Classificazione del Genere Musicale da File Audio

Relazione di Progetto - Esame di Machine Learning

Santi Lisi - Matricola 1000044181
Francesco Granata - Matricola 1000046547

23 giugno 2025

Indice

1 Problema

L'obiettivo del progetto è classificare automaticamente il genere musicale di una traccia audio. La classificazione dei generi musicali rappresenta un problema tipico di apprendimento supervisionato, in cui si predice una delle categorie predefinite a partire da feature audio estratte da file MP3.

1.1 Contesti applicativi

- Organizzazione automatica di librerie musicali
- Raccomandazione musicale personalizzata
- Riconoscimento musicale in app interattive

2 Dataset

Link al dataset su HuggingFace: Dataset. La spiegazione dei vari file si trova sul link.

Il dataset sviluppato per questo progetto ha l'obiettivo di associare un brano musicale a uno dei generi predefiniti, sulla base di caratteristiche audio significative.

2.1 Feature audio estratte

Le feature audio utilizzate rappresentano caratteristiche distintive del suono, in grado di catturare informazioni timbriche, ritmiche e armoniche del brano. Di seguito una breve descrizione delle 6 feature utilizzate:

- **MFCC (Mel Frequency Cepstral Coefficients)**: rappresentano l'inviluppo spettrale del suono secondo una scala logaritmica simile a quella dell'udito umano. Sono ampiamente usati per caratterizzare il timbro. Nel Dataset viene rappresentata come due matrici (Media e Deviazione Standard) di grandezza 13x90 (13: numero di coefficienti di MFCC; 90: numero di frame audio).

- **Chroma STFT**: misura la distribuzione dell'energia nelle 12 classi di altezza tonale (note) su una scala temperata, indipendentemente dall'ottava. È utile per catturare informazioni armoniche.Nel Dataset viene rappresentata come due matrici (Media e Deviazione Standard) di grandezza 12x90 (12: numero di coefficienti di MFCC; 90: numero di frame audio).

- **Spectral Contrast**: rappresenta la differenza tra picchi e valli nello spettro di frequenza. Evidenzia il contrasto tra componenti armoniche e rumorose del suono. Nel Dataset viene rappresentata come due matrici (Media e Deviazione Standard) di grandezza 7x90 (7: numero di coefficienti di MFCC; 90: numero di frame audio).

- **Zero-Crossing Rate (ZCR)**: misura quante volte il segnale attraversa lo zero. È un indicatore della rumorosità del segnale, tipicamente alto per suoni acusticamente frastagliati.Nel Dataset viene rappresentata come due matrici (Media e Deviazione Standard) di grandezza 1x90 (1: numero di coefficienti di MFCC; 90: numero di frame audio).

- **Tempo**: stima la velocità ritmica del brano (battiti per minuto), utile per distinguere generi con tempi differenti.Nel Dataset viene rappresentata come un float che indica i bpm (es. 150.00 bpm).

- **Statistiche sui battiti**: includono il numero di battiti rilevati, l'intervallo medio tra battiti e la deviazione standard degli intervalli. Offrono una descrizione più granulare della struttura ritmica.

Per ciascuna feature su frame temporali, sono stati calcolati: **media** e **deviazione standard** per ciascuno dei 90 secondi centrali della traccia, ottenendo un vettore numerico rappresentativo di ogni canzone.

2.2 Fase di acquisizione

Il dataset è stato costruito manualmente con le API di Spotify tramite `spotipy`, selezionando oltre 25 generi e scaricando 2-3 playlist da 100 brani ciascuna per genere. I metadati sono stati salvati in un file CSV.

2.3 Estrazione della traccia audio

Le tracce audio sono state ottenute con `pytube`, usando YouTube come sorgente. Ogni brano è stato convertito e tagliato nei 90 secondi centrali tramite `yt_dlp` e `pydub`.

2.4 Estrazione delle feature

Tramite la libreria `librosa` sono state estratte le feature audio descritte in precedenza. Per la maggior parte delle feature, i valori calcolati nel tempo sono stati sintetizzati in due matrici: una contenente le **medie** e l'altra le **deviazioni standard**, su finestre temporali di un secondo. Questo approccio ha permesso di ridurre significativamente la dimensionalità dei dati, passando da circa **8430 frame** (per i 90 secondi centrali) moltiplicati per ciascun coefficiente di ogni feature — oltre **280.000 valori** (13*8430 + 12*8430 + 7*8430 + 1*8430 + 1+ 1*8430) per traccia — a un totale di **90 frame**(1 al secondo) e **5944 valori** per brano.

Ogni secondo è rappresentato da una colonna in ciascuna matrice, fornendo una rappresentazione compatta ma informativa del brano. I risultati sono stati salvati in un secondo file `CSV`, strutturato in modo da mantenere la coerenza con il file dei metadati grazie all'ID Spotify. Questa strategia ha anche consentito di contenere le dimensioni del dataset finale, evitando file di dimensioni eccessive (nell'ordine di centinaia di gigabyte o più).

2.5 Gestione del download incrementale

Un sistema automatico controllava i file CSV per evitare di ripetere download già effettuati, rendendo il processo robusto e riprendibile.

2.6 Statistiche prima dell'oversampling

- Totale tracce raccolte: **oltre 7000**
- **29** generi possibili

2.7 Preparazione e Mappatura del Dataset

Il dataset iniziale, `TrackFeatures4.csv`, è stato sottoposto a una fase di pulizia e riorganizzazione per aggregare i generi musicali in macro-generi più ampi e gestibili. Questo processo è cruciale per ridurre la complessità del problema di classificazione e migliorare la robustezza del modello. Le fasi principali includono:

1. **Caricamento e Rimozione Duplicati:** Il dataset originale viene caricato in un DataFrame Pandas. La prima operazione consiste nella rimozione di eventuali righe duplicate, assicurando l'unicità di ogni record e prevenendo bias o sovrastima delle statistiche.

2. **Mappatura a Macro-Generi:** Viene definita una mappatura predefinita (`genre_to_macro`) che associa i generi musicali specifici (es. 'Grunge', 'Trance') a categorie più ampie (es. 'Rock', 'Electronic'). Questa aggregazione riduce il numero totale di classi da 29 (generi originali) a 9 macro-generi distinti:

   Ambient/Other, Classical, Electronic, Folk/Country, Hip-Hop, Jazz/Blues, Pop, Reggae/Afrobeat, Rock

3. **Gestione Generi Non Mappati:** Durante il processo di mappatura, alcune righe potrebbero contenere generi che non sono stati esplicitamente inclusi nella mappa `genre_to_macro`. Queste righe vengono identificate e rimosse dal dataset, poiché non rientrano nelle categorie di interesse per la classificazione. Alcuni generi nel dataset iniziale erano troppo generali ed erano tra più macro-generi quindi si è deciso di eliminarli.

4. **Generazione ID Numerici:** Per facilitare l'addestramento dei modelli di machine learning, i macro-generi testuali vengono convertiti in ID numerici univoci. Viene creata una nuova colonna (`macro_genre_id`) nel DataFrame che associa a ciascun macro-genere un ID intero, partendo da 0 e seguendo un ordine alfabetico dei macro-generi. La corrispondenza macro-genere $\leftrightarrow$ ID è la seguente:

   - 0: Ambient/Other

- 1: Classical
- 2: Electronic
- 3: Folk/Country
- 4: Hip-Hop
- 5: Jazz/Blues
- 6: Pop
- 7: Reggae/Afrobeat
- 8: Rock

5. **Salvataggio del Dataset Pulito:** Il DataFrame risultante, contenente le feature originali più le nuove colonne `macro_genre` e `macro_genre_id`, viene salvato in un nuovo file CSV (`dataset_macro_generi.csv`). Questo file rappresenta il dataset finale utilizzato per le fasi successive di pre-elaborazione delle feature numeriche e l'addestramento dei modelli.

Questa fase di preparazione del dataset è fondamentale per standardizzare le etichette, consolidare categorie simili e fornire un input pulito e strutturato ai successivi passaggi di analisi e modellazione.

2.8 Split del Dataset e Bilanciamento delle Classi

Dopo la fase di preparazione e mappatura dei generi musicali, il dataset è stato ulteriormente elaborato per essere suddiviso in set di training e test e per affrontare l'eventuale problema dello sbilanciamento delle classi. Questo passaggio è cruciale per addestrare modelli robusti e valutarne le prestazioni in modo affidabile su dati non visti.

Le fasi principali di questa elaborazione sono le seguenti:

1. **Caricamento del Dataset Pre-elaborato:** Il processo inizia con il caricamento del file `dataset_macro_generi.csv`, che contiene i dati musicali già puliti e mappati ai macro-generi con i relativi ID numerici.

2. **Split Stratificato in Training e Test Set:** Il dataset viene diviso in un set di training e un set di test utilizzando la funzione `train_test_split` della libreria `scikit-learn`.

    - La dimensione del set di test è fissata al **20%** del dataset totale (`test_size=0.2`).

- Viene applicata la **stratificazione**(`stratify=df["macro_genre_id"]`). Questo assicura che la distribuzione delle classi (macro-generi) sia mantenuta proporzionale sia nel set di training che in quello di test, prevenendo che alcune classi siano sovra o sottorappresentate in uno dei due subset.
- Un `random_state` fisso (`42`) è impostato per garantire la riproducibilità dello split.

Questa separazione è fondamentale per valutare la capacità di generalizzazione del modello su dati che non ha mai "visto" durante l'addestramento.

3. **Bilanciamento del Training Set tramite Sovracampionamento/Sottocampionamento:** Il set di training viene poi bilanciato per mitigare l'impatto di eventuali sbilanciamenti nella distribuzione delle classi. Un dataset sbilanciato può portare il modello a favorire le classi maggioritarie, compromettendo le performance sulle classi minoritarie.

   - Viene calcolata la dimensione della classe maggioritaria nel set di training (`class_counts.max()`). Questa dimensione viene scelta come **dimensione target** per tutte le classi (`mean_size`).
   - Per ogni macro-genere nel set di training:
     - Se la classe ha meno campioni della dimensione target, viene applicato il **sovracampionamento (oversampling)** tramite `resample(..., replace=True, n_samples=mean_size)`. Questo duplica casualmente i campioni esistenti della classe fino a raggiungere la dimensione desiderata.
     - Se la classe ha più campioni della dimensione target, viene applicato il **sottocampionamento (undersampling)** tramite `group.sample(n=mean_size)`. Questo seleziona casualmente un sottoinsieme di campioni della classe fino a raggiungere la dimensione desiderata. Visto che si è scelto come dimensione target la dimensione della classe più grande, questo caso non viene mai effettivamente utilizzato.
     - Se la classe ha già la dimensione target, rimane invariata.
   - Un `random_state` fisso (`42`) è utilizzato anche per le operazioni di campionamento per garantire la riproducibilità.

   È importante sottolineare che il bilanciamento viene applicato **esclusivamente al set di training** per evitare data leakage e per mantenere il set di test rappresentativo della distribuzione reale dei dati.

4. **Unione e Mescolamento dei Dati Bilanciati:** I frammenti di DataFrame bilanciati per ogni classe vengono concatenati per formare il nuovo set di training bilanciato (`train_balanced_df`). Questo DataFrame viene poi **mescolato casualmente** (`sample(frac=1)`) per distribuire uniformemente i campioni delle diverse classi.

5. **Salvataggio dei Dataset Finali:** I set di training bilanciato e di test pulito vengono salvati in file CSV separati: `train_balanced.csv` e `test_clean.csv`. Questi file saranno gli input per le successive fasi di pre-elaborazione delle feature (scalatura) e l'addestramento dei modelli.

2.9 Statistiche finali

- Totale tracce **11529**
- **9** generi possibili

3 Metodi

Questa sezione descrive in dettaglio l'approccio metodologico adottato per affrontare il problema della classificazione dei generi musicali. È strutturata in diverse sottosezioni, ognuna delle quali illustra un aspetto specifico della metodologia impiegata. Inizieremo con le fasi di pre-elaborazione dei dati e la preparazione del dataset, seguite dalla descrizione delle architetture dei modelli di apprendimento profondo che abbiamo proposto. Infine, presenteremo la strategia di addestramento e valutazione utilizzata. Una sottosezione aggiuntiva sarà dedicata a un metodo sperimentale che ha previsto una modifica specifica al dataset, volta a esplorare l'impatto di tale alterazione sulle performance del modello.

3.1 Pre-elaborazione dei Dati

La qualità e il formato dei dati di input sono cruciali per le prestazioni di un modello di machine learning. Il dataset originale, composto da file CSV, contiene feature numeriche estratte da brani musicali. La fase di pre-elaborazione è stata gestita dalla funzione `doPreprocessing` (definita nella libreria `libProject`), che esegue le seguenti operazioni:

1. **Parsing delle Feature:** Le feature estratte, come MFCCs (Mel-frequency cepstral coefficients), Chroma, Spectral Contrast, Zero Crossing Rate (ZCR), Beats e Tempo, sono inizialmente memorizzate co-

me stringhe (spesso rappresentazioni di dizionari o liste). La funzione `preprocess_row` converte queste stringhe in strutture dati Python (dizionari o liste) utilizzando `ast.literal_eval`.

2. **Appiattimento e Concatenazione:** Molte feature, come MFCCs, Chroma, Spectral Contrast e ZCR, sono rappresentate con valori di media (`mean`) e deviazione standard (`std`) per ogni coefficiente. Queste matrici o liste bidimensionali vengono appiattite in vettori unidimensionali e successivamente concatenate insieme a feature scalari come il tempo e i parametri dei beat (conteggio, media e deviazione standard dell'intervallo). Questo produce un unico vettore di feature per ogni brano, con una dimensione fissa di 5944 elementi.

3. **Normalizzazione/Scalatura:** Dopo la creazione del vettore di feature unificato (`X`), i dati vengono scalati utilizzando lo **StandardScaler** di scikit-learn. Questo processo standardizza le feature sottraendo la media e dividendo per la deviazione standard, garantendo che tutte le feature abbiano una media di 0 e una deviazione standard di 1. Lo scaler viene addestrato solo sul set di training e poi riutilizzato per trasformare il set di test per evitare data leakage. Lo scaler addestrato viene salvato su disco come `scaler_sicuro.pkl` per poter essere riutilizzato in futuro o in fase di inferenza.

4. **Encoding delle Etichette:** Le etichette di genere (`y`) vengono convertite in un formato numerico (integer), adatto per l'input ai modelli di classificazione in PyTorch.

I dati pre-elaborati vengono poi incapsulati in oggetti `AudioFeaturesDataset` personalizzati, derivati da `torch.utils.data.Dataset`, che facilitano il caricamento batch durante l'addestramento tramite `DataLoader`.

3.2 Architetture dei Modelli

Per la classificazione dei generi musicali, sono state esplorate e implementate tre diverse architetture di reti neurali. Questi modelli sono stati definiti come classi `nn.Module` di PyTorch.

SoftMaxRegressor

Questo modello rappresenta una regressione logistica multiclasse, ovvero la forma più semplice di rete neurale senza hidden layer. È stato incluso come baseline per valutare l'efficacia di architetture più complesse.

- **Struttura:** Consiste in un singolo strato lineare (`nn.Linear`) che mappa direttamente l'input di dimensione `in_size` all'output di dimensione `out_size` (corrispondente al numero di classi).
- **Funzione di Attivazione:** Non è specificata esplicitamente una funzione di attivazione dopo lo strato lineare, implicando che l'output siano i logits, che verranno poi passati a una funzione di loss (come Cross-Entropy) che include implicitamente una Softmax.

MLPClassifier

Questo è un MLP con un singolo strato nascosto. Rappresenta un passo avanti in complessità rispetto al SoftMaxRegressor, permettendo al modello di apprendere relazioni non lineari più elaborate.

- **Struttura:**

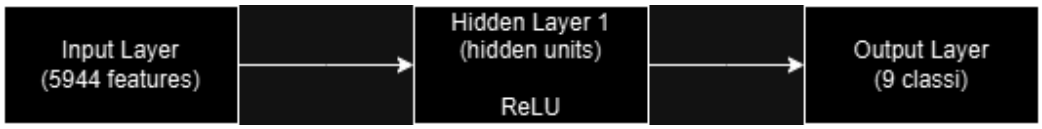

Figura 1: MLP

  - Uno strato lineare di input (`nn.Linear(in_features, hidden_units)`).
  - Una funzione di attivazione **ReLU** (`nn.ReLU()`) applicata all'output dello strato nascosto, introducendo non linearità.
  - Uno strato lineare di output (`nn.Linear(hidden_units, out_classes)`) che mappa le feature apprese dallo strato nascosto alle probabilità delle classi.
- **Parametri:** Prende in ingresso `in_features` (dimensione dell'input), `hidden_units` (numero di neuroni nello strato nascosto) e `out_classes` (numero di classi di output).

DeepMLPClassifier

Questo modello è un MLP più profondo, caratterizzato dalla presenza di due strati nascosti. È stato progettato per catturare pattern più complessi nei dati.

- **Struttura:** Il modello è definito tramite `nn.Sequential`, che incapsula una sequenza di operazioni:

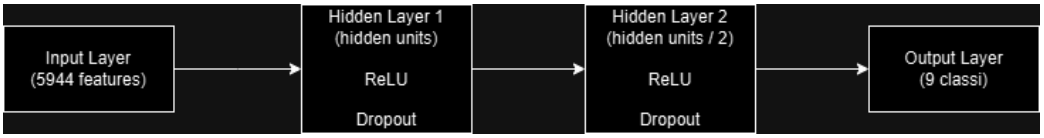

Figura 2: DeepMLP

1. Primo strato lineare: `nn.Linear(in_features, hidden_units)`.
2. Prima attivazione ReLU: `nn.ReLU()`.
3. Primo strato di Dropout: `nn.Dropout(dropout)`, applicato per prevenire l'overfitting.
4. Secondo strato lineare: `nn.Linear(hidden_units, int(hidden_units/2))`. Il numero di neuroni in questo strato è la metà di quelli del primo strato nascosto, introducendo una compressione delle feature.
5. Seconda attivazione ReLU: `nn.ReLU()`.
6. Secondo strato di Dropout: `nn.Dropout(dropout)`.
7. Strato lineare di output: `nn.Linear(int(hidden_units/2), out_classes)`.

- **Parametri:** Similmente all'MLPClassifier, accetta `in_features`, `hidden_units` e `out_classes`, oltre a un parametro `dropout` per controllare l'intensità della regolarizzazione.

3.3 Strategia di Addestramento e Valutazione

Il processo di addestramento e valutazione è orchestrato dalla funzione `train_classifier` e `test_classifier` (entrambe in `libProject`).

Addestramento del Modello (`train_classifier`)

- **Funzione di Loss:** Viene utilizzata la **CrossEntropyLoss** (`nn.CrossEntropyLoss()`), che è la scelta standard per problemi di classificazione multi-classe. Questa funzione combina log softmax e Negative Log Likelihood Loss.
- **Ottimizzatore:** L'ottimizzatore scelto è **SGD (Stochastic Gradient Descent)** con momentum (`momentum`) e weight decay (`weight_decay`).
- **Iterazione su Iperparametri:** Il processo di addestramento itera su una griglia di iperparametri (`param_grid`) caricata dal file `par.json`. Questi parametri includono `learning_rate`, `batch_size`, `momentum`, `weight_decay`, `dropout` e `hidden_units`. Questo approccio permette di esplorare diverse configurazioni e trovare quella ottimale. troavata

quella ottimale si è rpovato a variare di poco i parametri di quella per migliorare i risultati

- **Gestione Batch e Dispositivo:** I dati vengono caricati in batch tramite `DataLoader` e spostati sul dispositivo disponibile (GPU se `cuda` è disponibile, altrimenti CPU).
- **Logging con TensorBoard:** Durante l'addestramento, le metriche (loss, accuracy, top-3 accuracy) vengono registrate utilizzando `SummaryWriter` di TensorBoard, consentendo una visualizzazione grafica dell'andamento dell'addestramento e della validazione.
- **Early Stopping:** Per prevenire l'overfitting e ottimizzare il tempo di addestramento, è implementato un meccanismo di **early stopping**. Questo ferma l'addestramento se la loss sul set di test non migliora per un numero predefinito di epoche (`early_stopping_patience`). Il modello con la migliore loss di validazione viene salvato e ripristinato alla fine dell'addestramento in caso di early stopping. I pesi dei modelli migliori vengono salvati nella cartella `models_weights`.

Valutazione del Modello (`test_classifier`)

- **Modalità di Valutazione:** La funzione `test_classifier` pone il modello in modalità di valutazione (`model.eval()`) e disabilita il calcolo del gradiente (`torch.no_grad()`) per velocizzare l'inferenza e prevenire aggiornamenti accidentali dei pesi.
- **Metriche di Valutazione:**
  - **Accuracy:** Misurata come percentuale di predizioni corrette.
  - **Top-k Accuracy:** Viene calcolata anche la top-3 accuracy, utile per valutare se la classe corretta rientra tra le prime 3 predizioni più probabili del modello.
  - **Report di Classificazione:** Per ogni modello addestrato, viene generato un report di classificazione dettagliato (`classification_report` di scikit-learn) che include precisione, recall, f1-score e supporto per ogni classe, oltre alle medie macro e weighted.
  - **Matrice di Confusione:** La matrice di confusione vuole visualizzare le performance del modello in termini di veri positivi, falsi positivi, veri negativi e falsi negativi, fornendo una visione granulare degli errori di classificazione.

- **Salvataggio delle Metriche:** I report di classificazione per i set di train e test, insieme ai parametri del modello, vengono salvati in un file JSON. Questo permette di tenere traccia e confrontare facilmente le performance di tutti gli esperimenti.

- **Stampa dei Risultati:** La funzione `print_metrics_from_json` legge il file JSON e stampa una sintesi delle performance di ogni modello, evidenziando i modelli migliori basati su accuratezza, f1-score macro e f1-score weighted sul set di test.

3.4 Integrazione di Feature Testuali: Il Titolo del Brano

Per esplorare l'impatto di informazioni non acustiche sulle performance di classificazione, abbiamo condotto un esperimento aggiuntivo che ha previsto l'integrazione di una feature testuale: il **titolo del brano**. L'ipotesi sottostante è che, sebbene primariamente acustici, i generi musicali possano avere associazioni lessicali nei titoli che, se catturate, possono fornire un segnale complementare alle feature audio.

Questo approccio si è articolato nelle seguenti fasi:

1. **Caricamento e Unione dei Dati:** I set di training (`train_balanced.csv`) e di test (`test_clean.csv`) pre-esistenti sono stati caricati. Contestualmente, è stato caricato anche un file aggiuntivo, `Tracks1.csv`, contenente gli ID dei brani e i loro rispettivi titoli. I titoli sono stati uniti ai DataFrame di training e test utilizzando `id_track` come chiave di unione. Questo garantisce che ogni riga del nostro dataset (train e test) contenga, oltre alle feature audio e all'etichetta di genere, anche il titolo associato.

2. **Pre-elaborazione della Feature Testuale:** Per rendere il testo del titolo utilizzabile da un modello numerico, è stata applicata la seguente pre-elaborazione:

    - Tutti i titoli sono stati convertiti in minuscolo (`str.lower()`) per standardizzare la rappresentazione e trattare parole come "Rock" e "rock" allo stesso modo.
    - Eventuali valori mancanti (NaN) nei titoli sono stati sostituiti con stringhe vuote (`fillna('')`).
    - È stato utilizzato un **TfidfVectorizer** (*Term Frequency-Inverse Document Frequency*) per convertire il testo in un vettore numerico. Il TF-IDF è una tecnica statistica che riflette l'importanza

di una parola in un documento rispetto a una collezione di documenti. Abbiamo configurato il `TfidfVectorizer` per estrarre le `max_features=50` parole più significative, bilanciando la complessità del modello con la capacità di catturare i termini più rilevanti. L'addestramento del TF-IDF è avvenuto solo sul set di training (`fit_transform`), mentre sul set di test è stata applicata solo la trasformazione (`transform`) per evitare data leakage.

3. **Concatenazione delle Feature:** I vettori di feature testuali (generati dal TF-IDF) sono stati concatenati orizzontalmente con i vettori di feature audio pre-elaborate. Questo ha creato un nuovo vettore di input per ogni brano, che ora include sia le caratteristiche acustiche che quelle testuali. La dimensione finale del vettore di input per il modello è stata quindi aumentata, passando da 5944 feature a **5944 + 50 = 5994** feature.

4. **Addestramento del Modello:** Con il dataset aggiornato includendo la feature testuale, il modello `MLPClassifier` è stato addestrato utilizzando la stessa strategia di addestramento.

5. **Valutazione e Registrazione delle Metriche:** Le performance del modello con le feature testuali aggiuntive sono state valutate sui set di train e test, e le metriche di classificazione (accuratezza, precisione, recall, f1-score) sono state calcolate e aggiunte al file `metrics2.json`, consentendo un confronto diretto con i modelli addestrati esclusivamente sulle feature audio.

4 Esperimenti

Questa sezione è dedicata alla descrizione e all'analisi degli esperimenti condotti per valutare le performance dei modelli sviluppati per la classificazione dei generi musicali. L'obiettivo primario è confrontare l'efficacia di diverse architetture, inclusa l'integrazione di feature testuali, e identificare la configurazione più performante.

4.1 Setup Sperimentale

Tutti gli esperimenti sono stati eseguiti in un ambiente di sviluppo Python, utilizzando `PyTorch` per l'implementazione delle reti neurali e `scikit-learn` per le metriche di valutazione. Per garantire la riproducibilità dei risultati, è stato impostato un `seed` fisso a `17` per le librerie `random`, `numpy` e `torch`.

Il dataset utilizzato deriva da `TrackFeatures4.csv`, pre-elaborato come descritto nella Sezione 3. Il processo ha portato a un dataset con 5944 feature acustiche per brano e 9 macro-generi di destinazione. Un'ulteriore fase di pre-elaborazione ha generato i set di training e test bilanciati, rispettivamente `train_balanced.csv` e `test_clean.csv`. Per il modello che include feature testuali, i dataset sono stati arricchiti con 50 feature TF-IDF estratte dai titoli, portando la dimensione dell'input a 5994 feature.

Per l'addestramento, la funzione di loss impiegata è stata la **CrossEntropyLoss** (`nn.CrossEntropyLoss()`), standard per i problemi di classificazione multiclasse. L'ottimizzatore scelto per tutti i modelli è stato lo **Stochastic Gradient Descent (SGD)**, configurato con iperparametri quali `learning_rate`, `momentum` e `weight_decay`.

Un meccanismo di **early stopping** è stato implementato con una `patience` di 10 epoche. Questo ha permesso di interrompere l'addestramento quando la loss sul set di test non mostrava miglioramenti per un numero predefinito di epoche, ripristinando il modello allo stato con la migliore loss di validazione e prevenendo l'overfitting. Il monitoraggio dell'andamento di loss e accuratezza è stato effettuato tramite `TensorBoard` per ogni esecuzione.

Gli iperparametri che sono stati variati per trovare una soluzione ottimale:

- **Epochs:** Fissati a 1000, con early stopping.
- **Learning Rate (`lr`):** Tasso di apprendimento dell'ottimizzatore.
- **Batch Size (`batch_size`):** Numero di campioni per ogni aggiornamento del gradiente.
- **Momentum (`momentum`):** Termine per accelerare la convergenza dell'SGD.
- **Weight Decay (`weight_decay`):** Regolarizzazione L2 sui pesi del modello.
- **Dropout Rate (`dropout`):** Percentuale di neuroni disattivati casualmente (per MLP e DeepMLP).
- **Hidden Units (`hidden_units`):** Dimensione del primo strato nascosto (per MLP e DeepMLP).

4.2 Architetture dei Modelli a Confronto

Per questo studio, sono state confrontate quattro distinte architetture di modelli, ognuna con le proprie caratteristiche e complessità.

1. **SoftMaxRegressor**
    - lr: 0.0005
    - batch_size: 128
    - momentum: 0.8
    - weight_decay: 0.01
    - dropout: ///
    - hidden_units: ///
2. **MLPClassifier (Solo Audio)**
    - lr: 0.0005
    - batch_size: 256
    - momentum: 0.95
    - weight_decay: 0.01
    - dropout: ///
    - hidden_units: 1024
3. **DeepMLPClassifier (Solo Audio)**
    - lr: 0.0005
    - batch_size: 128
    - momentum: 0.8
    - weight_decay: 0.005
    - dropout: 0.4
    - hidden_units: 512
4. **MLPClassifier (Audio + Testo)**
    - lr: 0.0005
    - batch_size: 256
    - momentum: 0.95
    - weight_decay: 0.01
    - dropout: ///
    - hidden_units: 1024

Di ogni architettura questi sono gli iperparametri con cui si è raggiunto il risultato migliore.

4.3 Risultati e Analisi Comparativa

Gli esperimenti sono stati eseguiti per ogni architettura, testando diverse combinazioni di iperparametri tramite una griglia di ricerca. Per ciascuna architettura, è stato selezionato il modello con le migliori performance sul set di test, basandosi principalmente sull'accuratezza e sull'F1-score pesato.

La Tabella 1 riassume le metriche chiave di questi quattro modelli migliori, offrendo un confronto diretto delle loro capacità di generalizzazione sul compito di classificazione dei generi musicali.

Tabella 1: Confronto delle performance dei migliori modelli di ciascuna architettura sul set di test e di training.

Architettura	**Acc. Test**	**F1-Macro Test**	**F1-Weighted Test**	**Acc. Train**
1	0.536	0.508	0.539	0.832
2 (Solo Audio)	**0.598**	**0.584**	**0.592**	0.947
3 (Solo Audio)	0.579	0.566	0.571	0.906
4 (Audio + Testo)	0.585	0.573	0.577	0.942

Dalla Tabella 1, possiamo trarre le seguenti osservazioni preliminari:

- **Osservazione Generale:** È evidente un pronunciato fenomeno di overfitting in tutte le architetture testate, come dimostrato dalla marcata differenza tra le elevate accuratezze sul training set (che raggiungono valori fino al 94.7%) e le accuratezze significativamente inferiori sul test set (che si attestano in un intervallo tra 53.6% e 59.8%). Le performance di generalizzazione tra le diverse architetture sul test set sono relativamente vicine, variando di circa 6 punti percentuali.

- **Analisi SoftMaxRegressor vs MLPClassifier (Solo Audio)**: Il SoftMaxRegressor, in quanto modello lineare, mostra la più bassa capacità di generalizzazione con un'Accuratezza Test di 0.536, indicando che il problema di classificazione dei generi musicali richiede l'apprendimento di relazioni non lineari. L'introduzione di un singolo strato nascosto nel MLPClassifier (Solo Audio) migliora significativamente le metriche sul test set (Acc. Test: 0.598), dimostrando una maggiore capacità rappresentativa, nonostante persista un marcato overfitting sul training set (Acc. Train: 0.947).

- **Analisi MLPClassifier (Solo Audio) vs DeepMLPClassifier (Solo Audio)**: Il passaggio da un singolo strato nascosto (MLPClassifier)

a due strati (DeepMLPClassifier), nonostante l'inclusione di Dropout per mitigare l'overfitting, non ha portato a un miglioramento delle performance sul set di test (Acc. Test: 0.579 vs 0.598 del MLPClassifier). Sebbene l'accuratezza sul training set del DeepMLP (0.906) indichi un overfitting leggermente inferiore rispetto al MLPClassifier (0.947), questa maggiore complessità dell'architettura non si è tradotta in una migliore capacità di generalizzazione per questo specifico compito, suggerendo che un modello più profondo potrebbe essere eccessivo o non ottimamente configurato per le feature disponibili.

- **Analisi MLPClassifier (Solo Audio) vs MLPClassifier (Audio + Testo)**: L'integrazione della feature testuale (il titolo del brano) nel MLPClassifier non ha portato a un miglioramento significativo delle performance. Le metriche del MLPClassifier (Audio + Testo) (Acc. Test: 0.585) sono molto simili, se non leggermente inferiori in alcuni casi, a quelle del MLPClassifier che utilizza solo feature audio (Acc. Test: 0.598). Ciò suggerisce che la rappresentazione TF-IDF del titolo, con le 50 feature selezionate, non fornisce un segnale discriminante sufficientemente forte o complementare per questo compito, o che le feature audio sono già prevalenti nel determinare il genere.

- **Identificazione del Modello Migliore:** Sulla base delle accuratezze e degli F1-score sul set di test, il **MLPClassifier (Solo Audio)** è risultato essere il modello più performante, sebbene con un margine limitato rispetto alle altre architetture testate.

La Figura 3 mostra l'andamento della loss e dell'accuratezza (su training e test set) per il modello che ha conseguito le migliori performance complessive in questo confronto. Questo grafico è utile per visualizzare la sua curva di apprendimento.

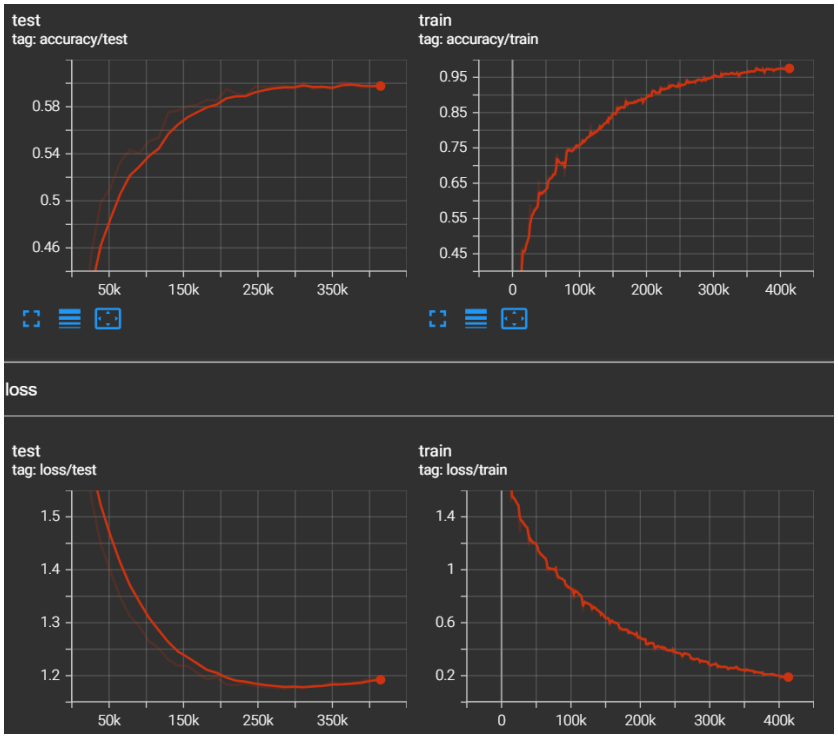

Figura 3: Andamento della Loss e Accuratezza (Train/Test) per il MLPClassifier.

4.4 Analisi Dettagliata del Modello Migliore: MLPClassifier (solo audio)

In base ai risultati comparativi della sezione precedente, il modello più performante è stato il **MLPClassifier (solo audio)**. Per comprendere meglio le sue capacità e i pattern di errore, è fondamentale analizzare in dettaglio la sua matrice di confusione sul set di test.

La matrice di confusione (Figura 4) visualizza la distribuzione delle predizioni rispetto alle classi reali.

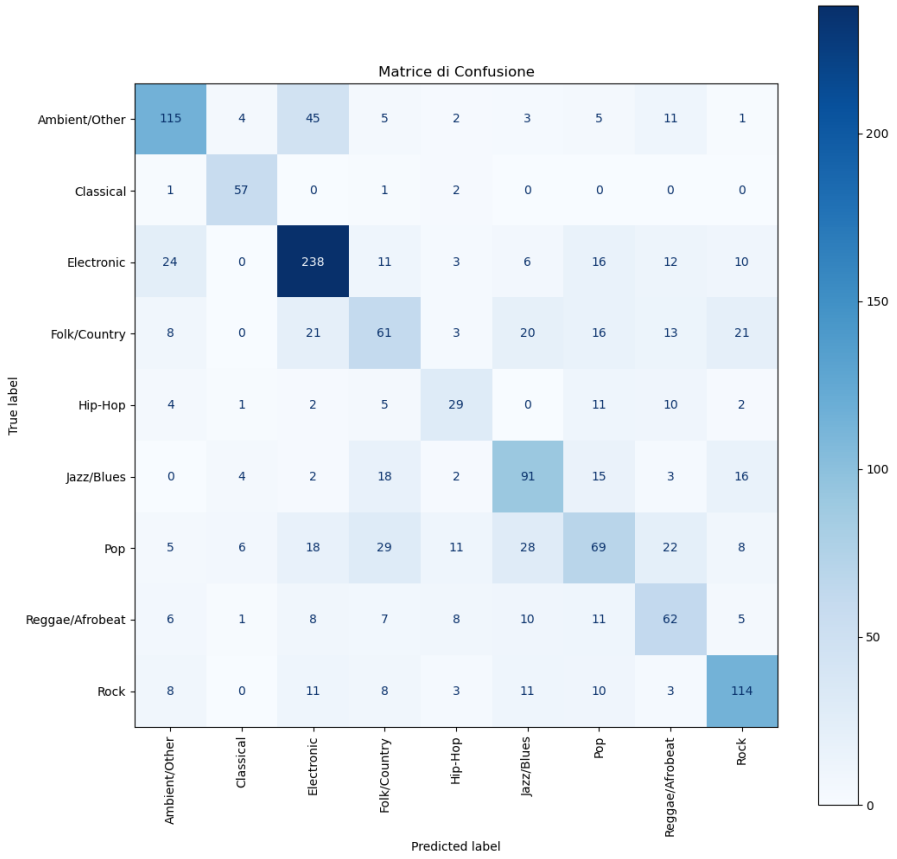

Figura 4: Matrice di confusione per il modello **MLPClassifier (solo audio)** sul set di test. Gli assi rappresentano i macro-generi musicali.

Osservazioni:

- **Classi con alta accuratezza:** Il genere `Electronic` presenta il numero più elevato di classificazioni corrette (238), indicando che il modello riesce a distinguere efficacemente le sue caratteristiche. Anche i generi `Rock` (114), `Ambient/Other` (115) e `Jazz/Blues` (91) mostrano buone performance di classificazione.
- **Classi con bassa precisione:** Il genere `Hip-Hop` conta solo 29 predizioni corrette, con un'elevata confusione verso altre classi come `Jazz/Blues`,

`Pop` e `Reggae/Afrobeat`. `Folk/Country` e `Pop` mostrano anch'essi notevoli sovrapposizioni con altri generi.

- **Coppie frequentemente confuse:** Il modello tende a confondere `Ambient/Other` con `Electronic` (45 errori) e in misura minore con `Rock` e `Folk/Country`. Notevole è anche la confusione tra `Folk/Country` e `Pop` (29 errori), probabilmente dovuta a similarità timbriche. `Jazz/Blues` viene frequentemente scambiato con `Pop` e `Rock`.

- **Distribuzione degli errori:** Gli errori non risultano simmetrici. Ad esempio, `Pop` è confuso con `Electronic` in 18 casi, mentre il contrario accade solo 16 volte. Inoltre, i generi con un numero ridotto di esempi (come `Classical` e `Hip-Hop`) tendono a presentare una precisione inferiore.

- **Considerazioni generali:** Il modello mostra difficoltà nel distinguere generi con caratteristiche acustiche simili, come `Pop`, `Jazz/Blues` e `Folk/Country`.

4.5 Discussione e Conclusioni degli Esperimenti

Gli esperimenti condotti hanno fornito preziose intuizioni sull'efficacia di diverse architetture di reti neurali nella classificazione automatica dei generi musicali, basata su feature acustiche e testuali.

Durante la sperimentazione, è emerso chiaramente che il modello **SoftMaxRegressor** si è rivelato troppo semplice e limitato per affrontare efficacemente un compito complesso come la classificazione dei generi musicali. Al contrario, i modelli **MLPClassifier** e **DeepMLPClassifier** hanno mostrato performance migliori, con una lieve superiorità del primo. Tuttavia, un aspetto interessante osservato è che la variazione degli iperparametri di entrambi i modelli, anche in maniera significativa, non ha portato a cambiamenti sostanziali nelle prestazioni: l'accuratezza è rimasta mediamente attorno a 0,57, con un picco massimo di 0,59.

Questa stabilità delle performance può essere attribuita a due fattori principali:

- *Limiti informativi delle feature:* le caratteristiche audio estratte potrebbero non essere sufficientemente rappresentative o discriminative per distinguere efficacemente i diversi generi musicali. Nel nostro caso, da ogni brano sono stati considerati 90 secondi, segmentati in frame da un secondo (90 frame totali, con frequenza di campionamento a 90 Hz). Per ciascun secondo, sono stati calcolati la media e la deviazione

standard delle feature acustiche. Tuttavia, questo tipo di aggregazione statistica potrebbe appiattire variazioni temporali significative e limitare la quantità di informazione utile per la classificazione, riducendo così l'impatto positivo che si potrebbe ottenere tramite un'ottimizzazione più sofisticata dei parametri del modello.

- *Qualità e bilanciamento del dataset:* un dataset rumoroso o sbilanciato può impedire al modello di apprendere relazioni robuste, limitando le performance complessive.

Inoltre, l'integrazione delle feature testuali, in particolare il titolo del brano, non ha apportato benefici al modello. Al contrario, ha introdotto rumore, riducendo l'accuratezza complessiva. Questo è probabilmente dovuto alla scarsa rilevanza semantica dei titoli rispetto al genere musicale, nonché alla possibile ambiguità linguistica.

Un altro problema rilevato è stato l'**overfitting**, chiaramente visibile nell'andamento delle funzioni di loss mostrate in Figura 3. In particolare, la differenza tra la loss di training e quella di test evidenzia che il modello ha imparato troppo bene i pattern presenti nei dati di addestramento, generalizzando poco sui dati nuovi.

Il problema principale, tuttavia, risiede nel dataset. Quest'ultimo non solo è **sbilanciato** (come evidenziato nella matrice di confusione in cui alcune classi contengono pochi esempi), ma presenta anche problemi legati alla qualità dei dati. Il processo di costruzione del dataset, basato sullo scraping automatico di brani da playlist di Spotify associate a ciascun genere, ha introdotto variabilità e potenziali errori di etichettatura. La mancanza di un controllo manuale ha compromesso l'affidabilità delle etichette di classe.

In conclusione, questo studio ha dimostrato la fattibilità della classificazione automatica dei generi musicali tramite modelli di apprendimento profondo, sottolineando al contempo l'importanza:

- della selezione di un'architettura adeguata;
- della qualità e bilanciamento del dataset;
- della rilevanza delle feature impiegate.

Futuri miglioramenti potrebbero derivare dall'ampliamento e dalla pulizia del dataset e dall'uso di tecniche di data augmentation.

5 Demo

5.1 Funzioni della Demo

- Predire il genere musicale di una canzone, mostrando come risultato i **tre generi più probabili**. Se disponibile, viene anche mostrato il **genere reale** della canzone, per consentire un confronto diretto con la predizione. Tuttavia, non sempre il genere reale è presente: questo accade perché le canzoni nel dataset di inferenza sono aggiunte dinamicamente tramite la demo e, in alcuni casi, il recupero automatico delle informazioni fallisce. Ciò può essere dovuto a errori nel titolo o nel nome dell'artista, oppure a discrepanze tra i risultati restituiti dalla libreria Python `musicbrainzngs` e quelli presenti nel nostro dataset originale. Usiamo questa libreria per cercare automaticamente il genere disponibile nel database del sito MusicBrainz.

- Predire il genere selezionando una canzone già presente nel dataset di inferenza (file `CSV`), tramite uno *slider* che permette di scegliere la traccia desiderata.

- Predire il genere di una nuova canzone, inserendo manualmente il **titolo** e l'**artista** negli appositi campi. Il sistema cercherà automaticamente il brano, effettuerà il **sampling di 90 secondi**, estrarrà le **feature audio**, e mostrerà un **player** per l'ascolto. Tramite un apposito tasto sarà possibile ottenere la predizione del genere. Inoltre, la canzone verrà automaticamente aggiunta al dataset di inferenza, così da poterla riutilizzare per ulteriori test in futuro.

5.2 Come usare la Demo

Per utilizzare la demo, è necessario clonare la repository Git del progetto "MusicGenreClassification". Dopodiché, dal terminale, eseguire il seguente comando all'interno della directory del progetto:

```
streamlit run Demo/StreamLitDemo.py
```

Una volta avviata la demo, sarà possibile:

- Selezionare la sezione **"Seleziona canzone esistente"** per effettuare l'inferenza su canzoni e feature audio già presenti nel dataset. È sufficiente scegliere la canzone tramite uno slider e premere il pulsante **"Predici il genere musicale"**: verranno mostrati i tre generi più

probabili e, se disponibile, anche il genere reale della canzone, così da poter confrontare la predizione con il valore corretto. *Nota: non tutte le canzoni dispongono del genere reale. Questo perché i brani nel dataset di inferenza vengono inseriti tramite la demo, che non sempre riesce a recuperare il genere corretto — a causa di errori nei nomi o discrepanze tra le API di MusicBrainz e il nostro dataset. La libreria Python* `musicbrainzngs` *viene utilizzata per cercare automaticamente il genere della canzone.*

- Selezionare la sezione **"Inserisci nuova canzone"**: dopo aver inserito il titolo e l'artista (è importante inserire i nomi corretti), il programma cercherà la canzone online, effettuerà un campionamento di 90 secondi, estrarrà automaticamente le feature e mostrerà un player per ascoltare il brano. A quel punto sarà possibile cliccare sullo stesso pulsante per predirne il genere musicale.
- La canzone appena inserita verrà automaticamente aggiunta al dataset di inferenza, rendendola disponibile anche nella sezione precedente.
- È inoltre presente un pulsante **"Vedi distribuzione dei generi"**, che mostra statistiche relative al training set utilizzato.

5.3 Video dimostrativo

Su git: `VideoMl.mp4`

6 Codice

6.1 Librerie Python Utilizzate

Il progetto è stato sviluppato all'interno di un ambiente virtuale Anaconda. Nel repository GitHub è presente un'esportazione dell'ambiente utilizzato, che può essere importato ed utilizzato tramite il comando:

```
conda env create -f environment.yaml
```

Attenzione: per far funzionare correttamente il progetto e la demo è necessario avere installato `ffmpeg` con l'encoder `libmp3lame`. Questo non è incluso in Anaconda e deve essere scaricato separatamente dal seguente link: `https://www.gyan.dev/ffmpeg/builds/ffmpeg-release-full.7z`.

Di seguito sono riportate le principali librerie, già presenti all'interno di Anaconda, utilizzate durante lo sviluppo del progetto. Queste includono sia

librerie generiche che specifiche per il machine learning e l'elaborazione dei dati.

- **Gestione Dati e Operazioni Numeriche:**
  - `pandas`: Essenziale per la manipolazione e l'analisi di dataset in formato tabellare (CSV, DataFrame).
  - `numpy`: Fondamentale per operazioni vettoriali e matriciali ad alte prestazioni, base per le operazioni sui tensori.
  - `json`: Utilizzato per la lettura e scrittura di file JSON, impiegato per salvare configurazioni (es. parametri) e i report delle metriche dei modelli.
  - `joblib`: Per la serializzazione e il salvataggio di oggetti Python complessi, come gli scaler addestrati, garantendone la persistenza.
- **Apprendimento Automatico e Reti Neurali:**
  - `torch` (`PyTorch`): Il framework principale per la costruzione, l'addestramento e l'inferenza delle reti neurali artificiali. Include:
    - `torch.nn`: Modulo che fornisce le classi per costruire strati (layer) e modelli neurali.
    - `torch.utils.data`: Per la gestione efficiente dei dataset e la creazione di `DataLoader` per l'addestramento batch.
    - `torch.optim`: Contiene gli algoritmi di ottimizzazione, come `SGD`, per l'aggiornamento dei pesi del modello.
    - `torch.utils.tensorboard`: Per l'integrazione con TensorBoard, utile per il monitoraggio e la visualizzazione del processo di addestramento.
  - `scikit-learn` (`sklearn`): Un pilastro per l'apprendimento automatico tradizionale e per utility essenziali:
    - `sklearn.preprocessing.StandardScaler`: Per la normalizzazione (standardizzazione) delle feature numeriche.
    - `sklearn.model_selection.train_test_split`: Per suddividere il dataset in set di training e test in modo stratificato.
    - `sklearn.utils.resample`: Utilizzato per il bilanciamento delle classi tramite oversampling e undersampling.
    - `sklearn.metrics`: Per il calcolo di metriche di classificazione cruciali (`accuracy_score`, `classification_report`, `confusion_matrix`) e la generazione delle relative visualizzazioni.

    * `sklearn.feature_extraction.text.TfidfVectorizer`: Per la vettorizzazione delle feature testuali (titoli dei brani) nel formato TF-IDF.

- **Elaborazione e Acquisizione Audio:**
  - `spotipy`: Client Python per l'API di Spotify, impiegato per la ricerca e l'acquisizione di metadati musicali (titoli, generi).
  - `pytube`: Per la ricerca e il download di contenuti audio da YouTube.
  - `yt-dlp`: Un'alternativa robusta e aggiornata per il download di file multimediali da varie piattaforme online, inclusa l'estrazione del solo audio.
  - `pydub`: Libreria versatile per la manipolazione di file audio (es. campionamento di segmenti, conversione di formati). **Questa libreria si interfaccia con FFmpeg e FFprobe per le sue funzionalità di elaborazione audio.**
  - `librosa`: Una libreria leader per l'analisi audio e l'estrazione di feature musicali di basso livello (es. MFCC, Zero Crossing Rate, Spettrogrammi), fondamentali per la rappresentazione dei brani.
  - `FFmpeg`: Non è una libreria Python, ma un potente **strumento a riga di comando** open-source per la manipolazione di file multimediali. È utilizzato per compiti quali la transcodifica audio/video, il ridimensionamento, il campionamento e la conversione di formati. Il progetto lo sfrutta implicitamente tramite librerie come `pydub` per le operazioni di pre-elaborazione audio.
  - FFprobe: Complementare a FFmpeg, è anch'esso uno **strumento a riga di comando** che permette di analizzare e ispezionare le proprietà dei file multimediali (durata, codec, bitrate, ecc.). Viene utilizzato per ottenere informazioni dettagliate sui file audio prima della loro elaborazione.

- **Visualizzazione e Strumenti di Sviluppo:**
  - `matplotlib.pyplot`: Per la creazione di grafici e visualizzazioni, come le curve di apprendimento e le matrici di confusione.
  - `streamlit`: Framework per lo sviluppo di applicazioni web interattive, utilizzato per la creazione della demo del progetto.
  - `os`: Per operazioni di gestione del sistema operativo, come la manipolazione di percorsi file e directory.

- `random`: Utilizzato per impostare i semi per la riproducibilità degli esperimenti.
- `dotenv`: Per la gestione sicura delle variabili d'ambiente (es. credenziali API).

6.2 Organizzazione del progetto Git

Link alla repository: Repository

Di seguito è riportata la struttura della repository, con una descrizione dei file e delle cartelle principali:

File `videoDemoML.mp4`

Video dimostrativo del funzionamento della Demo realizzata per il progetto.

File `environment.yaml`

Esportazione dll'ambiente di sviluppo utilizzato per il progetto contenente tutte le librerie usate e citate sopra. Contiene anche streamlit per l'esecuzione della Demo. **Attenzione:** per far funzionare correttamente il progetto e la demo è necessario avere installato `ffmpeg` con l'encoder `libmp3lame`. Questo non è incluso in Anaconda e deve essere scaricato separatamente dal seguente link:
`https://www.gyan.dev/ffmpeg/builds/ffmpeg-release-full.7z`.

File `relazioneML.pdf`

PDF della relazione di qi questo progetto.

File `relazioneML.docs`

File .docs della relazione di questo progetto

Cartella `Dataset`

- `BuildDataset.py`: Script che utilizza le API di Spotify (via la libreria `spotipy`) per cercare e salvare in un file CSV i titoli delle canzoni e i relativi generi, utili per la costruzione del dataset.
- `MusicDownloader.py`: Script che contiene una funzione per scaricare una canzone (dato titolo e artista) tramite la libreria `pytube`. Viene effettuato un campionamento di x secondi al centro della traccia utilizzando le librerie `pydub` e `yt-dlp`.

- `FeatureExtractor.py`: Contiene una funzione per estrarre le feature audio precedentemente descritte, secondo la modalità spiegata nel relativo paragrafo.
- `DEF-batch.py`: Script che, ad ogni esecuzione, legge il file CSV generato da `BuildDataset.py` e, riprendendo da dove era stato interrotto, salva in un secondo file CSV le feature audio ottenute grazie alle funzioni dei due file precedenti.
- `sistemaDataset.py`: A partire dal dataset originale contenente 29 generi musicali, questo script effettua il merge di alcuni generi simili, riducendoli a 9 macro-generi principali.
- `id-genere.txt`: File contenente gli identificatori numerici e i nomi associati ai 9 macro-generi utilizzati nel progetto.
- `overSampling.py`: Utilizza il dataset riorganizzato da `sistemaDataset.py` e lo bilancia tramite una procedura di *over-sampling*, in modo da ottenere lo stesso numero di brani per ciascun genere.
  L'output consiste in due file CSV:
  - un training set bilanciato;
  - un test set privo di duplicati.
- `titleOnDataset.py`: Script che legge i file CSV del training set e del test set, unendoli con un file separato contenente gli ID e i titoli dei brani. Aggiunge una nuova colonna al dataset, contenente il titolo del brano in formato testuale (lowercase).

Cartella `Demo`

- `Cartella Dataset`: Contiene alcuni degli script già descritti nella cartella `Dataset`.
- `Cartella Modello`: Contiene i file ausiliari utilizzati nella demo, tra cui:
  - `modello-sicuro.pth`: Pesi del modello addestrato.
  - `scaler-sicuro.pkl`: Parametri per la normalizzazione dei dati.
  - `ModelTraining.py`: Script con la definizione del modello.
  - `preprocessing.py`: Funzioni di preprocessing.
  - `UseModel.py`: Funzione per effettuare inferenza con il modello.

  - `TestTracks.csv`: Dataset di inferenza con le informazioni delle canzoni (titolo, artista, genere, ecc.).
  - `TestTracksFeature.csv`: Feature audio associate alle tracce di cui sopra.
  - `InsertData.py`: Funzioni per inserire nuove tracce nei due file CSV, rendendole disponibili per l'inferenza nella demo.

- `Cartella Track`: Directory in cui viene salvata temporaneamente la canzone scaricata, sia in versione completa che in versione campionata (90 secondi).

- `count.py`: Script che conta il numero totale di canzoni presenti nel training set, suddivise per genere.

- `StreamLitDemo.py`: File principale che lancia la demo su porta locale 8501, permettendo di interagire con tutte le funzionalità implementate.

Cartella `Modello`

Questa cartella contiene i file utilizzati per l'addestramento e la valutazione dei modelli di classificazione del genere musicale.

- `musicClassificator.py`: Script principale per l'addestramento del modello. Gestisce il caricamento dei dati, la definizione del modello e l'intero processo di training.

- `musicClassificator copy.py`: Modifica dell `musicClassificator.py` per il dataset con il testo.

- `libProject.py`: Modulo ausiliario contenente tutte le funzioni utilizzate da `musicClassificator.py`, come la creazione del modello, il preprocessing dei dati e il calcolo delle metriche.

- `metrics.json`: File contenente le metriche di valutazione (accuratezza, loss, ecc.) di alcuni dei modelli testati durante la fase sperimentale.

- `par.json`: File con alcune delle combinazioni di iperparametri provate sui vari modelli durante la fase di training.

Per una descrizione dettagliata degli algoritmi e delle architetture utilizzate, si rimanda alla Sezione 3 *Metodi*.

7 Conclusioni

Il progetto ha permesso di esplorare in profondità le sfide legate alla classificazione automatica dei generi musicali, evidenziando i limiti e le potenzialità delle tecniche di apprendimento automatico applicate a dati audio.

Sebbene i risultati ottenuti in termini di accuratezza non siano stati particolarmente elevati, il lavoro ha fornito importanti spunti di riflessione. In particolare, è emerso che modelli più complessi, come `MLPClassifier` e `DeepMLPClassifier`, sono in grado di superare approcci più semplici come la regressione SoftMax, pur mantenendo una certa stabilità nelle prestazioni anche al variare degli iperparametri.

Tuttavia, la qualità e la rappresentatività delle feature utilizzate, così come la composizione del dataset, si sono rivelati fattori critici. Le feature acustiche, ottenute tramite medie e deviazioni standard calcolate su finestre temporali di un secondo, si sono dimostrate informative solo in parte, mentre l'aggiunta di feature testuali (come il titolo del brano) ha portato più rumore che beneficio. Inoltre, problemi di bilanciamento e rumore nel dataset – derivanti dalla raccolta automatica dei dati e dalla mancanza di un controllo manuale delle etichette – hanno inciso negativamente sulla capacità di generalizzazione dei modelli.

Nonostante queste criticità, l'esperienza ha confermato la fattibilità del problema e ha tracciato una chiara direzione per sviluppi futuri. Tra le possibili evoluzioni del progetto si suggeriscono:

- l'arricchimento e la pulizia del dataset, anche attraverso una revisione manuale delle etichette di genere;
- l'introduzione di tecniche di *data augmentation* audio;
- l'uso di architetture più avanzate, come reti convoluzionali o ricorrenti, in grado di catturare meglio le dinamiche temporali del segnale acustico.

In sintesi, il lavoro ha messo in luce la complessità del problema trattato, ma ha anche mostrato che, con un dataset più robusto e tecniche più sofisticate, è possibile ottenere risultati significativamente migliori nella classificazione automatica dei generi musicali.