cronologia da LH12 a LH21.

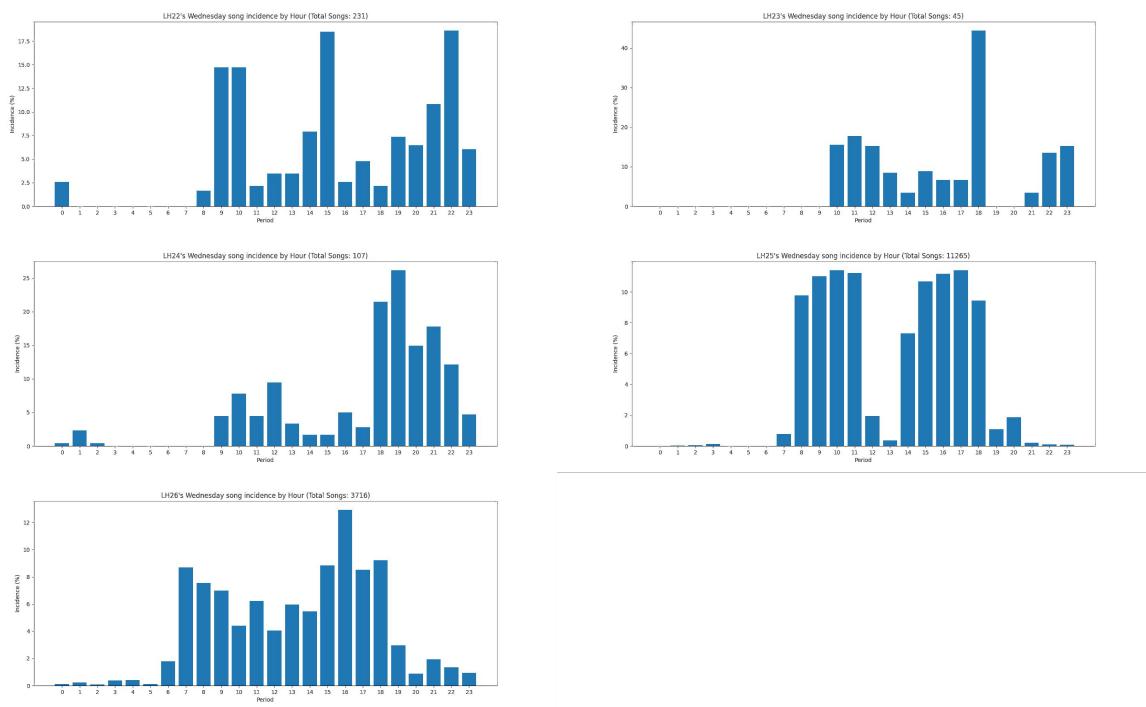


Figura 90: Incidenza percentuale oraria di ascolto durante il giorno Mercoledì per i file di cronologia da LH22 a LH26.

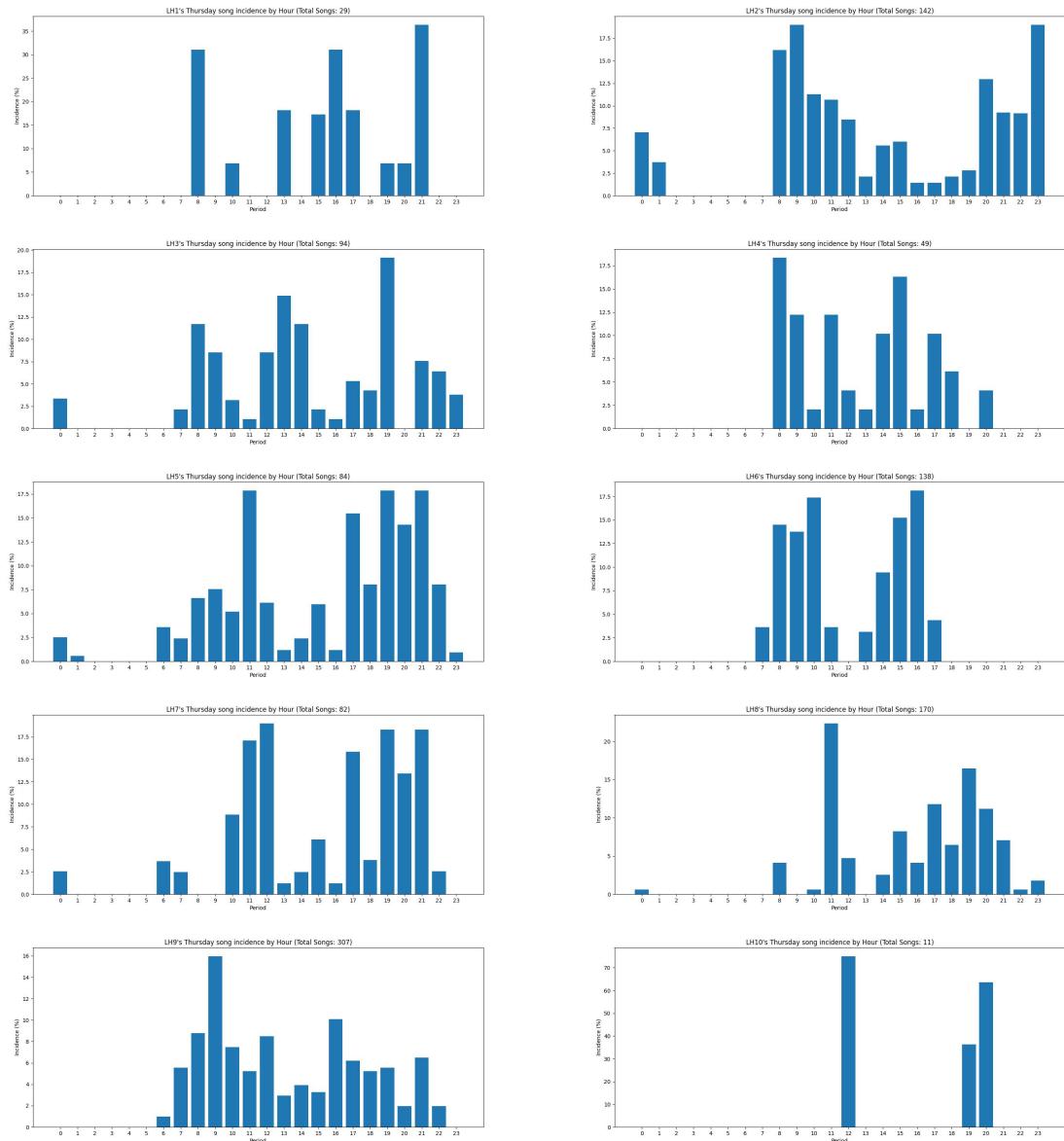


Figura 91: Incidenza percentuale oraria di ascolto durante il giorno Giovedì per i file di cronologia da LH1 a LH10.

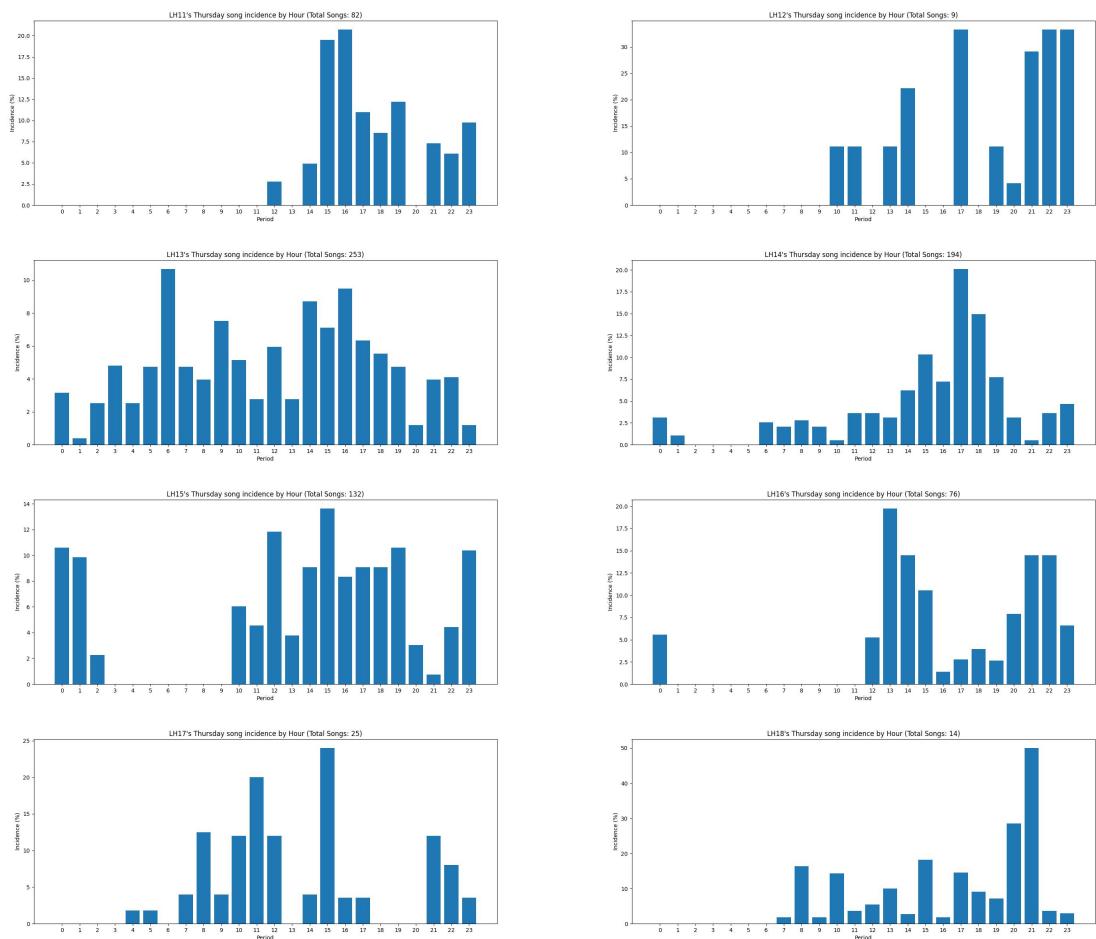


Figura 92: Incidenza percentuale oraria di ascolto durante il giorno Giovedì per i file di cronologia da LH11 a LH18.

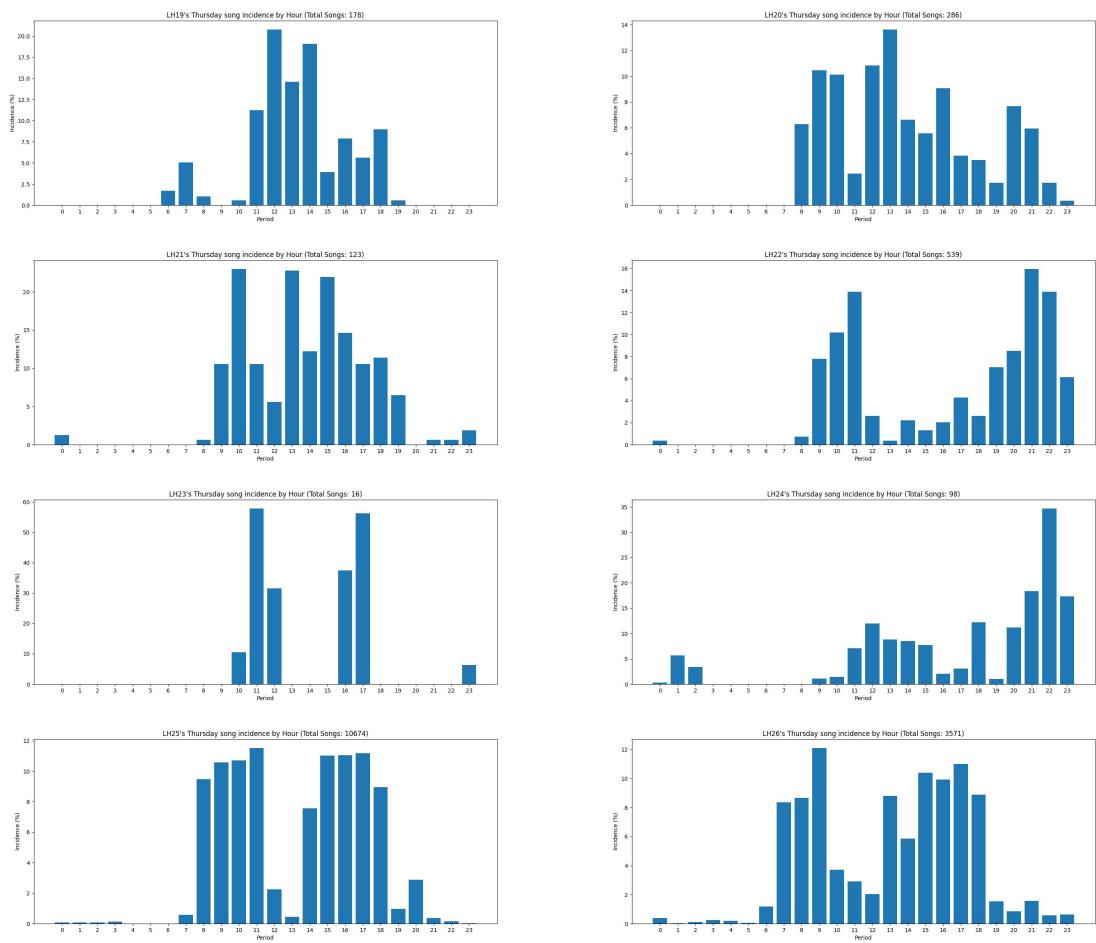


Figura 93: Incidenza percentuale oraria di ascolto durante il giorno Giovedì per i file di cronologia da LH19 a LH26.

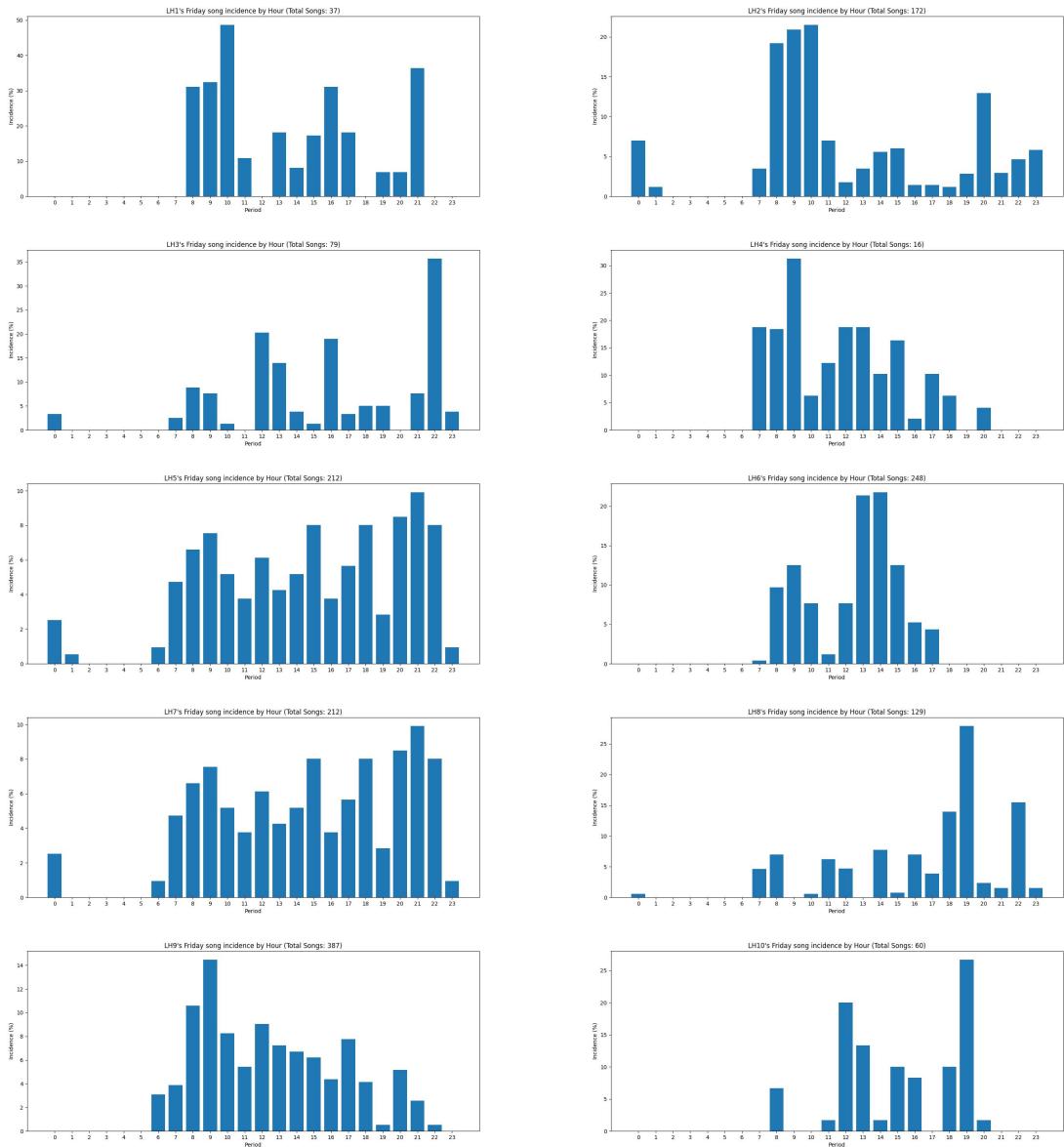


Figura 94: Incidenza percentuale oraria di ascolto durante il giorno Venerdì per i file di cronologia da LH1 a LH10.

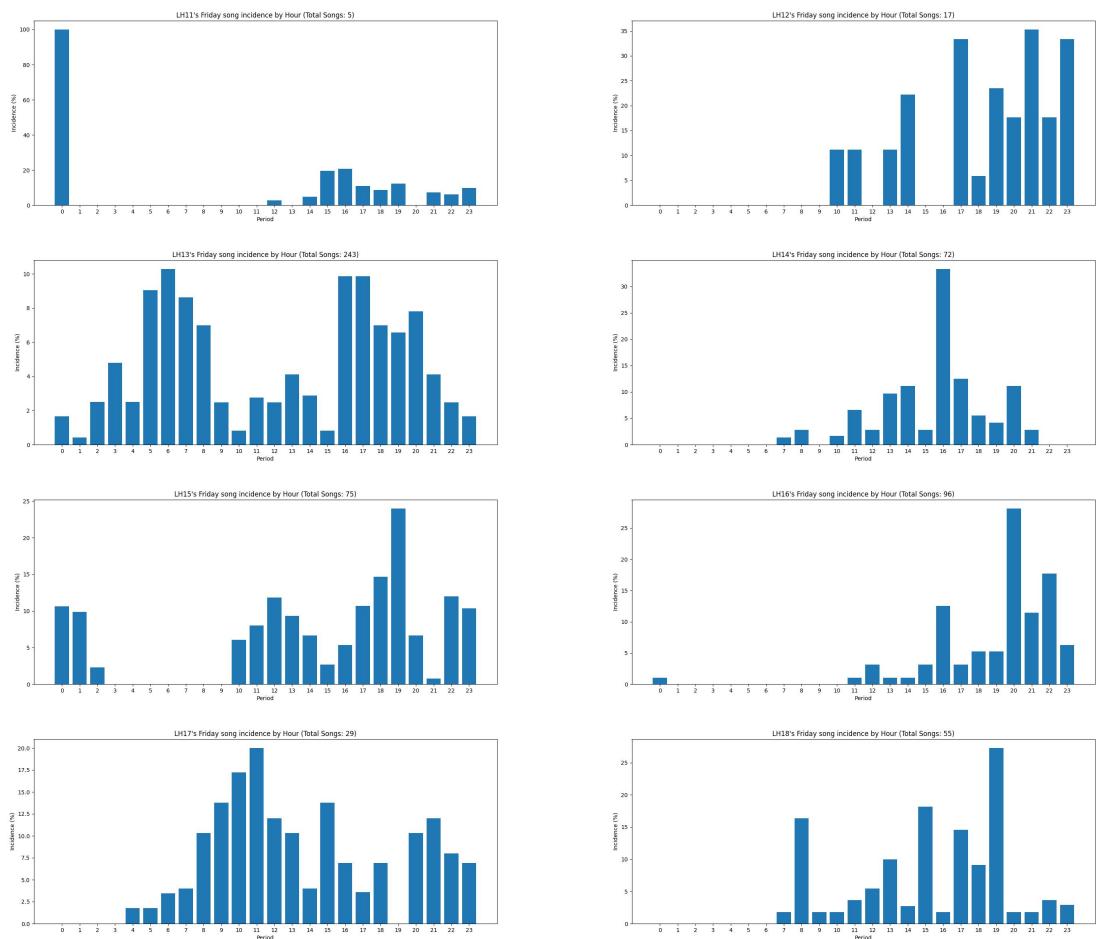


Figura 95: Incidenza percentuale oraria di ascolto durante il giorno Venerdì per i file di cronologia da LH11 a LH18.

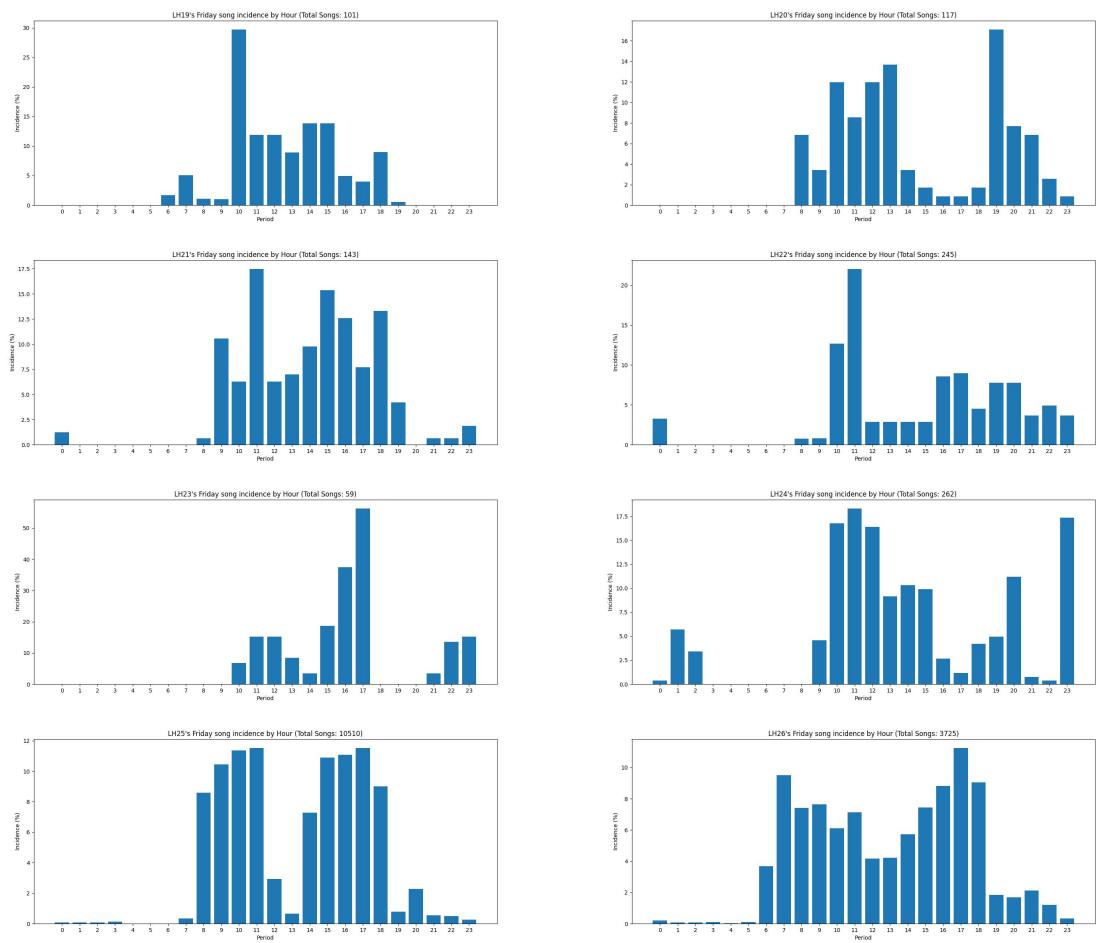


Figura 96: Incidenza percentuale oraria di ascolto durante il giorno Venerdì per i file di cronologia da LH19 a LH26.

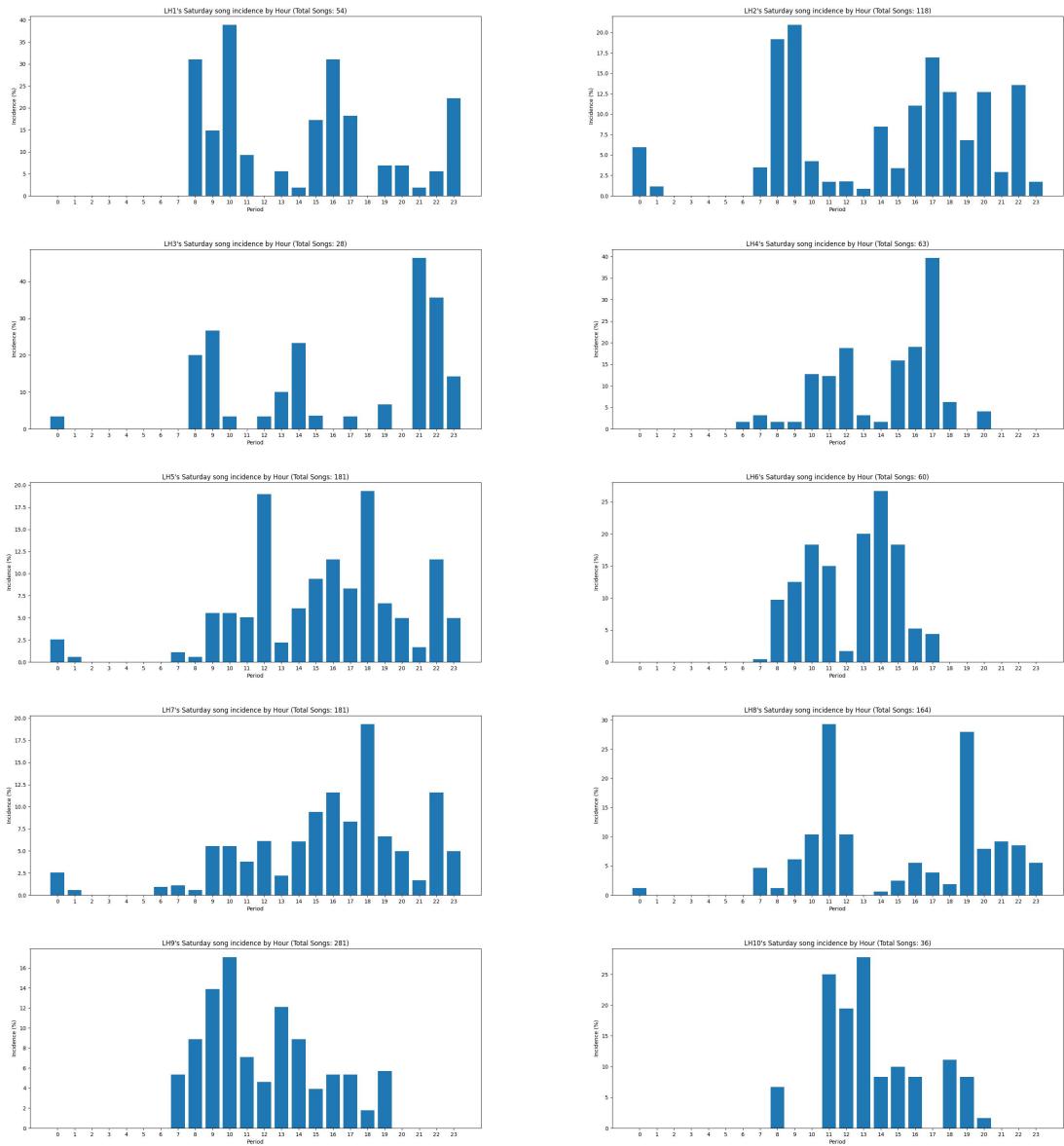


Figura 97: Incidenza percentuale oraria di ascolto durante il giorno Sabato per i file di cronologia da LH1 a LH10.

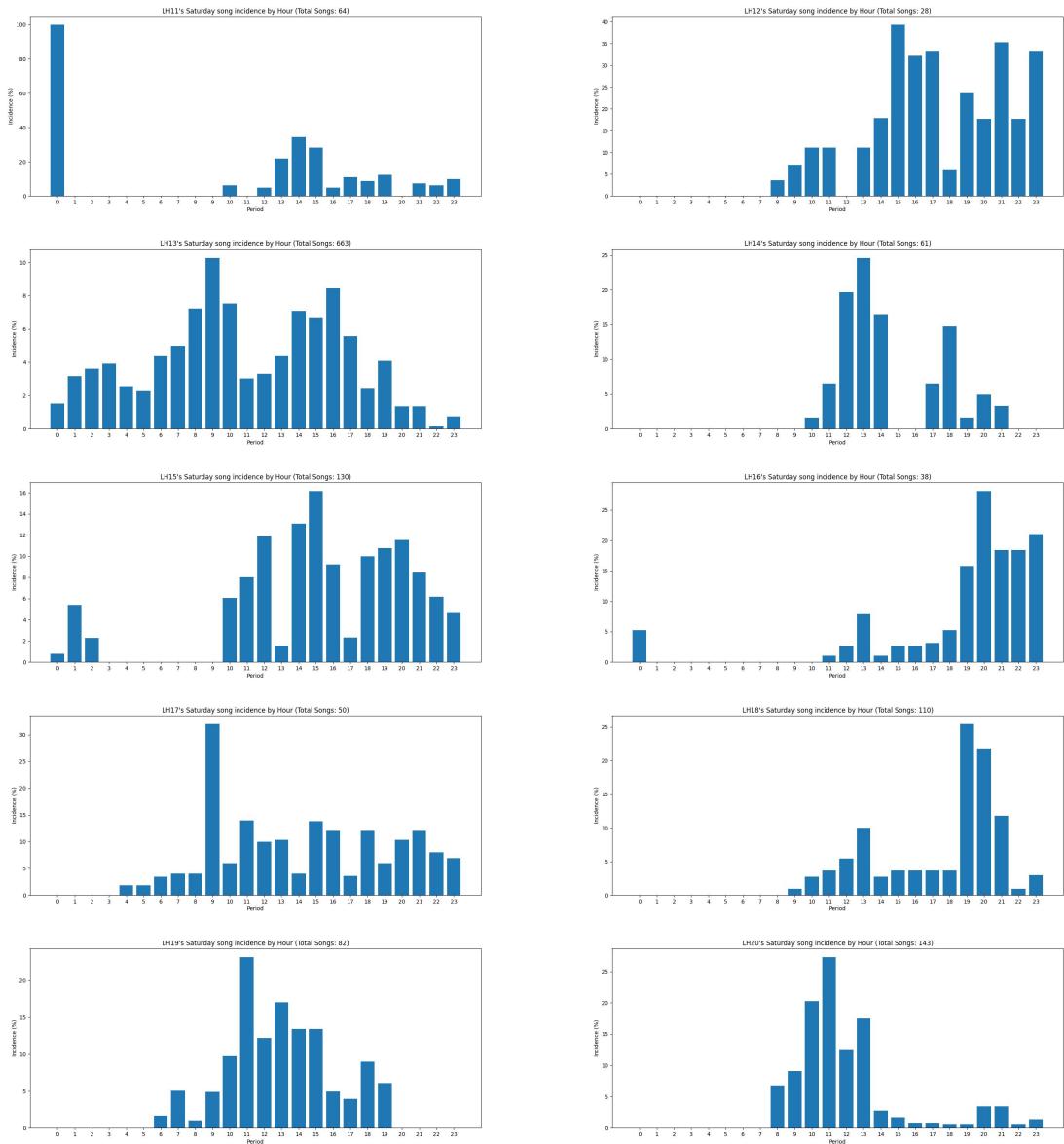


Figura 98: Incidenza percentuale oraria di ascolto durante il giorno Sabato per i file di cronologia da LH11 a LH20.

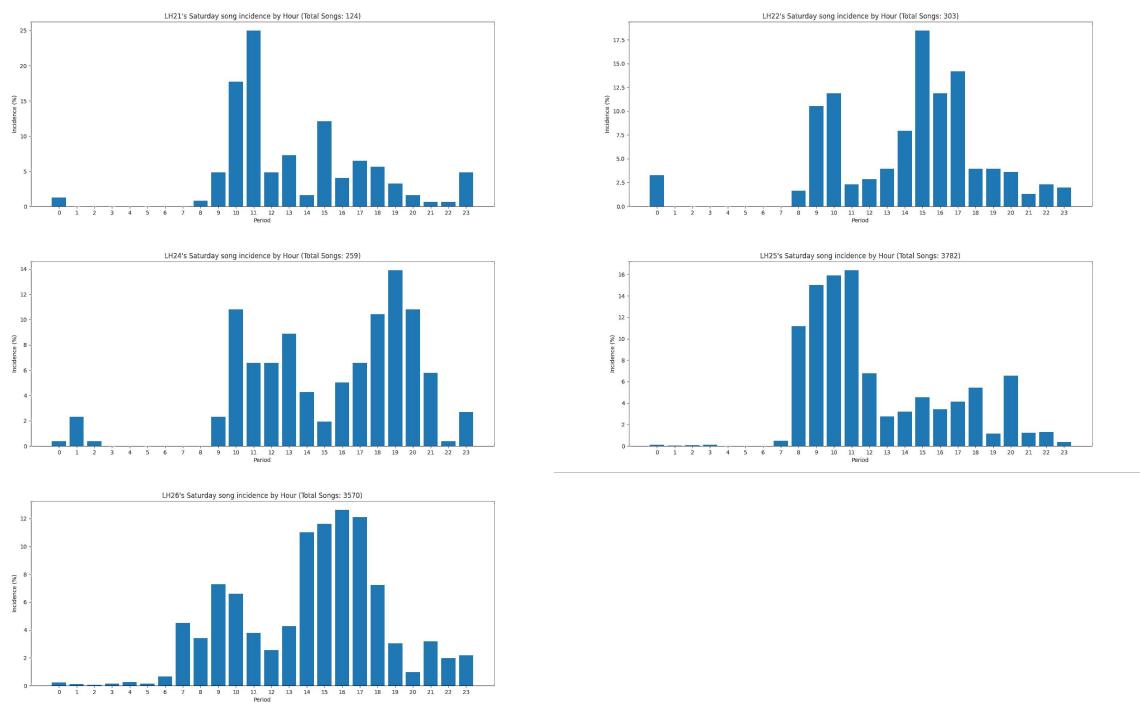


Figura 99: Incidenza percentuale oraria di ascolto durante il giorno Sabato per i file di cronologia da LH21 a LH26.

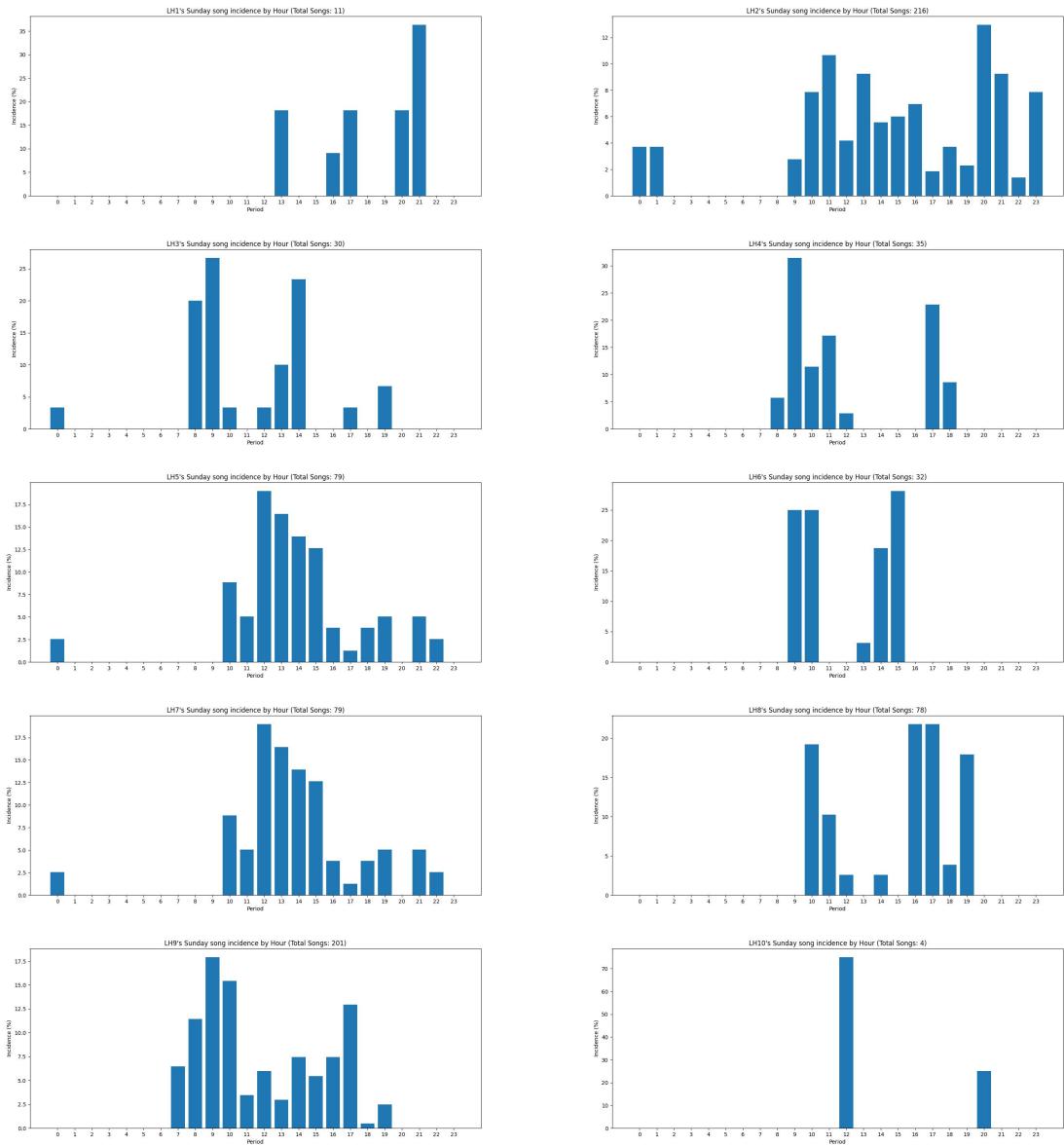


Figura 100: Incidenza percentuale oraria di ascolto durante il giorno Domenica per i file di cronologia da LH1 a LH10.

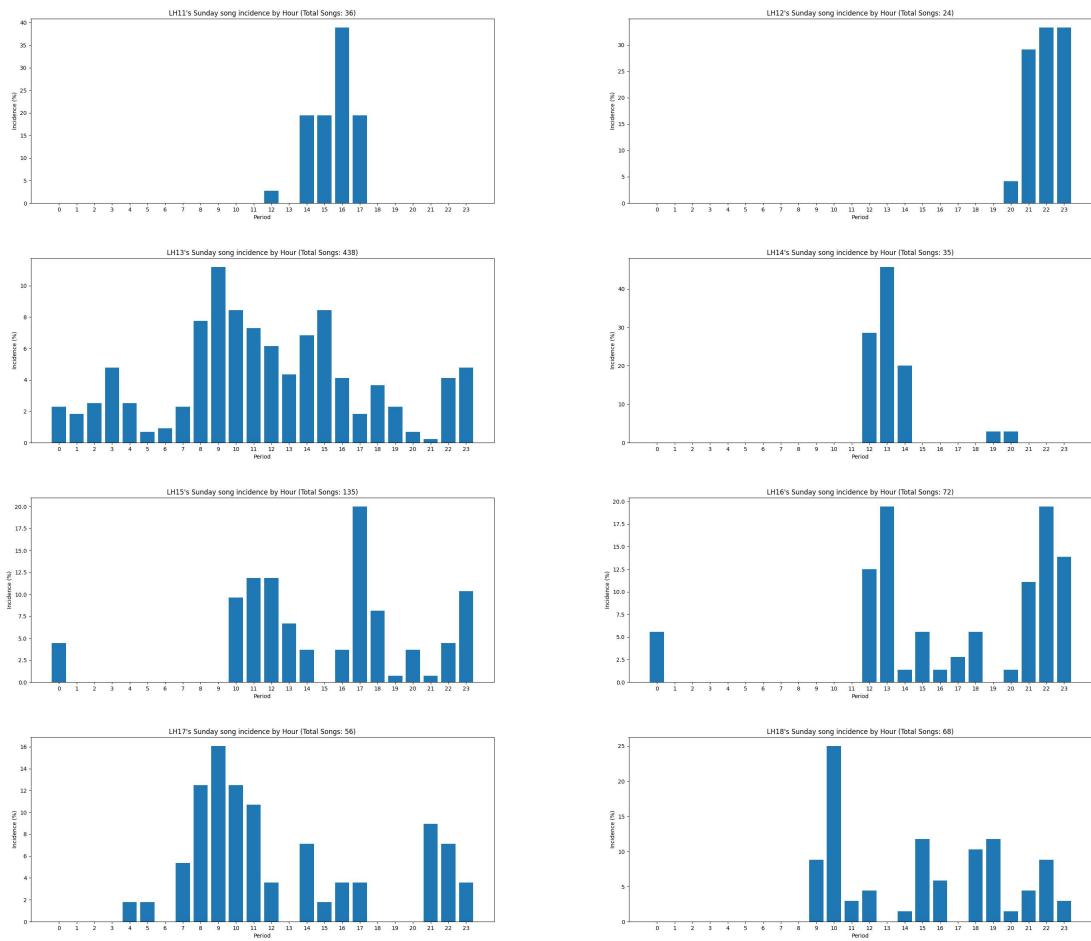


Figura 101: Incidenza percentuale oraria di ascolto durante il giorno Domenica per i file di cronologia da LH11 a LH18.

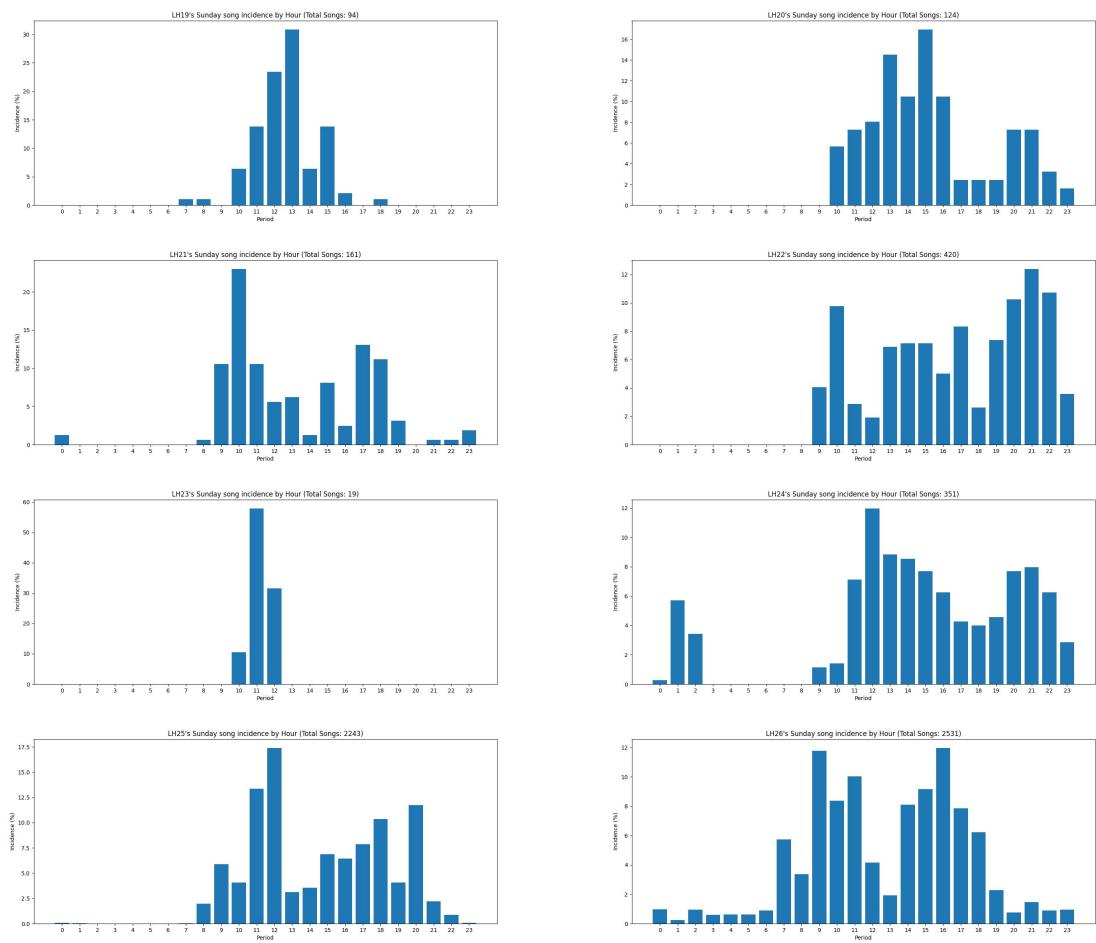


Figura 102: Incidenza percentuale oraria di ascolto durante il giorno Domenica per i file di cronologia da LH19 a LH26.

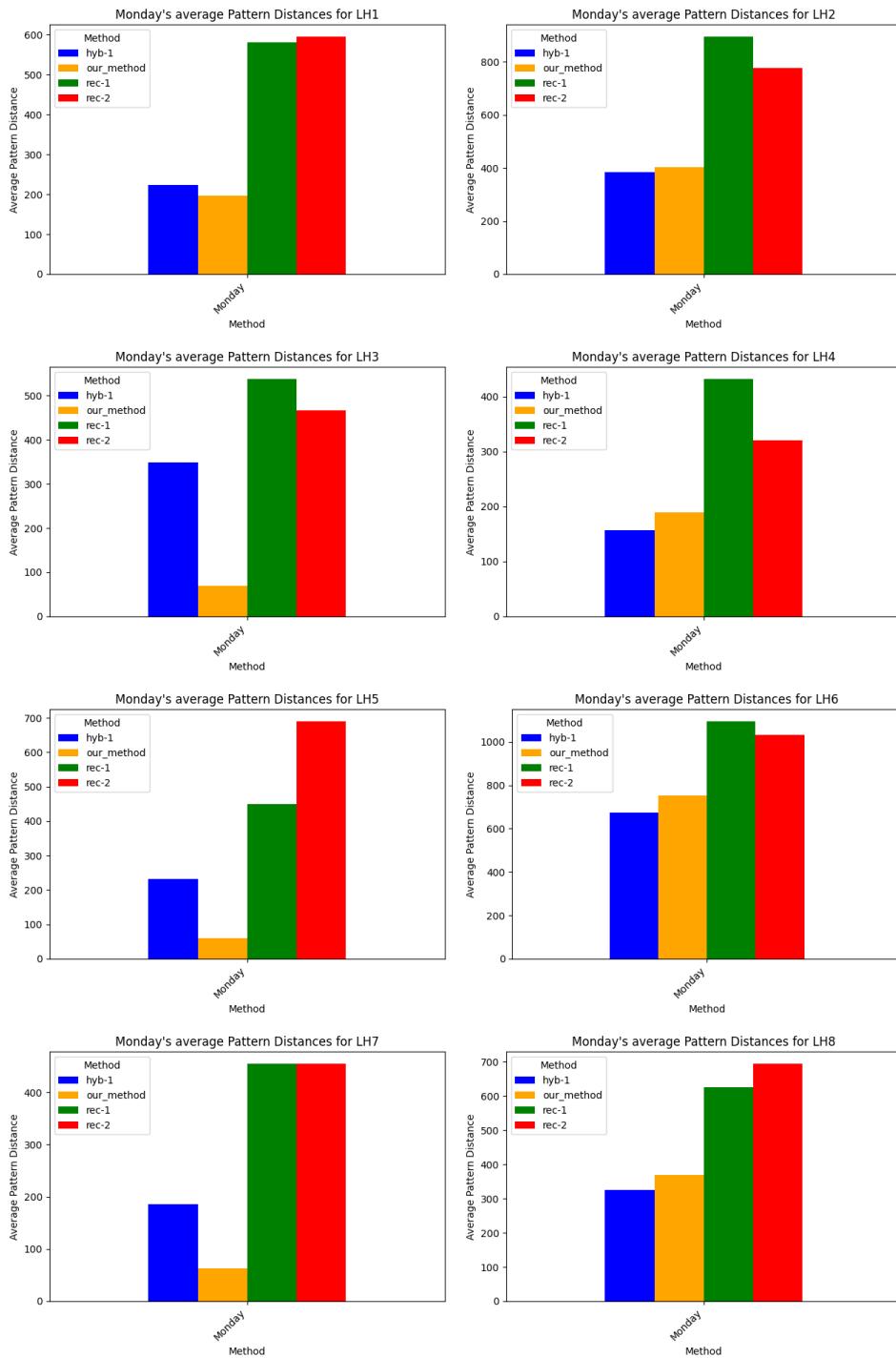


Figura 103: Playlist Pattern Distance media di ogni metodo durante il giorno Lunedì per i file di cronologia da LH1 a LH8.

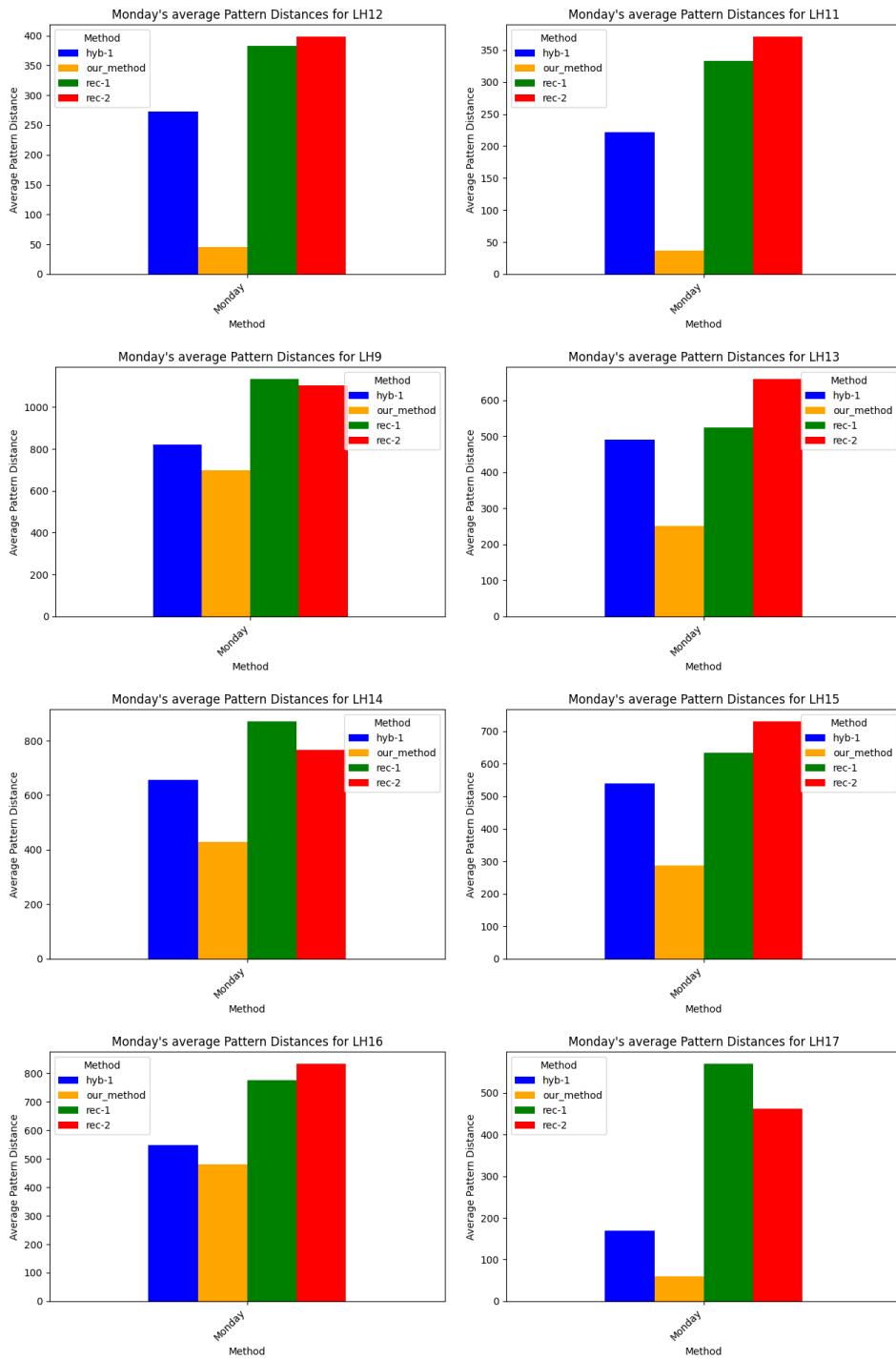


Figura 104: Playlist Pattern Distance media di ogni metodo durante il giorno Lunedì per i file di cronologia da LH9 a LH17.

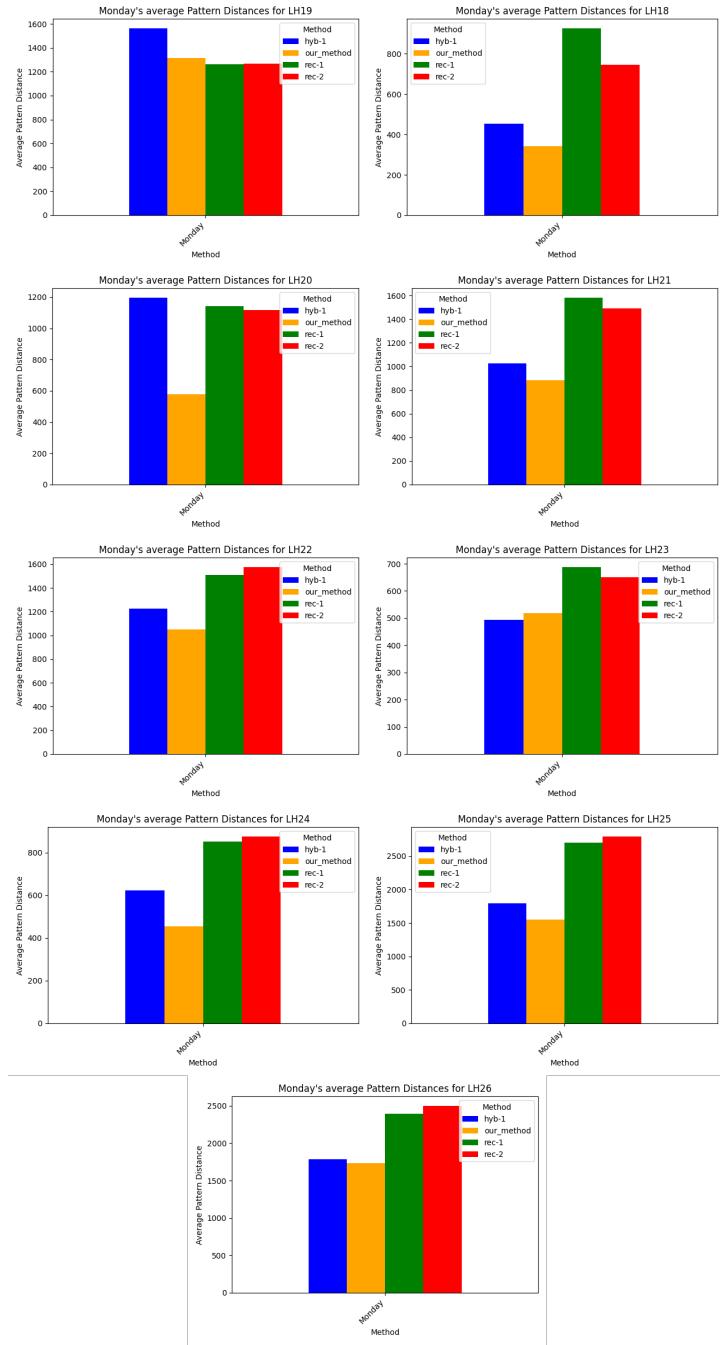


Figura 105: Playlist Pattern Distance media di ogni metodo durante il giorno Lunedì per i file di cronologia da LH18 a LH26.

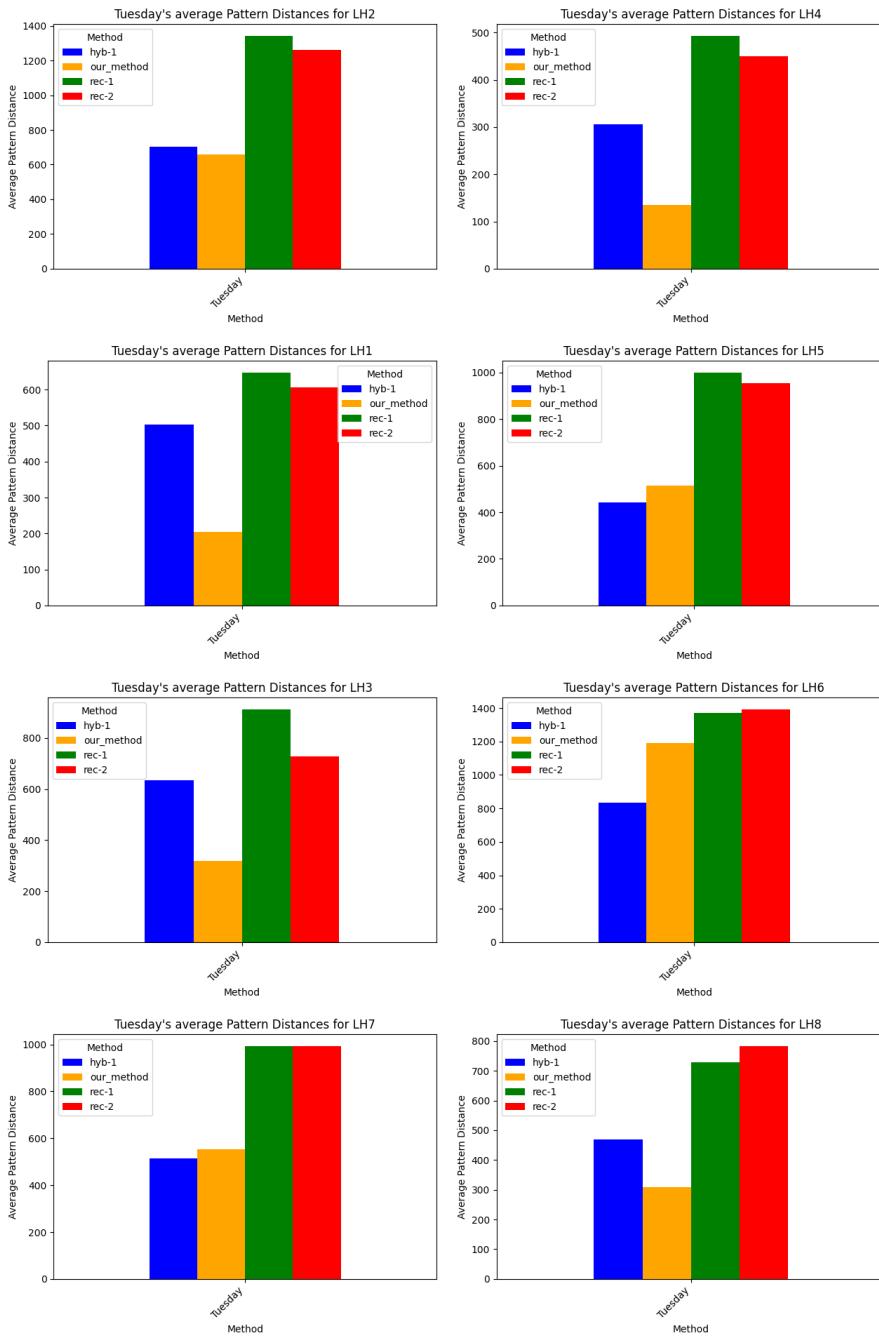


Figura 106: Playlist Pattern Distance media di ogni metodo durante il giorno Martedì per i file di cronologia da LH1 a LH8.

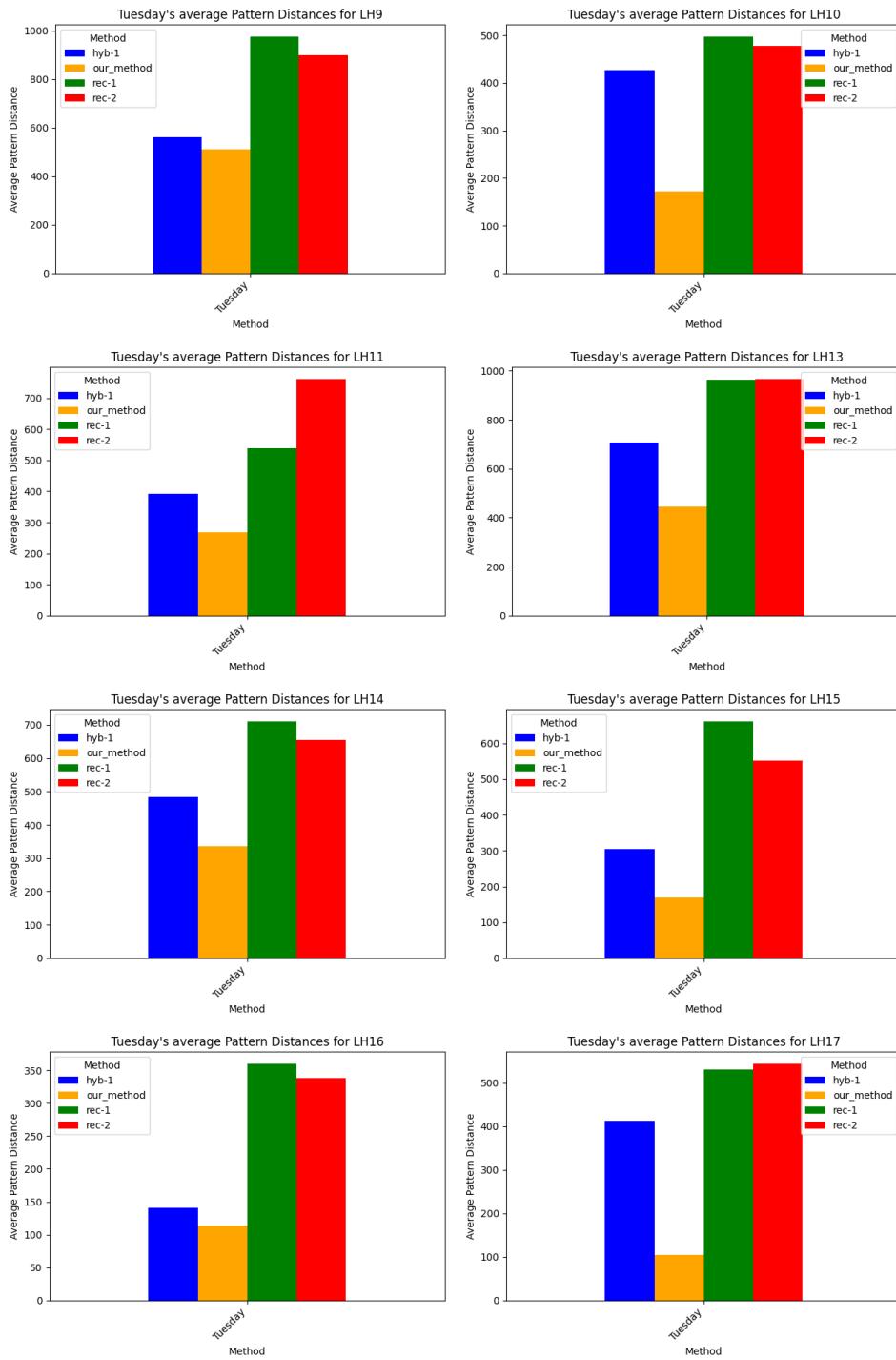


Figura 107: Playlist Pattern Distance media di ogni metodo durante il giorno Martedì per i file di cronologia da LH9 a LH17.

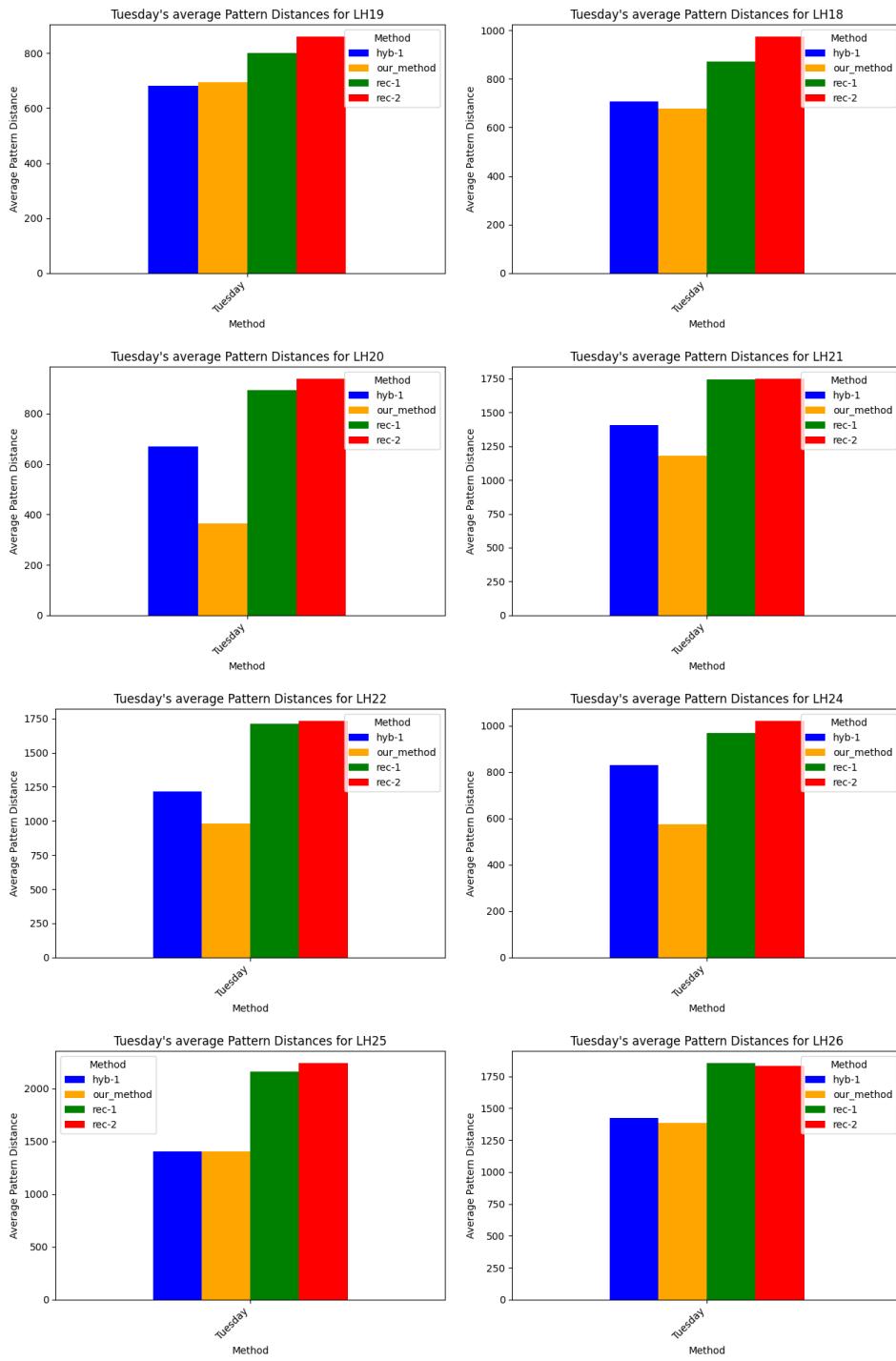


Figura 108: Playlist Pattern Distance media di ogni metodo durante il giorno Martedì per i file di cronologia da LH18 a LH26.

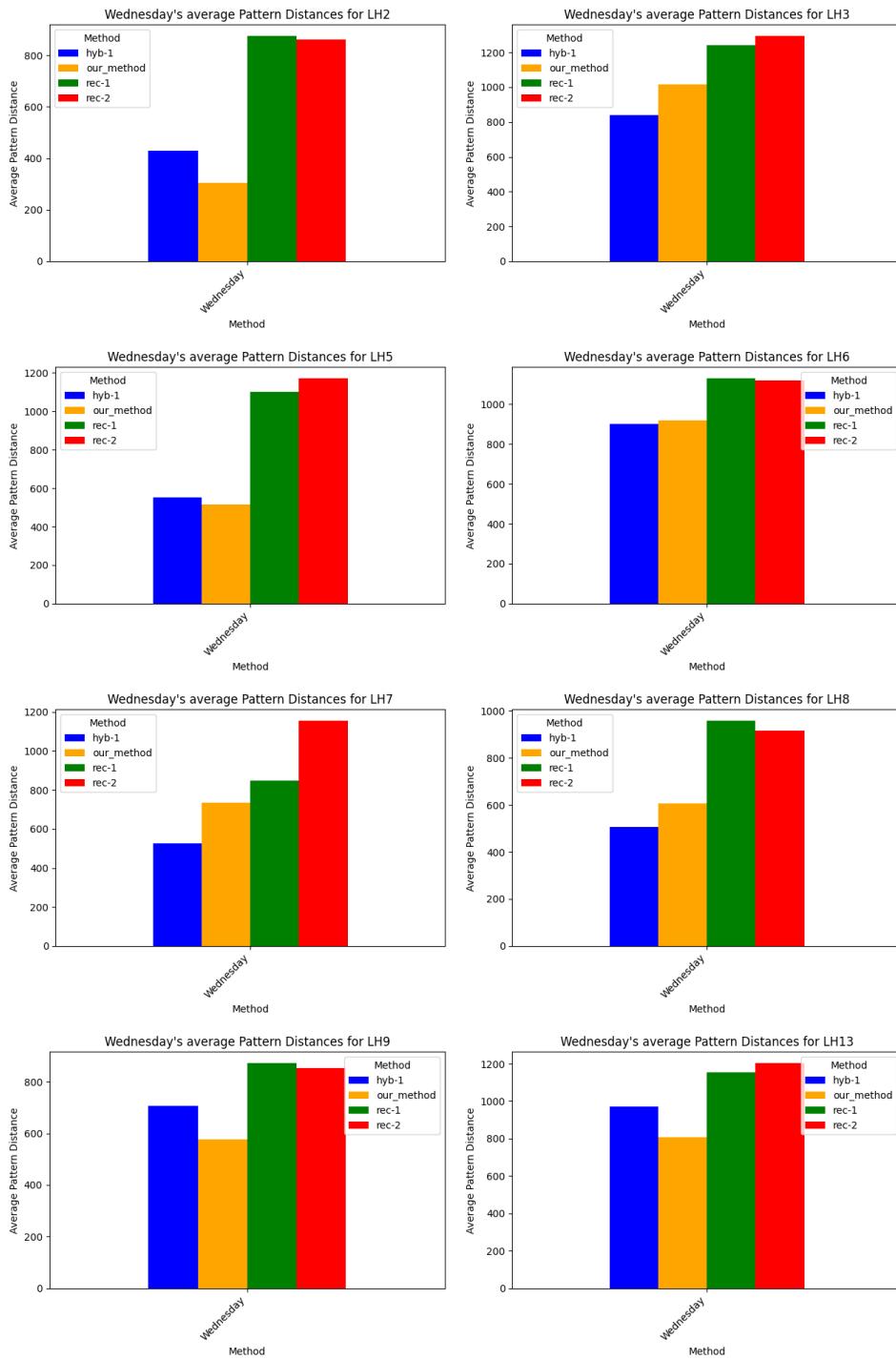


Figura 109: Playlist Pattern Distance media di ogni metodo durante il giorno Mercoledì per i file di cronologia da LH2 a LH13.

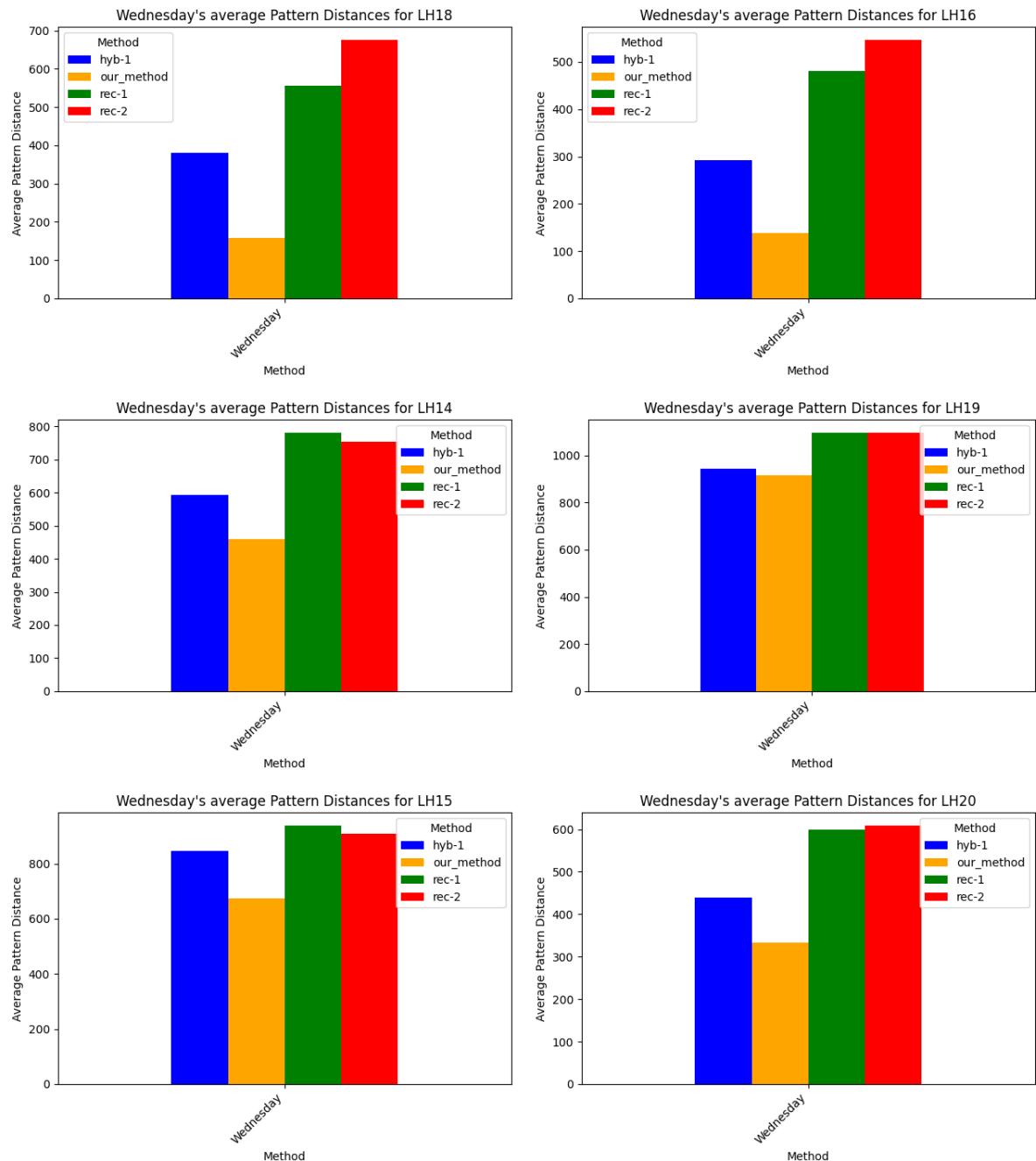


Figura 110: Playlist Pattern Distance media di ogni metodo durante il giorno Mercoledì per i file di cronologia da LH14 a LH20.

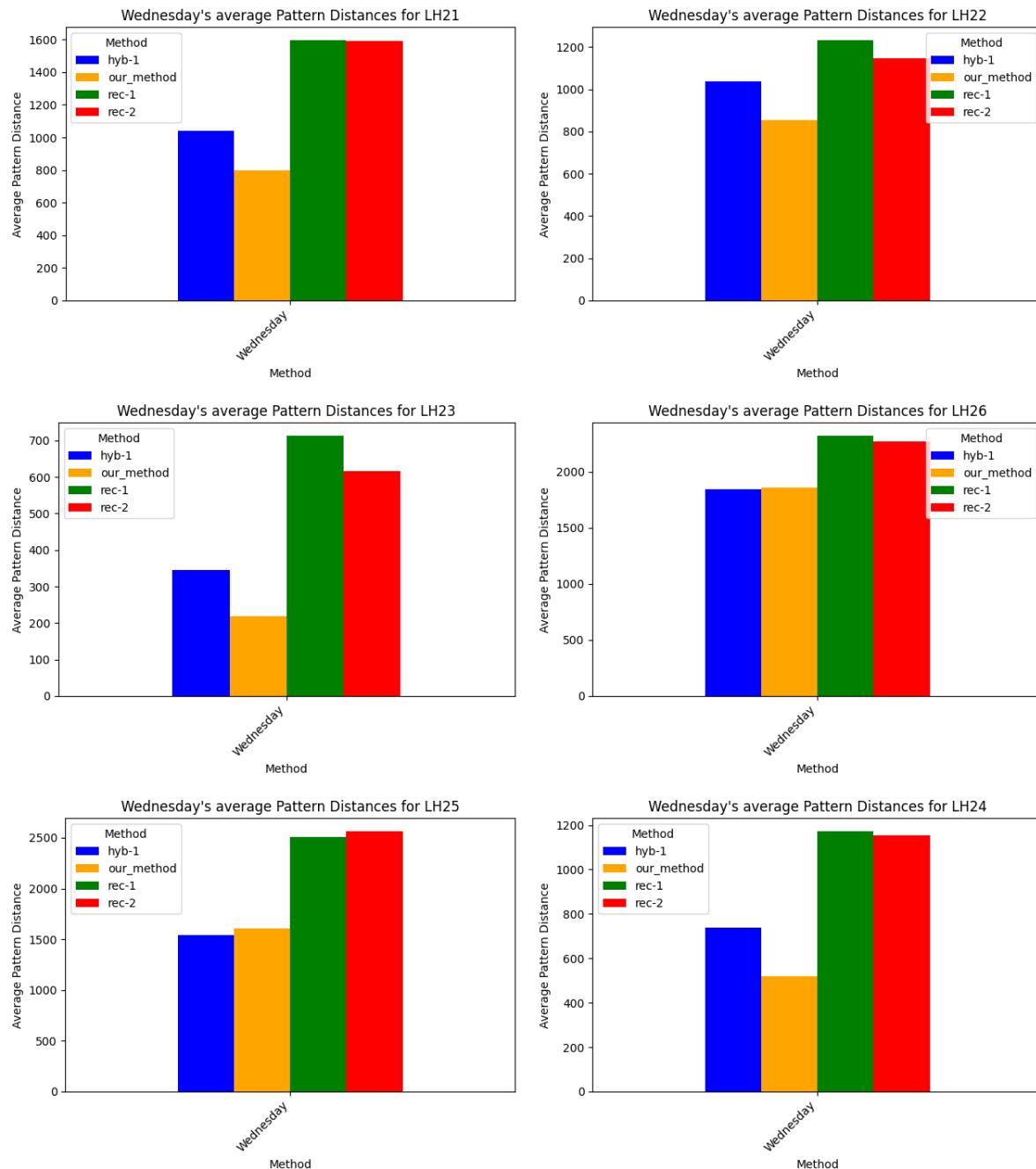


Figura 111: Playlist Pattern Distance media di ogni metodo durante il giorno Mercoledì per i file di cronologia da LH21 a LH26.

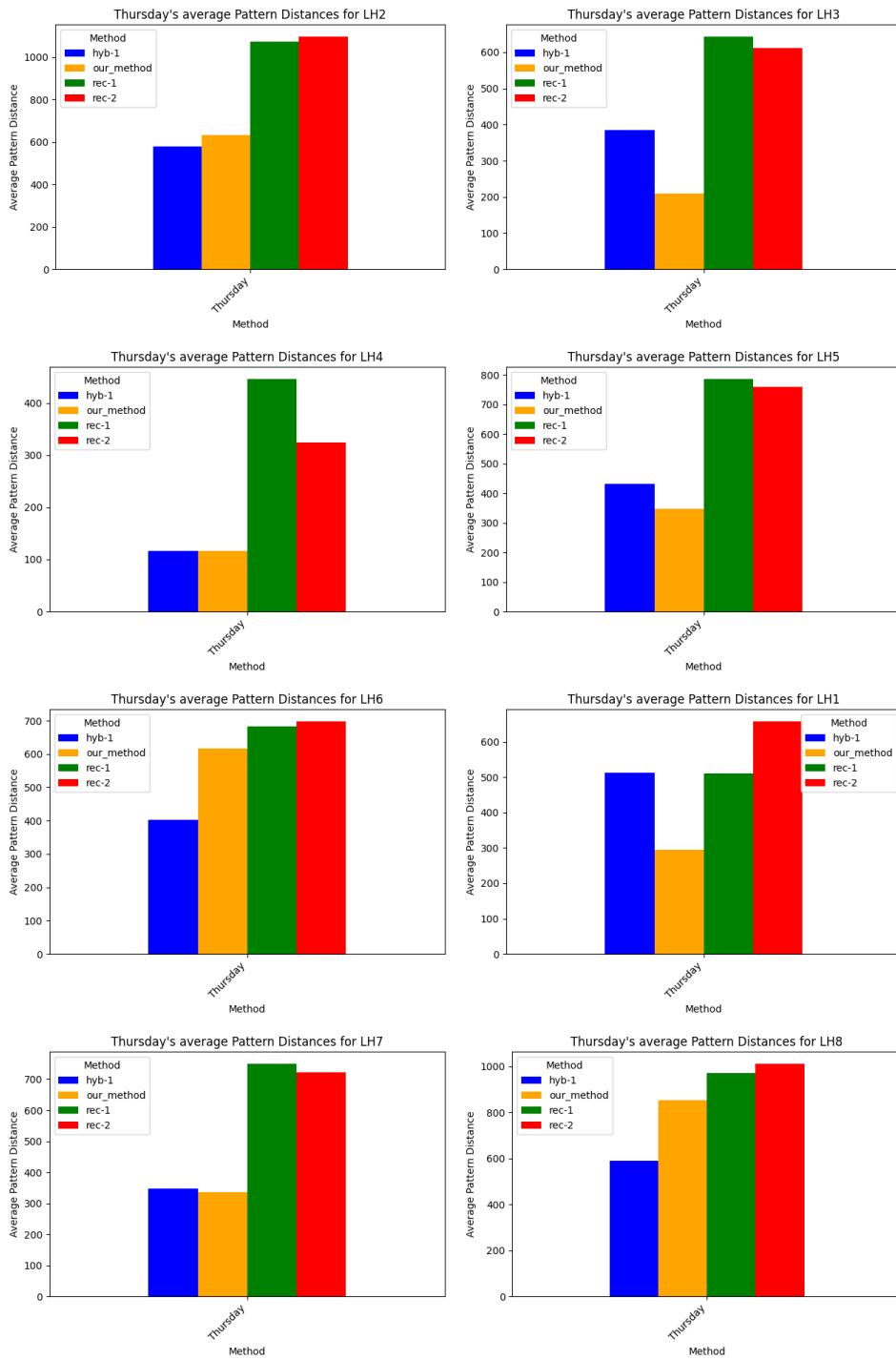


Figura 112: Playlist Pattern Distance media di ogni metodo durante il giorno Giovedì per i file di cronologia da LH1 a LH8.

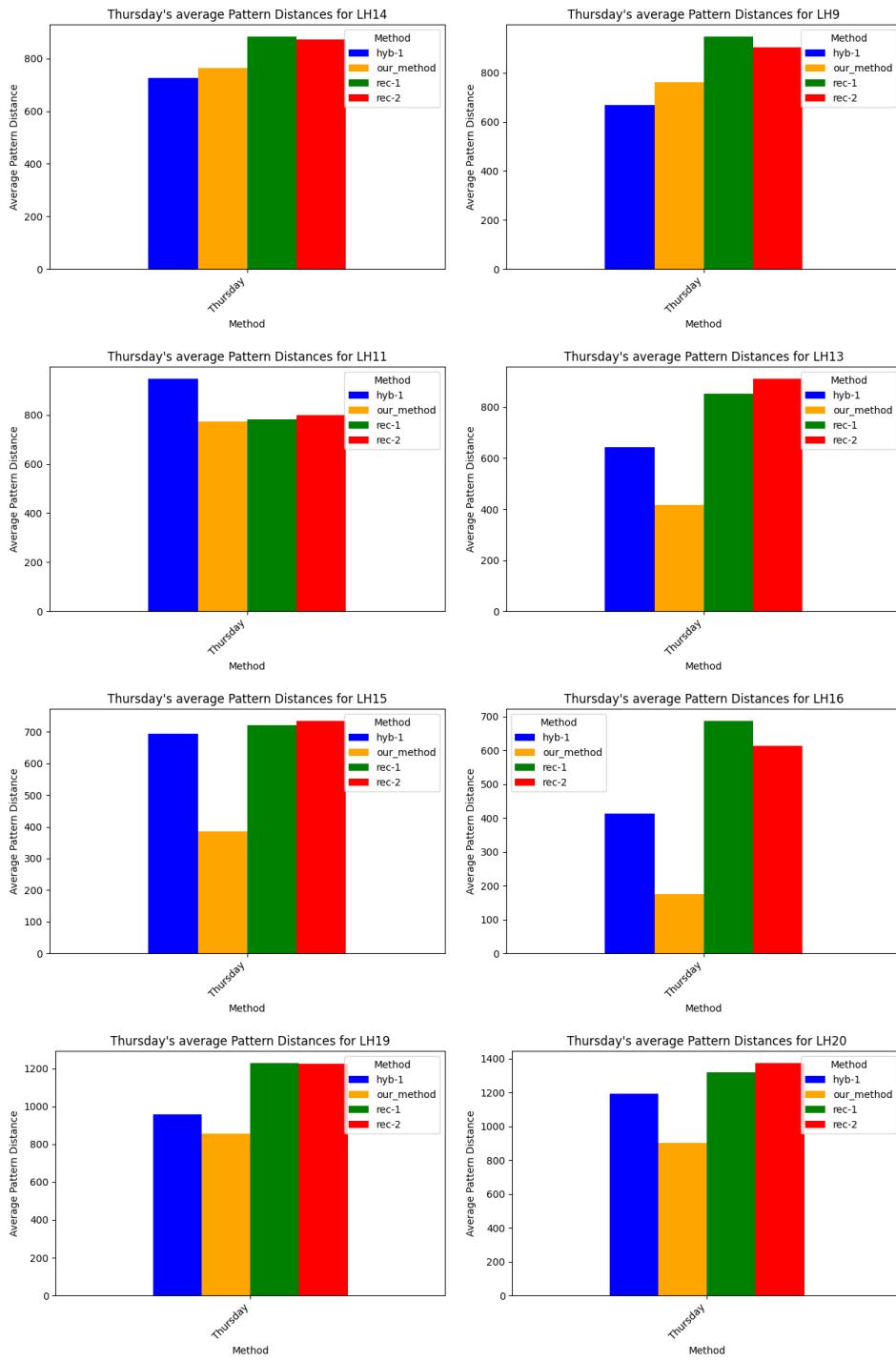


Figura 113: Playlist Pattern Distance media di ogni metodo durante il giorno Giovedì per i file di cronologia da LH9 a LH20.

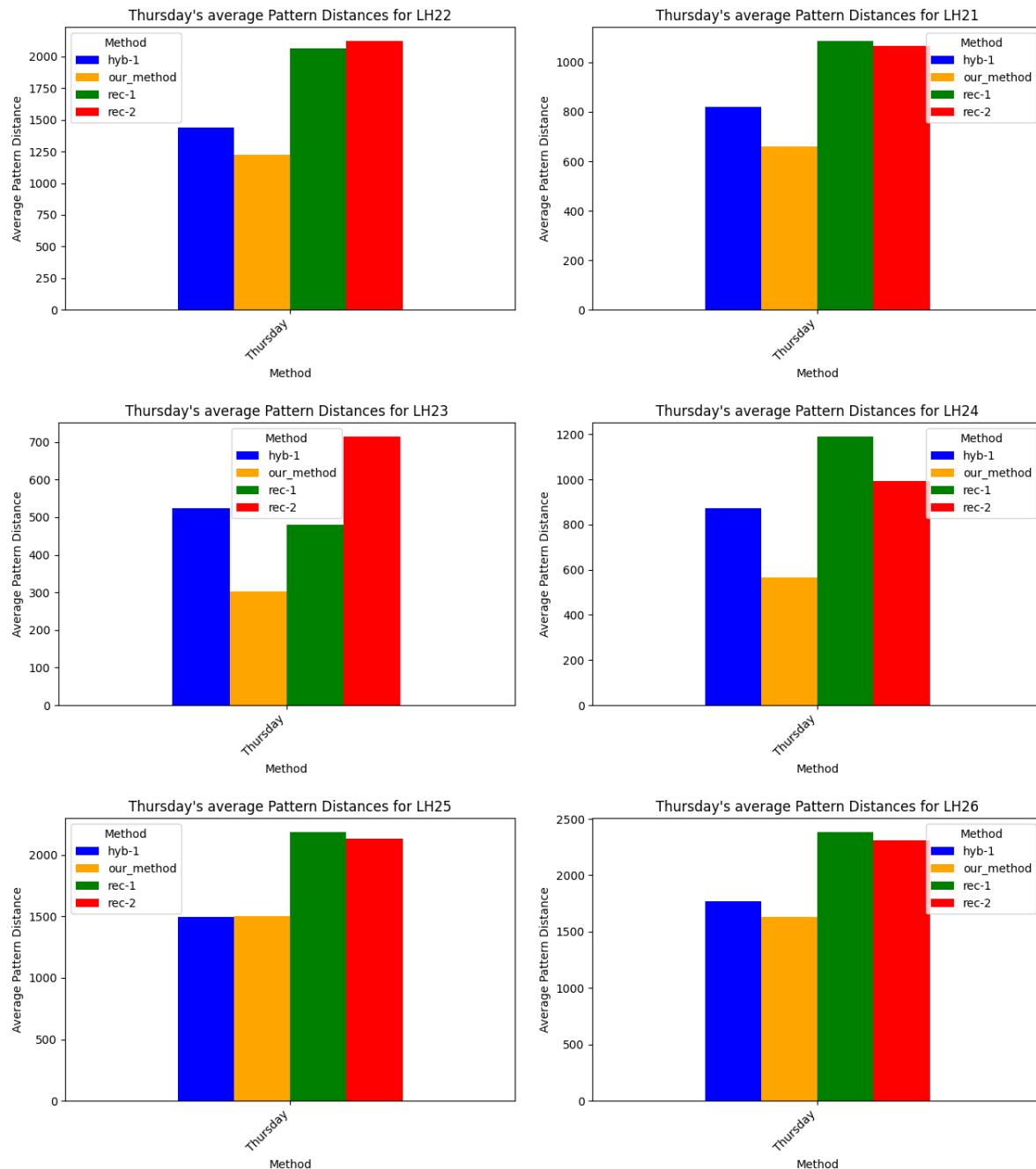


Figura 114: Playlist Pattern Distance media di ogni metodo durante il giorno Giovedì per i file di cronologia da LH21 a LH26.

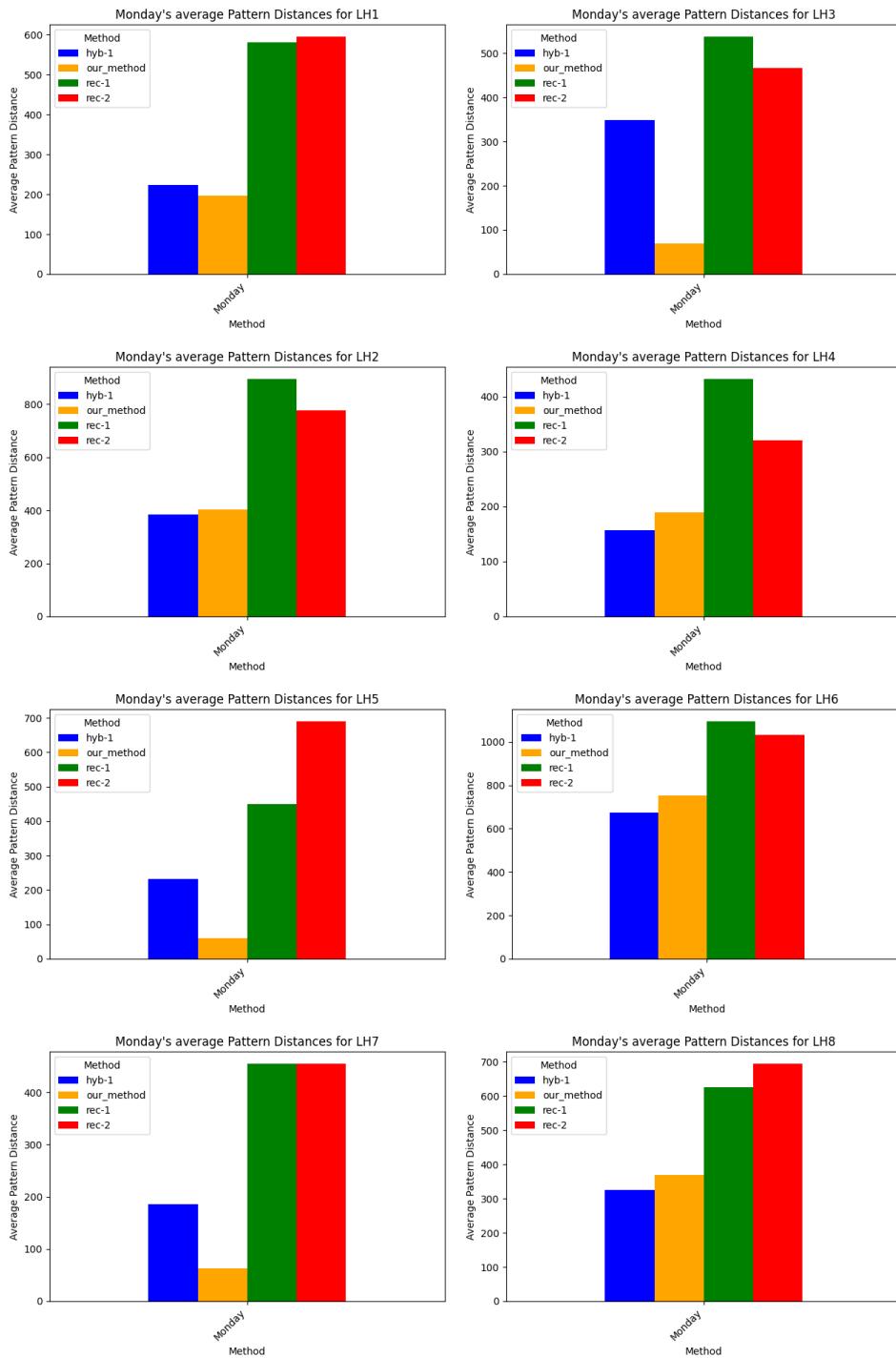


Figura 115: Playlist Pattern Distance media di ogni metodo durante il giorno Venerdì per i file di cronologia da LH1 a LH8.

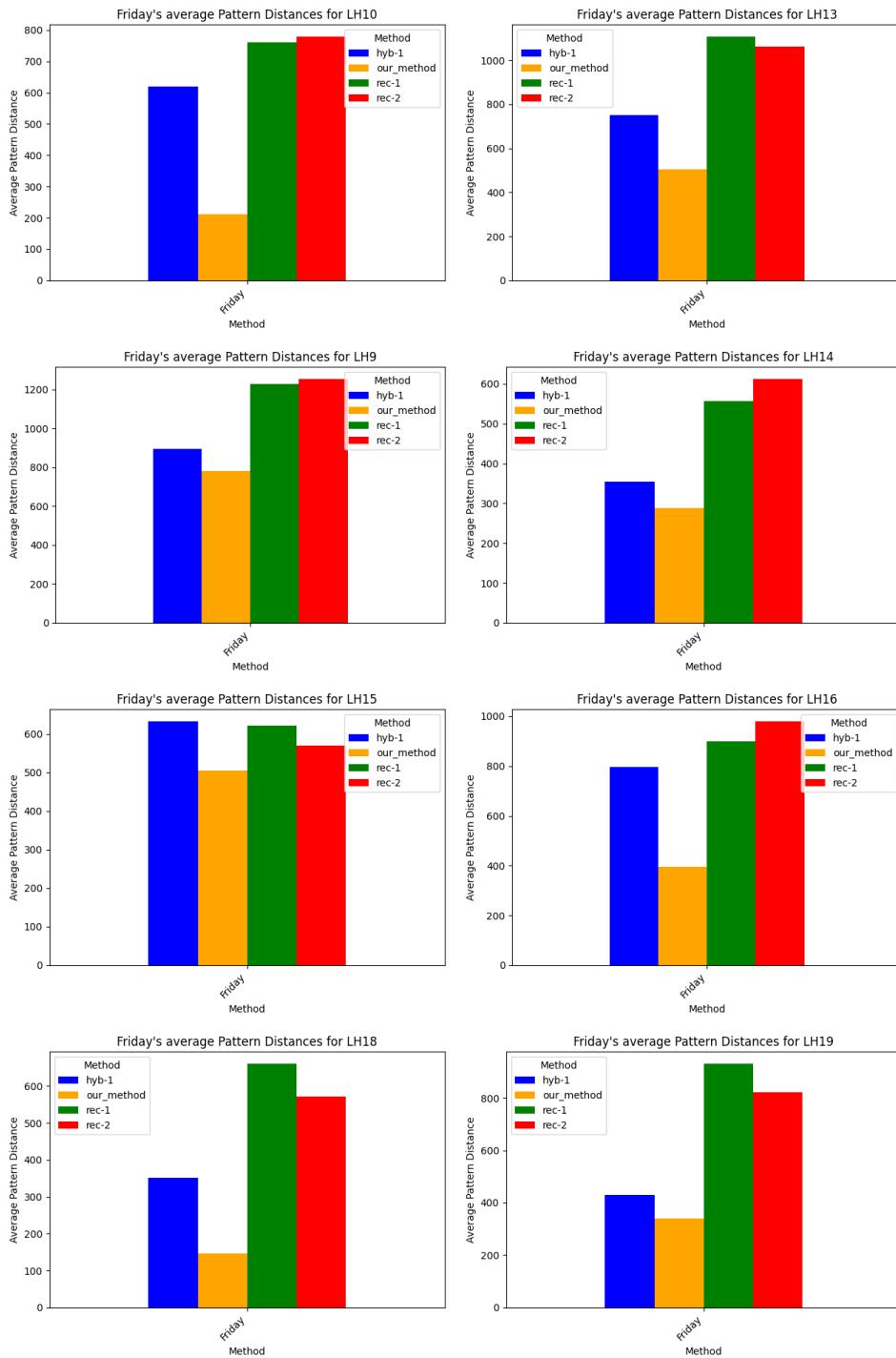


Figura 116: Playlist Pattern Distance media di ogni metodo durante il giorno Venerdì per i file di cronologia da LH9 a LH19.

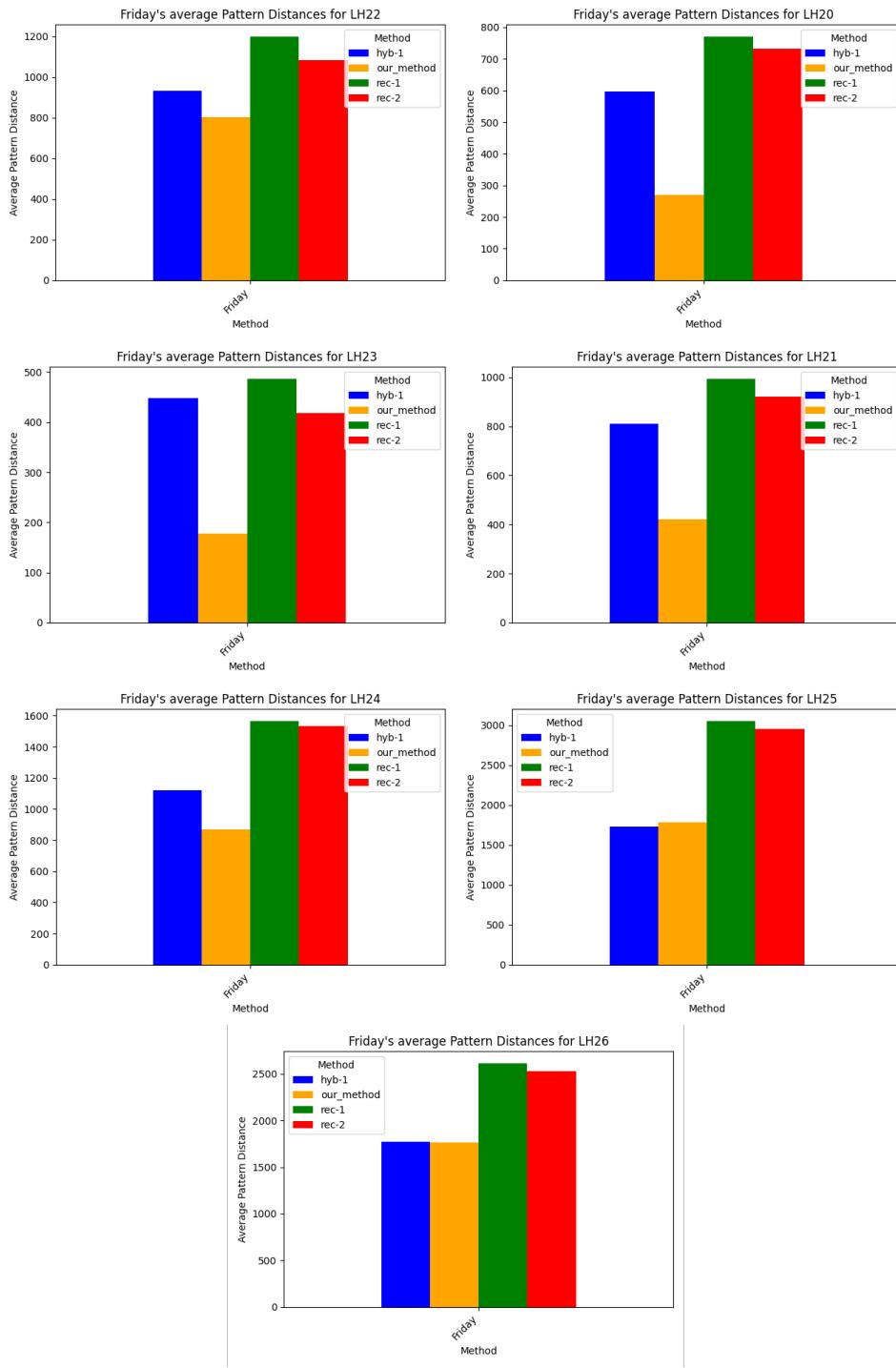


Figura 117: Playlist Pattern Distance media di ogni metodo durante il giorno Venerdì per i file di cronologia da LH21 a LH26.

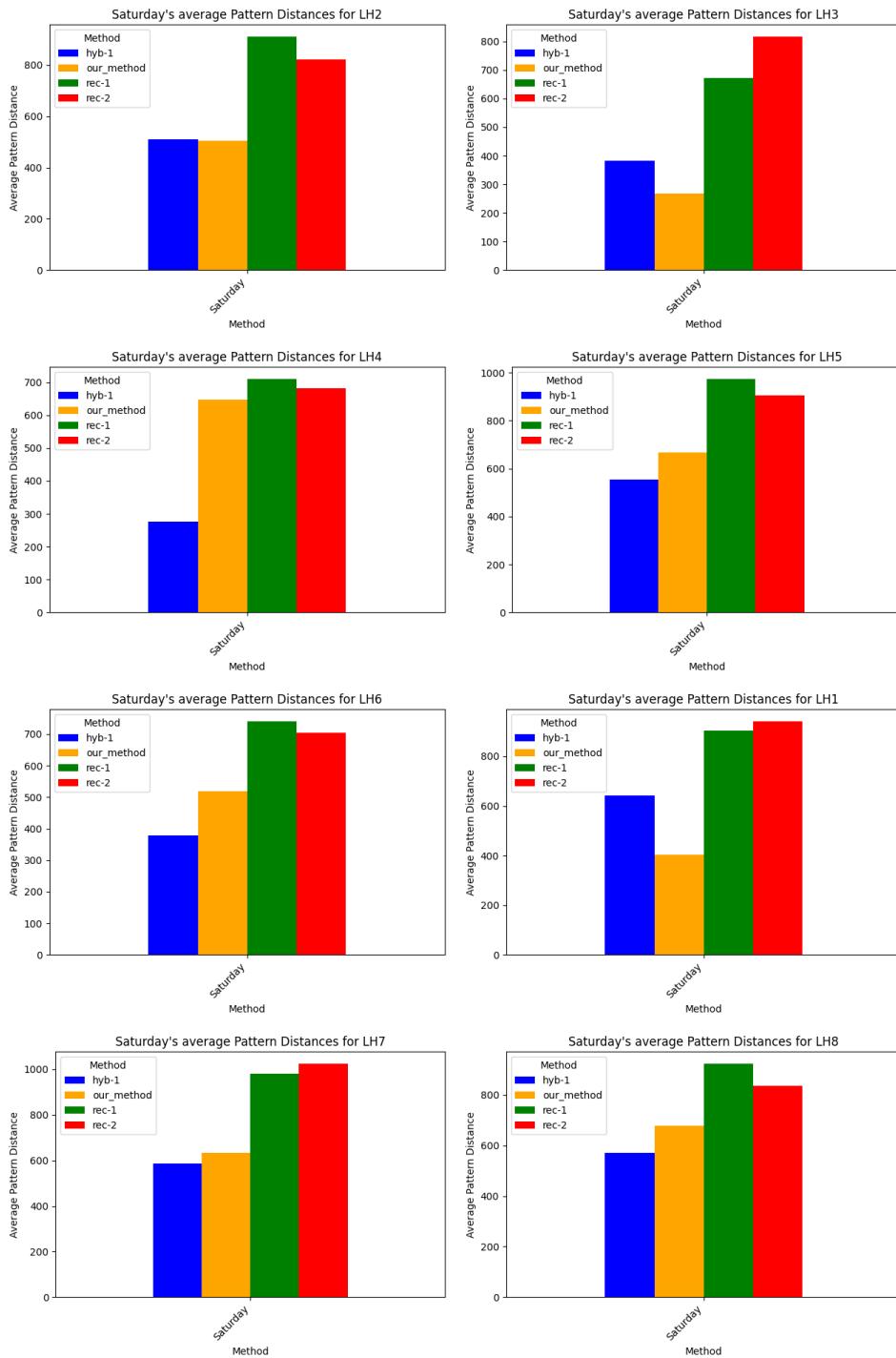


Figura 118: Playlist Pattern Distance media di ogni metodo durante il giorno Sabato per i file di cronologia da LH1 a LH8.

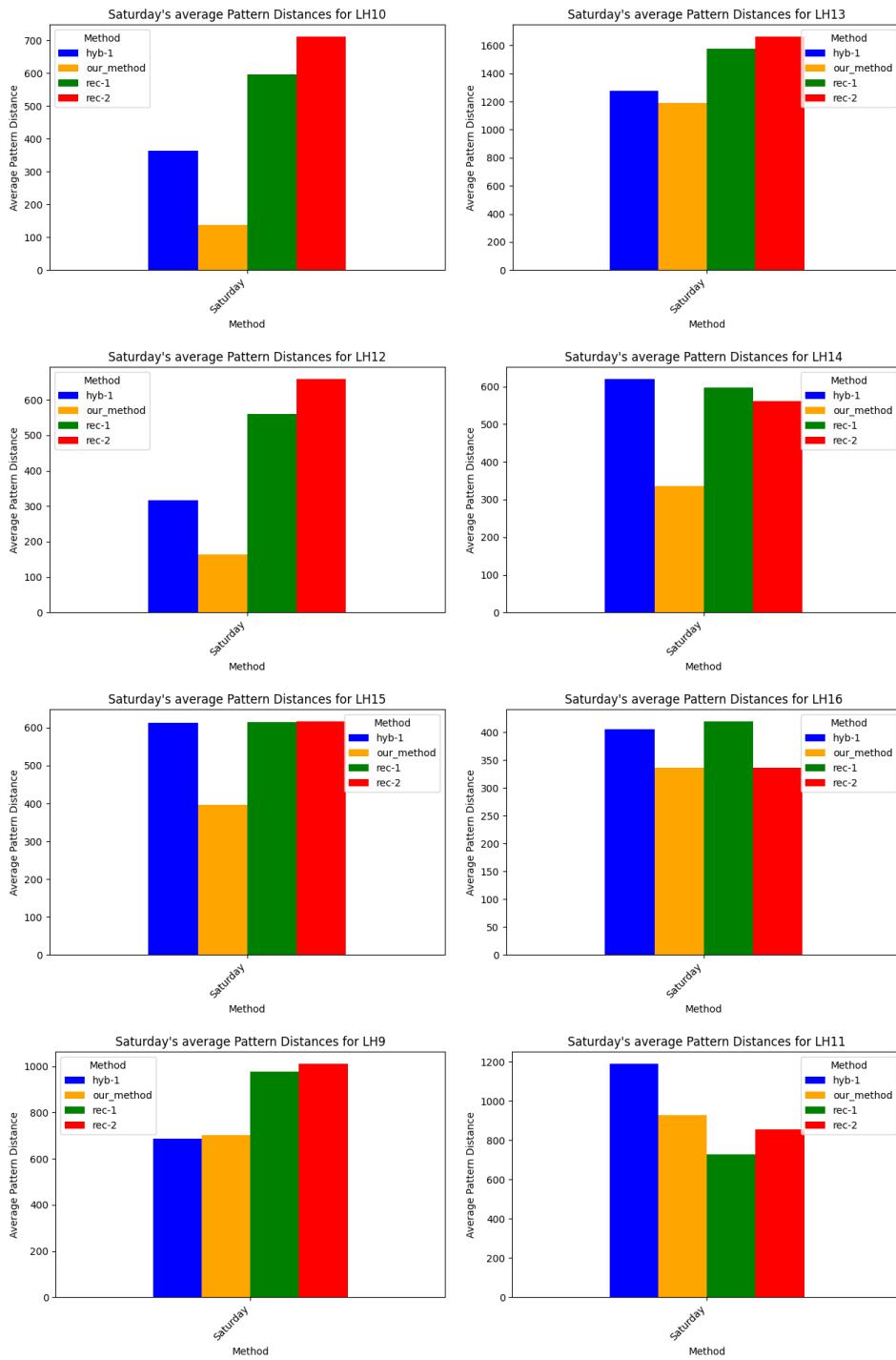


Figura 119: Playlist Pattern Distance media di ogni metodo durante il giorno Sabato per i file di cronologia da LH9 a LH16.

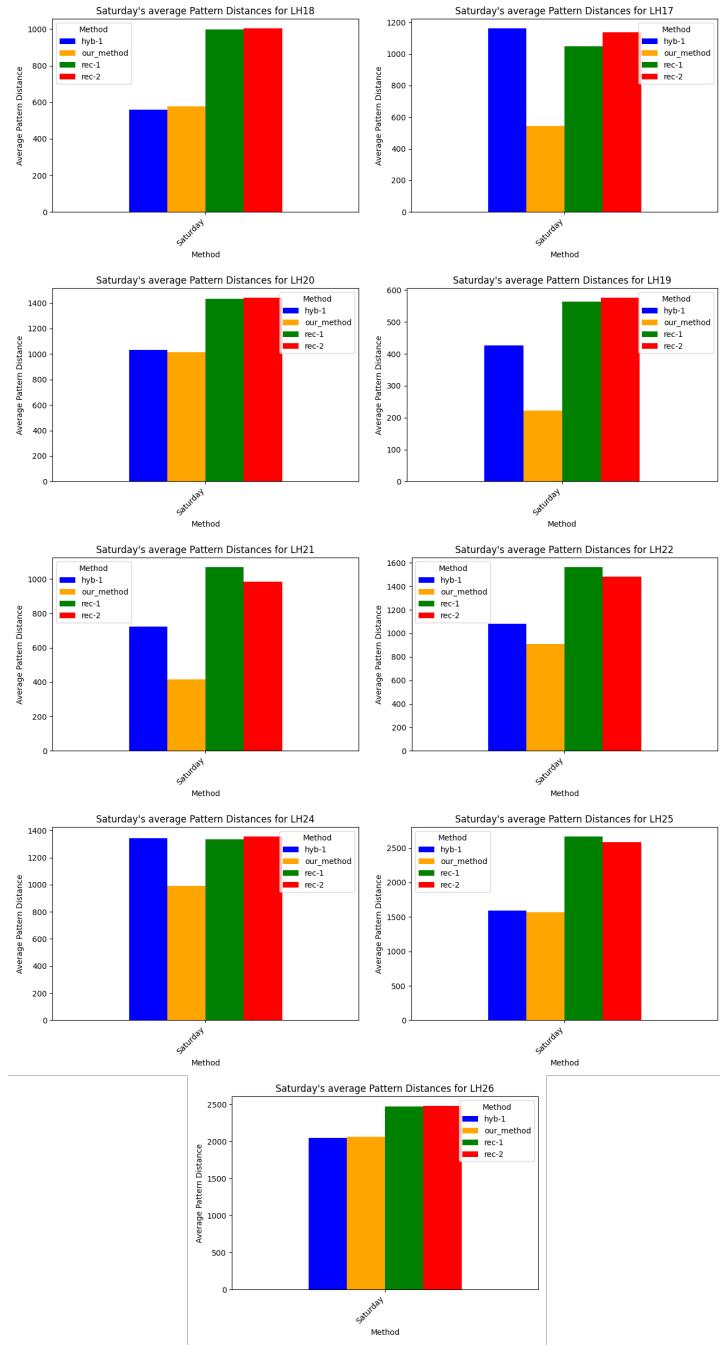


Figura 120: Playlist Pattern Distance media di ogni metodo durante il giorno Sabato per i file di cronologia da LH17 a LH26.

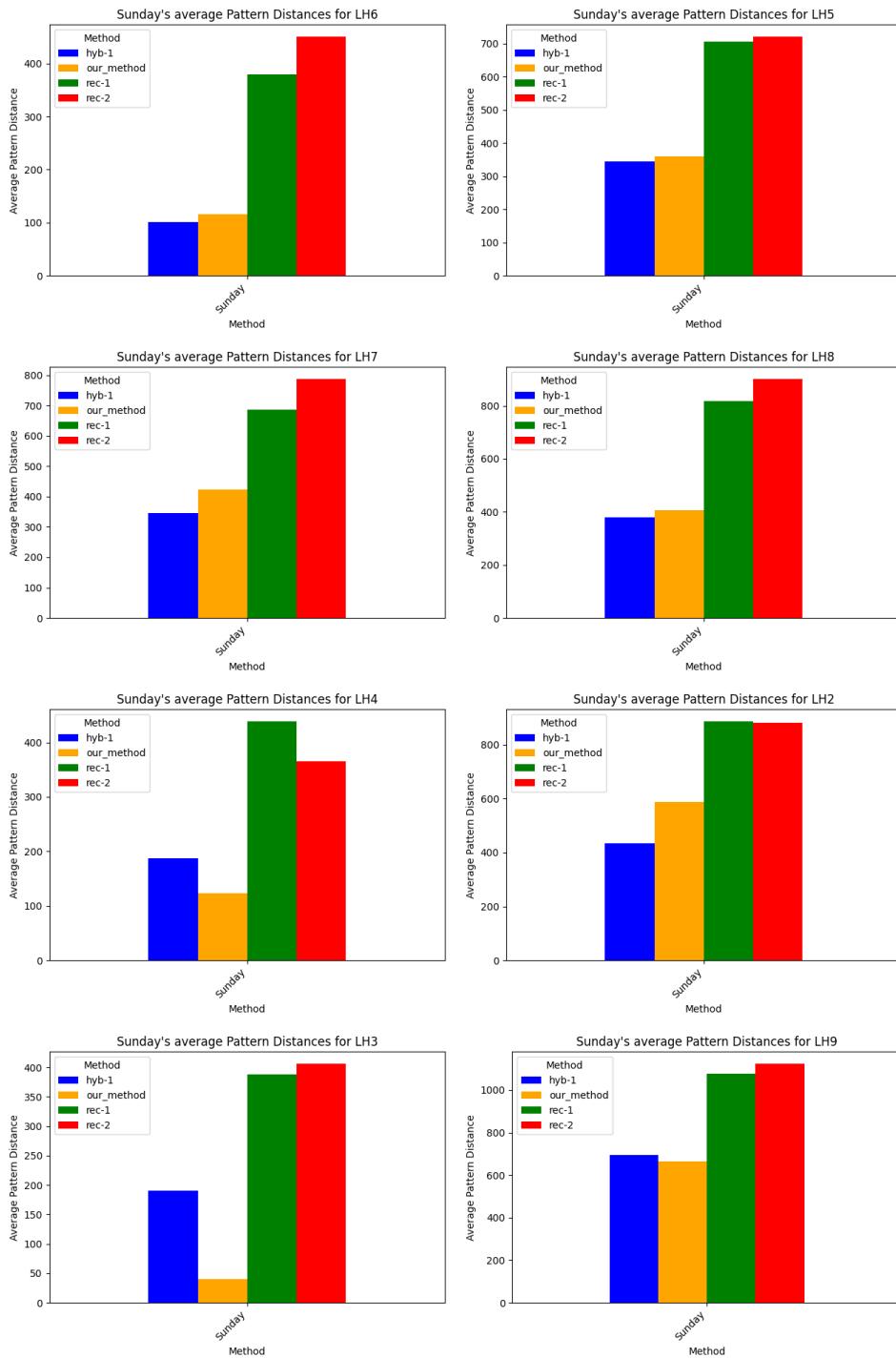


Figura 121: Playlist Pattern Distance media di ogni metodo durante il giorno Domenica per i file di cronologia da LH2 a LH9.

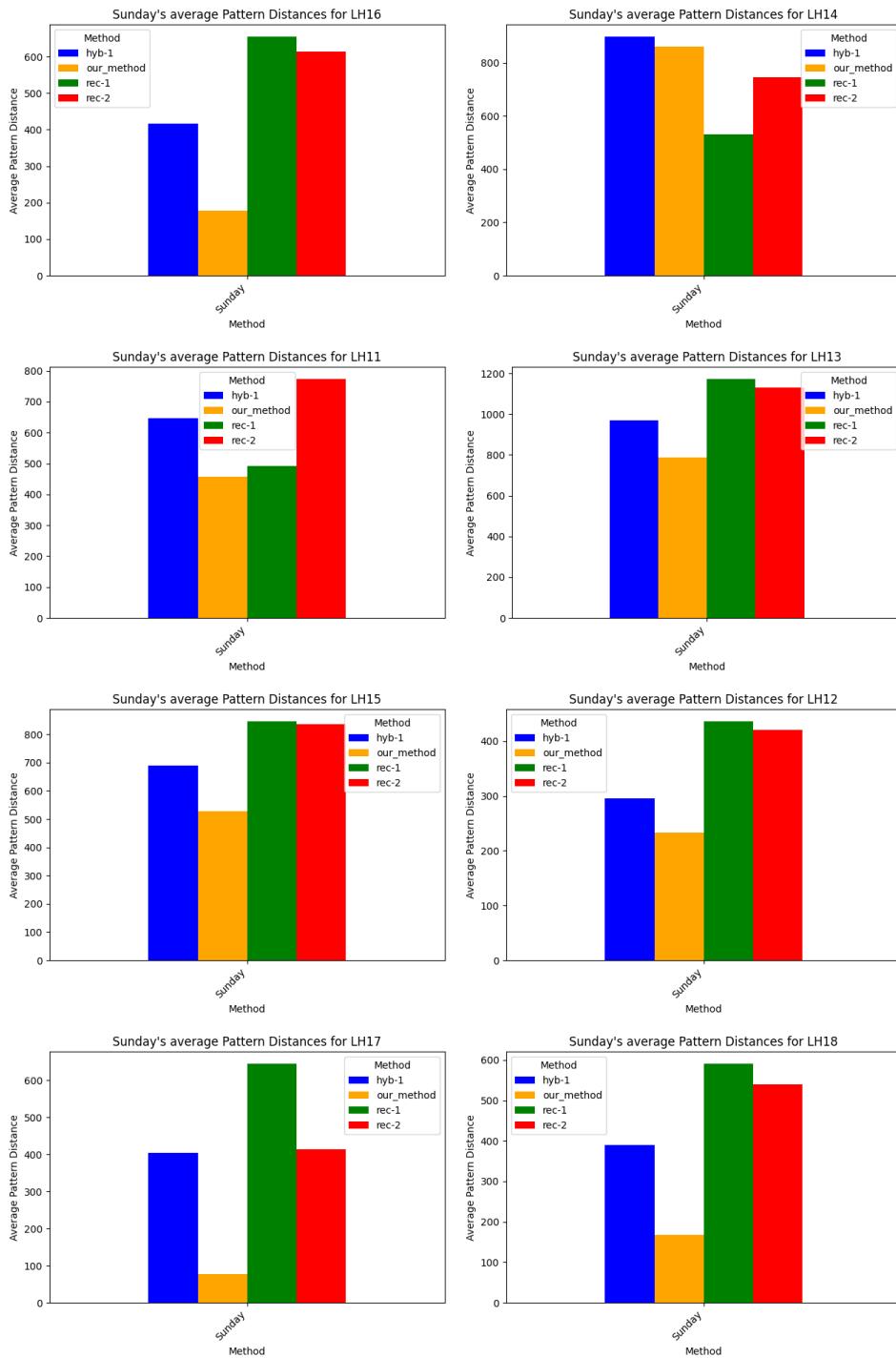


Figura 122: Playlist Pattern Distance media di ogni metodo durante il giorno Domenica per i file di cronologia da LH11 a LH18.

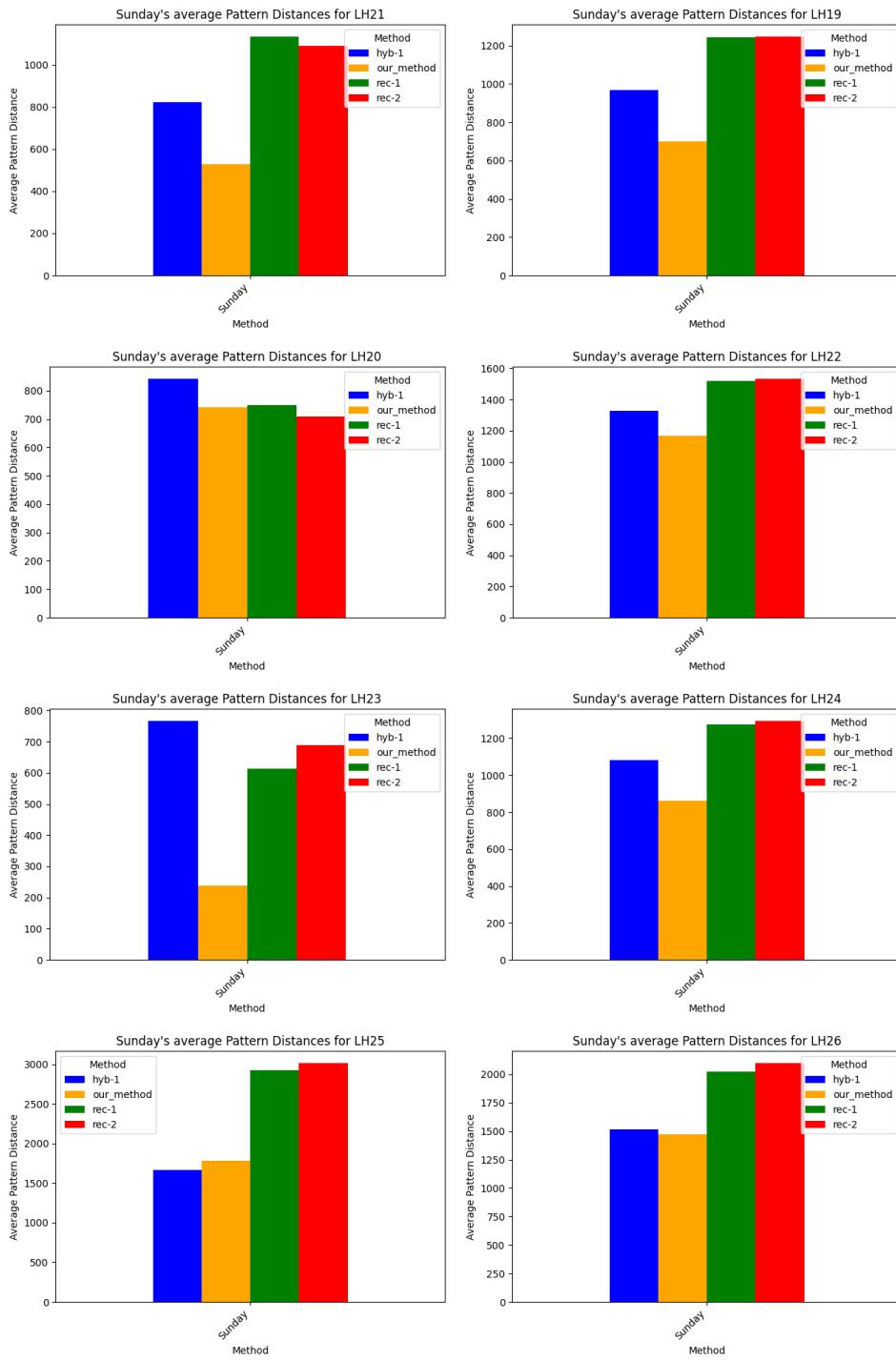


Figura 123: Playlist Pattern Distance media di ogni metodo durante il giorno Domenica per i file di cronologia da LH19 a LH26.

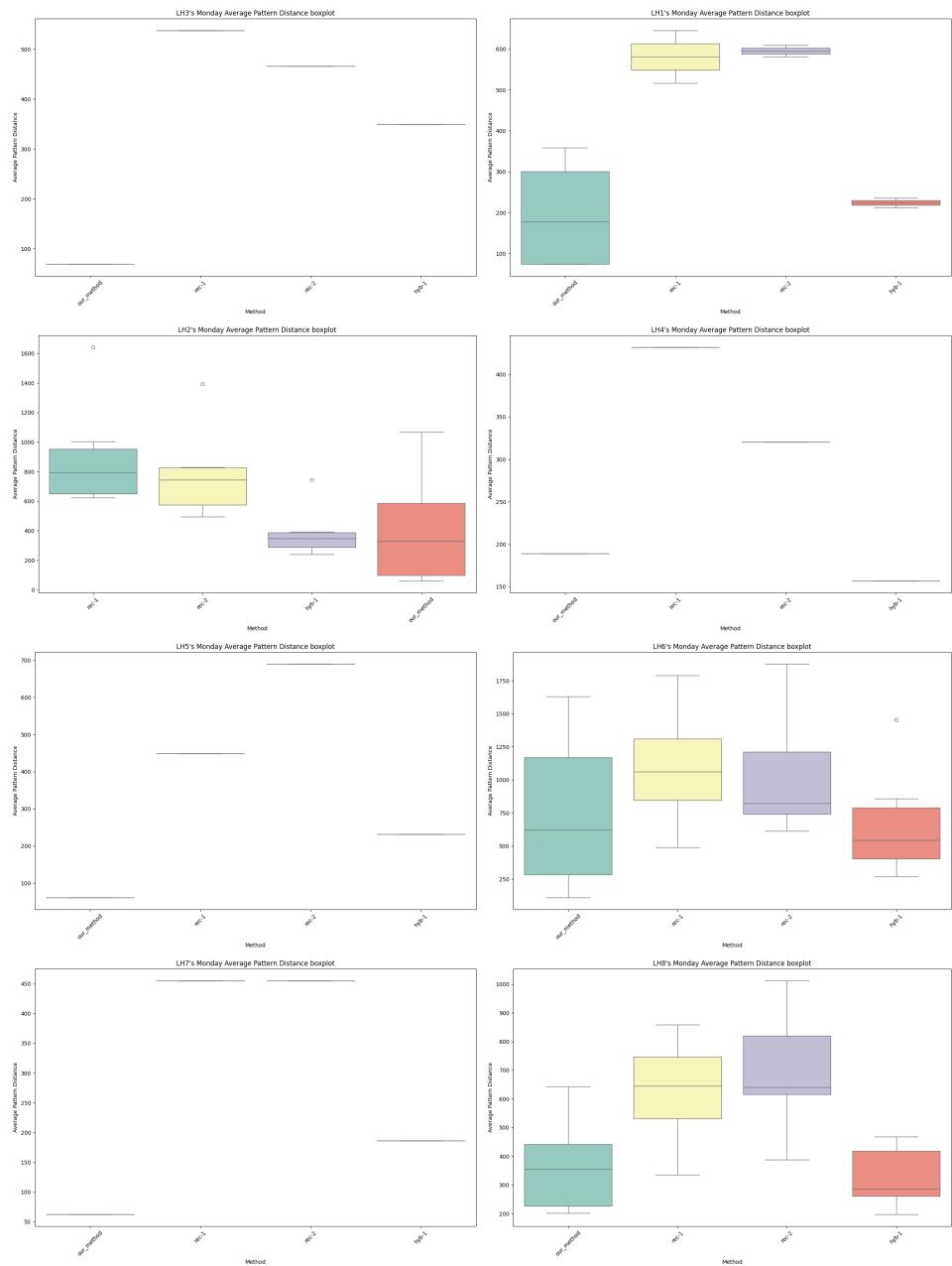


Figura 124: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Lunedì per i file di cronologia da LH1 a LH8.

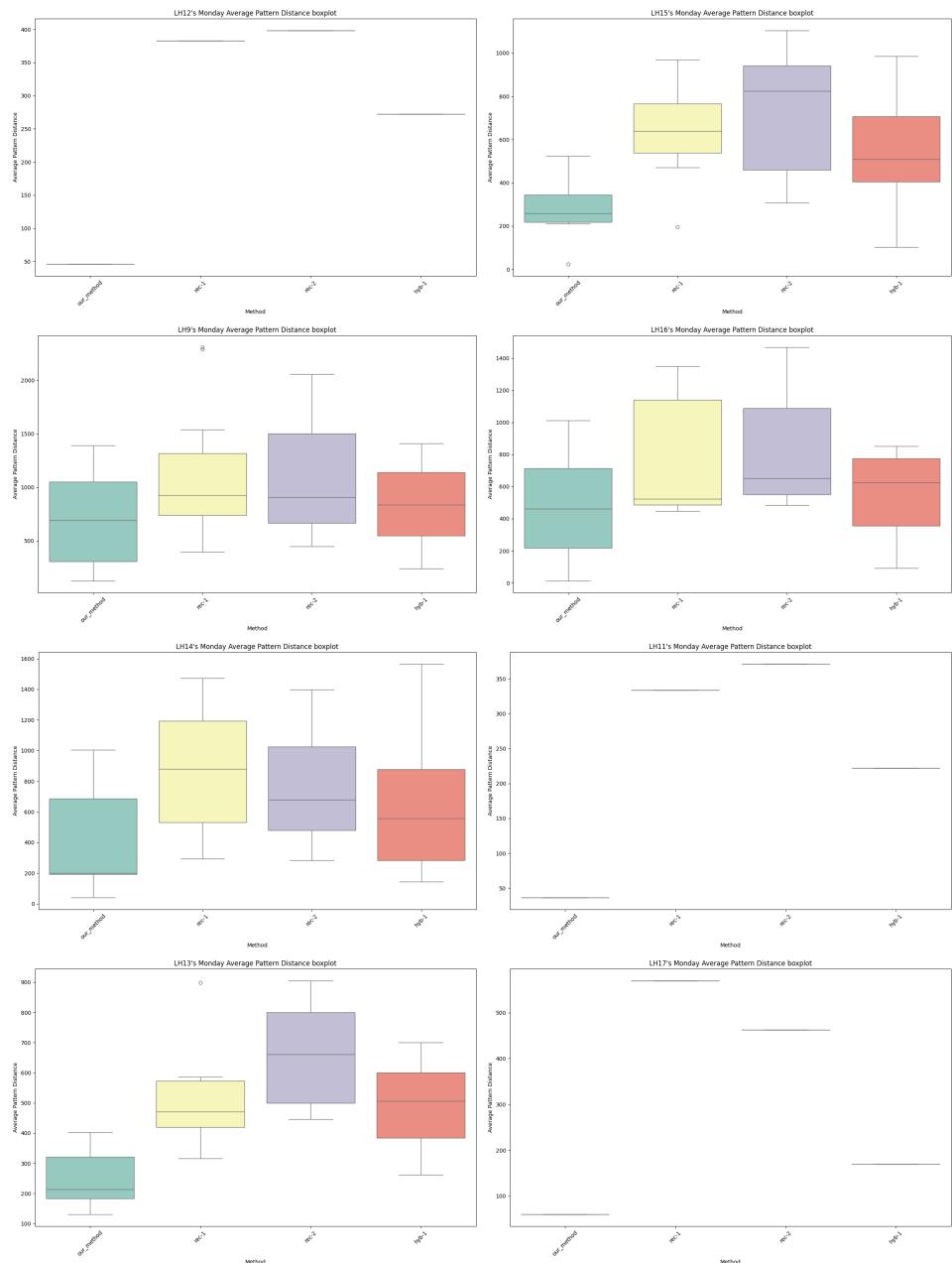


Figura 125: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Lunedì per i file di cronologia da LH9 a LH17.

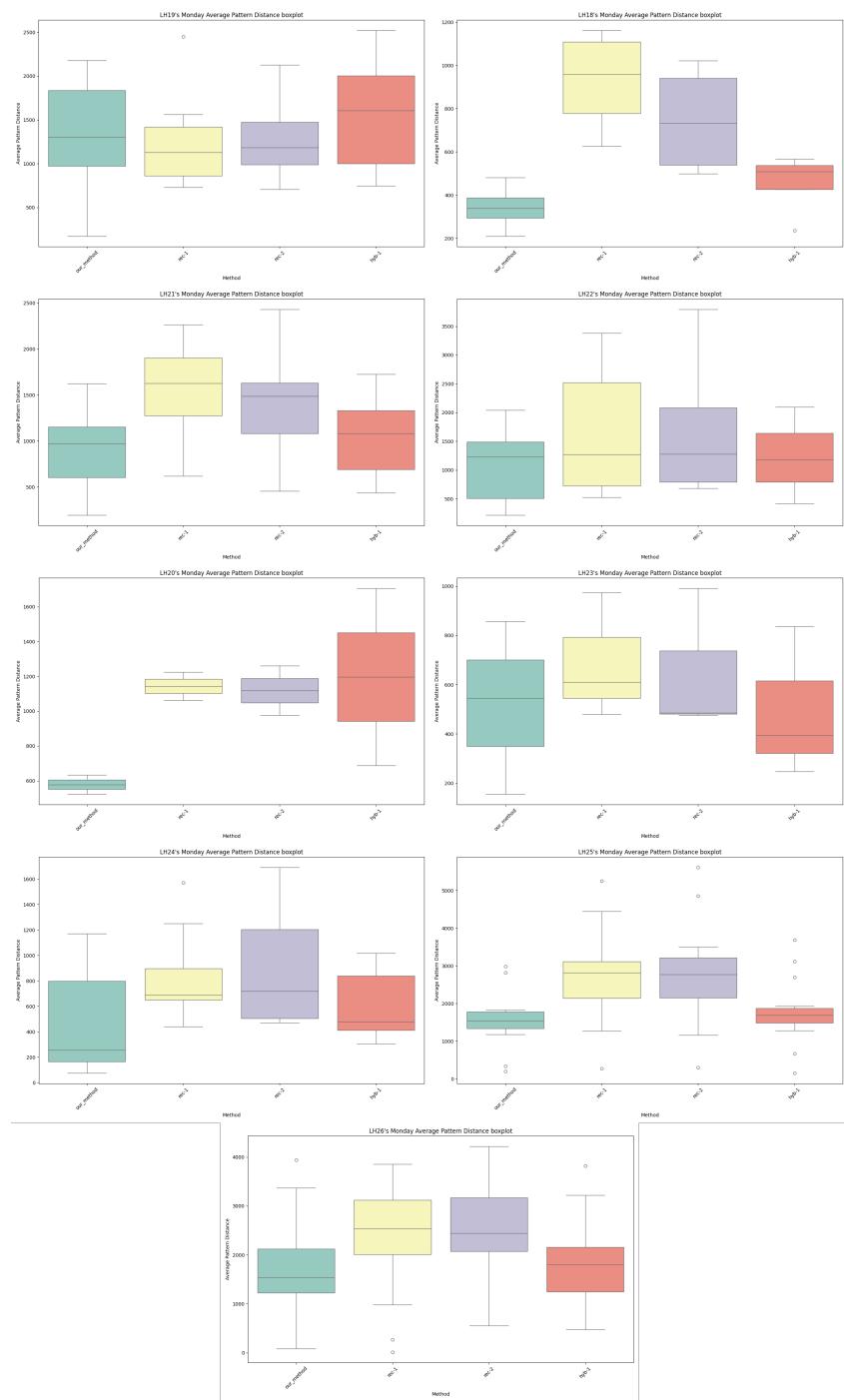


Figura 126: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Lunedì per i file di cronologia da LH18 a LH26.

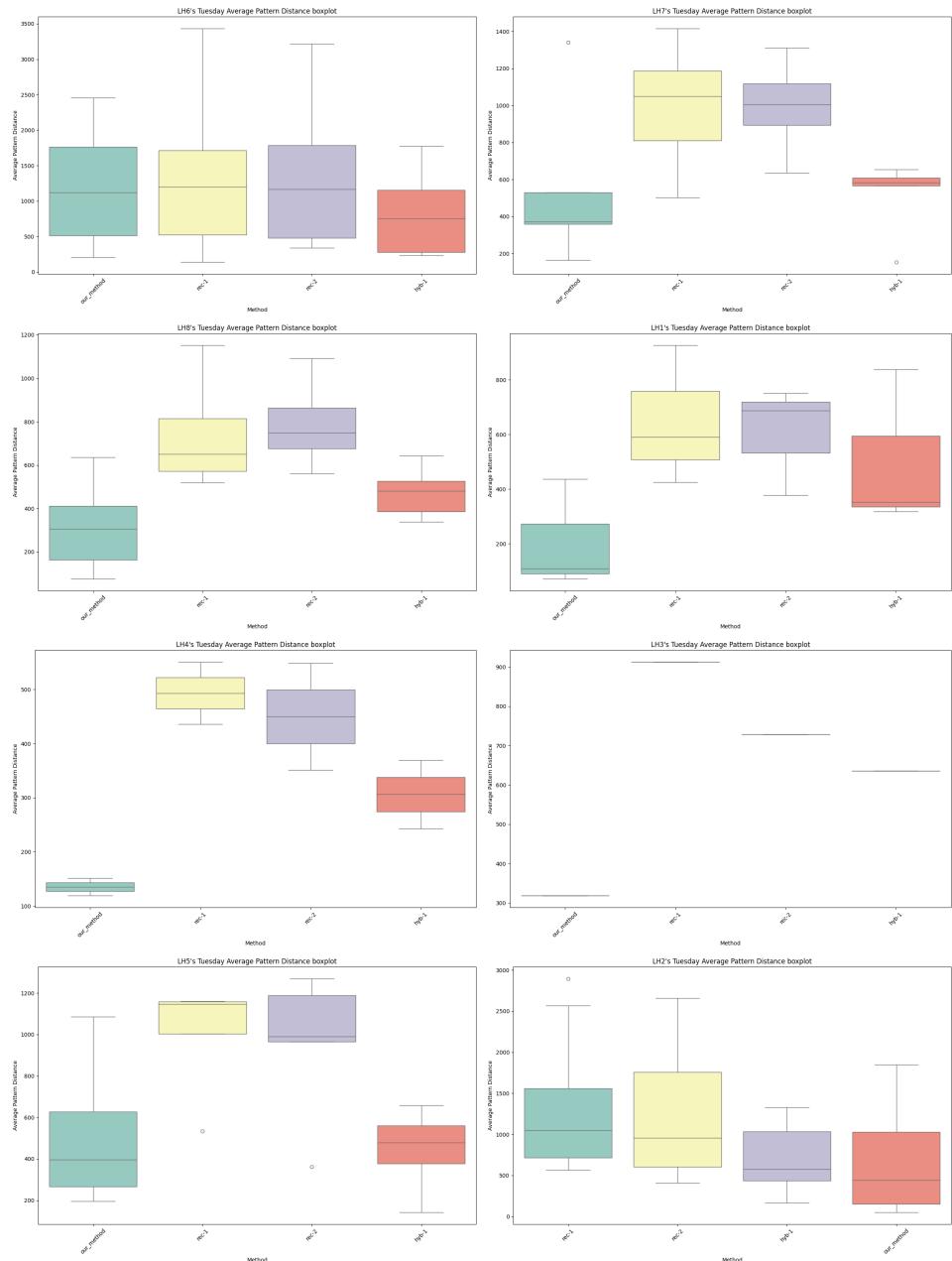


Figura 127: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Martedì per i file di cronologia da LH1 a LH8.

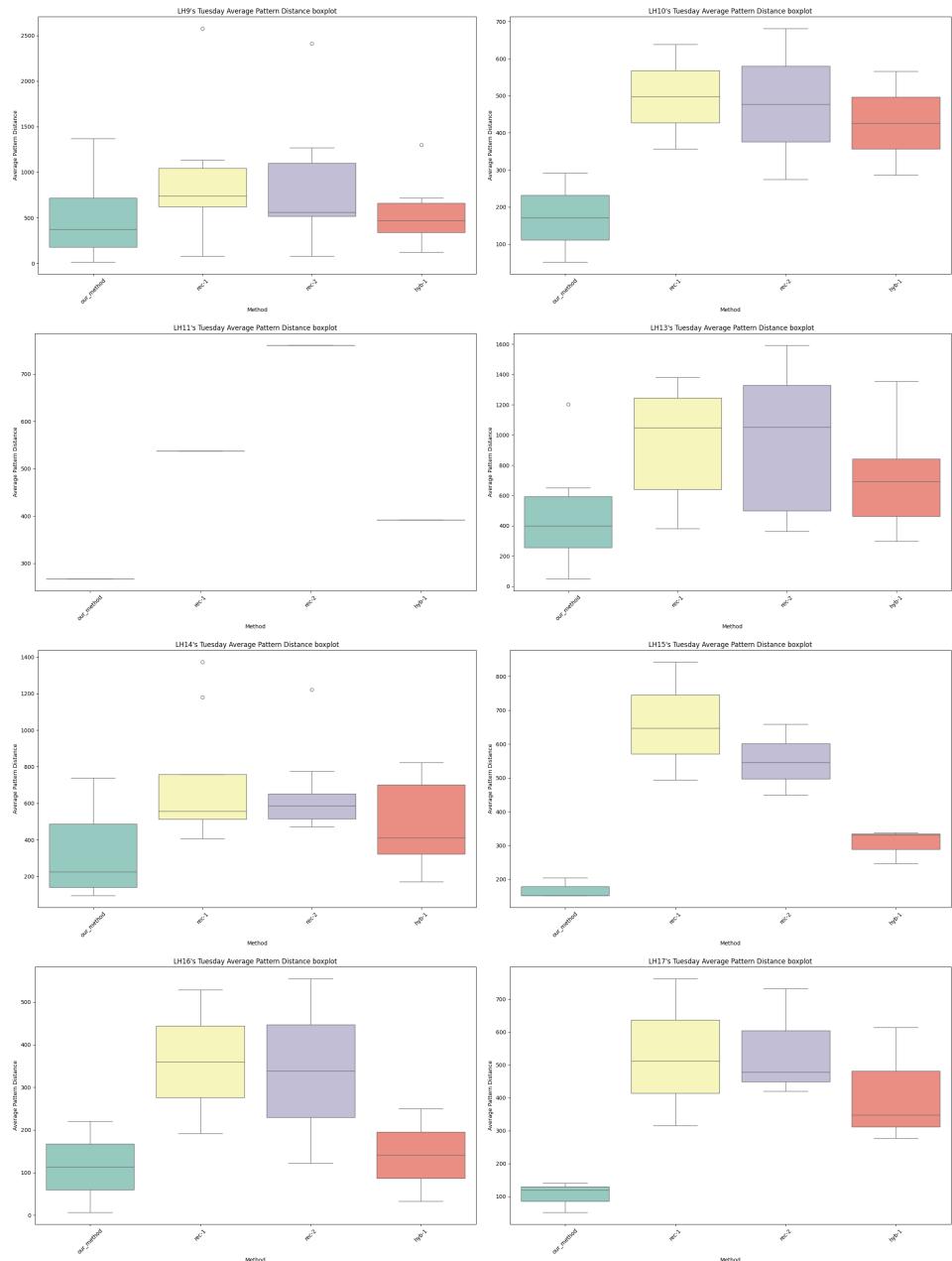


Figura 128: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Martedì per i file di cronologia da LH9 a LH17.

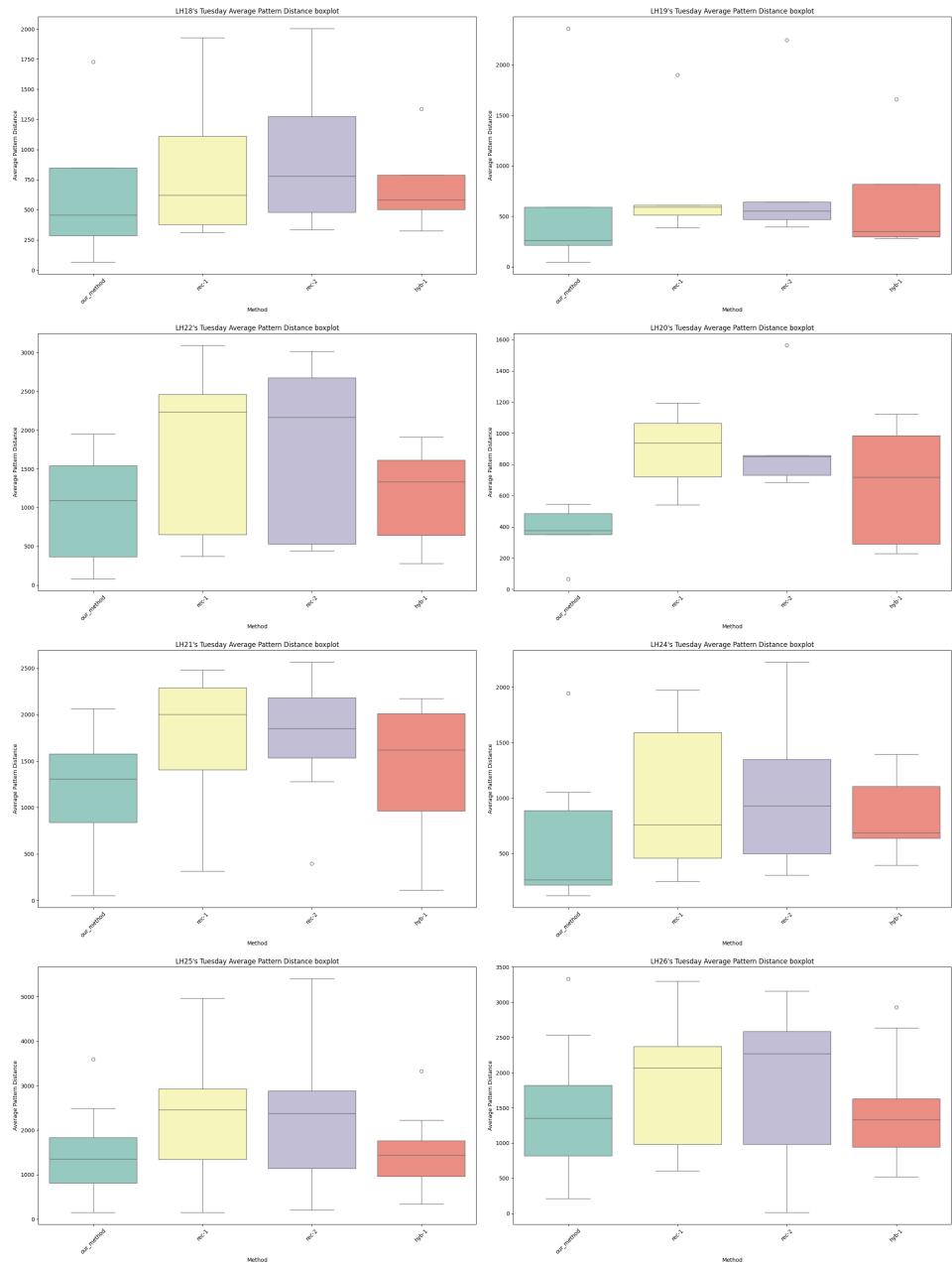


Figura 129: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Martedì per i file di cronologia da LH18 a LH26.

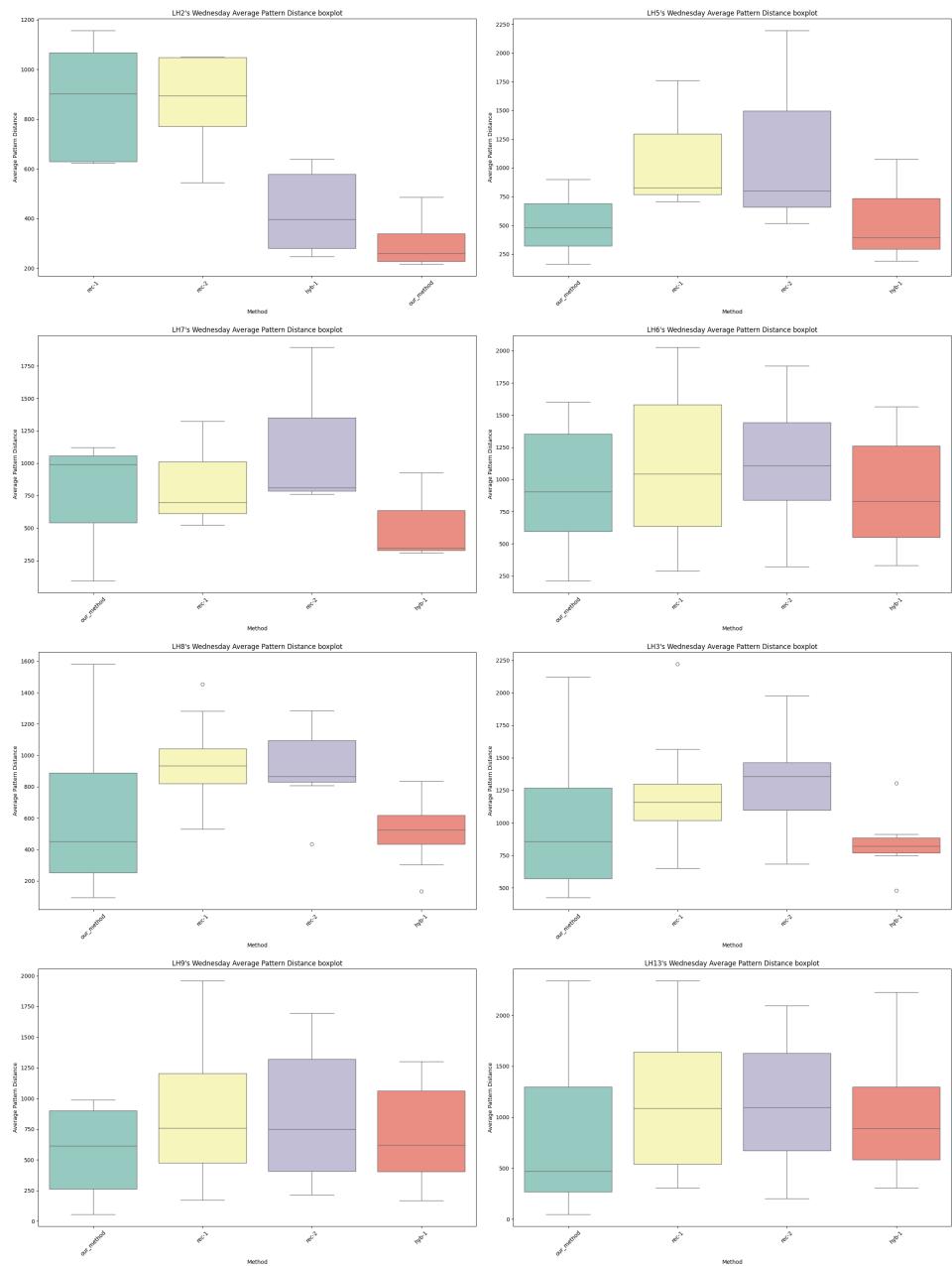


Figura 130: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Mercoledì per i file di cronologia da LH2 a LH13.

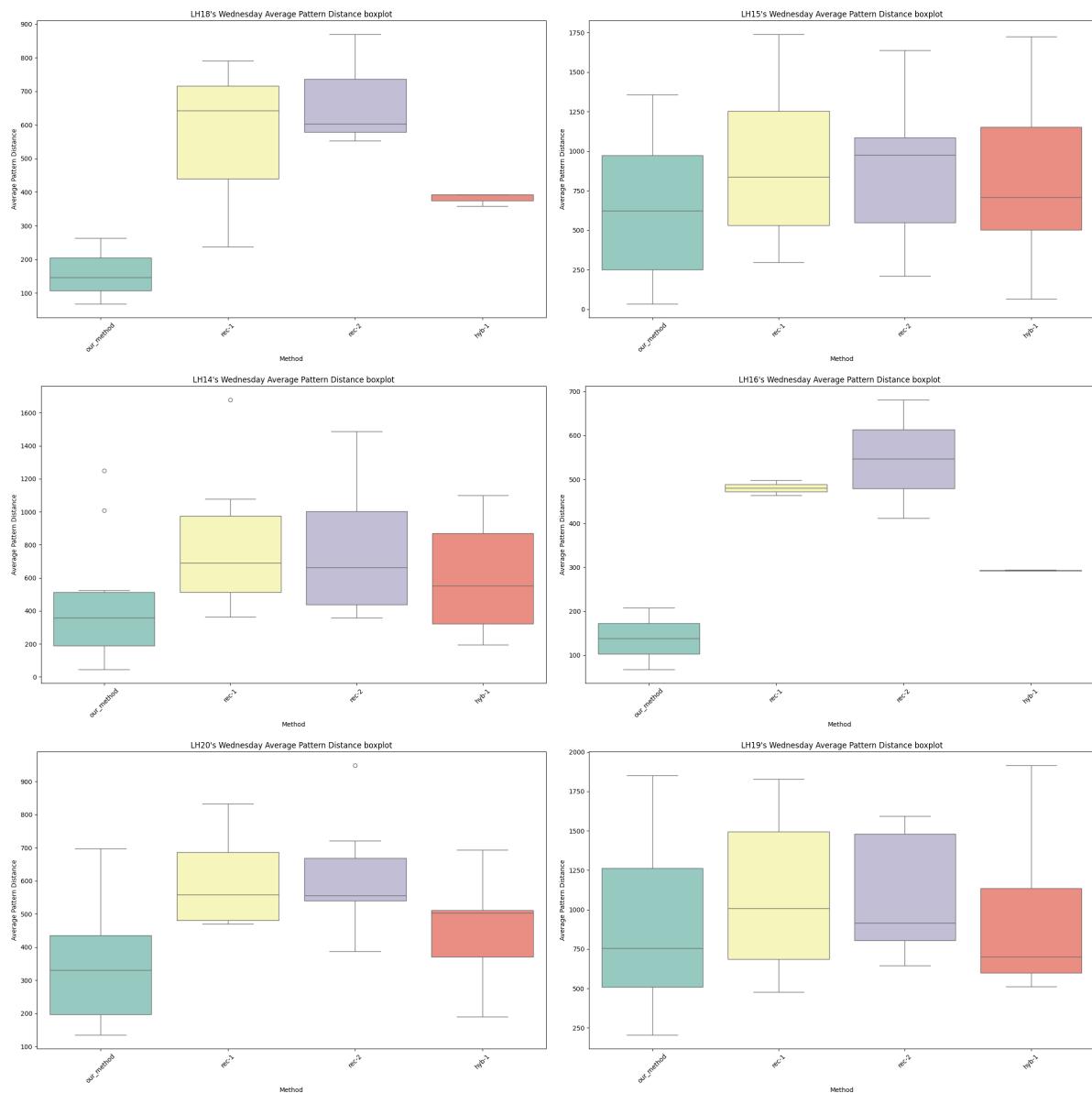


Figura 131: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Mercoledì per i file di cronologia da LH14 a LH20.

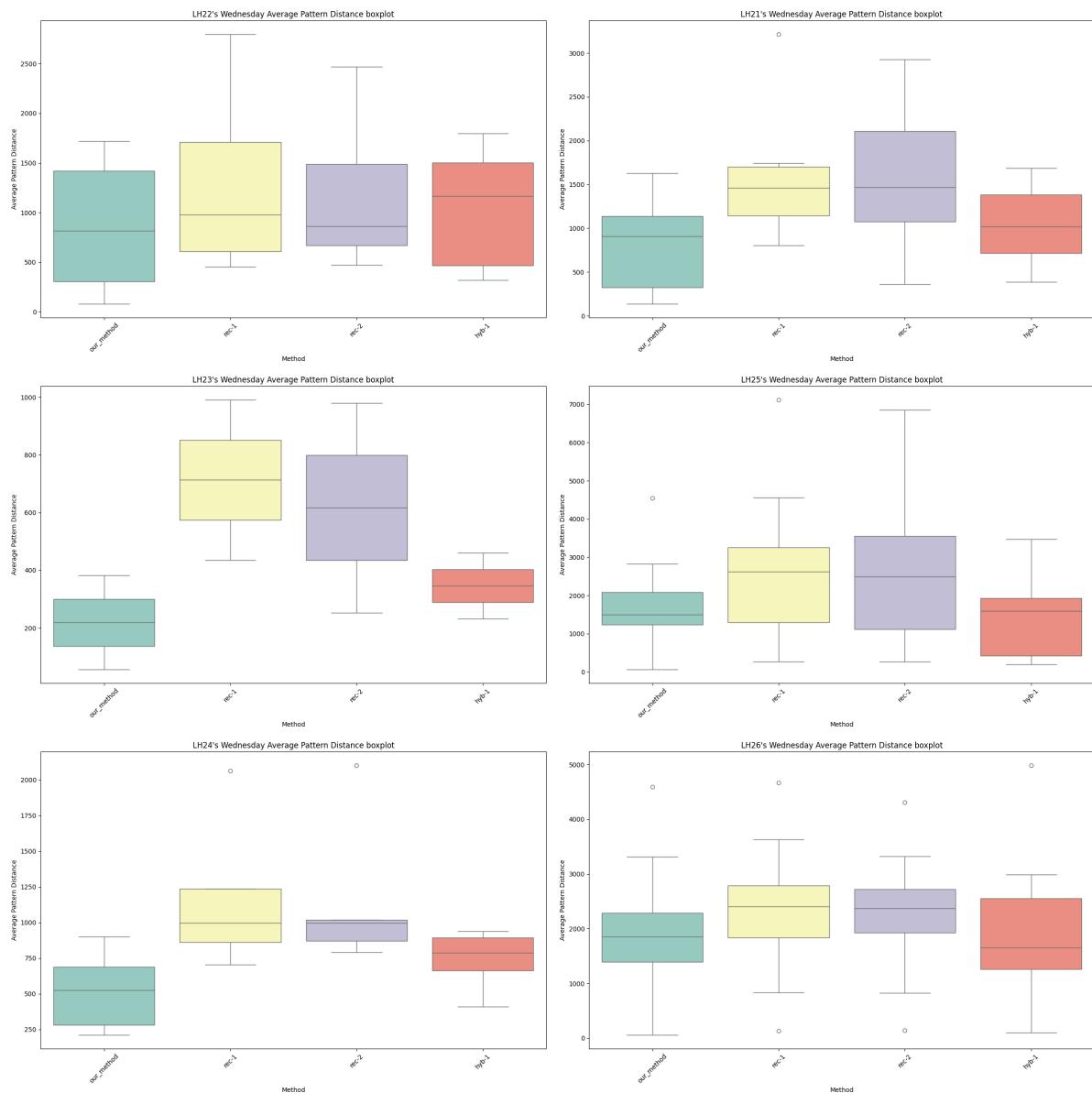


Figura 132: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Mercoledì per i file di cronologia da LH21 a LH26.

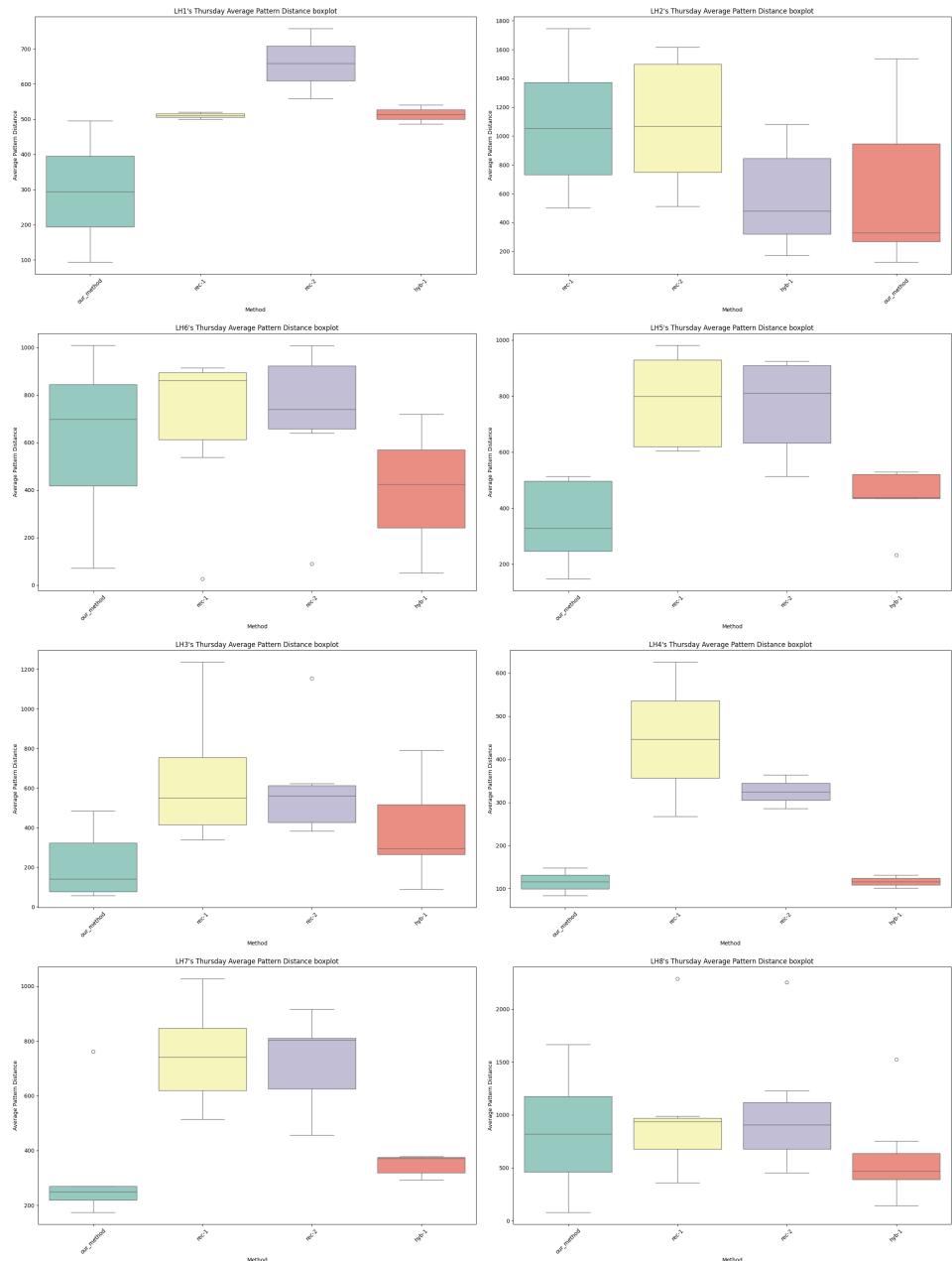


Figura 133: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Giovedì per i file di cronologia da LH1 a LH8.

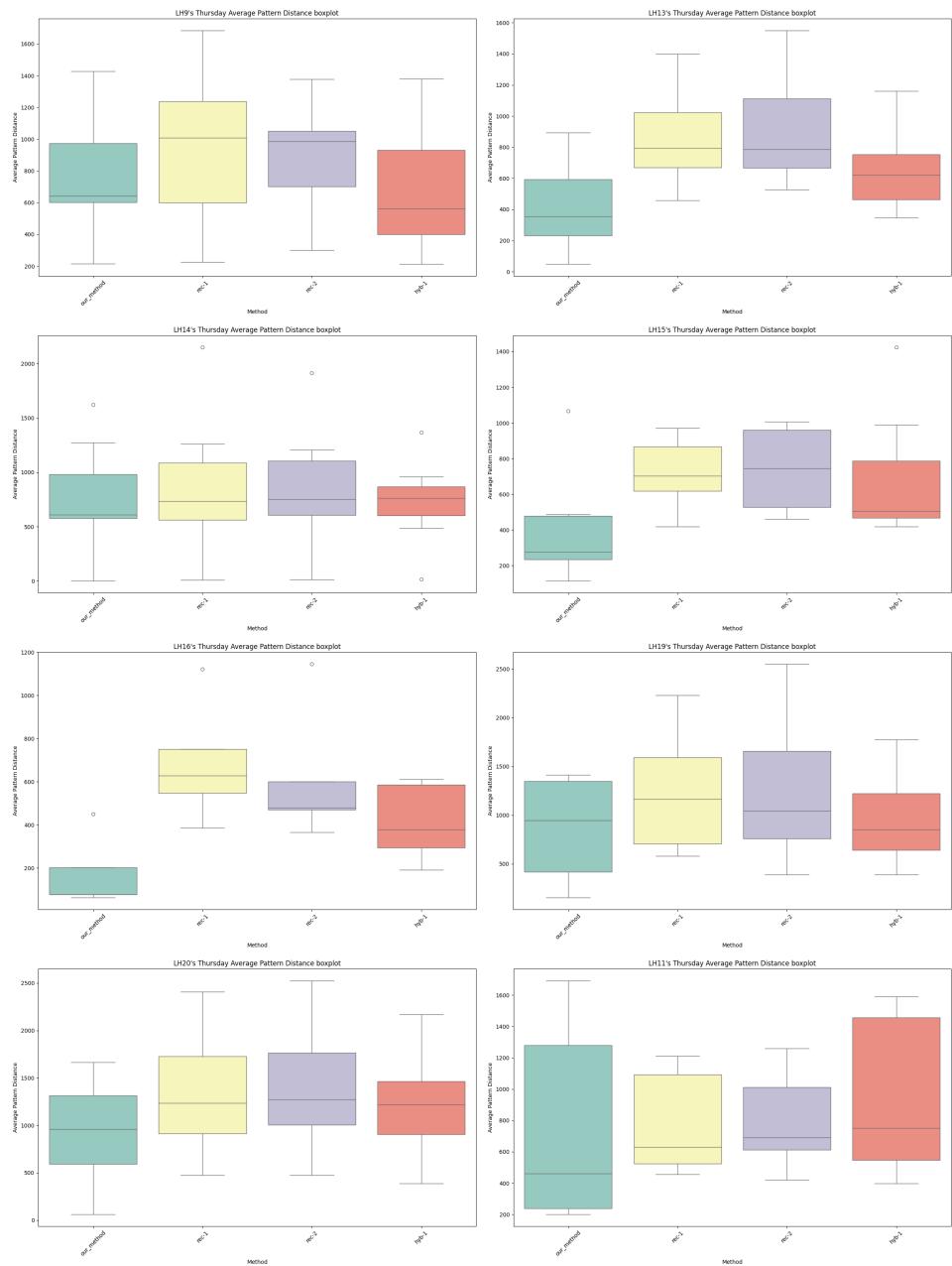


Figura 134: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Giovedì per i file di cronologia da LH9 a LH20.

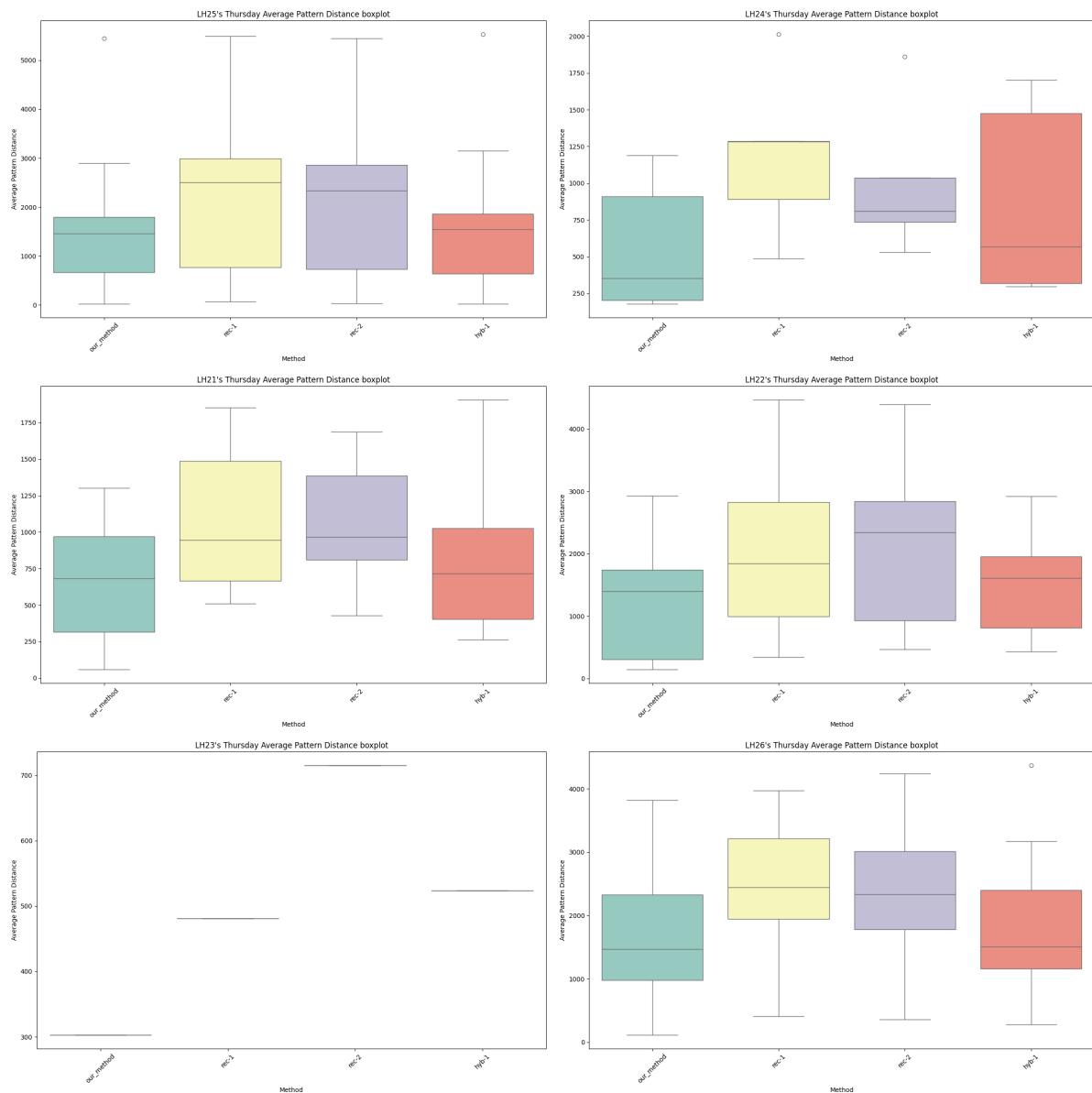


Figura 135: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Giovedì per i file di cronologia da LH21 a LH26.

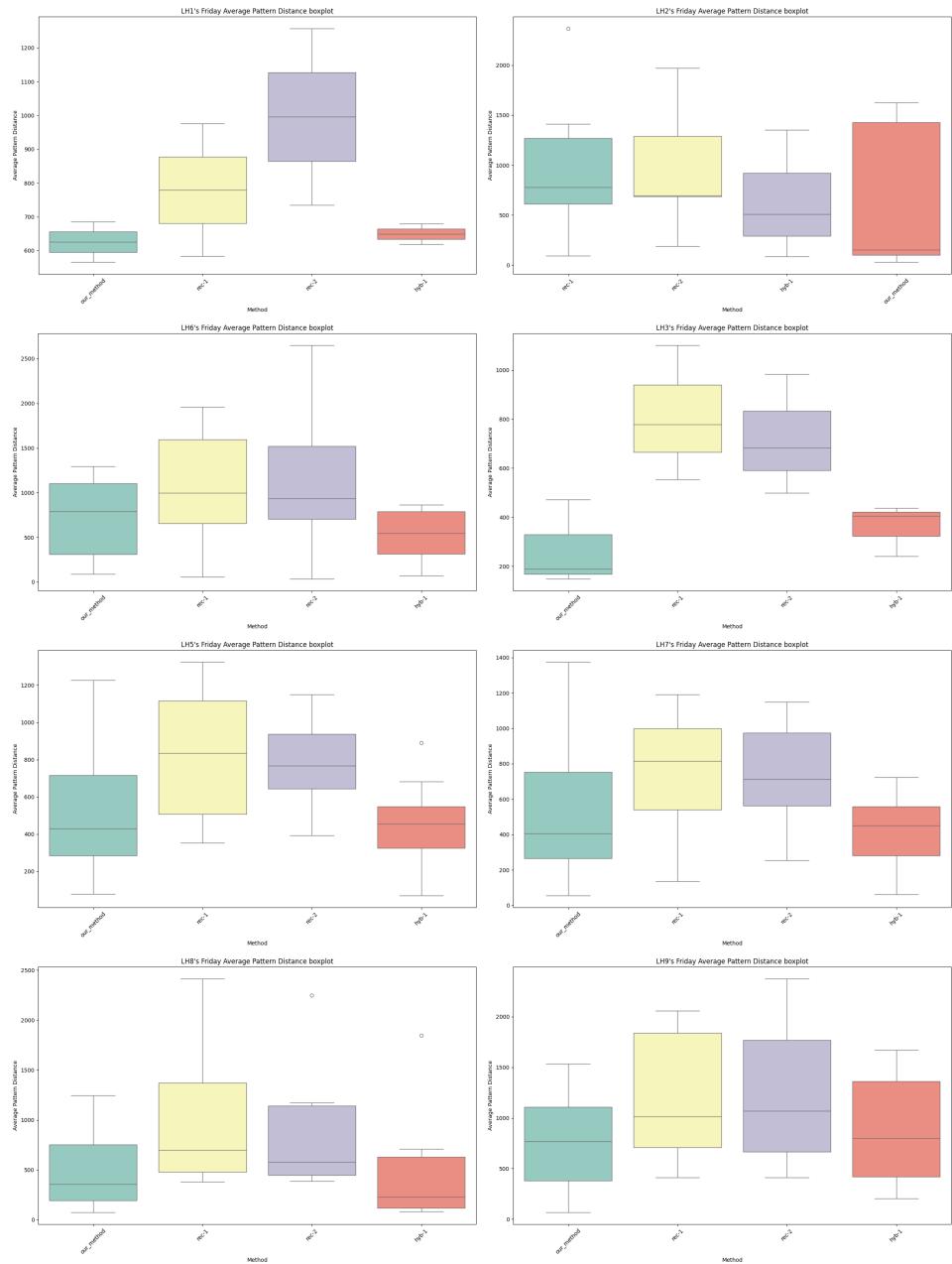


Figura 136: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Venerdì per i file di cronologia da LH1 a LH9.

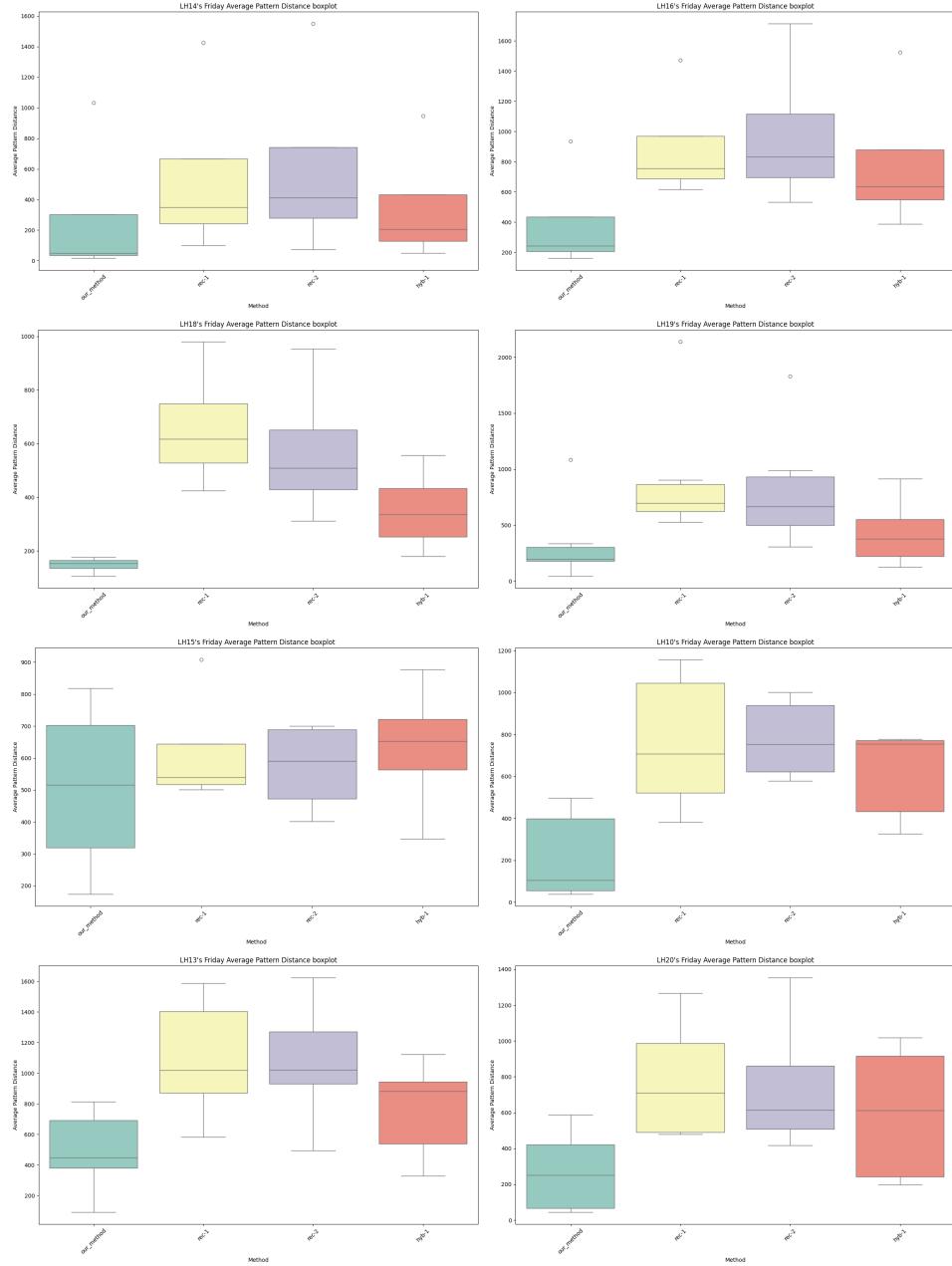


Figura 137: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Venerdì per i file di cronologia da LH10 a LH20.

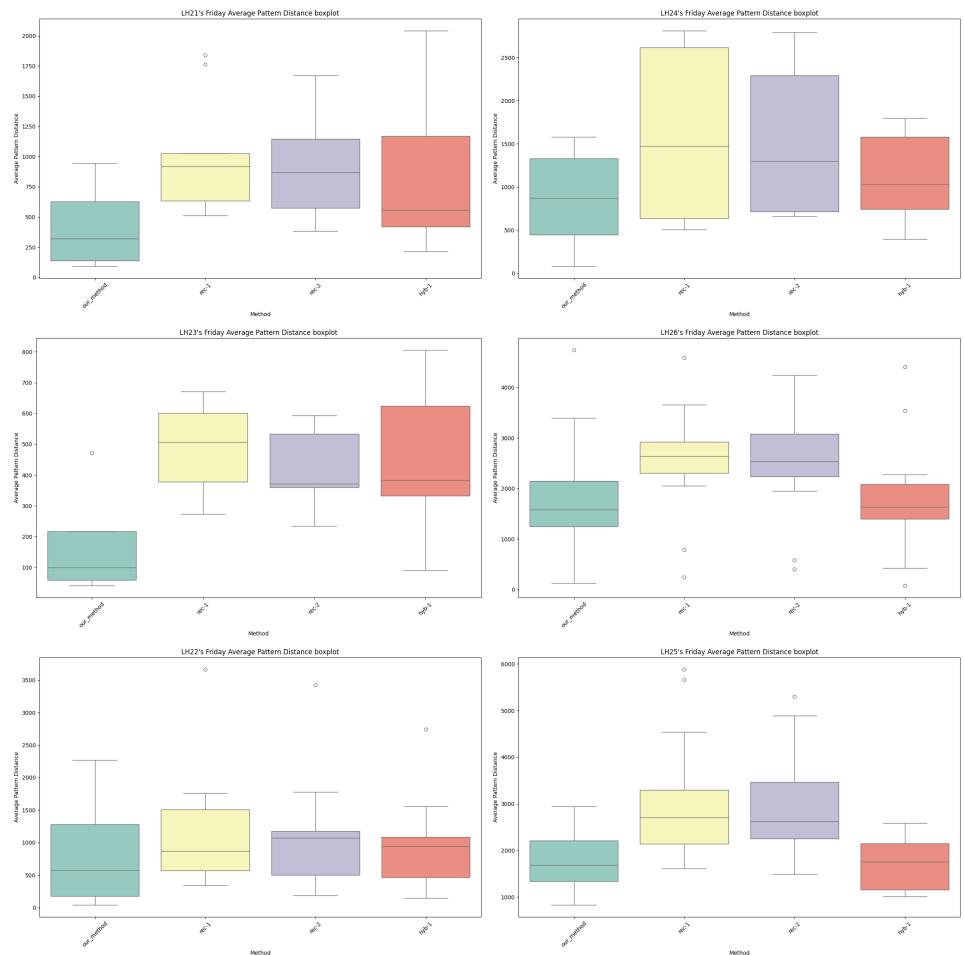


Figura 138: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Venerdì per i file di cronologia da LH21 a LH26.

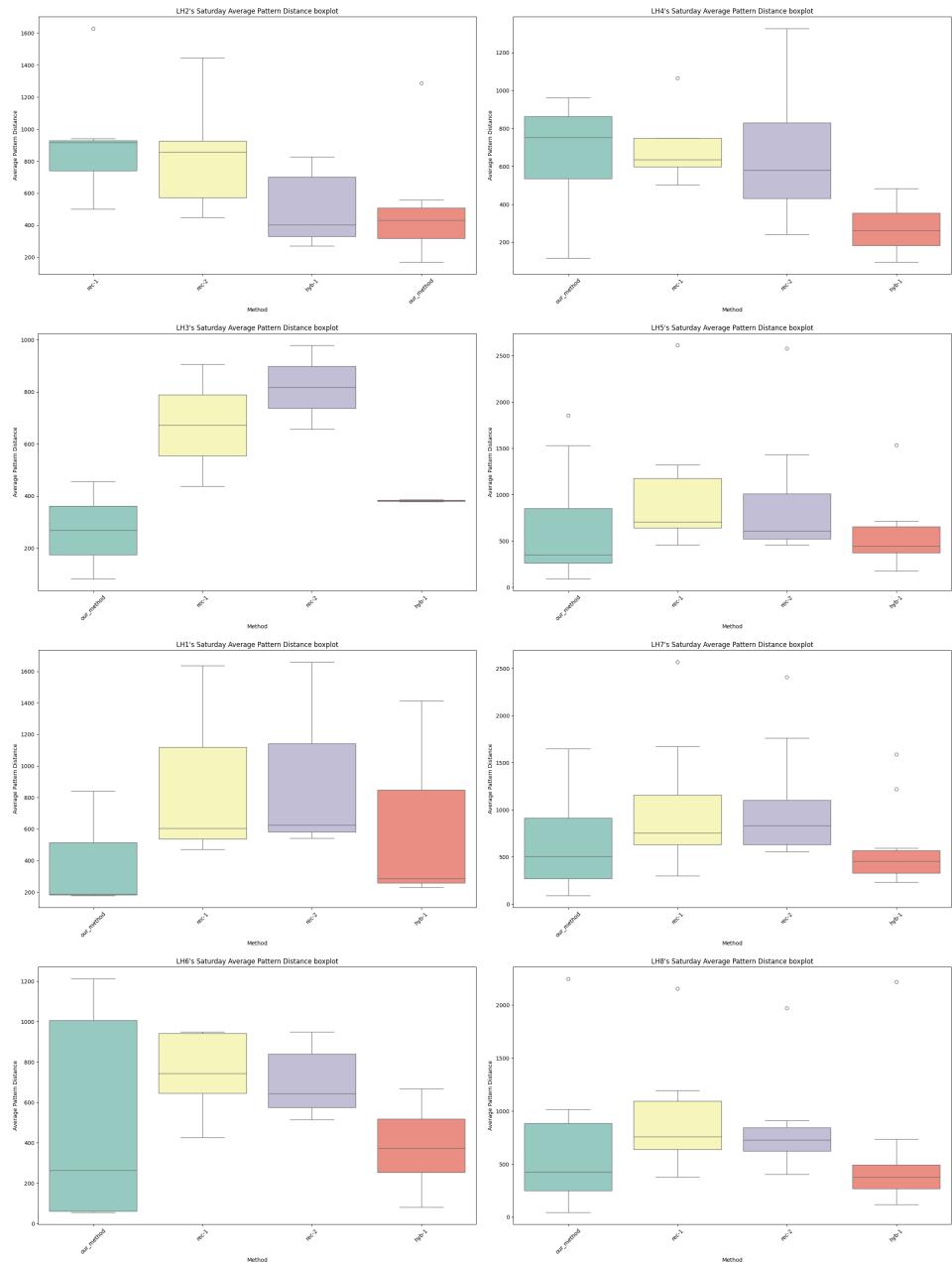


Figura 139: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Sabato per i file di cronologia da LH1 a LH8.

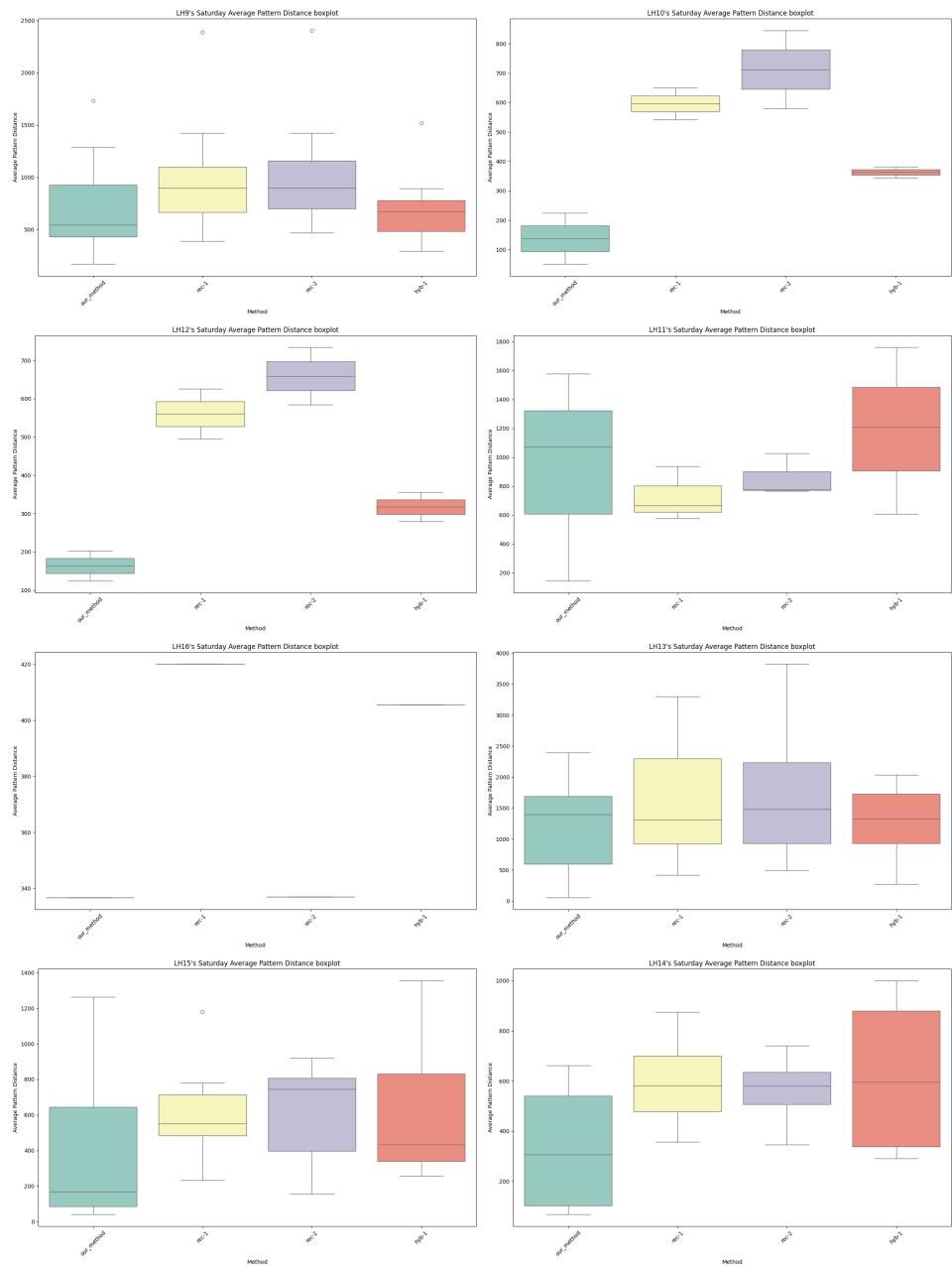


Figura 140: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Sabato per i file di cronologia da LH9 a LH16.

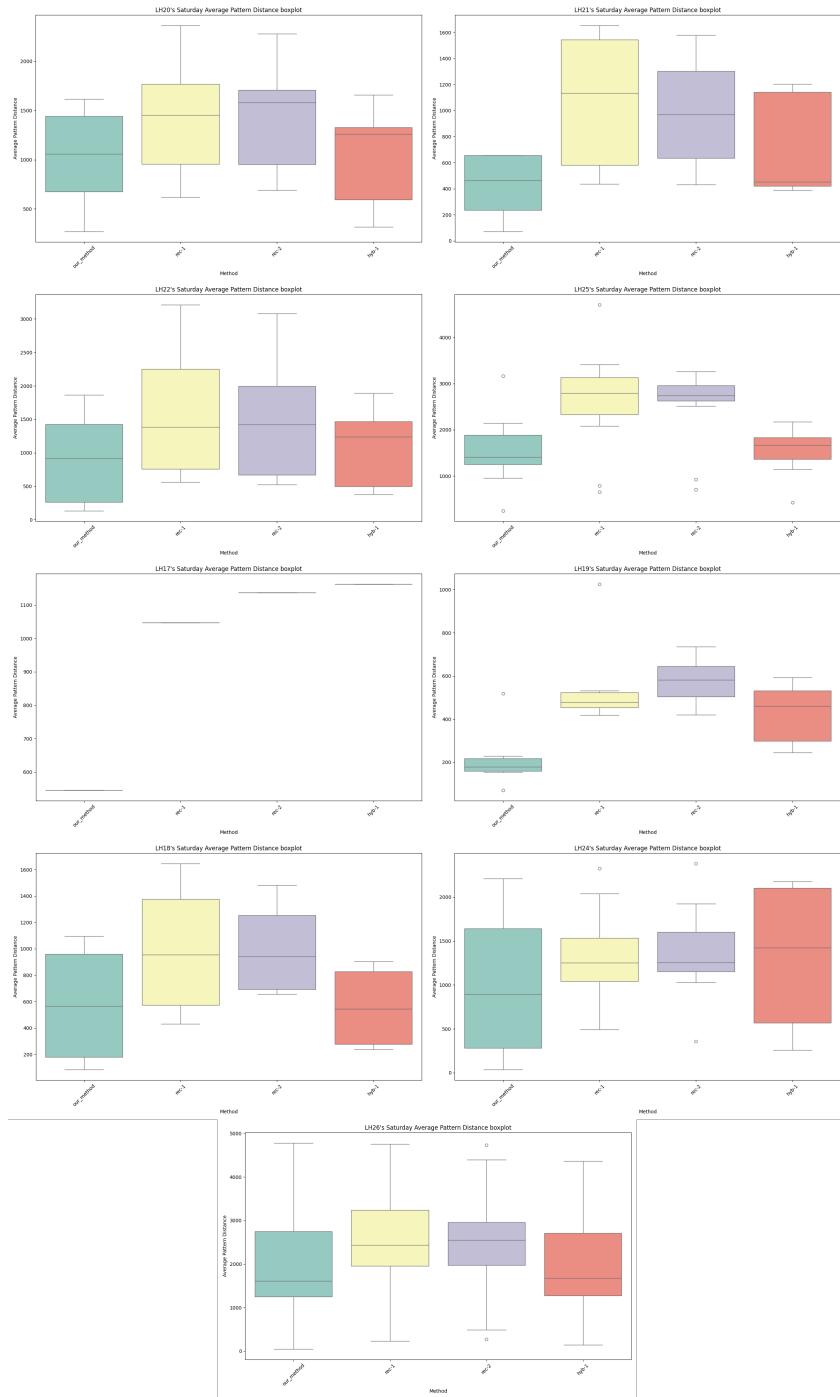


Figura 141: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Sabato per i file di cronologia da LH17 a LH26.

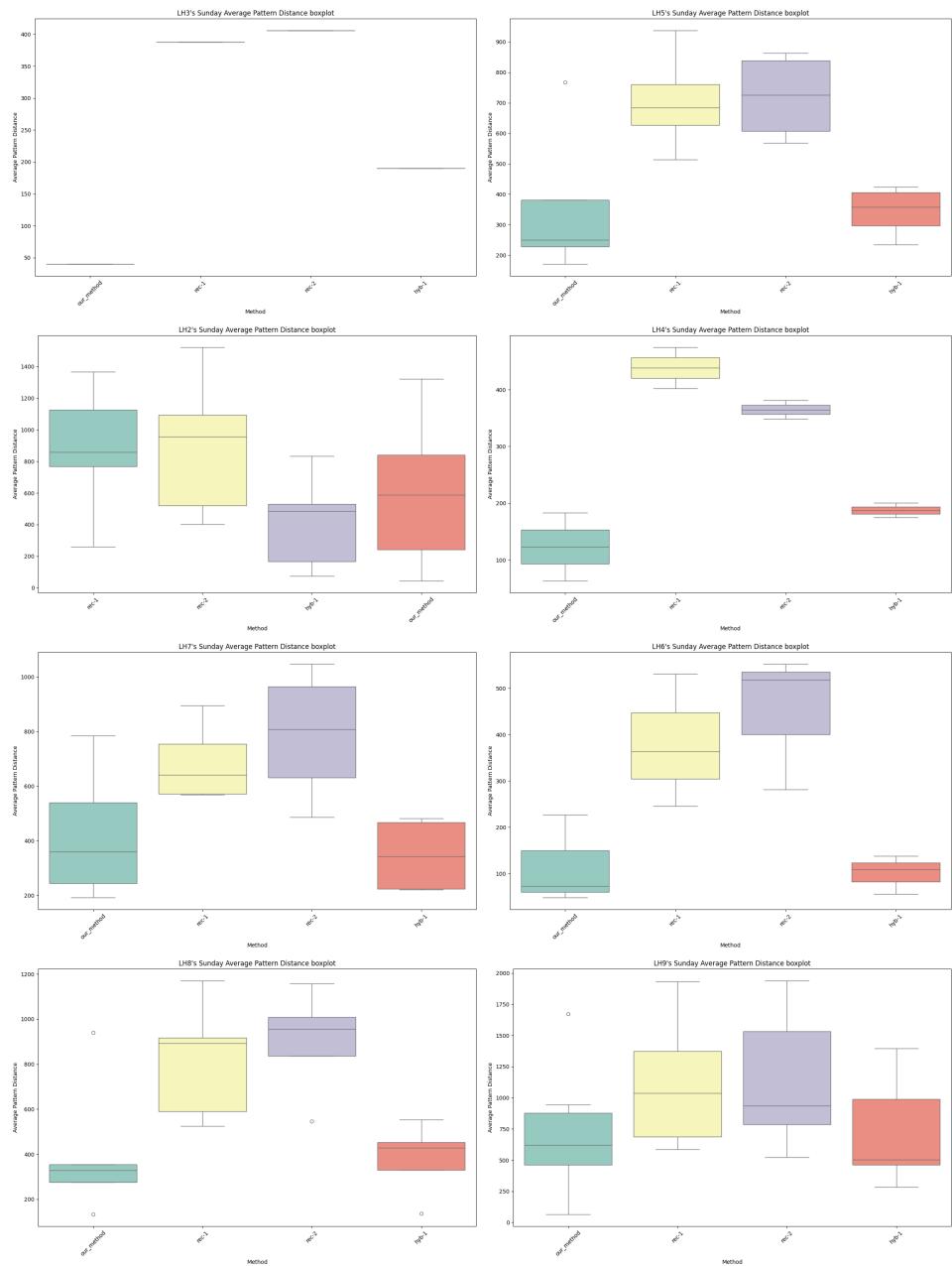


Figura 142: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Domenica per i file di cronologia da LH2 a LH9.

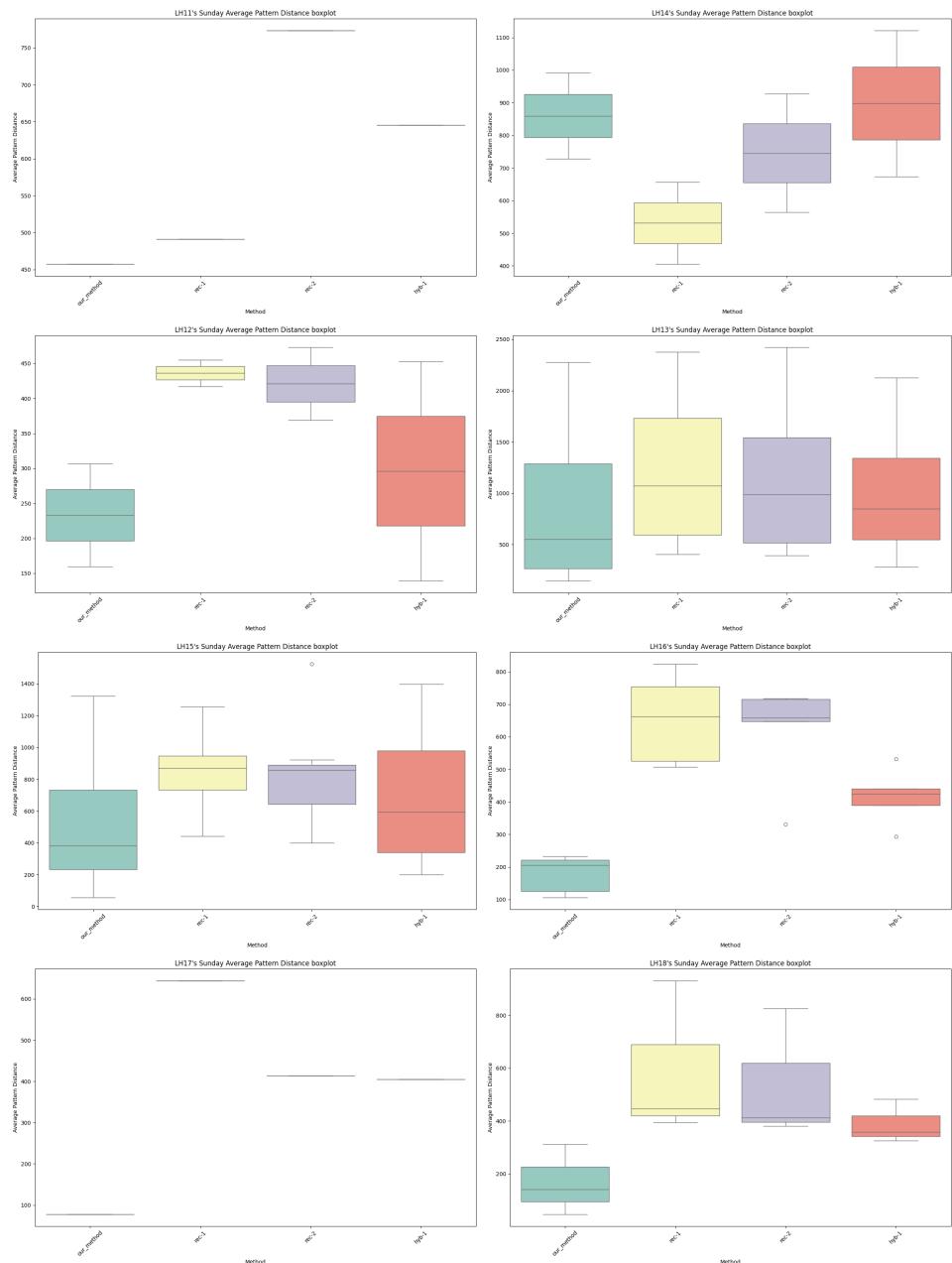


Figura 143: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Domenica per i file di cronologia da LH11 a LH18.

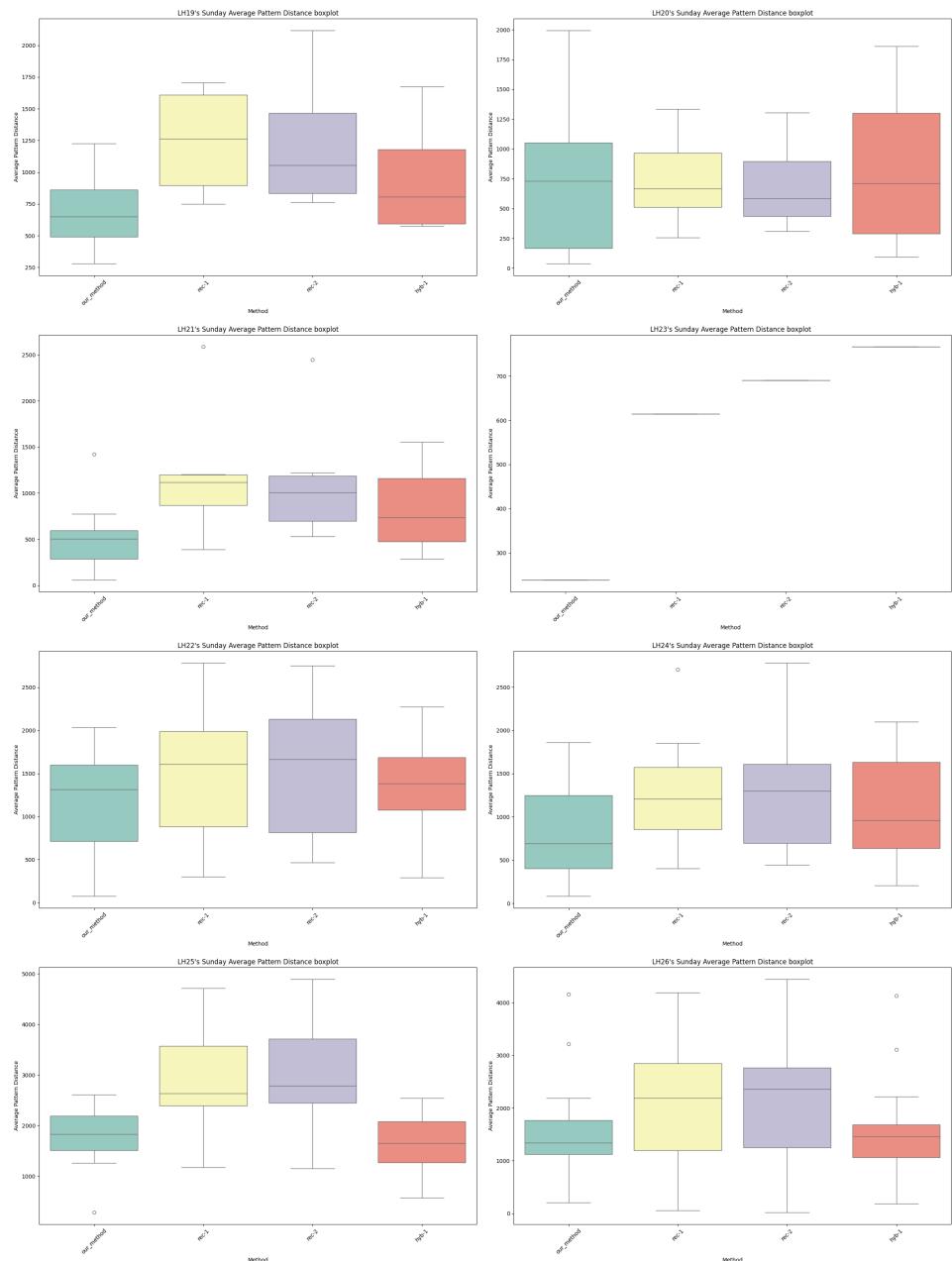


Figura 144: Boxplot della Playlist Pattern Distance media di ogni metodo durante il giorno Domenica per i file di cronologia da LH19 a LH26.

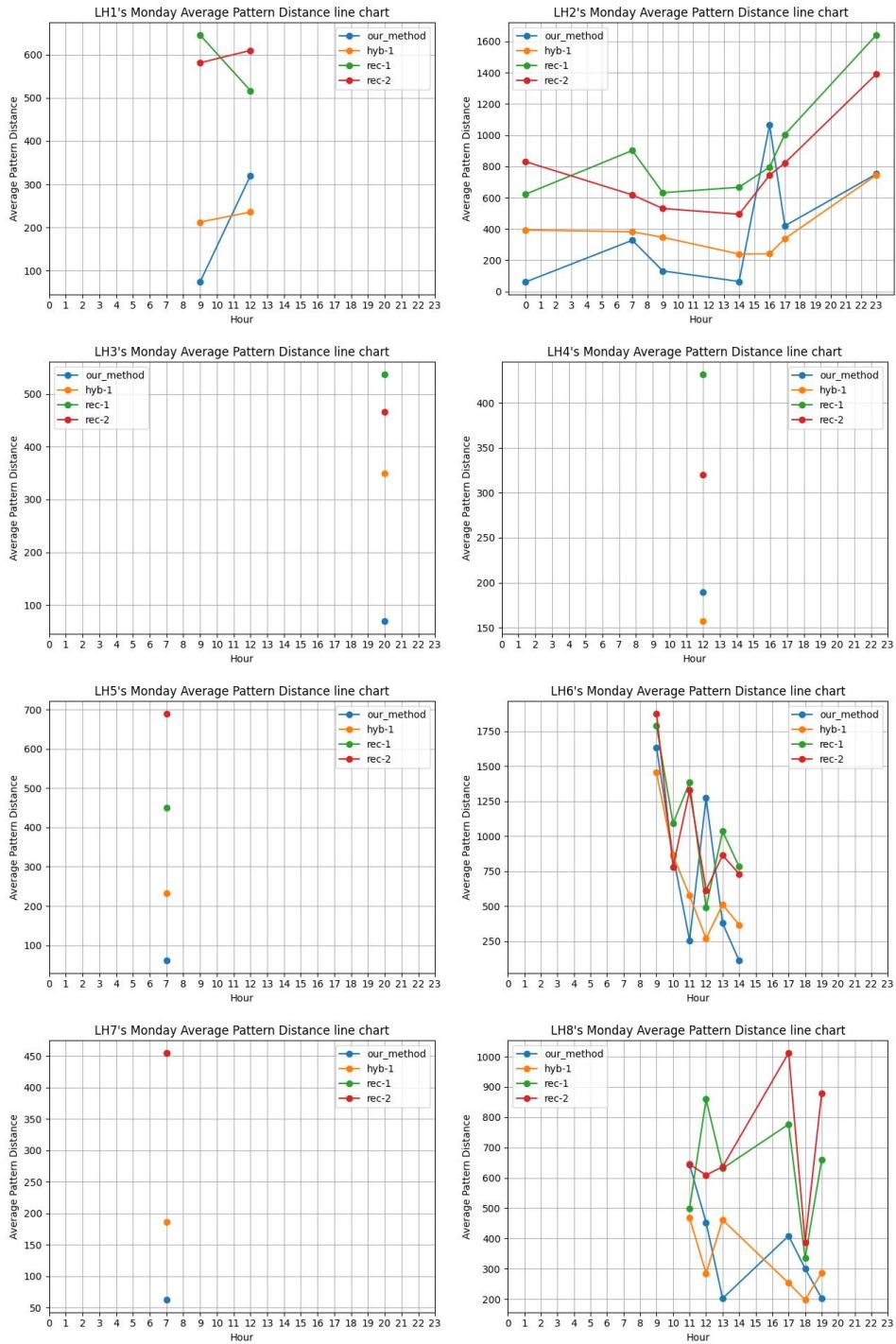


Figura 145: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Lunedì per i file di cronologia da LH1 a LH8.

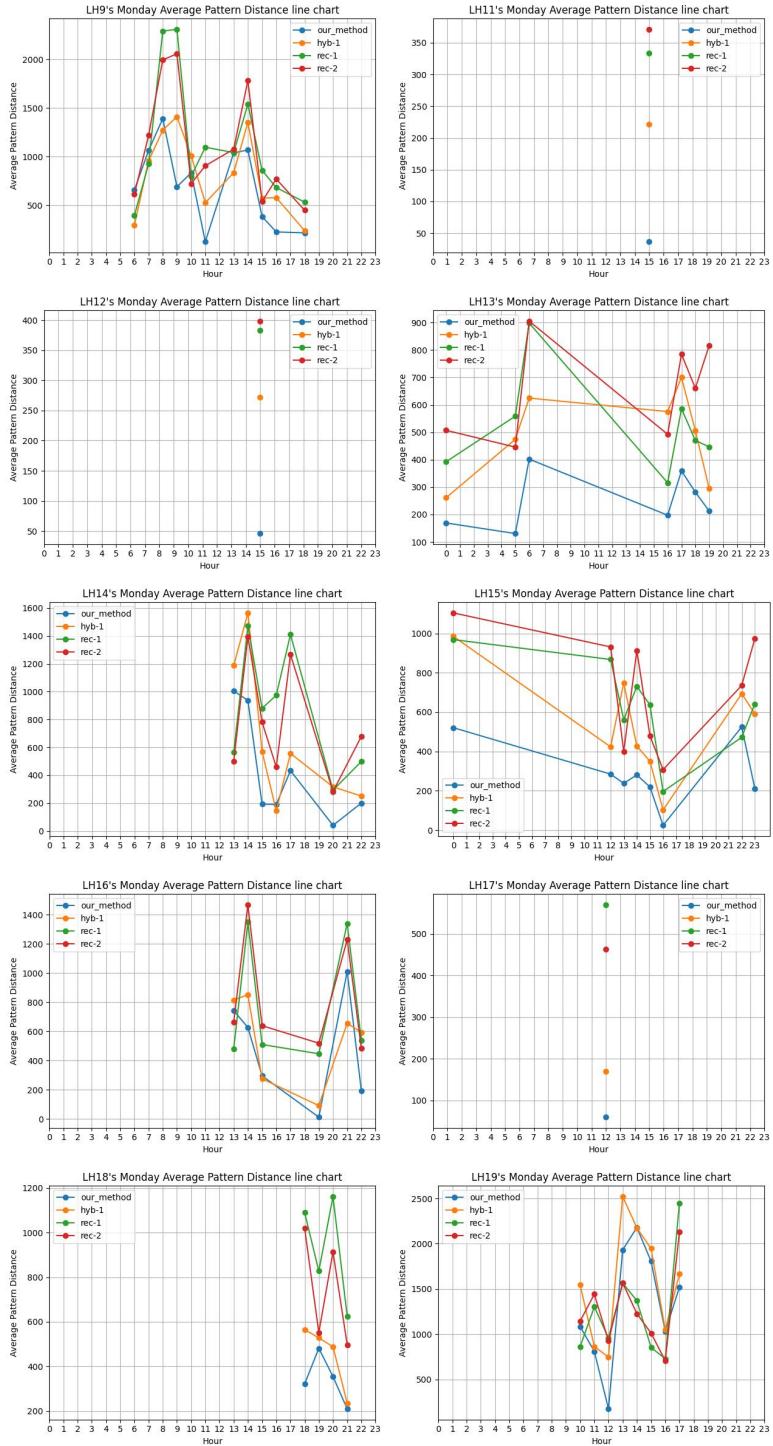


Figura 146: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Lunedì per i file di cronologia da LH9 a LH19.



Figura 147: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Lunedì per i file di cronologia da LH20 a LH26.

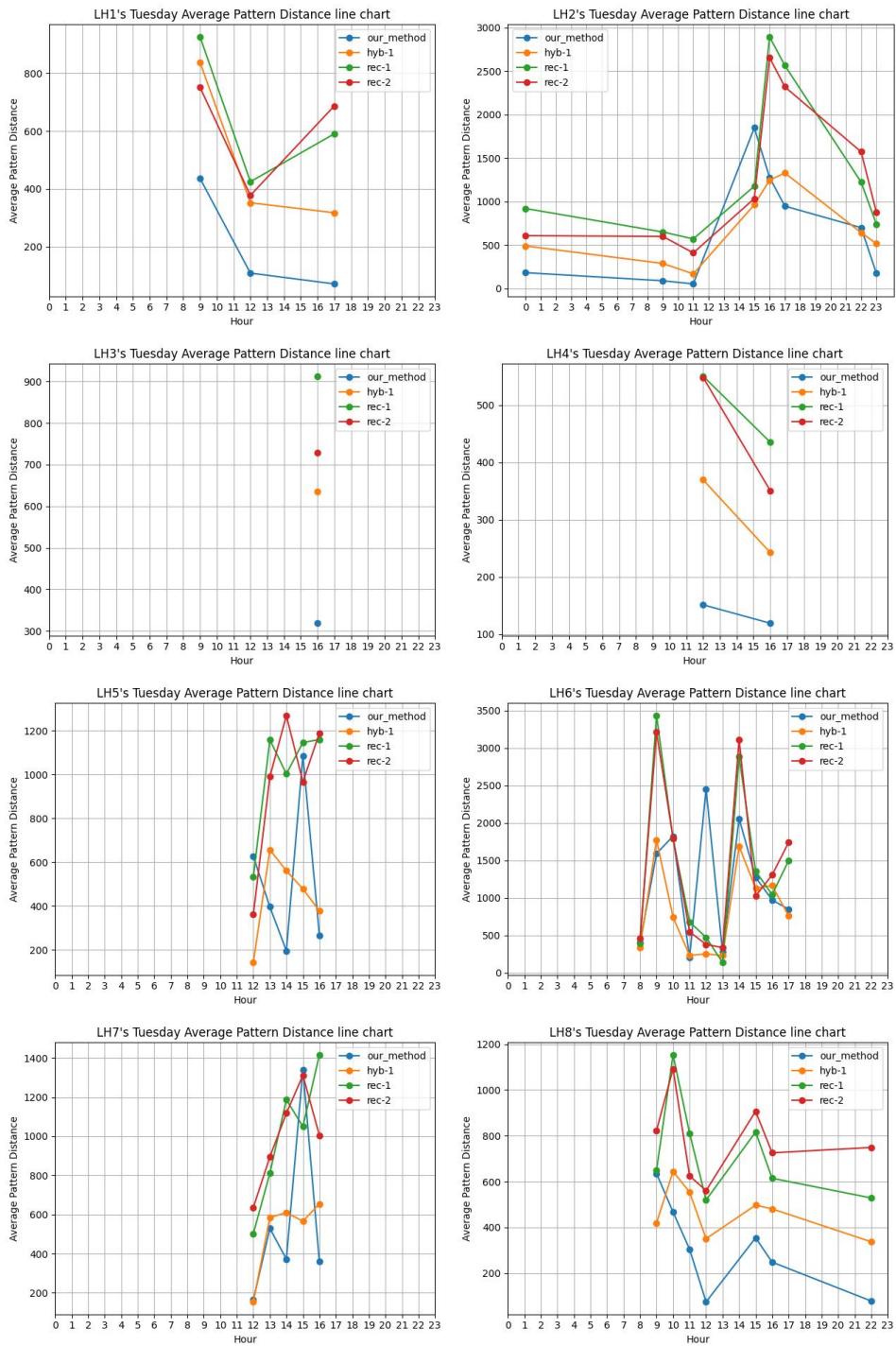


Figura 148: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Martedì per i file di cronologia da LH1 a LH8.

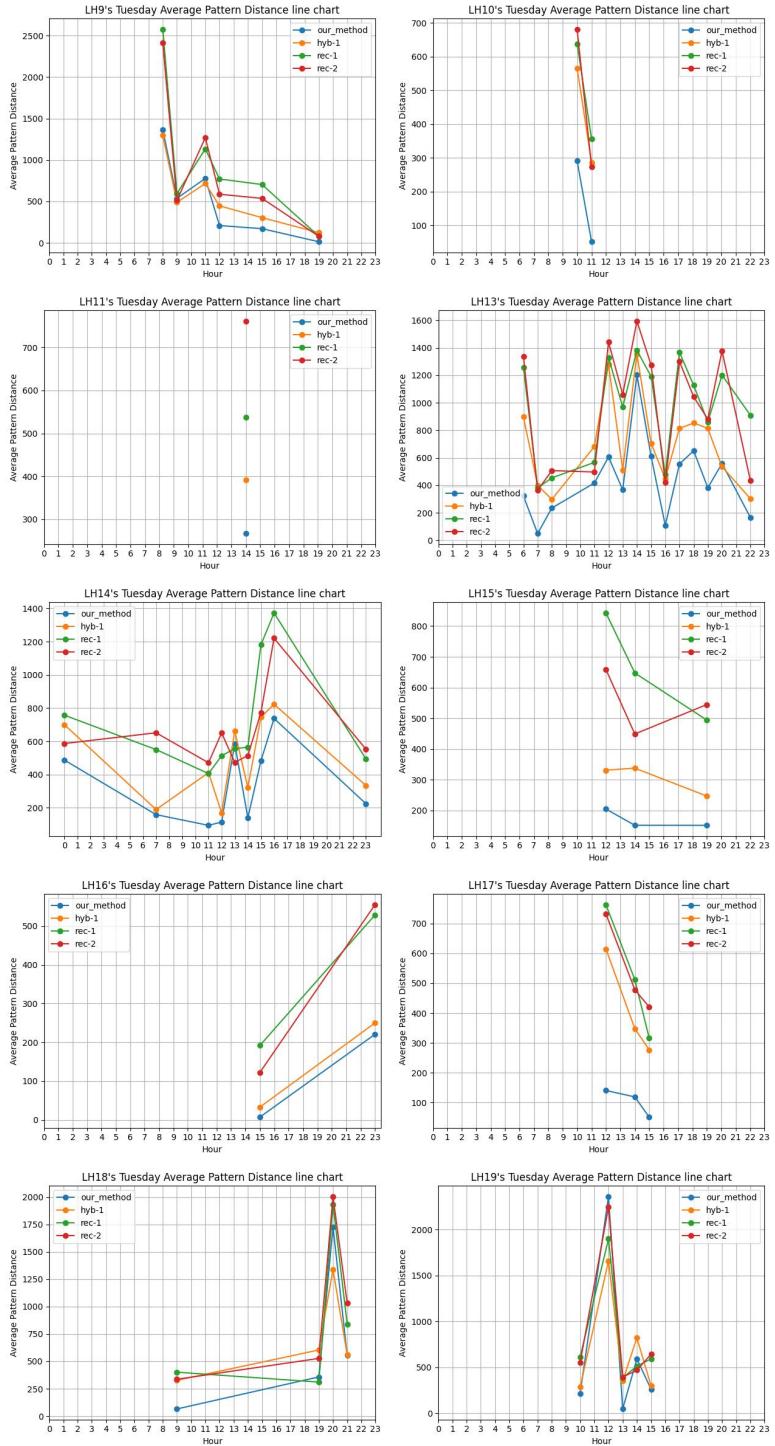


Figura 149: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Martedì per i file di cronologia da LH9 a LH19.

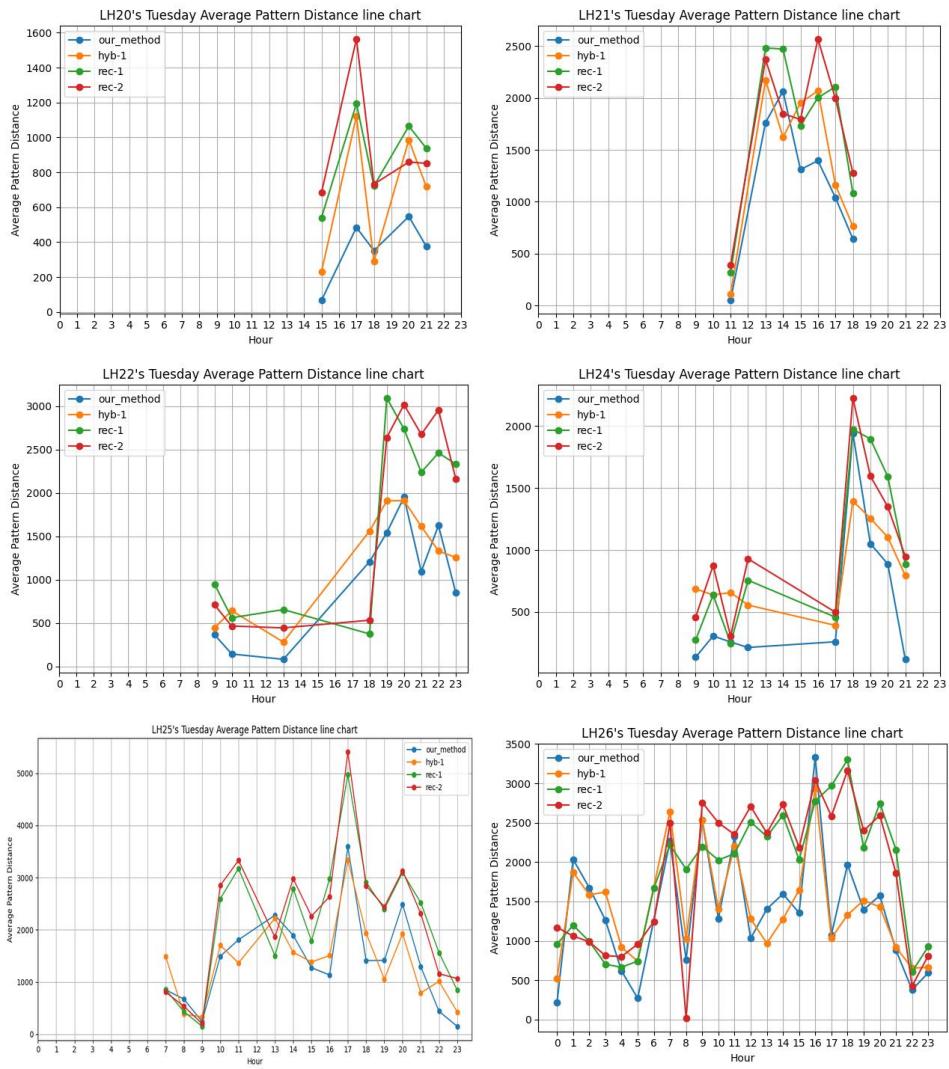


Figura 150: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Martedì per i file di cronologia da LH20 a LH26.

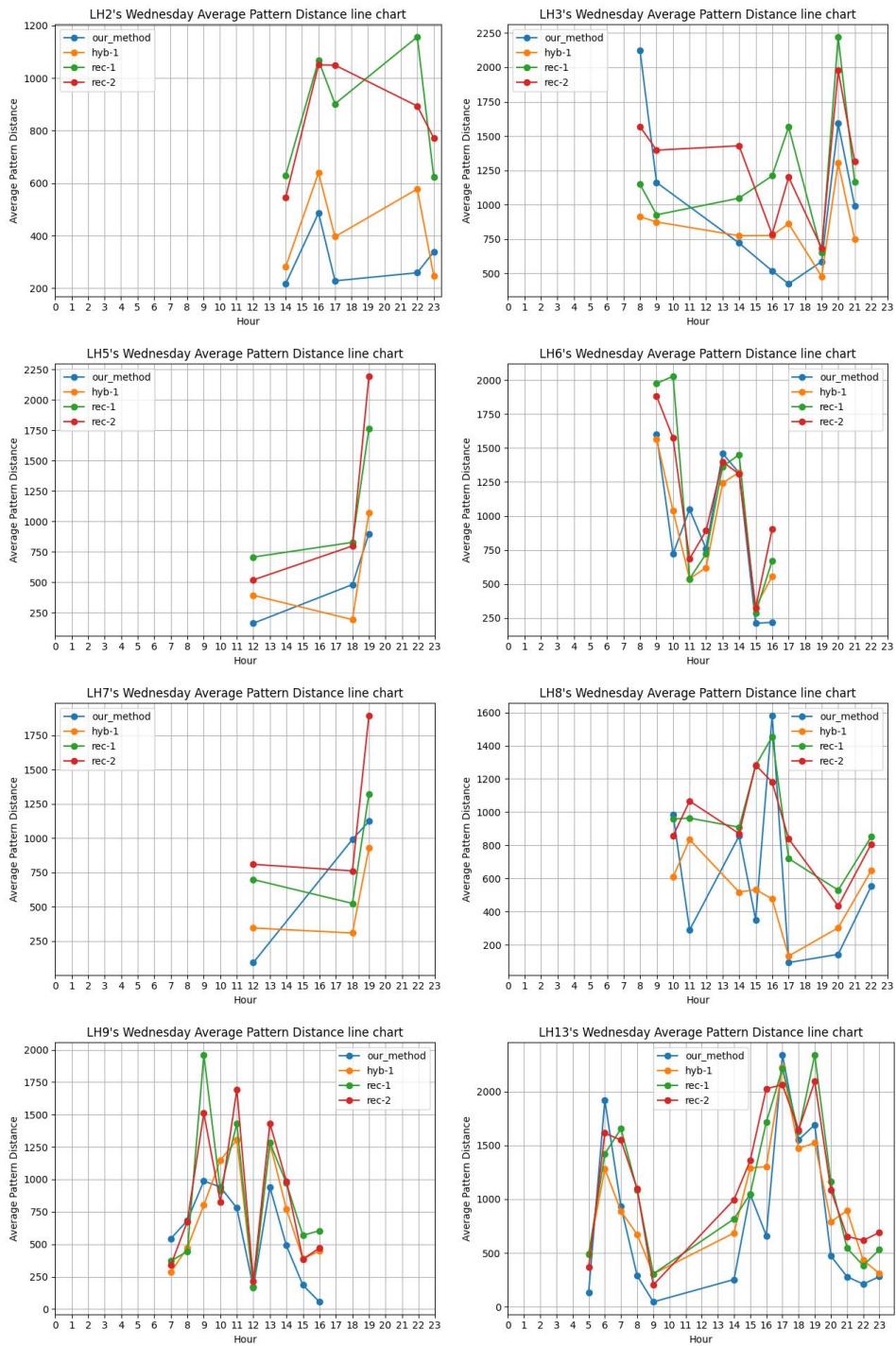


Figura 151: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Mercoledì per i file di cronologia da LH2 a LH13.

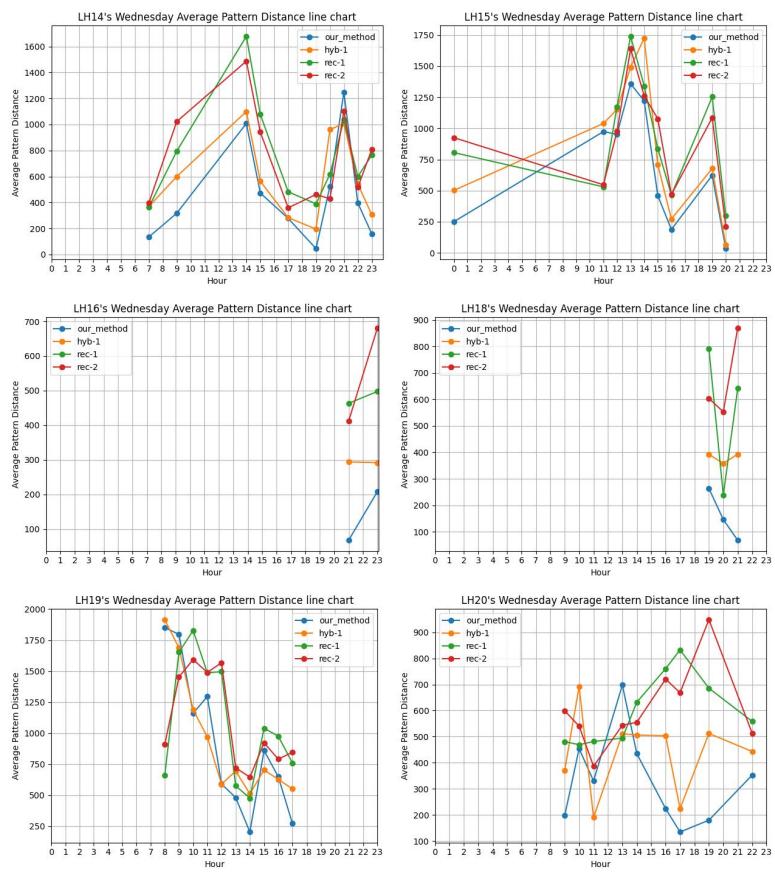


Figura 152: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Mercoledì per i file di cronologia da LH14 a LH20.

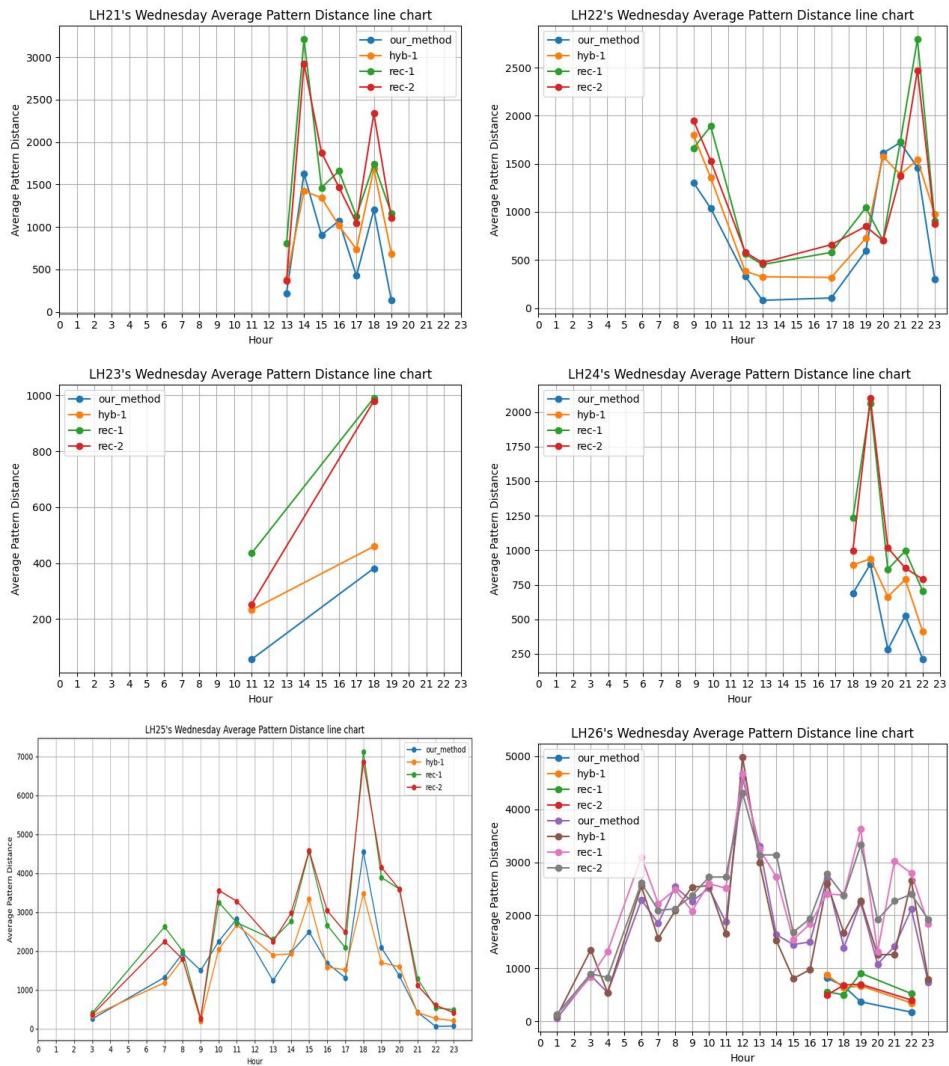


Figura 153: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Mercoledì per i file di cronologia da LH21 a LH26.

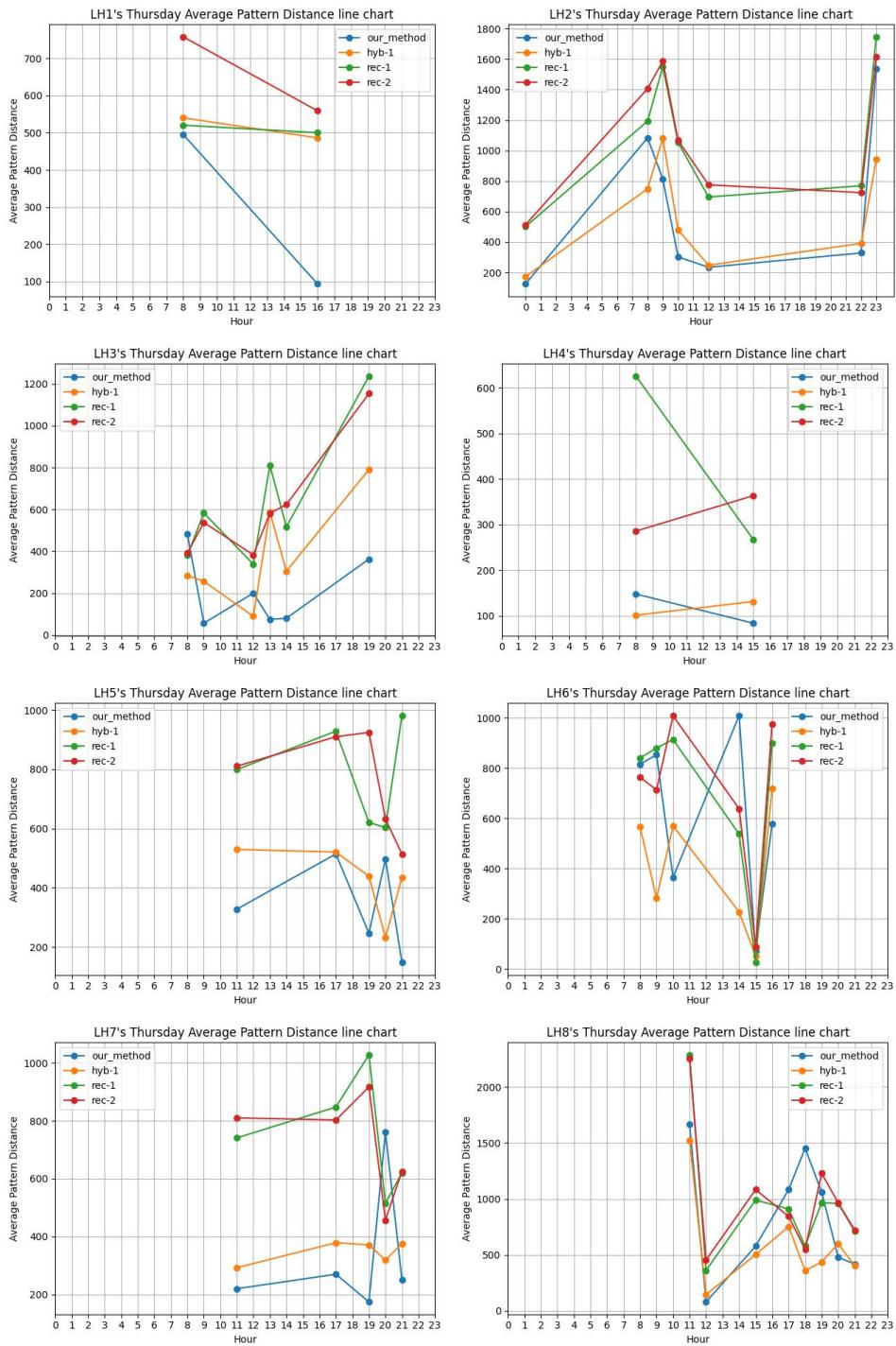


Figura 154: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Giovedì per i file di cronologia da LH1 a LH8.

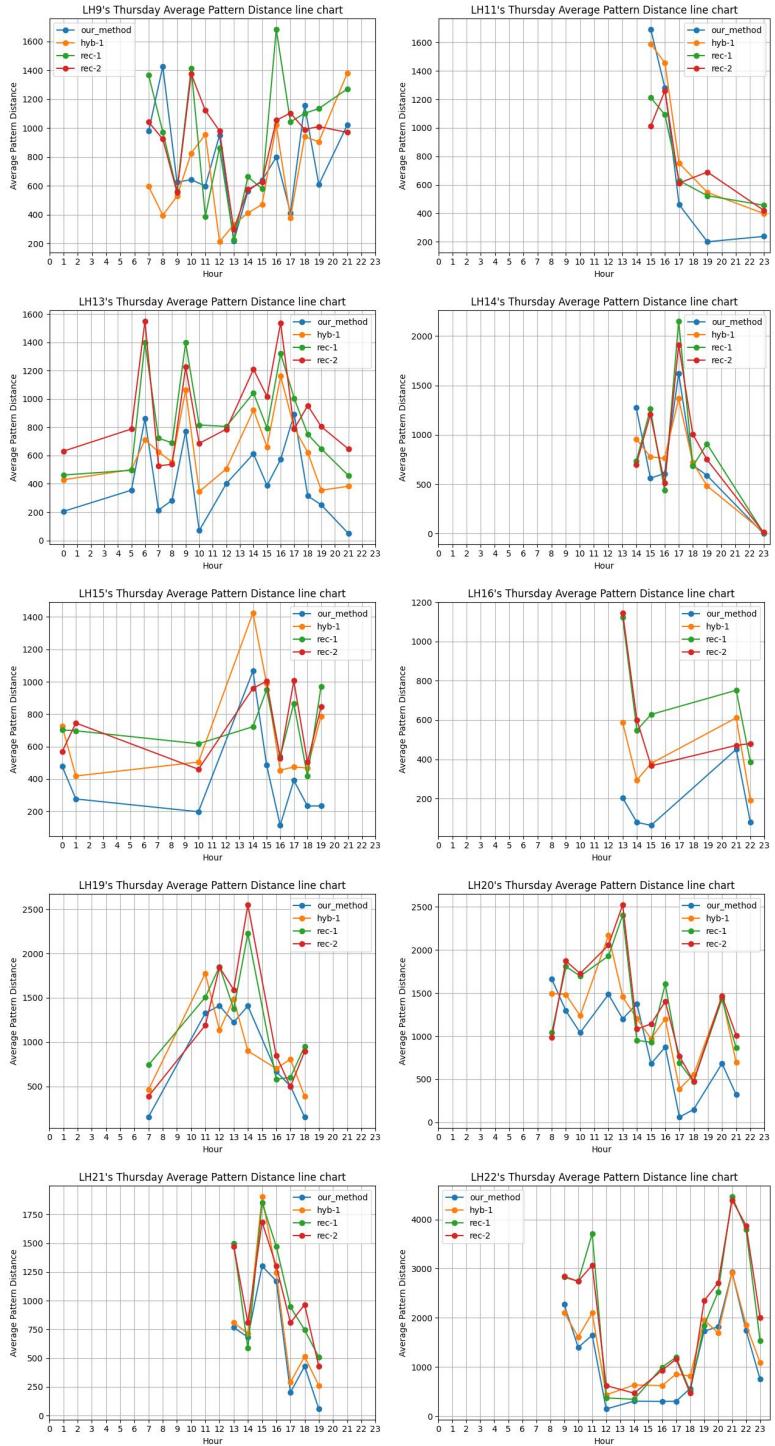


Figura 155: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Giovedì per i file di cronologia da LH9 a LH22.

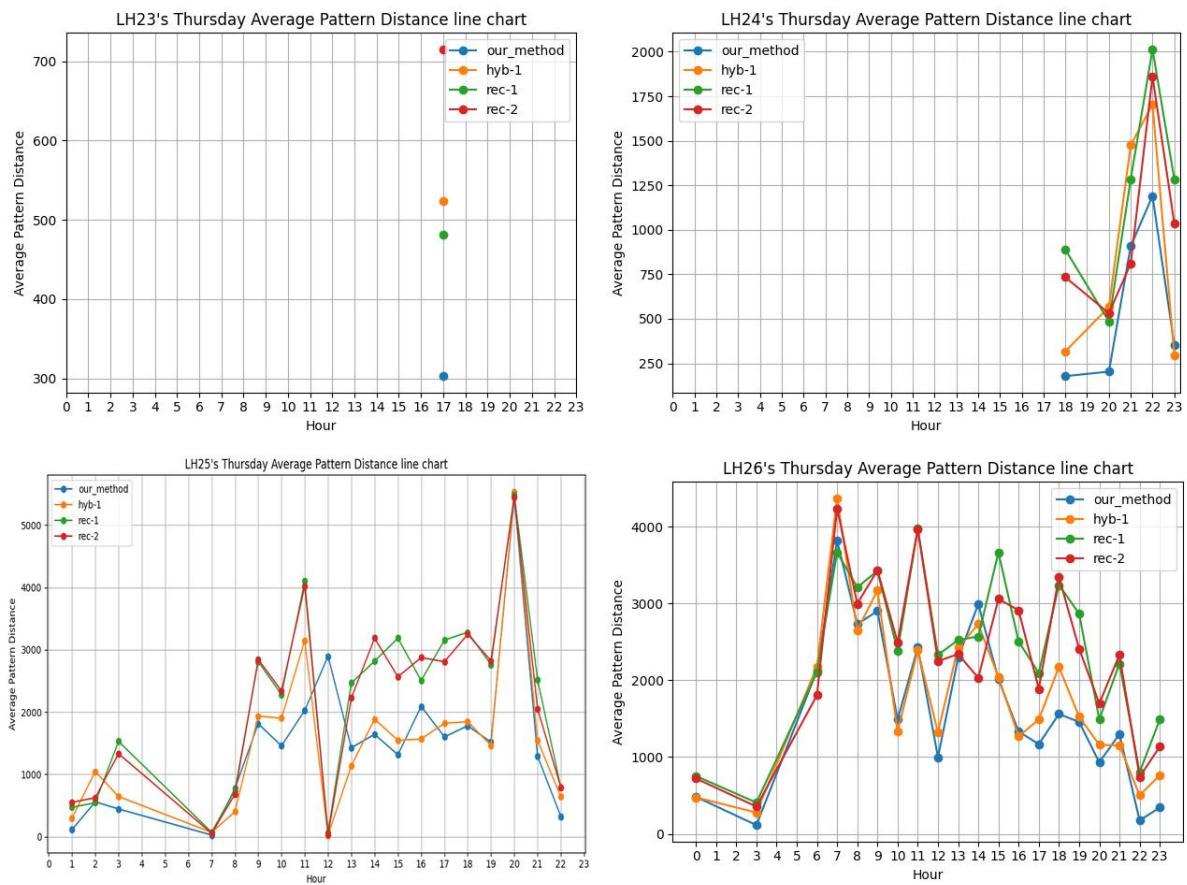


Figura 156: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Giovedì per i file di cronologia da LH23 a LH26.

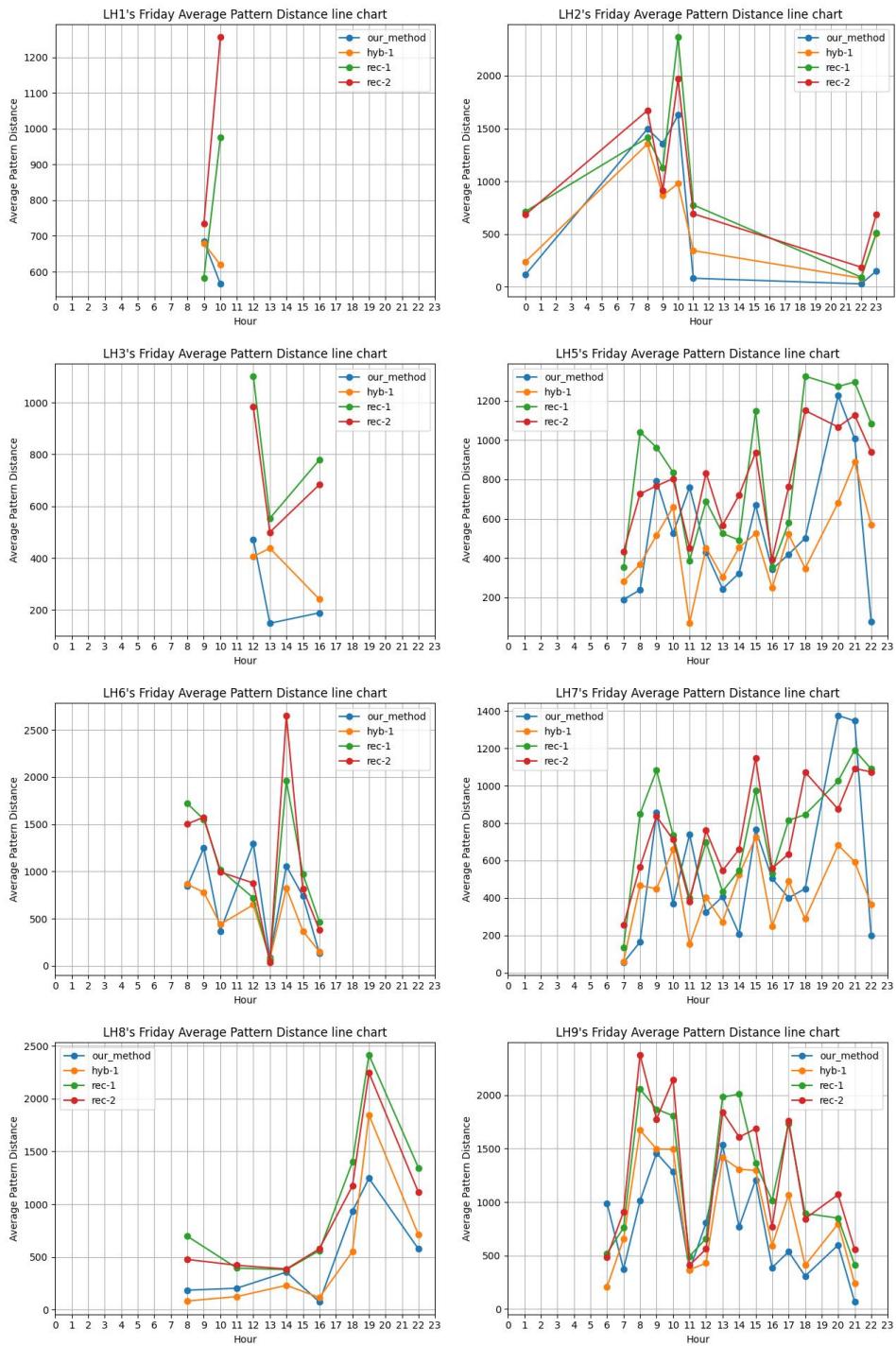


Figura 157: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Venerdì per i file di cronologia da LH1 a LH9.

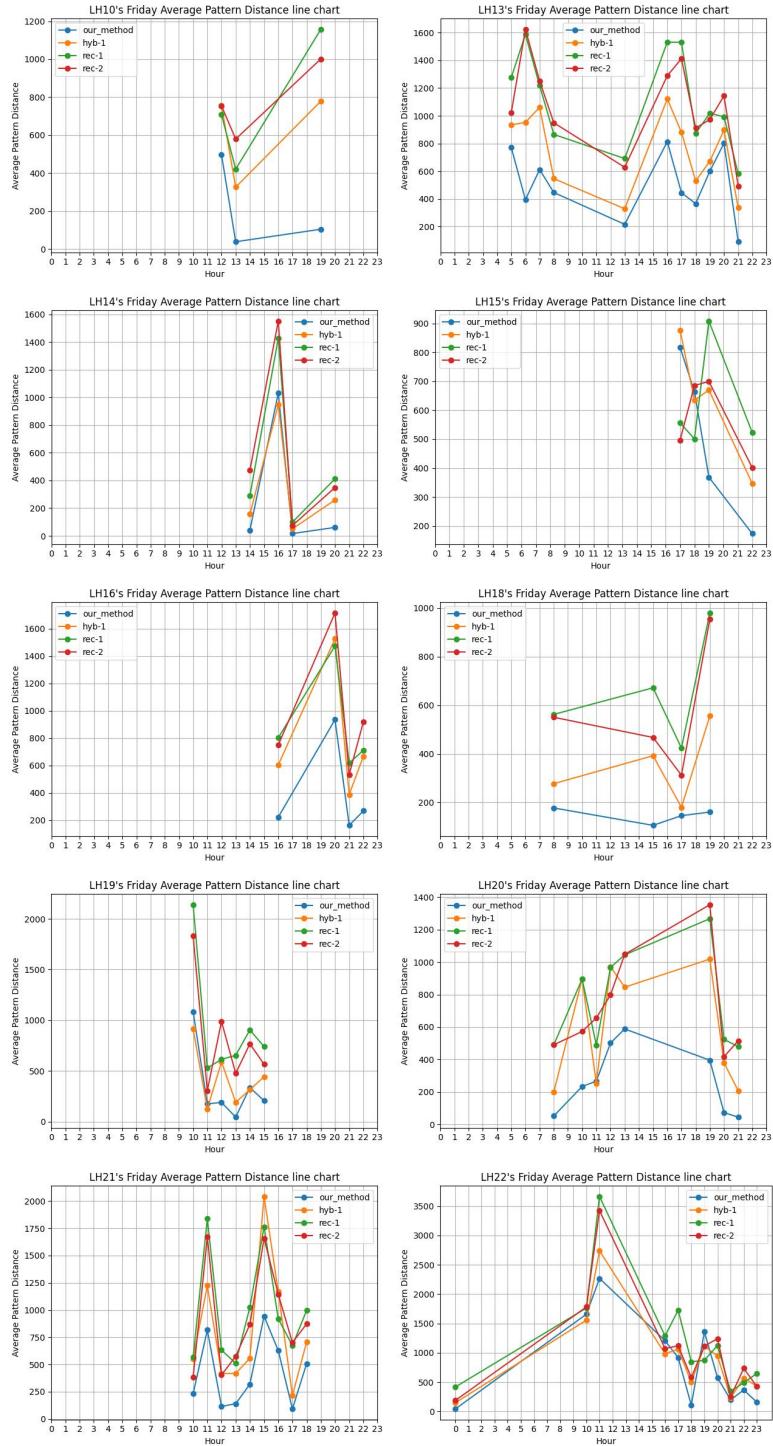


Figura 158: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Venerdì per i file di cronologia da LH10 a LH22.

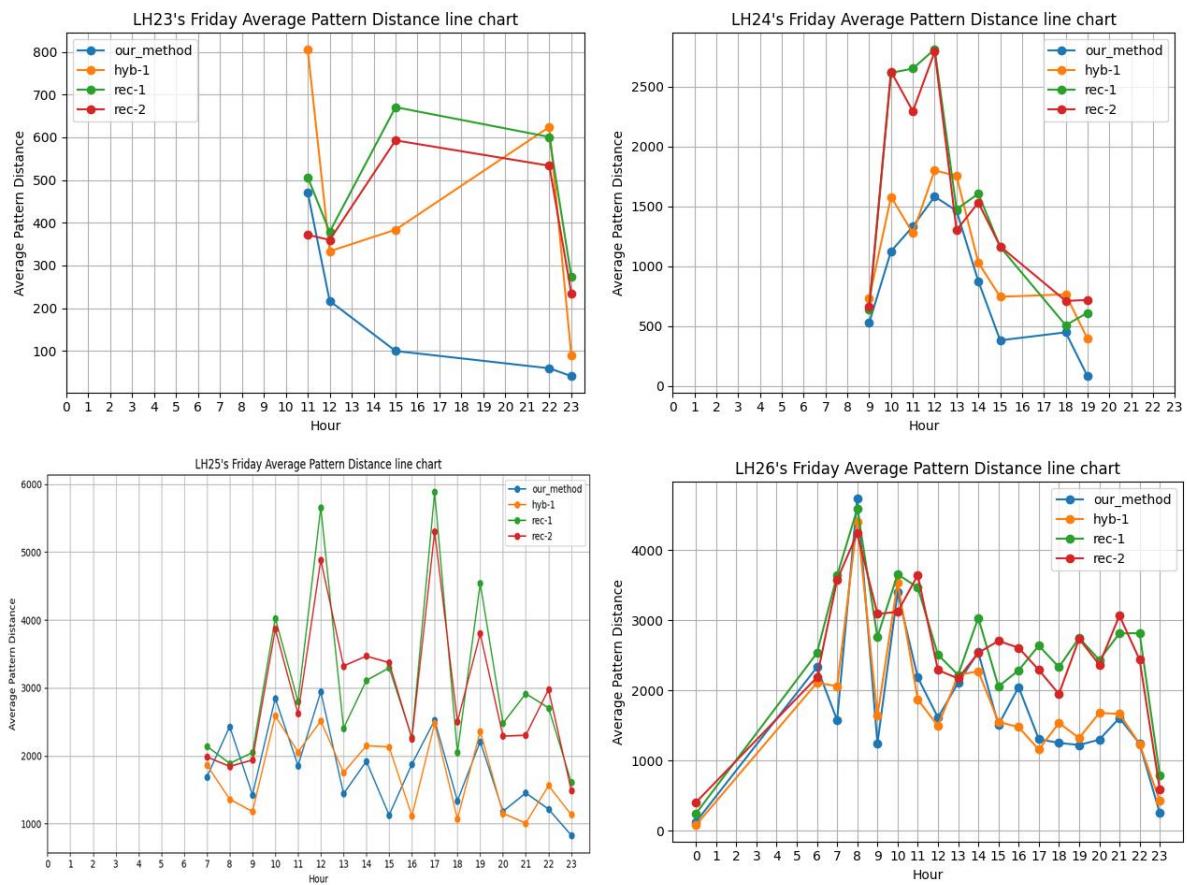


Figura 159: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Venerdì per i file di cronologia da LH23 a LH26.

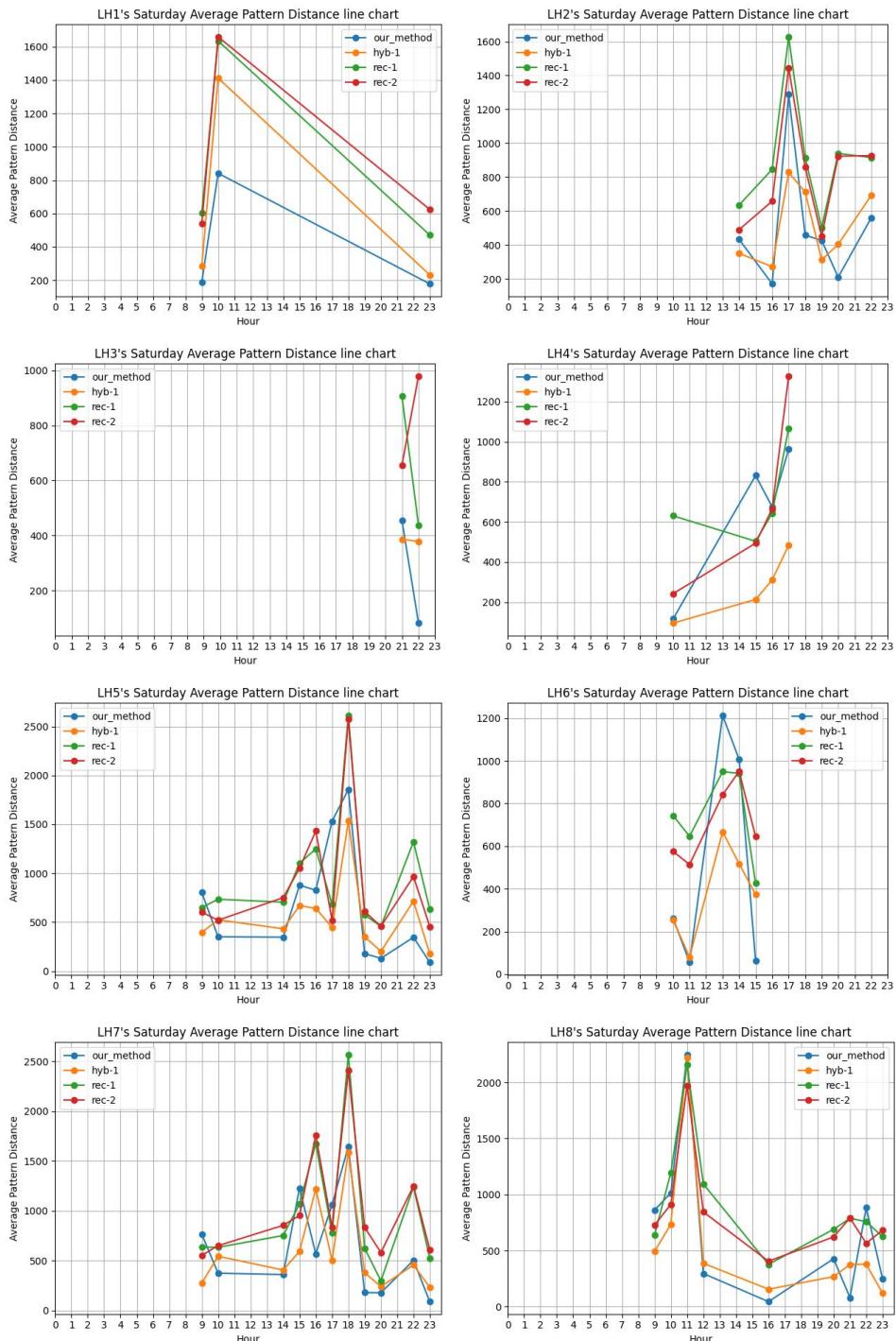


Figura 160: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Sabato per i file di cronologia da LH1 a LH8.

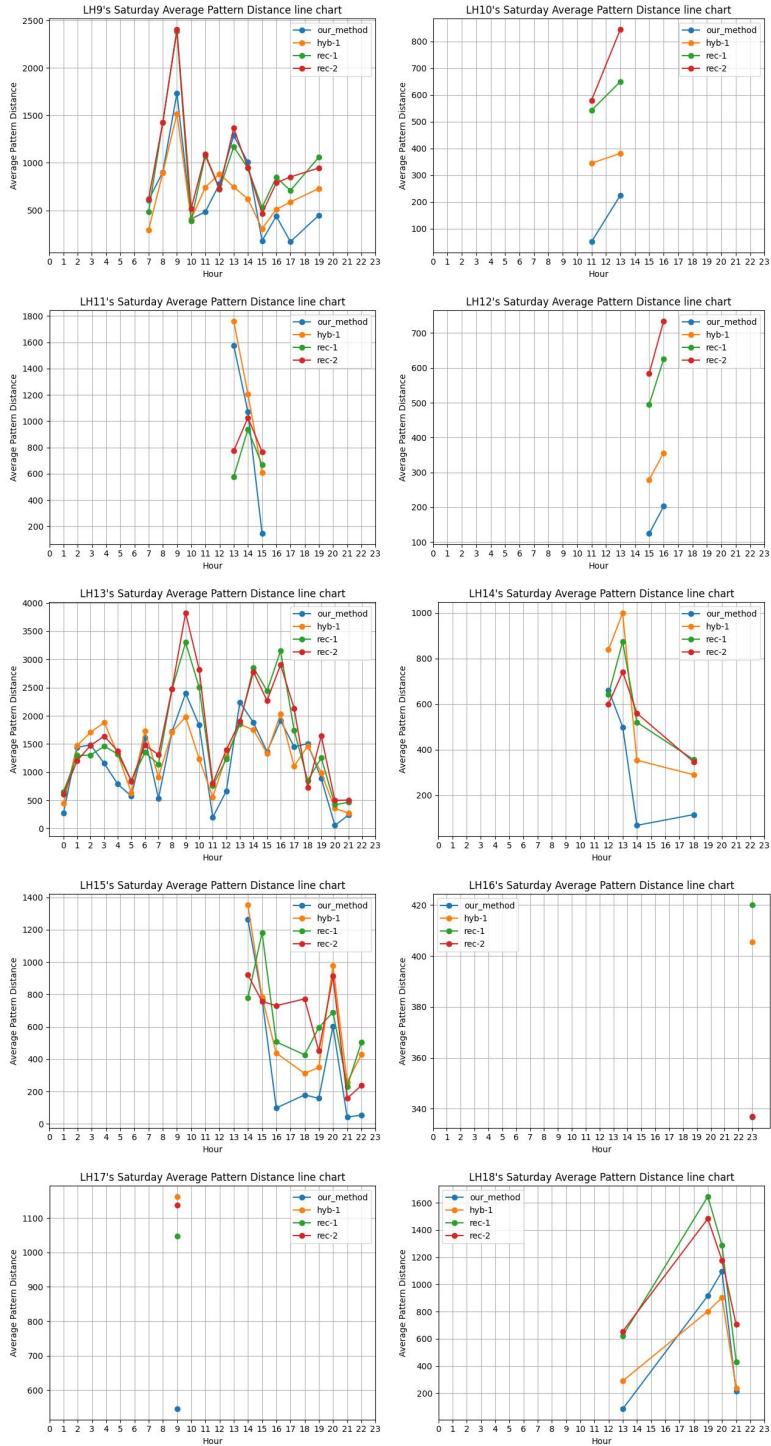


Figura 161: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Sabato per i file di cronologia da LH9 a LH18.



Figura 162: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Sabato per i file di cronologia da LH19 a LH26.

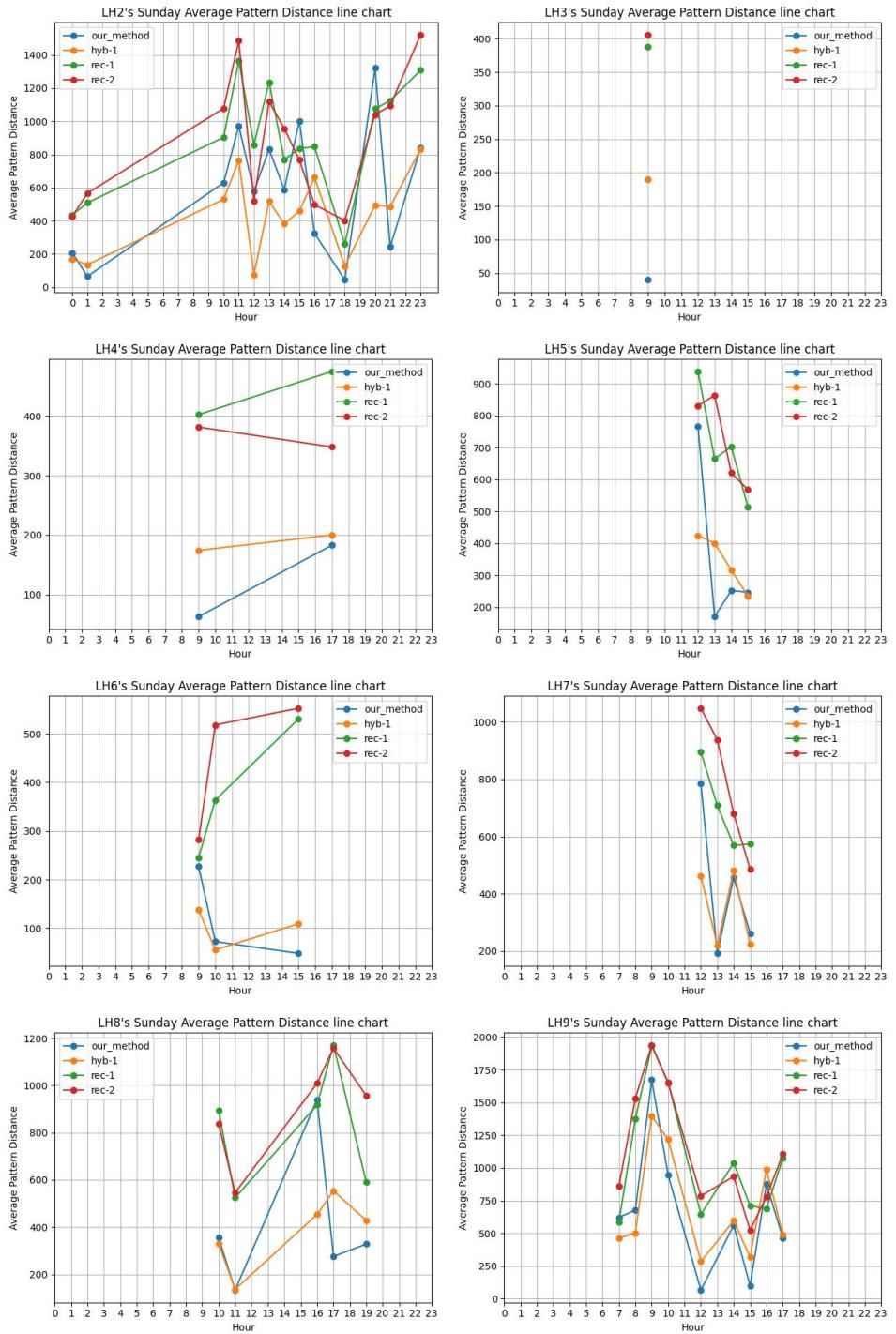


Figura 163: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Domenica per i file di cronologia da LH2 a LH9.

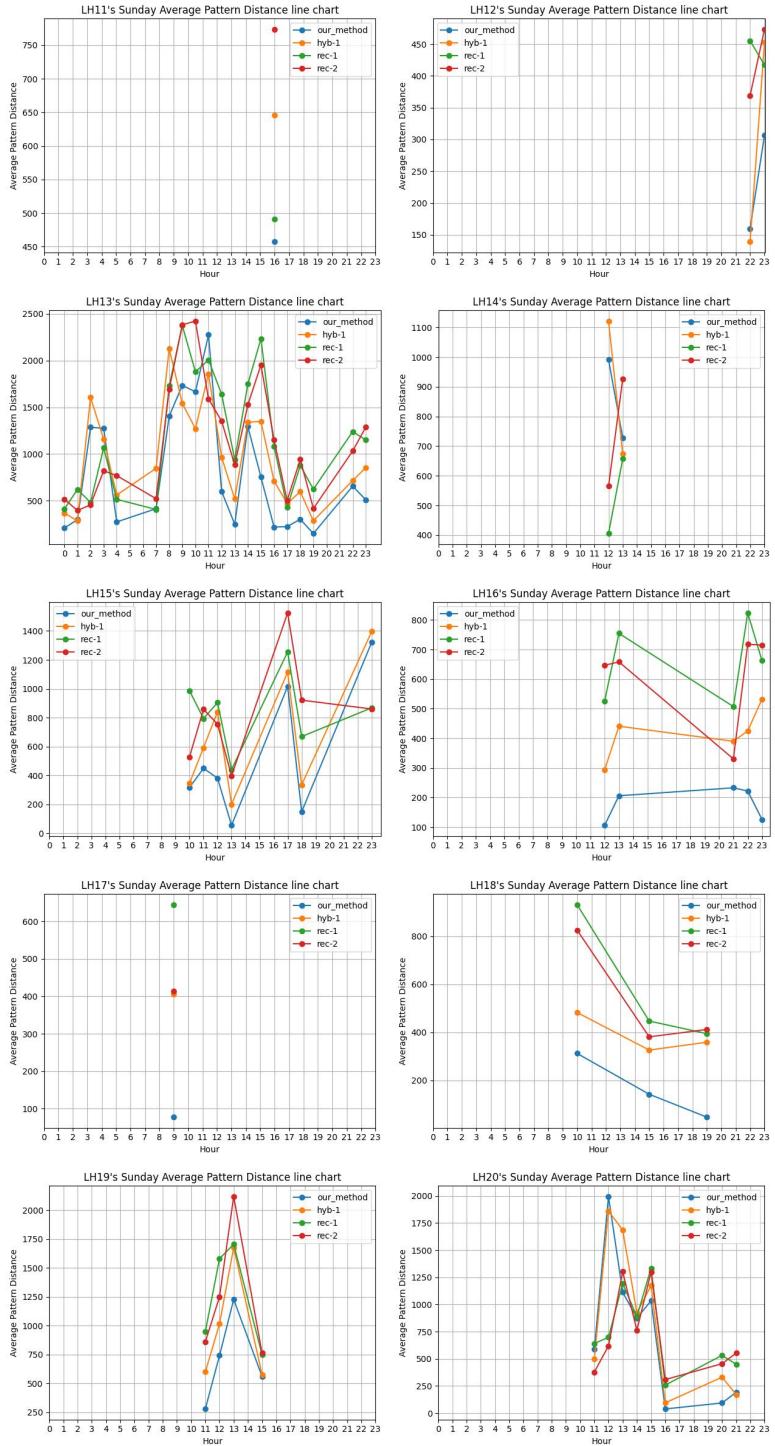


Figura 164: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Domenica per i file di cronologia da LH11 a LH20.



Figura 165: Andamento della Playlist Pattern Distance media di ogni metodo durante il giorno Domenica per i file di cronologia da LH21 a LH26.