



Laurea Triennale in Informatica – Università di Salerno  
Studenti : Panico Alessandro (matr. 0512119501)  
Parente Cataldo (matr. 0512119039)

---

## PlayMoodify

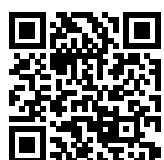


Panico Alessandro, Parente Cataldo

Gennaio 2026

[GitHub Repository](#)

[Notebook Colab](#)





# Sommario

## Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Descrizione del sistema</b>	<b>3</b>
2.1	Obiettivi . . . . .	3
2.2	Specifiche P.E.A.S. . . . .	3
2.3	Analisi del problema . . . . .	4
<b>3</b>	<b>Raccolta, analisi e preprocessing dei dati</b>	<b>4</b>
3.1	Scelta del dataset . . . . .	4
3.2	Analisi del dataset . . . . .	4
3.2.1	Feature del dataset . . . . .	4
3.3	Data Understanding . . . . .	6
3.3.1	Distribuzione delle feature nel dataset . . . . .	7
3.3.2	Matrice di correlazione . . . . .	8
3.3.3	Individuazione di righe non significative . . . . .	9
3.3.4	Individuazione di feature non significative . . . . .	9
3.4	Data Preparation . . . . .	10
3.4.1	Rimozione dei record non significativi . . . . .	10
3.4.2	Rimozione delle feature non significative . . . . .	10
3.4.3	Split del dataset . . . . .	12
3.4.4	Scaling delle feature . . . . .	12
3.4.5	Data Balancing . . . . .	13
3.5	Evaluation . . . . .	14
3.5.1	Valutazione del modello: Random Forest . . . . .	14
3.5.2	Valutazione del modello: Logistic Regression . . . . .	17
3.5.3	Confronto tra i modelli e scelta del modello ottimale . . . . .	18
3.5.4	Creazione del modello Random Forest . . . . .	19
<b>4</b>	<b>Piattaforma PlayMoodify</b>	<b>20</b>
4.1	Implementazione del modello . . . . .	20
4.1.1	Backend . . . . .	20
4.1.2	Frontend . . . . .	29
4.1.3	Interfaccia della piattaforma . . . . .	29
<b>5</b>	<b>Miglioramenti futuri</b>	<b>31</b>
5.1	Modello di classificazione . . . . .	31
5.2	Miglioramento efficienza in risposta . . . . .	31



## 1 Introduzione

Esistono numerose piattaforme online progettate per suggerire brani agli utenti sulla base dei loro ascolti, con l'obiettivo di aiutarli a scoprire nuova musica affine ai loro gusti. Tuttavia, nessuna di queste offre la possibilità di classificare il mood — ovvero l'umore — di una canzone o di un'intera playlist.

Per questo motivo si intende sviluppare PlayMoodify, un sistema capace di analizzare e classificare l'umore delle playlist degli utenti, contenente un sistema di raccomandazione che propone brani utili a colmare eventuali “gap” tra i diversi stati d'animo rappresentati nella loro libreria musicale.

## 2 Descrizione del sistema

### 2.1 Obiettivi

Lo scopo del nostro progetto è quello di realizzare un sistema in grado di:

- Valutare l'umore delle singole canzoni, al fine di esprimere una valutazione totale della playlist inserita dall'utente.
- Suggerire nuovi brani, suddivisi per umore, che rispecchiano le caratteristiche delle canzoni già presenti nella playlist dell'utente e che il sistema ha classificato in base al loro ”mood”.

### 2.2 Specifica P.E.A.S.

P.E.A.S.	
<b>Performance</b>	La misura di performance dell'agente è la capacità di quest'ultimo nel classificare correttamente le canzoni presenti nella playlist dell'utente e della classificazione della playlist generale.
<b>Environment</b>	L'ambiente in cui opera l'agente può essere suddiviso in 2 aree: <ul style="list-style-type: none"><li>• L'insieme di tutte le tracce disponibili reperibili tramite API, che restituiscono i metadati associati a tali tracce.</li><li>• Playlist inserita dall'utente: l'agente opera all'interno della playlist per classificare i brani contenuti in essa e la playlist stessa. Questo sotto-ambiente varia ogni qualvolta l'utente inserisce una nuova playlist.</li></ul> Pertanto, l'ambiente può essere classificato secondo i seguenti criteri: <ul style="list-style-type: none"><li>• <b>Parzialmente osservabile:</b> l'agente non conosce l'umore esatto della singola canzone finché non la analizza.</li><li>• <b>Non deterministico:</b> a parità di metadati audio, l'associazione tra brano e umore può non essere univoca, poiché la percezione emotiva della musica è soggettiva.</li><li>• <b>Dinamico:</b> l'ambiente cambia continuamente con l'inserimento di playlist differenti da parte dell'utente.</li><li>• <b>Discreto:</b> alle canzoni vengono assegnati dei metadati strutturati tramite le API.</li></ul>
<b>Actuators</b>	L'agente agisce producendo una predizione dell'umore associato a ciascun brano e, a livello aggregato, alla playlist dell'utente..
<b>Sensors</b>	I sensori dell'agente sono costituiti dai metadati audio estratti dalle canzoni tramite API, che rappresentano l'input osservabile utilizzato per la classificazione.



## 2.3 Analisi del problema

La problematica principale da risolvere riguarda la classificazione dell'umore delle canzoni presenti nella playlist, al fine di valutare correttamente l'umore di quest'ultima. Per poterlo fare, abbiamo deciso di utilizzare tecniche di machine learning che permettessero di effettuare la classificazione dei brani attraverso l'uso di etichette. L'idea di base è, dunque, quella di richiedere tramite un'API i dati riguardanti le singole canzoni della playlist e, in base a questi, classificarle con l'etichetta più adatta. Terminata la classificazione delle singole canzoni, si valuta l'umore della playlist totale calcolando media e modi degli umori contenuti al suo interno.

Per generare nuovi suggerimenti, il sistema analizza la playlist dell'utente e suggerisce, per ogni mood, una canzone simile a quelle contenute nella playlist.

# 3 Raccolta, analisi e preprocessing dei dati

## 3.1 Scelta del dataset

Per quanto riguarda la scelta del dataset necessario per lo sviluppo del modello, le strade possibili erano molteplici:

- **Creare un dataset artificialmente:** ottenere metadati sintetici tramite l'utilizzo di tecnologie di supporto secondarie.
- **Creare un dataset da zero:** ottenere metadati di  $n$  brani tramite API e classificarle manualmente affinchè non ci fossero errori di classificazione.
- **Reperire un dataset online:** in questo caso si ottiene un dataset dal web con brani etichettati e suddivisi in modo corretto in base ai metadati, andando ad evidenziare le feature caratterizzanti per il nostro problema e effettuare sul dataset operazioni di preprocessing e feature engineering.

La nostra scelta è dipesa principalmente da alcune principali problematiche che sorgono con le prime due soluzioni. Entrambe le soluzioni potrebbero, infatti, portare a dataset di limitata dimensione, oppure, dati inconsistenti dovuti ad una sintetizzazione poco veritiera dei dati. Un'altra problematica evidente risiede nella quantità di tempo necessaria per la creazione del dataset in questione.

Proprio queste motivazioni ci hanno portato a ricercare online la presenza di dataset contenenti metadati già etichettati, in modo da rendere lo sviluppo del modello molto più efficace. Dopo svariate ricerche, tramite la piattaforma **Kaggle**, siamo riusciti ad ottenere un dataset strutturato correttamente per le nostre esigenze.

## 3.2 Analisi del dataset

Il dataset selezionato per lo sviluppo del nostro progetto è reperibile su **Kaggle**, tramite questo [link](#), all'interno di quest'ultimo sono presenti circa **280mila** brani etichettati in base ai rispettivi metadati e suddivisi in **4 etichette**. Il dataset presenta **13 feature**, alcune di queste non significative per il nostro progetto e dunque trascurabili. Inoltre c'è uno sbilanciamento delle etichette e feature non normalizzate, le quali verranno normalizzate per rendere efficiente l'apprendimento del modello. Infine, sono necessarie operazioni per l'eliminazione di feature non necessarie.

### 3.2.1 Feature del dataset

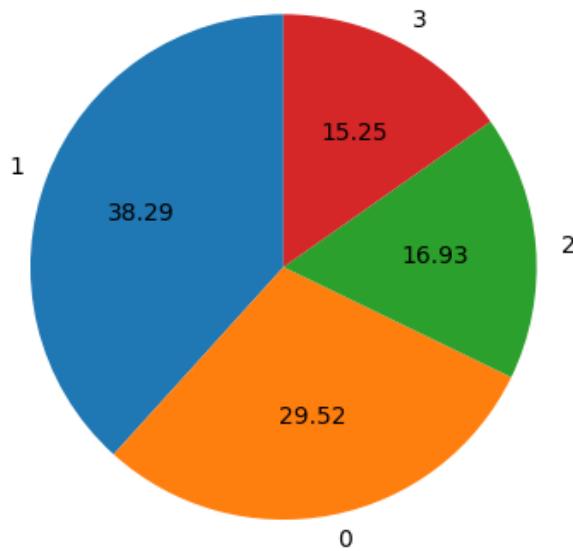
FEATURE	
Labels	Sono presenti 4 labels (etichette) che rappresentano l'umore di una determinata canzone. In particolare, queste sono: {'sad': 0, 'happy': 1, 'energetic': 2, 'calm': 3}.



FEATURE	
<b>Acousticness</b>	E' una misura che indica il livello di acusicità della canzone. I valori vanno da 0.0 a 1.0, e un valore di acousticness pari a 1.0 rappresenta un'alta probabilità che la traccia sia acustica.
<b>Danceability</b>	E' una misura che descrive quanto un brano sia adatto per ballare, in base ad una combinazione di elementi musicali quali: tempo, stabilità ritmica, intensità del beat e regolarità generale. Il valore può variare tra 0.0 e 1.0, il primo indica che il brano è poco ballabile, mentre il secondo indica che è molto ballabile.
<b>Energy</b>	Rappresenta una misura percettiva di intensità e attività. Tipicamente, le tracce energetiche risultano veloci, forti e rumorose. Ad esempio, il Death Metal ha un'energia elevata, mentre un Preludio di Bach ha un valore di energia molto basso. Le caratteristiche percettive che contribuiscono a questo attributo includono: gamma dinamica, intensità percepita, timbro, velocità di attacco e entropia generale. La scala dei valori va da 0.0 a 1.0, il primo indica una traccia poco energetica, mentre il secondo una traccia molto energetica.
<b>Instrumentalness</b>	E' una misura che prevede se una traccia non contiene parti vocali. I suoni come "ooh" e "aah" sono considerati strumentali in questo contesto. Le tracce Rap o Spoken Word sono chiaramente vocali. Più il valore di Instrumentalness è alto, maggiore sarà la probabilità che la traccia non contenga parti vocali.
<b>Liveness</b>	E' una misura che rileva la presenza di pubblico all'interno della registrazione. Alti valori di Liveness indicano una maggiore probabilità che il brano sia stato eseguito dal vivo. La scala dei valori va da 0.0 a 1.0, il primo indica che il brano ha poca probabilità che sia stato eseguito dal vivo, il secondo invece indica che il brano è stato quasi sicuramente eseguito dal vivo.
<b>Loudness</b>	E' una misura che indica il volume complessivo di una traccia in decibel (dB). Il valore di Loudness è calcolato come media sull'intera traccia ed è utile per confrontare il volume relativo delle varie tracce. I valori di questa misura variano solitamente tra -60 dB e 0 dB.
<b>Speechiness</b>	E' una misura che rileva la presenza di parti parlate in una traccia. I valori variano tra 0.0 e 1.0 e, più un brano è parlato, più il valore è vicino ad 1.0. Valori superiori a 0.66 rappresentano tracce che sono probabilmente composte da parti parlate. Valori compresi tra 0.33 e 0.66 descrivono brani che possono contenere sia parti parlate che parti musicali. Infine, valori al di sotto di 0.33 rappresentano tracce non parlate composte di sola musica.
<b>Valence</b>	E' una misura che varia tra 0.0 e 1.0 e descrive la positività trasmessa dalla canzone. Tracce con alto valore di Valence suonano molto positive (felici, euforiche), mentre tracce con basso valore di Valence risultano più negative (tristi, depresse).
<b>Tempo</b>	Il tempo complessivo di un brano misurato in BPM (battiti al minuto). Indica la velocità o il ritmo del brano.
<b>Duration</b>	Indica la durata del brano in millisecondi.
#	Indica l'ID della canzone all'interno del dataset.
<b>Spec_rate</b>	Indica la frequenza di campionamento del brano.

### 3.3 Data Understanding

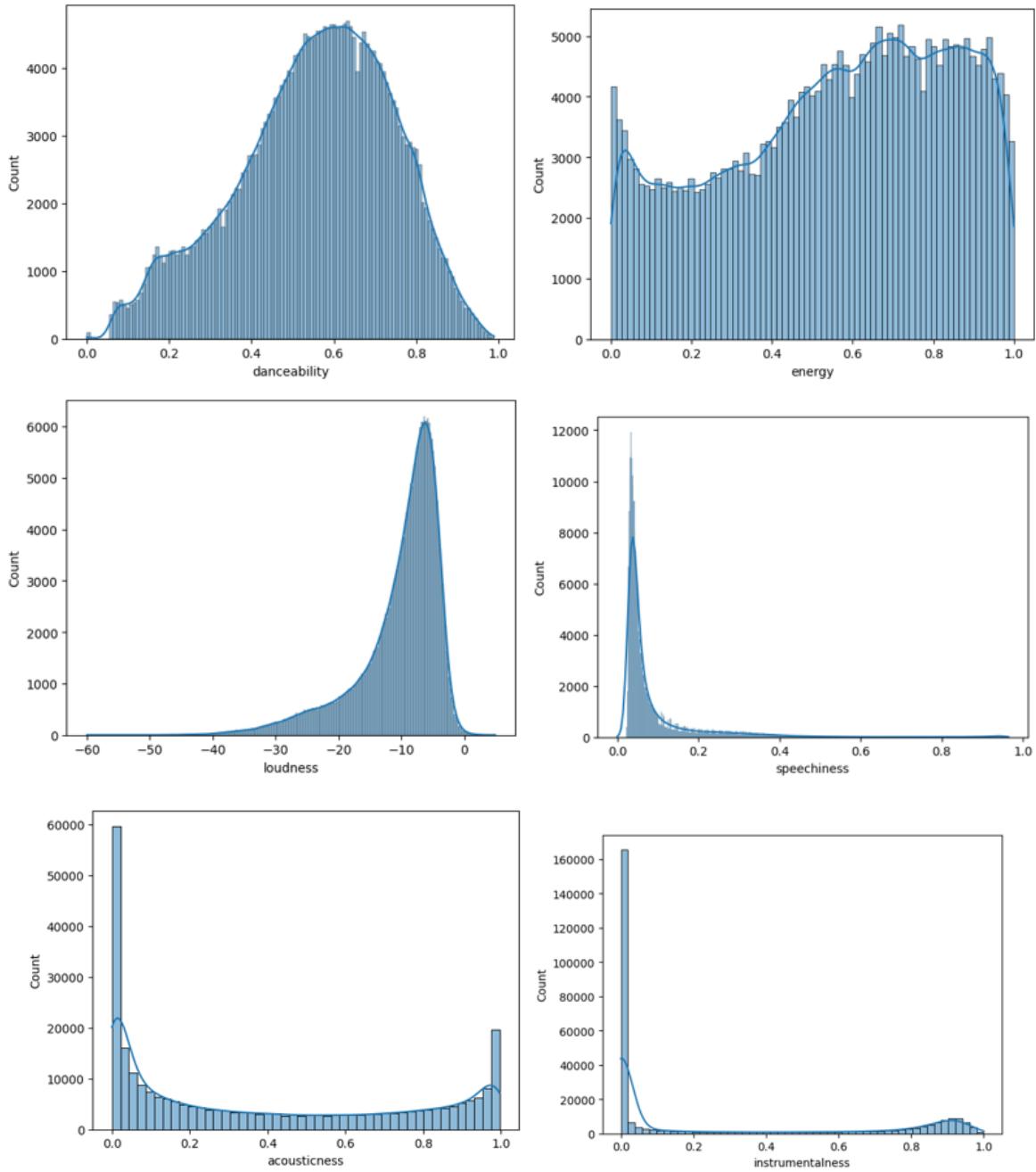
Dopo aver approfondito il significato delle singole feature del dataset in esame, abbiamo evidenziato quelle che sono le criticità di quest'ultimo, come la presenza di righe non significative, la presenza di feature non significative, lo sbilanciamento delle classi e la necessità di feature scaling. Di seguito è riportato nel dettaglio la distribuzione delle etichette all'interno del dataset.

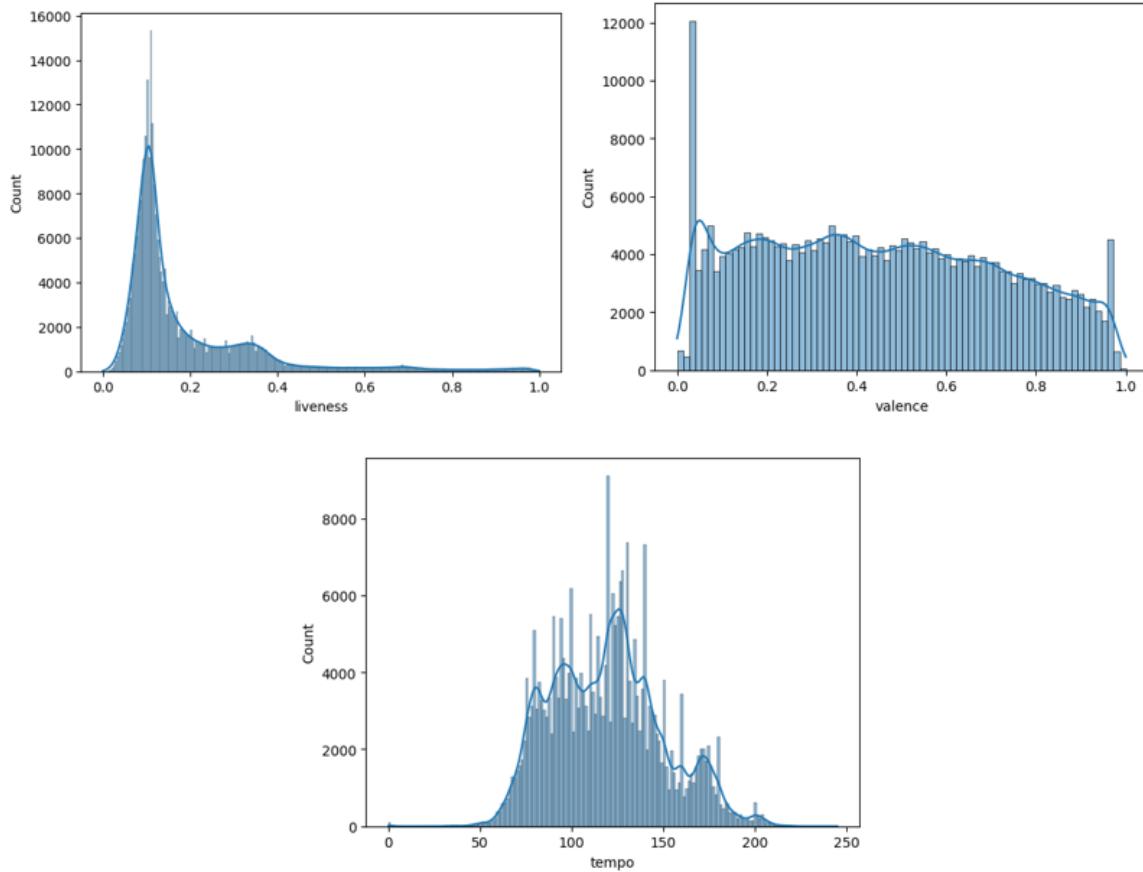


Analizzando tale distribuzione, possiamo notare che è presente un lieve sbilanciamento delle classi. Pertanto, risulta utile un bilanciamento di queste ultime nelle fasi successive.

Per quanto riguarda le altre criticità individuate, sono elencate nei paragrafi successivi le tecniche risolutive da noi adottate.

### 3.3.1 Distribuzione delle feature nel dataset



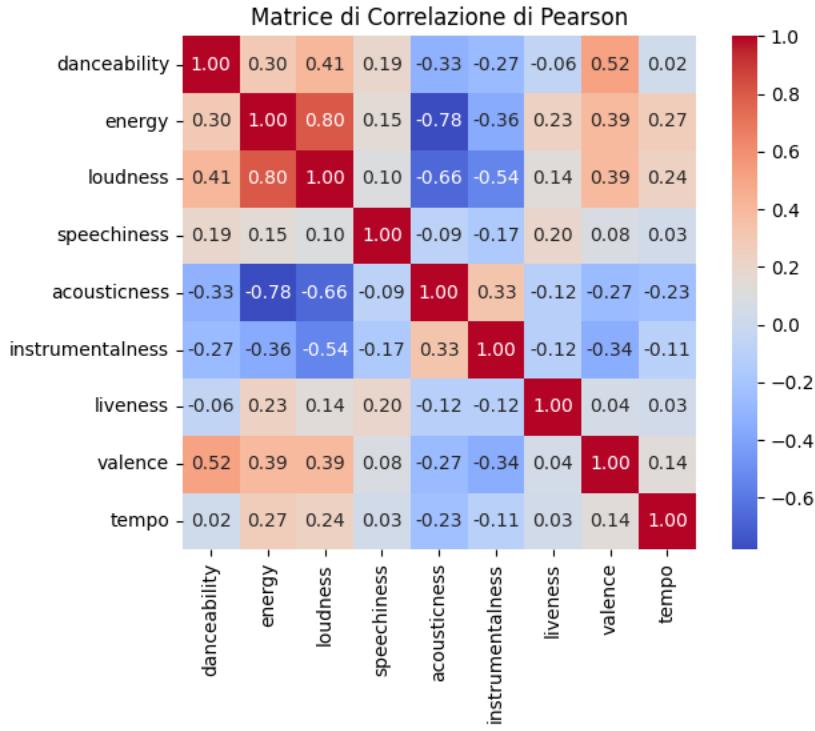


Dall'osservazione dei grafici sopra riportati possiamo notare che:

- La feature Energy risulta essere ben distribuita nella scala tra 0 e 1;
- Le feature Speechiness e Instrumentalness hanno un alto picco su 0;
- Le feature Acousticness e Valence hanno picchi sia su 0 che su 1;
- La feature Danceability ha un picco nei valori che vanno tra 0.4 e 0.8;
- La feature Liveness ha un picco tra 0 e 0.2, per poi decrescere drasticamente fino ad 1.

### 3.3.2 Matrice di correlazione

Per poter analizzare le correlazioni tra le varie feature, abbiamo deciso di calcolare una Matrice di Correlazione attraverso il metodo Pearson. Di seguito è riportata tale Matrice.



Dall’analisi della Matrice di Correlazione emerge un legame molto stretto tra le feature **energy** e **loudness**. In fase di Data Preparation vedremo il modo in cui abbiamo gestito tale correlazione.

### 3.3.3 Individuazione di righe non significative

Prima di andare a rimuovere le feature non significative, poichè queste possono essere utili nella fase di data cleaning, le utilizziamo per rilevare le righe non significative, ovvero quelle che non rappresentano canzoni, ma ad esempio suoni bianchi e/o podcast. La rimozione delle righe è basata sui seguenti aspetti:

- **Presenza di brani analizzati in modo non corretto:** ad esempio, canzoni con Spec rate pari a 0 o Tempo pari a 0, che identifica canzoni campionate in modo errato (`spec_rate = 0.0`) oppure tracce contenenti suoni bianchi, poesie o letture di testo (`Tempo = 0.0`).
- **Presenza di tracce che non rappresentano brani:** ad esempio, canzoni con una durata superiore ai 10 minuti potrebbero essere podcast, letture di parti di poesie e/o testi in generale ( $Duration \geq 10 \text{ min}$ ).

### 3.3.4 Individuazione di feature non significative

All’interno del dataset sono presenti delle feature che hanno un impatto quasi nullo sulla classificazione dell’umore dei vari brani. Le feature in questione sono:

- `#`, in quanto è un semplice ID utilizzato all’interno del dataset per l’identificazione delle righe;
- **Spec\_rate**, che non è un descrittore percettivo, ma indica soltanto la frequenza di campionamento del brano. Quest’ultimo assume valori estremamente bassi, il che potrebbe portare all’aggiunta di rumore nel dataset, aumentando la complessità nella classificazione;

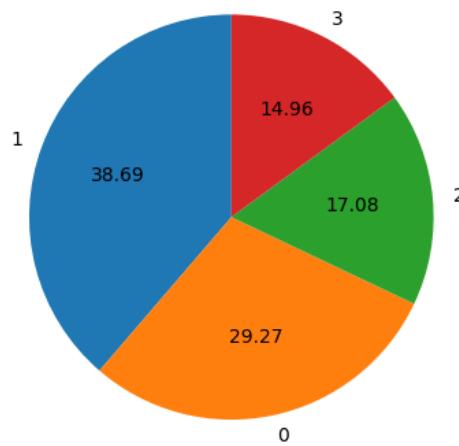
- **Duration**, anche questo, come lo spec\_rate, non è un descrittore percettivo, ma indica soltanto la durata del brano in millisecondi (ms).
- **Loudness**. Abbiamo deciso di escludere **loudness** per ridurre la ridondanza dei dati, privilegiando **energy** in quanto variabile più indicativa della componente emotiva del brano, rispetto al dato prettamente tecnico del volume.

## 3.4 Data Preparation

### 3.4.1 Rimozione dei record non significativi

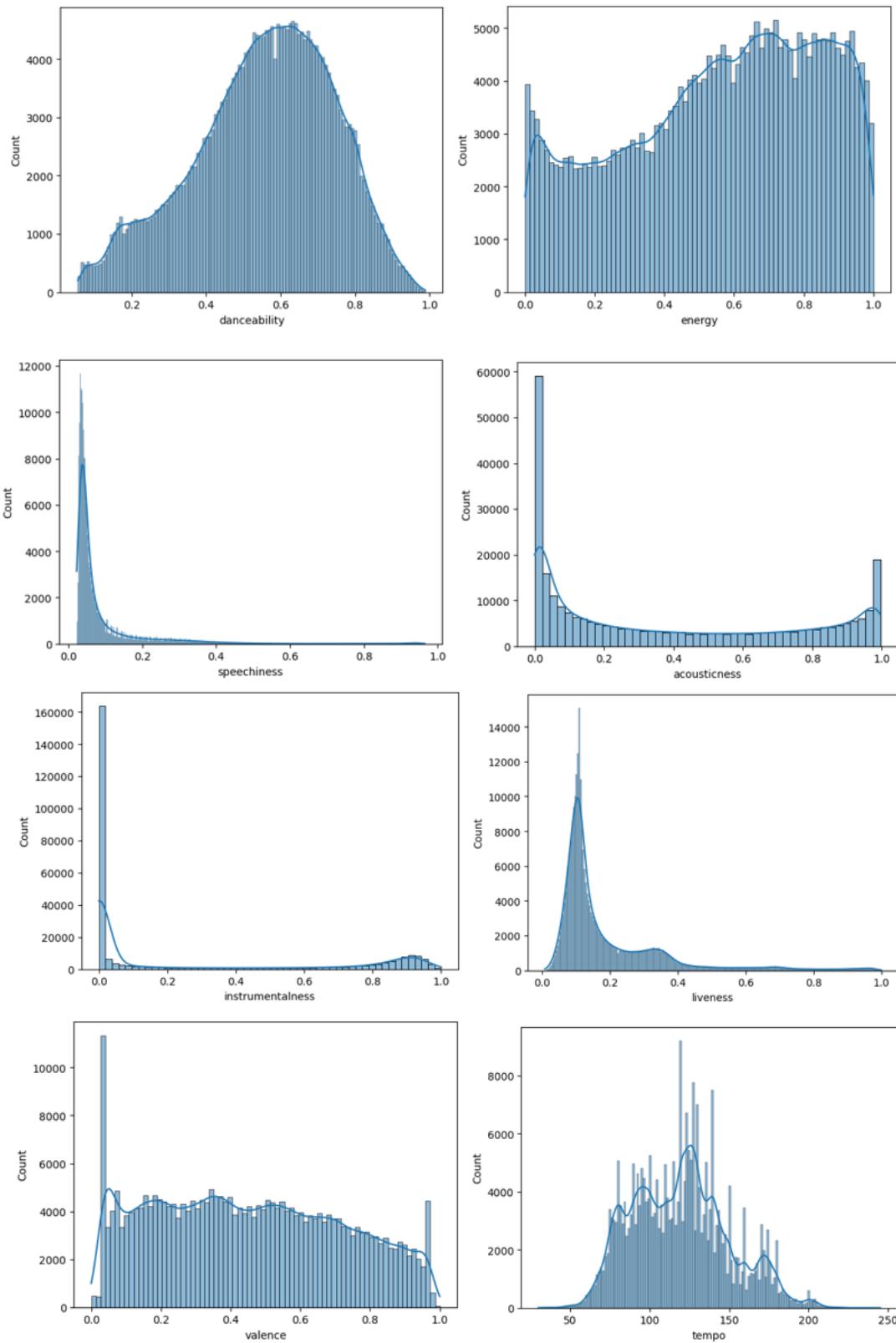
Una volta trovati i record non significativi all'interno del dataset, procediamo alla rimozione di questi.

Di seguito è riportato il grafico che descrive la distribuzione delle etichette del dataset dopo la rimozione.

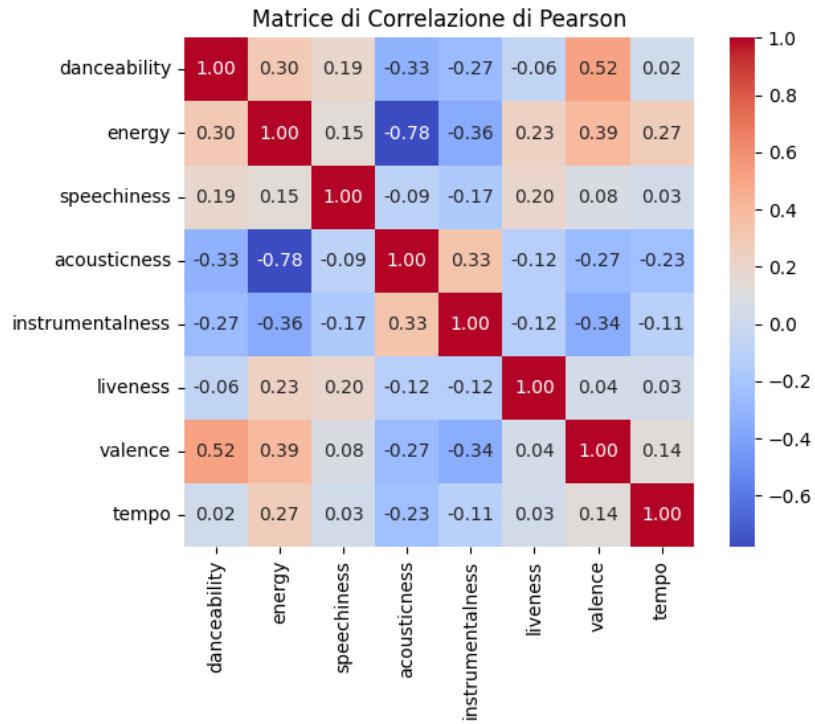


### 3.4.2 Rimozione delle feature non significative

In modo analogo a come fatto per i record, andiamo a rimuovere le feature non significative presenti nel dataset, precedentemente individuate (**#**, **Spec\_rate**, **Duration**, **Loudness**). I grafici riguardanti la distribuzione delle feature dopo la rimozione dei record e la Matrice di Correlazione ricalcolata sono di seguito indicati.



Dopo la rimozione della feature **loudness** abbiamo deciso di ricalcolare la Matrice di Correlazione per verificare la presenza di ulteriori correlazioni tra feature:



Possiamo, quindi, notare l'assenza di feature strettamente legate tra loro. Pertanto possiamo proseguire con la Data Preparation.

### 3.4.3 Split del dataset

Per la fase di splitting del dataset, si è scelto di applicare il Principio di Pareto, adottando una ripartizione 80/20: l'80% dei dati è stato destinato all'addestramento (*training set*), mentre il restante 20% è stato riservato alla validazione (*test set*). Lo split è stato effettuato con **random\_state = 42** che permette di effettuare uno split pseudo-casuale e riproducibile tramite una funzione deterministica.

```

1 # Effettuiamo la suddivisione del dataset
2 from re import X
3 X = df.drop("labels", axis=1)
4 y = df["labels"]
5
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
7 random_state=42, stratify=y)
# Utilizziamo random_state = 42, poiche' permette di effettuare uno split pseudo-
casuale e riproducibile tramite una funzione deterministica

```

### 3.4.4 Scaling delle feature

Dopo la suddivisione del dataset in training set e test set, è stata valutata l'applicazione delle tecniche di scaling delle feature come parte della fase di preprocessing.



**Scaling Random Forest** Per la pipeline basata su Random Forest, non è stato applicato alcuno scaling delle feature. Questo perchè è un algoritmo tree-based, il cui processo decisionale si basa su split a soglia sulle singole variabili. Di conseguenza, il modello non migliora le proprie performance in presenza di standardizzazione o normalizzazione delle feature. Inoltre, il mantenimento delle scale originali delle feature consente una maggiore interpretabilità delle soglie di split apprese dal modello e contribuisce a ridurre la complessità dell'intera pipeline di preprocessing.

**Scaling Logistic Regretion** Per quanto riguarda la pipeline con Logistic Regression, quest'ultima necessita di scaling delle feature, poichè è un modello gradient-based e basato su ottimizzazione numerica. Dunque, lo scaling delle feature è necessario poichè evita effetti negativi come:

- Coefficienti distorti
- Instabilità numerica
- Interpretazione errata delle feature
- Performance peggiori

In particolare, abbiamo deciso di utilizzare la **standardizzazione**. Algoritmi gradient-based, come la Logistic Regression, risultano essere instabili in presenza di feature con ordini di grandezze diversi. Pertanto, la scelta più adeguata risulta essere la standardizzazione, poichè riduce il divario tra le grandezze di feature differenti, facendo sì che queste abbiano media pari a 0 e deviazione standard pari a 1. In questo modo, si evita la situazione in cui feature con scale maggiori dominino la stima dei pesi.

```
1 # Implementiamo la standardizzazione
2 scaler = StandardScaler()
3
4 # Applichiamo la standardizzazione sul training set
5 X_train_scaled = scaler.fit_transform(X_train)
6
7 # Applichiamo separatamente la standardizzazione sul test set corrispondente
8 X_test_scaled = scaler.transform(X_test)
9
```

### 3.4.5 Data Balancing

**Bilanciamento Random Forest** Per la pipeline basata su Random Forest, la problematica dello sbilanciamento delle classi è stata affrontata mediante l'utilizzo di pesi di classe, impostando il parametro `class_weight='balanced'`. Questo approccio consente nel modificare la funzione di costo del modello, assegnando un peso maggiore alle classi minoritarie, incentivando così l'algoritmo a prestare maggiore attenzione a tali classi durante la fase di costruzione degli alberi.

Tale scelta è motivata dalla natura tree-based del modello Random Forest, che permette una gestione dello sbilanciamento attraverso criteri di split sensibili ai pesi delle classi. L'utilizzo dei pesi di classe riduce inoltre il rischio di overfitting che può essere introdotto da tecniche di oversampling sintetico, come SMOTE, che generano campioni artificiali potenzialmente ridondanti o non rappresentativi della distribuzione reale dei dati.

L'utilizzo di SMOTE è stato considerato come approccio alternativo, ma non adottato come soluzione primaria per la pipeline Random Forest, dando priorità ad una strategia più coerente con le caratteristiche del modello, al fine di preservare la capacità di generalizzazione sul test set.



```
1      # Inserito durante la creazione del modello
2      class_weight='balanced', # da peso maggiore alle classi minoritarie, bilancia
3      senza inserimento di valori sintetici
```

**Bilanciamento Logistic Regression** Nel caso della Logistic Regression, a differenza dei modelli basati su Random Forest, la gestione dello sbilanciamento delle classi riveste un ruolo particolarmente critico, in quanto il modello è maggiormente sensibile alla distribuzione delle etichette durante la fase di ottimizzazione. Per questo motivo, è stata adottata la tecnica **SMOTE** (*Synthetic Minority Over-sampling Technique*), che consente di generare campioni sintetici per le classi minoritarie.

L'applicazione di SMOTE è stata limitata esclusivamente al training set, al fine di evitare fenomeni di data leakage e garantire una corretta valutazione delle performance sul test set. Tale strategia consente di riequilibrare la distribuzione delle classi durante l'addestramento, migliorando la capacità del modello di apprendere pattern rappresentativi anche per le classi meno frequenti e aumentando la sensibilità del classificatore nel riconoscimento delle classi minoritarie.

```
1      # Implementazione della SMOTE
2
3      smote_lr = SMOTE(random_state=42)
4      X_train_smote_lr, y_train_smote_lr = smote_lr.fit_resample(X_train_scaled,
5      y_train)
```

## 3.5 Evaluation

### 3.5.1 Valutazione del modello: Random Forest

Di seguito sono riportate le principali misure di performance e la Confusion Matrix relative al modello Random Forest. Il modello ha ottenuto un valore di accuracy pari a circa 0.919, indicando un'elevata capacità di classificazione complessiva sul test set.

È stato inoltre calcolato il valore dell'**OOB score** (*Out-Of-Bag score*), che rappresenta una stima interna delle performance di generalizzazione del modello Random Forest, ottenuta sfruttando i campioni non inclusi in quelli utilizzati per l'addestramento dei singoli alberi. Tale metrica consente di stimare le performance del modello senza la necessità di introdurre un set di validazione separato.

L'OOB Score è stato utilizzato come indicatore preliminare della capacità di generalizzazione e come strumento di monitoraggio del rischio di overfitting, permettendo di valutare la qualità del modello già in fase di training. Questo approccio rappresenta un'alternativa computazionalmente efficiente alla cross-validation.

```
1      # Effettuiamo una stima delle performance solo sul training set
2      rf_clf.fit(X_train, y_train)
3      print("OOB score:", rf_clf.oob_score_)
4
5      # Performance confrontando i valori predetti con quelli del test set
6      y_pred_rf = rf_clf.predict(X_test)
7      report_dict_rf = classification_report(y_test, y_pred_rf, output_dict=True)
8
9      print("Performance del modello:")
10
11     # Accuracy del modello
12     acc_rf = accuracy_score(y_test, y_pred_rf)
13     print(f"Accuracy: {acc_rf:.3f}")
```

```

15      # Selezione solo delle classi (esclude accuracy)
16      per_class_rf_df = report_rf_df.iloc[:,-3][['precision', 'recall', 'f1-score']]
17      # Visualizzazione Confusion Matrix
18      class_names = ['sad', 'calm', 'happy', 'energetic']
19
20      disp = ConfusionMatrixDisplay.from_estimator(
21          rf_clf,
22          X_test,
23          y_test,
24          display_labels=class_names,
25          normalize='true',
26          cmap='Blues'
27      )
28

```

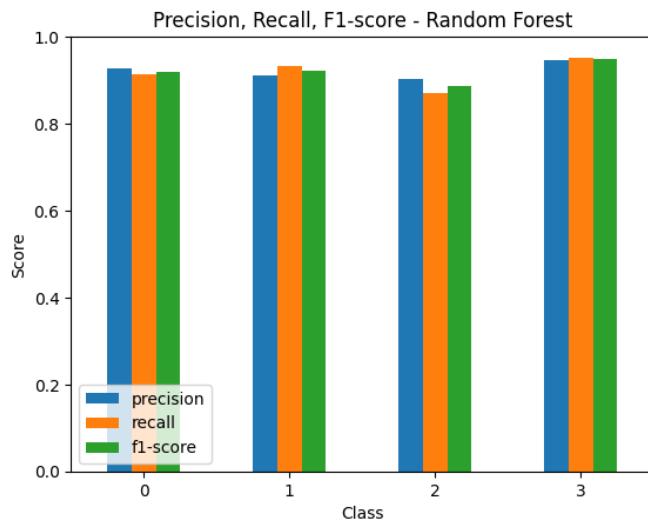
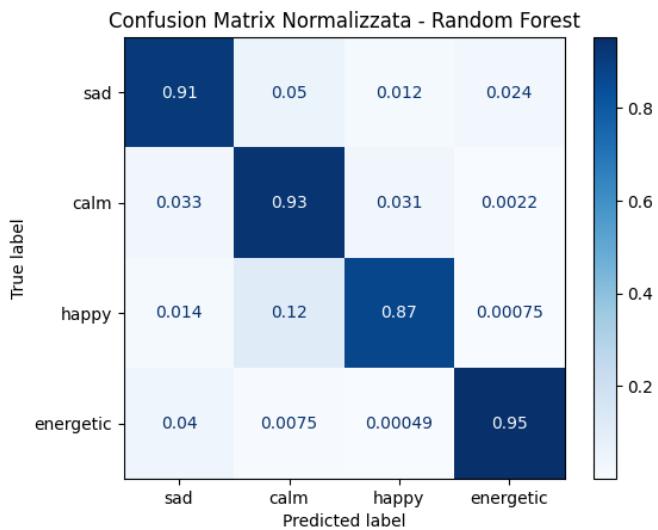


Tabella 2: Tabella specifica delle misure di prestazione

Classe	Precision	Recall	F1-score
sad: 0	0.927	0.914	0.920
happy: 1	0.910	0.934	0.922
energetic: 2	0.904	0.870	0.886
calm: 3	0.947	0.952	0.949



### 3.5.2 Valutazione del modello: Logistic Regression

Di seguito sono riportate le misure di performance e la Confusion Matrix riguardanti il modello Logistic Regression.

```

1 # Performance confrontando i valori predetti con quelli del test set
2 y_pred_lr = lr_clf.predict(X_test_scaled)
3 report_dict_lr = classification_report(y_test, y_pred_lr, output_dict=True)
4
5 print("Performance del modello:")
6
7 # Accuracy del modello
8 acc_lr = accuracy_score(y_test, y_pred_lr)
9 print(f"Accuracy: {acc_lr:.3f}")
10
11 # Conversione in DataFrame
12 report_lr_df = pd.DataFrame(report_dict_lr).T
13
14 # Selezione solo delle classi (esclude accuracy)
15 per_class_lr_df = report_lr_df.iloc[:-3][['precision', 'recall', 'f1-score']]
16 # Visualizzazione Confusion Matrix
17 class_names = ['sad', 'calm', 'happy', 'energetic']
18
19 disp = ConfusionMatrixDisplay.from_estimator(
20     rf_clf,
21     X_test,
22     y_test,
23     display_labels=class_names,
24     normalize='true',
25     cmap='Blues'
26 )
27

```

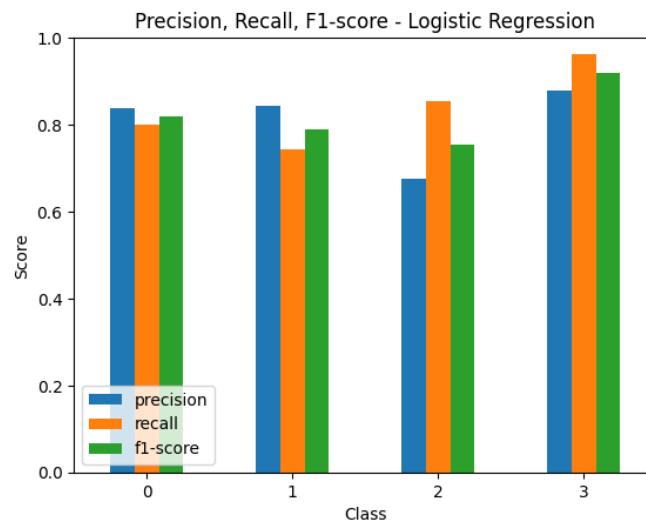
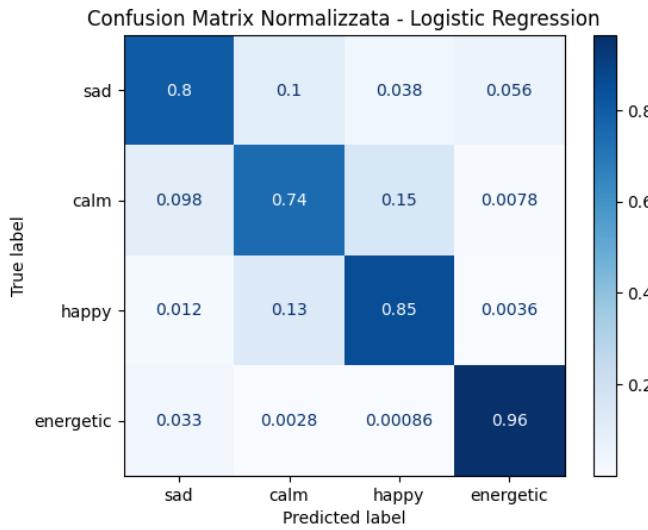


Tabella 3: Tabella specifica delle misure di prestazione

Classe	Precision	Recall	F1-score
<i>sad</i> : 0	0.839	0.801	0.820
<i>happy</i> : 1	0.844	0.743	0.790
<i>energetic</i> : 2	0.677	0.854	0.755
<i>calm</i> : 3	0.879	0.964	0.919

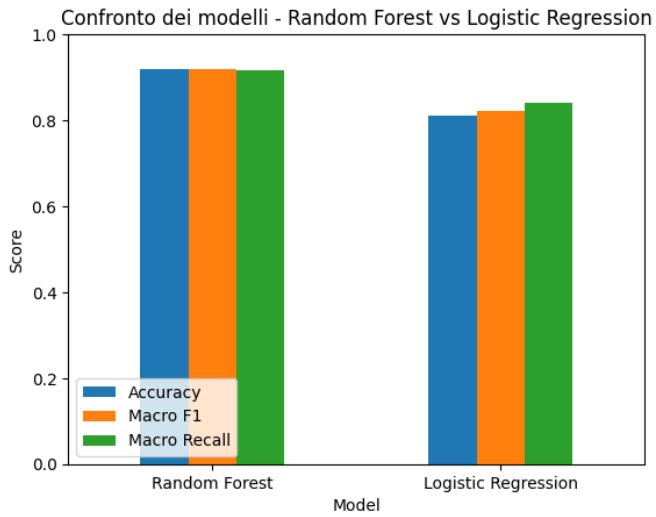


### 3.5.3 Confronto tra i modelli e scelta del modello ottimale

Il confronto tra le due pipeline ha evidenziato prestazioni superiori del modello Random Forest rispetto alla Logistic Regression su tutte le principali metriche di valutazione, ovvero **Accuracy**, **Macro F1-score** e **Macro Recall** (Tabella 4). In particolare, Random Forest ha mostrato una migliore capacità di riconoscere correttamente le classi minoritarie e di modellare relazioni non lineari tra le feature audio. Alla luce di questi risultati, Random Forest è stato selezionato come modello finale per il task di classificazione dell'umore delle canzoni.

Tabella 4: Tabella di confronto tra modelli

Modello	Accuracy	Macro F1-Score	Macro Recall
<i>Random Forest</i>	0.919	0.919	0.917
<i>Logistic Regression</i>	0.812	0.821	0.840



### 3.5.4 Creazione del modello Random Forest

```
1 # Creiamo un modello di Random Forest
2 rf_clf = RandomForestClassifier(
3     n_estimators=300, # numero di alberi decisionali che verrano creati
4     random_state=42,
5     class_weight='balanced', # da peso maggiore alle classi minoritarie,
6     bilancia senza inserimento di valori sintetici
7     n_jobs=-1, # velocizza il training utilizzando tutti i core, evitando
8     approccio sequenziale che sarebbe piu' lento
9     oob_score=True
)
```



## 4 Piattaforma PlayMoodify

### 4.1 Implementazione del modello

Durante l'implementazione del modello scelto per la classificazione dell'umore dei brani, ci sono stati alcuni problemi riguardanti la sua dimensione. Il modello, infatti, presentava una dimensione elevata e, sebbene avessimo potuto mantenerlo in locale, abbiamo deciso di caricarlo su una piattaforma online dal nome "**Hugging Face**" nella repository [Alepn04/PlayMoodifyModel](#). Tale piattaforma può essere definita come la "GitHub dei modelli ML" che, a differenza di GitHub, permette il caricamento online di modelli con dimensioni importanti.

Per quanto riguarda la struttura della piattaforma, abbiamo:

- Un **Backend** realizzato interamente in Python che si occupa del caricamento del modello da Hugging Face, dell'estrazione dei brani da un link di una playlist Spotify, della ricerca degli identificativi per ogni brano (*UUID*), della ricerca dei metadati audio dei vari brani tramite *UUID*, della valutazione dell'umore dei singoli brani tramite il modello, dei calcoli riguardanti moda e media degli umori e della raccomandazione di brani tramite API per ogni umore.
- Un **Frontend** realizzato, invece, con React in modo da ottenere un'interfaccia leggera e pulita, che si occupa soltanto di prendere in input un link di una playlist Spotify e di mostrare all'utente i risultati delle predizioni del modello.

#### 4.1.1 Backend

Il Backend è strutturato come segue:

- *linktocsvconverter.py* : si occupa di convertire un link di una playlist di Spotify in un file .csv contenente la coppia (title,artist) per ogni canzone presente nella playlist.

```
1  def spotify_playlist_to_csv(playlist_url, output_csv_path):
2      client = SpotifyClient()
3      playlist = client.get_playlist_info(playlist_url)
4
5      tracks = []
6
7      for track in playlist.get("tracks", []):
8          # Skip elementi non musicali o incompleti
9          if not track.get("artists") or not track.get("name"):
10              continue
11
12          title = track.get("name", "").strip()
13          artist = ", ".join([a.get("name", "").strip() for a in track.get("artists",
14          [])])
15
16          tracks.append({
17              "title": title,
18              "artist": artist
19          })
20
21      # Scrittura CSV
22      fieldnames = ["title", "artist"]
23
24      with open(output_csv_path, mode="w", newline="", encoding="utf-8") as f:
25          writer = csv.DictWriter(f, fieldnames=fieldnames)
26          writer.writeheader()
```



```
26     writer.writerows(tracks)
27
28     return output_csv_path
```

Questa funzione è stata realizzata manualmente poichè, durante la realizzazione del progetto, l'API ufficiale di Spotify (WEB API) era in manutenzione, pertanto inutilizzabile.

- **uuidfromname.py** : si occupa di recuperare, per ogni brano contenuto nel file .csv generato da *linktocsvconverter.py*, l'UUID corrispondente se presente, restituendo un nuovo file .csv contenente la tupla (title,artist,UUID). Per rendere più efficiente l'esecuzione del codice, abbiamo deciso di utilizzare funzioni parallele (linea 68-78).

```
1      # Ricerca l'UUID di una canzone tramite il titolo e l'artista su SoundCharts
2      API
3      def get_uuid_from_soundcharts(song_name: str, artist_name: Optional[str] = None
4          ) -> Optional[str]:
5          try:
6              query = song_name
7              if artist_name:
8                  query = f"{song_name} {artist_name}"
9
10             # Correggiamo l'encoding della query per URL
11             encoded_query = quote(query)
12
13             headers = {
14                 'x-app-id': '#####',
15                 'x-api-key': '#####',
16             }
17
18             params = {
19                 'offset': '0',
20                 'limit': '20',
21             }
22
23             response = requests.get(url, headers=headers, params=params, timeout=2)
24
25             if response.status_code == 200:
26                 data = response.json()
27
28                 # Estrai l'UUID dal primo risultato
29                 if data and "items" in data and len(data["items"]) > 0:
30                     uuid = data["items"][0].get("uuid")
31                     return uuid
32                 else:
33                     return None
34             else:
35                 return None
36
37             except Exception as e:
38                 print(f"Errore nella ricerca: {e}")
39                 return None
```



```
40
41
42     # Elabora una singola traccia e cerca l'UUID
43     def process_single_track(row: Dict) -> Optional[Dict[str, str]]:
44
45         title = row.get('title', '').strip()
46         artist = row.get('artist', '').strip()
47
48         uuid = get_uuid_from_soundcharts(title, artist)
49
50         return {
51             'title': title,
52             'artist': artist,
53             'uuid': uuid if uuid else 'N/A'
54         }
55
56     # Ricerca gli UUID per tutte le tracce in un CSV
57     def process_csv_and_get_uuids(csv_file_path: str, output_file_path: str) ->
58     List[Dict[str, str]]:
59
60         # Leggiamo il CSV di input
61         rows = []
62         with open(csv_file_path, 'r', encoding='utf-8') as infile:
63             reader = csv.DictReader(infile)
64             rows = list(reader)
65
66         results = []
67
68         # Utilizziamo ThreadPoolExecutor per elaborare in parallelo
69         with ThreadPoolExecutor(max_workers=6) as executor:
70             tasks = [
71                 executor.submit(process_single_track, row)
72                 for row in rows
73             ]
74
75             # Raccogliamo i risultati man mano
76             for future in as_completed(tasks):
77                 result = future.result()
78                 if result:
79                     results.append(result)
80
81         # Scrittura del CSV....
82
83         return results
```

- **soundchart.py** : si occupa di recuperare il file .csv generato da *uuidfromname.py* e, per ogni canzone, ricava le feature audio corrispondenti, caricandole su un file .csv contenente (title,artist,UUID, danceability, energy, speechiness, acousticness,instrumentalness,liveness,valence,tempo). Anche in questo caso, per motivi di efficienza, abbiamo utilizzato funzioni parallele (linea 71-80).

```
1     def get_audio_features_by_uuid(uuid: str) -> Optional[Dict]:
2         try:
3             url = f"https://customer.api.soundcharts.com/api/v2.25/song/{uuid}"
```



```
5         headers = {
6             'x-app-id': '#####',
7             'x-api-key': '#####',
8         }
9
10        response = requests.get(url, headers=headers, timeout=2)
11
12        if response.status_code != 200:
13            return None
14
15        data = response.json()
16
17        # Estraiamo le feature audio
18        audio = data.get("object", {}).get("audio")
19        if not audio:
20            return None
21
22        features = {
23            "danceability": audio.get("danceability"),
24            "energy": audio.get("energy"),
25            "speechiness": audio.get("speechiness"),
26            "acousticness": audio.get("acousticness"),
27            "instrumentalness": audio.get("instrumentalness"),
28            "liveness": audio.get("liveness"),
29            "valence": audio.get("valence"),
30            "tempo": audio.get("tempo")
31        }
32
33        return features
34
35    except Exception as e:
36        print(f"Errore recupero feature UUID {uuid}: {e}")
37        return None
38
39    # Elaboriamo una singola riga del CSV per recuperare le feature audio
40    def process_single_uuid(row: Dict, index: int) -> Optional[Dict]:
41        title = row.get('title', '').strip()
42        artist = row.get('artist', '').strip()
43        uuid = row.get('uuid', '').strip()
44
45        # Saltiamo la riga se l'UUID e' mancante
46        if not uuid or uuid == 'N/A':
47            return None
48
49        features = get_audio_features_by_uuid(uuid)
50
51        # Aggiungiamo le feature al risultato
52        result = {
53            "title": title,
54            "artist": artist,
55            "uuid": uuid,
56            **features
57        }
58
```



```
59         return result
60
61
62     # Funzione principale per lettura del CSV e ricavo delle feature audio
63     def process_csv_and_get_audio_features(csv_file_path: str, output_file_path:
64         str) -> List[Dict]:
65         rows = []
66
67         # Lettura del CSV...
68
69         results = []
70
71         # Elaboriamo le tracce in parallelo con 8 worker
72         with ThreadPoolExecutor(max_workers=8) as executor:
73             tasks = [
74                 executor.submit(process_single_uuid, row, i + 1)
75                 for i, row in enumerate(rows)
76             ]
77
78             for future in as_completed(tasks):
79                 result = future.result()
80                 if result:
81                     results.append(result)
82
83         # Scriviamo i risultati nel CSV di output
84         fieldnames = [
85             "title", "artist", "uuid",
86             "danceability", "energy", "speechiness", "acousticness",
87             "instrumentalness", "liveness", "valence", "tempo"
88         ]
89
90         # Scrittua effettiva del CSV...
91         return results
```



- **mood\_analysis.py** : si occupa di prevedere il mood delle singole canzoni presenti nel file .csv generato da *soundchart.py* e calcola moda (mood dominante), distribuzione dei mood e numero totale di brani per determinare il mood generale della playlist.

```
1     FEATURE_COLUMNS = [
2         "danceability",
3         "energy",
4         "speechiness",
5         "acousticness",
6         "instrumentalness",
7         "liveness",
8         "valence",
9         "tempo"
10    ]
11
12    # Prevediamo il mood delle singole tracce e calcoliamo statistiche complessive
13    def calculate_moods(csv_with_features: str, model):
14        # Lettura del CSV
15        df = pd.read_csv(csv_with_features)
16
17        # Controlliamo che tutte le colonne feature siano presenti
18        missing = [c for c in FEATURE_COLUMNS if c not in df.columns]
19        if missing:
20            print(f"[MOOD] Colonne feature mancanti: {missing}")
21            raise ValueError(f"Colonne feature mancanti: {missing}")
22
23        # Estrazione delle feature audio dal csv
24        X = df[FEATURE_COLUMNS]
25
26        # Tramite il modello prevediamo il mood per ogni traccia
27        df["label"] = model.predict(X).astype(int)
28
29        # Calcolo delle statistiche sul mood della playlist
30        overall = {
31            "mood_mode": int(df["label"].mode()[0]),
32            "mood_distribution": df["label"].value_counts(normalize=True).to_dict()
33        ,
34            "total_tracks": int(len(df))
35        }
36
37        # Salviamo le etichette nel CSV
38        df.to_csv(csv_with_features, index=False)
39
40        return df, overall
```

- **recommendations.py** : si occupa dell'intera gestione della raccomandazione dei brani. Tramite l'API *Last.fm*, raccomanda un brano per ogni mood musicale in base ai gusti dell'utente per quel mood. In assenza del mood, consiglia casualmente un brano dalla raccolta di Last.fm. Inoltre, tramite l'API di *Deezer*, recuperiamo l'immagine dei brani consigliati.

```
1     # Ricerca raccomandazioni per tutti i 4 mood in parallelo.
2     def get_similar_songs_by_mood(csv_with_features: str, lastfm_api_key: str = "
3         #####"):
4         from concurrent.futures import ThreadPoolExecutor, as_completed
```



```
4      import time
5
6      if not lastfm_api_key:
7          import os
8          lastfm_api_key = os.getenv("LASTFM_API_KEY", "#####")
9
10     # Leggi il CSV con i mood
11     try:
12         df = pd.read_csv(csv_with_features)
13         mood_counts = df["label"].value_counts().sort_index().to_dict()
14     except Exception as e:
15         return FALLBACK_RECOMMENDATIONS.copy()
16
17     already_recommended = set()
18     recommendations = []
19
20     # Ricerca per ogni mood in parallelo
21     with ThreadPoolExecutor(max_workers=4) as executor:
22         tasks = [
23             executor.submit(fetch_mood_recommendation, mood_id, MOOD_LABELS[mood_id], df, lastfm_api_key, already_recommended) for mood_id in range(4)
24         ]
25
26
27     # Raccogli risultati
28     for future in as_completed(tasks):
29         try:
30             result = future.result(timeout=10)
31             if result:
32                 recommendations.update(result)
33         except Exception as e:
34             print(f"[REC] Errore thread: {e}")
35
36
37     # Aggiunta di mood mancanti con fallback
38     for mood_id, mood_name in MOOD_LABELS.items():
39         if mood_name not in recommendations:
40             recommendations[mood_name] = FALLBACK_RECOMMENDATIONS[mood_name]
41
42     return recommendations
```

Il codice di recommendation.py risulta essere molto grande. Pertanto, abbiamo riportato solo la funzione principale richiamata da *app.py*, mentre il resto del codice è disponibile su [GitHub](#).

- ***utils.py*** : si occupa della pulizia dei file .csv temporanei dopo l'elaborazione e del caricamento del modello di classificazione.

```
1     load_dotenv()
2
3     MODEL_REPO = "Alepn04/PlayMoodifyModel"
4     MODEL_FILE = "PlayMoodify.pkl"
5
6     _model = None
7
8     # Carica il modello ML da HuggingFace Hub
```



```
9     def load_model():
10         global _model
11         if _model is None:
12             model_path = hf_hub_download(
13                 repo_id=MODEL_REPO,
14                 filename=MODEL_FILE,
15                 token=os.environ.get("HF_TOKEN")
16             )
17             _model = joblib.load(model_path)
18         return _model
19
20     # Eliminiamo i file CSV temporanei dopo l'elaborazione.
21     def cleanup_csv_files(base_dir: str = None):
22         if base_dir is None:
23             base_dir = os.path.dirname(os.path.abspath(__file__))
24
25         csv_files = [
26             os.path.join(base_dir, "playlist_tracks.csv"),
27             os.path.join(base_dir, "playlist_with_uuid.csv"),
28             os.path.join(base_dir, "playlist_with_features.csv")
29         ]
30
31         for csv_file in csv_files:
32             try:
33                 if os.path.exists(csv_file):
34                     os.remove(csv_file)
35             except Exception as e:
36                 print(f"[CLEANUP] Error deleting {csv_file}: {e}")
```

- *app.py* : rappresenta il nucleo del backend. In particolare, si occupa del caricamento delle API tramite *FastAPI*, del caricamento del modello e dell'esecuzione dell'intera pipeline che porta al risultato finale.

```
1     # Esegue la pipeline completa
2     def run_pipeline(playlist_url: str):
3
4         # Definiamo i percorsi dei file CSV temporanei
5         csv_1 = os.path.join(BASE_DIR, "playlist_tracks.csv")
6         csv_2 = os.path.join(BASE_DIR, "playlist_with_uuid.csv")
7         csv_3 = os.path.join(BASE_DIR, "playlist_with_features.csv")
8
9         # Otteniamo il CSV dalla playlist Spotify
10        p1 = subprocess.run(
11            [sys.executable, os.path.join(BASE_DIR, "linktocsconverter.py")],
12            playlist_url, csv_1,
13            check=False,
14            capture_output=True,
15            text=True,
16        )
17        if p1.returncode != 0:
18            raise subprocess.CalledProcessError(p1.returncode, p1.args, output=p1.
19            stdout, stderr=p1.stderr)
20
21         # Otteniamo gli UUID per le ogni traccia
22         p2 = subprocess.run(
```



```
21     [sys.executable, os.path.join(BASE_DIR, "uuidfromname.py"), csv_1,
22      csv_2],
23      check=False,
24      capture_output=True,
25      text=True,
26  )
27  if p2.returncode != 0:
28      raise subprocess.CalledProcessError(p2.returncode, p2.args, output=p2.
29      stdout, stderr=p2.stderr)
30
31      # Otteniamo le feature audio per ogni traccia
32  p3 = subprocess.run(
33      [sys.executable, os.path.join(BASE_DIR, "soundcharts.py"), csv_2, csv_3
34  ],
35      check=False,
36      capture_output=True,
37      text=True,
38  )
39  if p3.returncode != 0:
40      raise subprocess.CalledProcessError(p3.returncode, p3.args, output=p3.
41      stdout, stderr=p3.stderr)
42
43      return csv_3
44
45 @app.post("/process-playlist")
46
47      # Esegue l'intera pipeline per una playlist Spotify
48  def process_playlist(req: PlaylistRequest):
49      try:
50          final_csv = run_pipeline(req.playlist_url)
51          df, overall = calculate_moods(final_csv, model)
52          similar_songs = get_similar_songs_by_mood(final_csv)
53
54          response_data = {
55              "status": "success",
56              "playlist_url": req.playlist_url,
57              "overall_mood": overall,
58              "similar_songs_by_mood": similar_songs,
59              "tracks": df.to_dict(orient="records")
60          }
61
62          cleanup_csv_files(BASE_DIR)
63
64          return response_data
65
66      except subprocess.CalledProcessError as e:
67          cleanup_csv_files(BASE_DIR)
68          return {"status": "error", "error": f"Errore script: {str(e)}"}
69
70      except Exception as e:
71          cleanup_csv_files(BASE_DIR)
72          return {"status": "error", "error": str(e)}
```

Questa è la pipeline che eseguirà app.py, che verrà lanciata con l'endpoint a `@app.post("/process-playlist")`.



#### 4.1.2 Frontend

Per quanto riguarda il frontend, l'applicazione è stata sviluppata utilizzando React ed è organizzata nei seguenti componenti principali:

- **App.js**: gestisce il flusso principale dell'interfaccia grafica. Inizialmente mostra la pagina per l'inserimento del link della playlist, successivamente visualizza l'animazione di caricamento e, infine, presenta i risultati restituiti dal backend.
- **PlayListForm.js**: componente richiamato da App.js che si occupa di mostrare all'utente il form per l'inserimento dell'URL della playlist Spotify.
- **LoadingSpinner.js**: responsabile della visualizzazione dell'animazione di caricamento mostrata all'utente dopo l'invio del form.
- **ResultDisplay.js**: gestisce la presentazione dei risultati ottenuti dal backend. In particolare, mostra il mood prevalente della playlist, diversi grafici relativi alla distribuzione dei mood, i brani raccomandati per ciascun mood e l'elenco dettagliato dei brani della playlist, suddivisi per categoria emotiva.
- **api.js**: fa parte del layer dei servizi e gestisce l'intera comunicazione tra frontend e backend. Invia l'URL della playlist al backend, attende la risposta, ne verifica la validità e restituisce i dati elaborati ai componenti frontend.

#### 4.1.3 Interfaccia della piattaforma

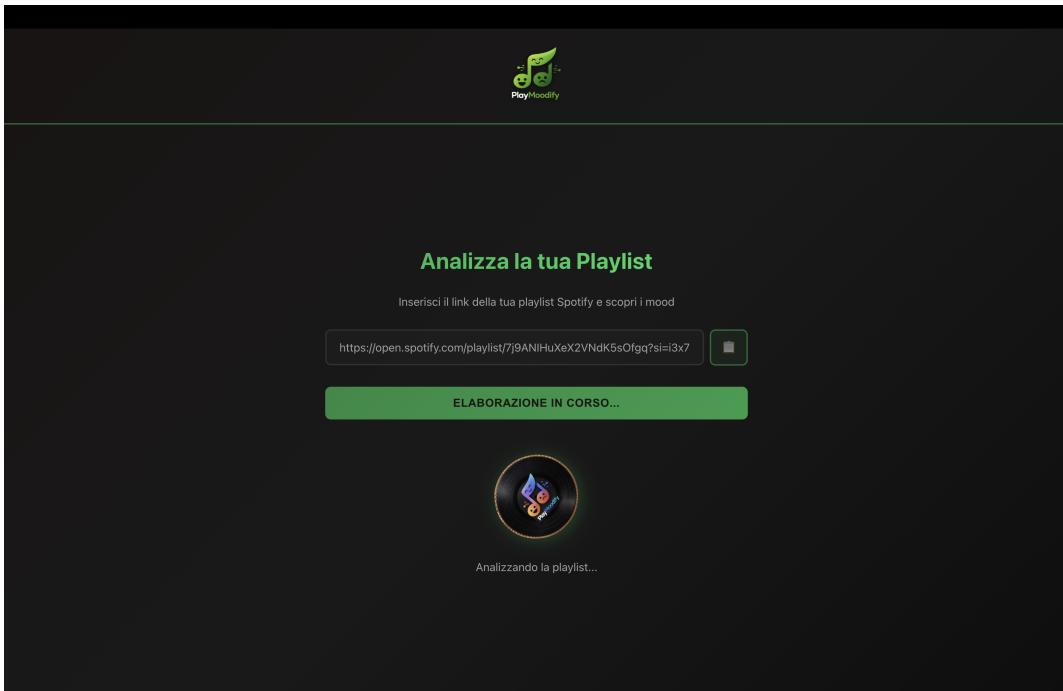


Figura 1: Home Page, inserimento link Spotify

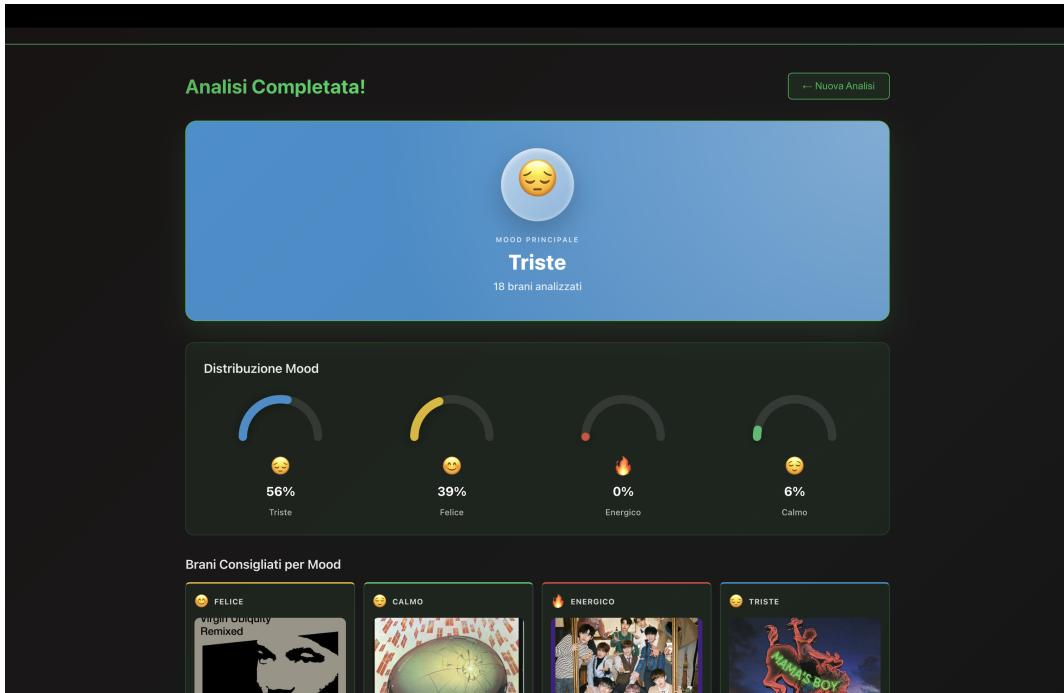


Figura 2: analisi generale della playlist e mood prevalente

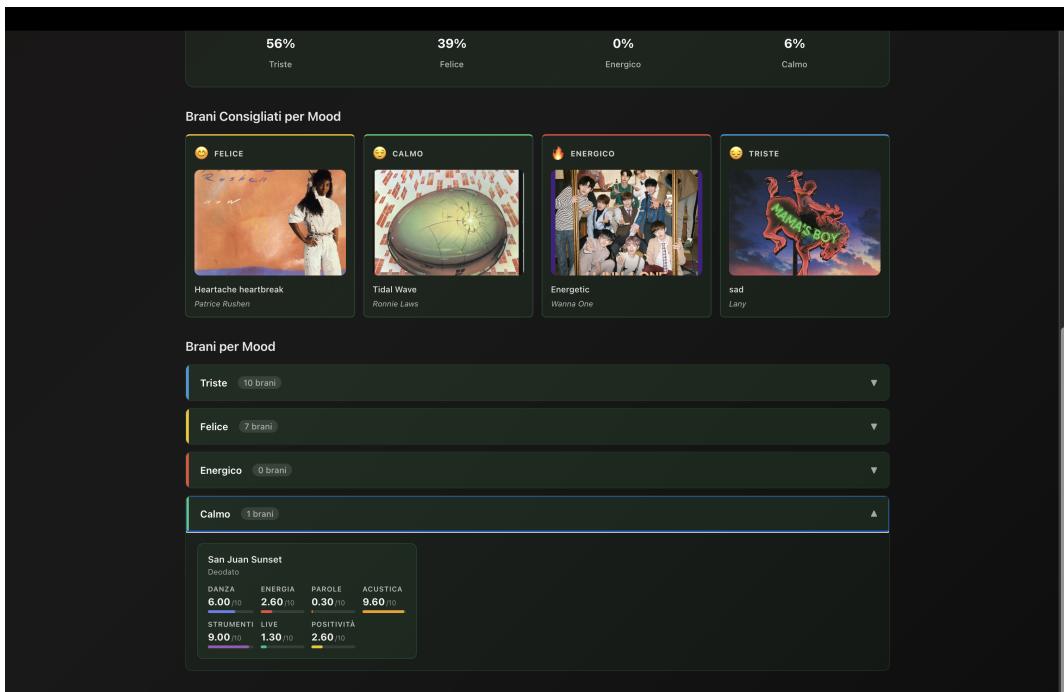


Figura 3: brani consigliati e catalogo delle canzoni suddivise per mood



## 5 Miglioramenti futuri

### 5.1 Modello di classificazione

Al fine di migliorare ulteriormente le performance del sistema PlayMoodify nel tempo, un possibile sviluppo futuro consiste nell'introduzione di una pipeline di addestramento continuo (*continuous training*). Tale approccio consentirebbe di aggiornare periodicamente il modello sfruttando nuovi dati provenienti dall'utilizzo reale del sistema, inclusi nuovi brani e feedback degli utenti, al fine di mitigare fenomeni di *data drift* e *concept drift* (cambiamenti nel tempo che possono degradare le performance di un modello addestrato).

In questo scenario, i nuovi dati verrebbero raccolti, validati e versionati, per poi essere integrati in un processo di *retraining* periodico del modello Random Forest. Le performance del nuovo modello verrebbero confrontate automaticamente con quelle della versione precedente, consentendo il deploy solo in caso di miglioramento delle metriche chiave. Questo approccio permetterebbe il miglioramento della capacità di generalizzazione del classificatore nel tempo.

### 5.2 Miglioramento efficienza in risposta

Un ulteriore ambito di miglioramento della piattaforma PlayMoodify riguarda l'ottimizzazione delle performance e la riduzione dei tempi di risposta, attraverso l'introduzione di meccanismi di ***session pooling***.

Il ***session pooling*** consente di riutilizzare connessioni e sessioni già inizializzate, evitando il costo computazionale associato alla creazione e chiusura/riapertura delle connessioni. Questo approccio permette di migliorare la scalabilità del sistema in presenza di richieste concorrenti.