Università Politecnica delle Marche
Facoltà di Ingegneria
Dipartimento di Ingegneria dell'Informazione

Lyrics Finder

NLP & BERT per l'Analisi dei Testi Musicali

Micol Zazzarini
Andrea Fiorani
Antonio Antonini

Indice

Elenco degli Snippet di Codice

Elenco delle tabelle

Elenco delle figure

Lyrics Finder

1 BERT: l'algoritmo che ha cambiato il mondo dell'NLP

Il **Natural Language Processing (NLP)** è un campo dell'intelligenza artificiale che si occupa di creare tecnologie in grado di interagire e comprendere il linguaggio umano. Tra le sue applicazioni più rilevanti troviamo la **classificazione automatica dei testi**, un ambito che spazia dalla *sentiment analysis* alla *moderazione dei contenuti*, fino alla *categorizzazione dei documenti*. In questo contesto, il modello **BERT** (acronimo di *Bidirectional Encoder Representations from Transformers*), sviluppato da Google, ha segnato una svolta rivoluzionaria nell'NLP grazie al suo approccio innovativo basato su reti neurali profonde pre-addestrate.
BERT utilizza una rete **Transformer** composta da più livelli di encoder per elaborare il linguaggio in maniera bidirezionale, ossia analizzando contemporaneamente sia il contesto precedente che quello successivo di ogni parola in una frase. Questa capacità gli consente di ottenere una comprensione più precisa e sfumata del testo rispetto ai modelli precedenti, che invece elaboravano il linguaggio in modo unidirezionale. Originariamente pre-addestrato su enormi quantità di dati non etichettati, tra cui l'intera *Wikipedia inglese* e il *Brown Corpus*, BERT può essere ulteriormente perfezionato (*fine-tuning*) per affrontare specifici compiti linguistici con un'accuratezza straordinaria.
Il suo successo si basa su due innovazioni principali:

1. **Masked Language Model (MLM)**, una tecnica di pre-addestramento in cui alcune parole del testo vengono mascherate e il modello deve prevederle basandosi sul contesto circostante. Questo approccio migliora la capacità del modello di cogliere le relazioni tra le parole.

2. **Next Sentence Prediction (NSP)**, che addestra il modello a prevedere se una frase segue logicamente un'altra, migliorando la comprensione delle connessioni tra frasi.

Grazie al suo **meccanismo di attenzione** (*multi-head attention*), BERT assegna un peso probabilistico a ciascun token in una frase, evidenziando quelli più rilevanti per il contesto. Questo processo risolve ambiguità linguistiche e polisemia, permettendo al modello di attribuire significati più precisi alle parole a seconda del loro contesto.
Un aspetto distintivo di BERT è l'adozione di un approccio completamente bidirezionale, che simula il modo in cui la mente umana analizza il linguaggio, valutando l'intero contesto di una frase piuttosto che procedere in una sola direzione. Ciò consente di catturare meglio le sfumature linguistiche e le interazioni tra i termini.
Oltre a essere uno dei più significativi miglioramenti introdotti nei sistemi di ricerca di Google negli ultimi anni, BERT rappresenta un fondamentale passo avanti nella comprensione del linguaggio naturale da parte delle macchine. Grazie alle sue capacità, è stato rapidamente adottato sia nella ricerca accademica che in ambito industriale, diventando uno strumento essenziale per molte applicazioni commerciali legate all'NLP.

2 Il nostro progetto

Un'area particolarmente interessante di applicazione di Bert e del Natural Language Processing è l'analisi dei testi musicali, dai quali è possibile estrarre informazioni significative, come il genere musicale o il contenuto tematico. Tuttavia, la comprensione semantica di testi complessi come quelli musicali, che spesso contengono metafore, giochi di parole e linguaggio informale, rappresenta una sfida per i modelli di machine learning tradizionali. Il nostro progetto si inserisce proprio in questo ambito, con

l'obiettivo di sviluppare una pipeline in grado di classificare automaticamente i generi musicali di una canzone a partire dal suo testo. Il valore di questo lavoro risiede nella sua capacità di automatizzare un compito che richiederebbe enormi risorse umane se svolto manualmente, aprendo così la strada a una serie di applicazioni nel campo della musica, del marketing, e della ricerca musicale. Ad esempio, tale sistema potrebbe essere utilizzato per creare playlist musicali in modo automatico, o per analizzare il contenuto di canzoni attraverso la loro classificazione, offrendo agli utenti un modo per esplorare e scoprire nuovi generi musicali. Il dataset utilizzato per l'addestramento del modello include canzoni provenienti da vari generi musicali, dai quali abbiamo dunque ricavato i testi e i generi stessi. L'approccio di classificazione si basa su una serie di operazioni che vanno a comporre l'intera pipeline di NPL, a partire dal cleaning e pre-processing, fino all'addestramento di un modello di Bert fine-tuned sui dati di testo, con l'obiettivo di ottenere una buona performance nella predizione del genere musicale associato a ciascun brano.

3 Dataset

Il dataset utilizzato contiene informazioni relative ai testi delle canzoni provenienti da vari generi musicali. Ogni riga del dataset rappresenta una singola canzone e include le seguenti colonne:

- **index**: L'indice univoco di ogni riga, che serve per identificare la posizione di ogni canzone nel dataset.
- **song**: Il nome della canzone.
- **year**: L'anno di rilascio della canzone, che fornisce informazioni temporali utili per l'analisi delle tendenze musicali.
- **artist**: Il nome dell'artista o del gruppo musicale che ha interpretato la canzone.
- **genre**: Il genere musicale della canzone, che rappresenta l'etichetta di classificazione principale del dataset. I generi presenti possono variare, includendo, ad esempio, Pop, Rock, Hip-Hop, Jazz e Country.
- **lyrics**: Il testo completo della canzone. Questa colonna è fondamentale per il nostro task, in quanto contiene il contenuto testuale che verrà utilizzato per l'analisi e la classificazione del genere musicale.

4 Exploratory Data Analysis (EDA)

La fase di *Exploratory Data Analysis* è stata fondamentale per comprendere la struttura e le caratteristiche principali del dataset, oltre a individuare eventuali problematiche che avrebbero potuto influenzare negativamente le prestazioni del nostro modello. In questa fase, sono emersi alcuni aspetti importanti relativi alla distribuzione dei generi musicali.
Il numero totale di campioni nel dataset è di 362,237, un ammontare significativo che fornisce una base solida per l'addestramento del modello. I generi musicali nel dataset sono vari, e sono stati identificati i seguenti generi unici: `Pop, Hip-Hop, Not Available, Other, Rock, Metal, Country, Jazz, Electronic, Folk, R&B, Indie`. Questi generi coprono una vasta gamma di stili musicali, ma è importante notare che alcune categorie, come `Not Available` e `Other`, non sono rappresentative di veri e propri generi musicali, ma potrebbero indicare dati incompleti o non classificabili, inutili dunque per il nostro task. Inoltre, la distribuzione dei generi nel dataset è fortemente sbilanciata. I generi più rappresentati sono `Rock`, con 131,377 campioni, e `Pop`, con 49,444 campioni. Al contrario, generi come `Indie` (5,732 campioni), `R&B` (5,935 campioni) e `Folk` (3,241 campioni) sono significativamente meno rappresentati. Questo sbilanciamento delle classi avrebbe potuto introdurre un bias nel modello, poiché avrebbe predetto maggiormente i generi più frequenti, ignorando quelli meno rappresentati. Per affrontare questo problema, come sarà descritto in seguito, abbiamo adottato tecniche di *undersampling*.
La distribuzione dei generi nel dataset è visualizzata nel seguente istogramma in Figura 1 e nella tabella in Figura 2:

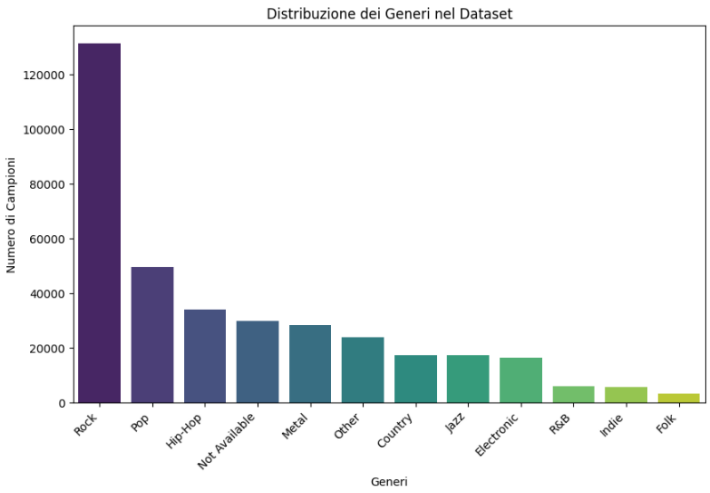

Figura 1: Distribuzione dei generi nel dataset originale.

Genere	Campioni
Rock	131,377
Pop	49,444
Hip-Hop	23,112
Metal	12,768
Electronic	10,231
Country	8,567
Jazz	7,421
R&B	5,935
Indie	5,732
Folk	3,241
Not Available	3,846
Other	1,763

Figura 2: Numero di campioni per genere musicale nel dataset originale.

Come si può osservare, la distribuzione delle classi non è uniforme, con un evidente predominio di generi come `Rock` e `Pop`. I generi minoritari, come `Indie`, `R&B` e `Folk`, sono significativamente meno rappresentati, il che evidenzia un problema di sbilanciamento delle classi. Questo è stato preso in considerazione durante la fase di addestramento del modello, in quanto avrebbe potuto influire negativamente sulla capacità del modello di generalizzare correttamente sui generi meno rappresentati.
Queste informazioni sono state cruciali per la progettazione del modello, poiché ci hanno permesso di prendere decisioni informate sul pre-processing dei dati, sull'implementazione di tecniche di bilanciamento delle classi e sulle metriche di valutazione da utilizzare per misurare la performance del modello.

5 Pipeline

5.1 Pulizia dei Dati

La fase di **cleaning** in una pipeline di Natural Language Processing (NLP), soprattutto in un task di classificazione con modelli come BERT, è il primo passaggio cruciale per preparare i dati in modo che il modello possa processarli efficacemente. Consiste in una serie di operazioni mirate a pulire e preelaborare il testo grezzo per rimuovere elementi indesiderati, normalizzare il contenuto, strutturarlo in una forma più adatta all'elaborazione, e risulta necessaria alla riduzione del rumore nei dati. Infatti, i dataset reali, come nel nostro caso, spesso contengono rumore come punteggiatura inutile, spazi extra, simboli non significativi, o caratteri non validi. Migliora di conseguenza l'efficienza computazionale, in quanto un testo pulito permette al modello di concentrarsi su contenuti rilevanti senza disperdere la capacità computazionale su dettagli irrilevanti. Facilita la comprensione del significato del testo da parte del modello e lo aiuta a generalizzare meglio, evitando che apprenda caratteristiche spurie.
Per quanto riguarda nello specifico Bert, questo risulta già molto robusto al rumore grazie alla sua architettura avanzata e al pre-addestramento su grandi dataset. Tuttavia, abbiamo valutato che pulire il testo in maniera particolarmente accurata potesse migliorare ancora di più la performance del modello e ridurre il carico computazionale. In dettaglio, abbiamo eseguito le seguenti operazioni di pulizia:

1. **Rimozione di tutte le righe del dataset nelle quali i campi `lyrics` o `genre` risultavano `null`, o `Not available`**, non essendo dunque utili per il nostro task.
2. **Rimozione degli spazi bianchi** all'inizio e alla fine dei brani nel campo `lyrics`.
3. **Rimozione di punteggiatura e caratteri speciali**, elementi non rilevanti per l'analisi, mantenendo solo il testo effettivo delle canzoni.
4. **Rimozione di identificatori legati alle canzoni**, che non contribuiscono al significato semantico del testo e possono introdurre rumore.

5. **Rimozione di simboli superflui** che non aggiungono valore semantico e sono probabilmente un residuo di formattazioni precedenti.
6. **Rimozione di moltiplicatori delle strofe**.
7. **Rimozione di testi strumentali o corrotti**, non contenenti informazioni testuali utili al nostro task.
8. **Rimozione di caratteri non ASCII**. Si presume che il modello lavori meglio con caratteri ASCII standard. I caratteri non ASCII possono essere rumore se il corpus è prevalentemente in inglese.
9. **Rimozione di righe vuote o solo spazi**

Il codice 1 mostra le operazioni di pulizia effettuate:

Listing 1: Operazioni di pulizia e filtraggio dei dati per il task di classificazione musicale

```
# Remove rows with missing genres or lyrics, entries where genre is 'Not Available',
    ↪ Strip leading/trailing whitespace from lyrics
data = data.dropna(subset=['genre', 'lyrics'])
data = data[data['genre'].str.lower() != 'not available']
data['lyrics'] = data['lyrics'].str.strip()

filtered = data[data['lyrics'].notnull()]
cleaned = filtered.copy()

# Remove punctuation and song-related identifiers
cleaned['lyrics'] = cleaned['lyrics'].str.replace("[-\?.,\/#!$%\^&\*;:{}=\_~()]", ' '
    ↪ )
cleaned['lyrics'] = cleaned['lyrics'].str.replace("\[(.*?)\]", ' ')
cleaned['lyrics'] = cleaned['lyrics'].str.replace("' | '", ' ')
cleaned['lyrics'] = cleaned['lyrics'].str.replace('x[0-9]+', ' ')

# Remove songs that are instrumental or with corrupted characters
cleaned = cleaned[cleaned['lyrics'].str.strip().str.lower() != 'instrumental']
cleaned = cleaned[~cleaned['lyrics'].str.contains(r'[^\x00-\x7F]+')]
cleaned = cleaned[cleaned['lyrics'].str.strip() != '']
cleaned = cleaned[cleaned['genre'].str.lower() != 'not available']

# Selezione di generi specifici
cleaned = filtered.loc[(filtered['genre'] == 'Pop') |
                       (filtered['genre'] == 'Country') |
                       (filtered['genre'] == 'Rock') |
                       (filtered['genre'] == 'Hip-Hop') |
                       (filtered['genre'] == 'Jazz')]
```

Una volta completata questa fase di pulizia, abbiamo eseguito di nuovo una visualizzazione dei dati per poter capire come proseguire. In dettaglio, in seguito alle precedeti operazioni abbiamo ottenuto un dataset composto da 229.558 elementi, con la distribuzione dei generi musicali mostrata nella tabella 4. Come risulta evidente dai dati, le classi risultano ancora fortemente sbilanciate. La nostra necessità a questo punto era quella di ricavarne un dataset in cui le etichette fossero bilanciate, contenenti lo stesso numero di campioni, e che allo stesso tempo fosse molto numeroso, per garantire buone prestazioni del modello di Bert.

Listing 2: Filtraggio dei generi di interessa e undersampling

```
# Selecting specific genres di interesse
cleaned = cleaned.loc[(cleaned['genre'] == 'Pop') |
                      (cleaned['genre'] == 'Rock') |
                      (cleaned['genre'] == 'Hip-Hop')|
                      (cleaned['genre'] == 'Metal')
                      ]
```

```
7
8  # Perform undersampling
9  balanced_samples = []
10 genres = cleaned['genre'].unique()
11 for genre in genres:
12     genre_subset = cleaned[cleaned['genre'] == genre]
13     if len(genre_subset) >= 23198:
14             balanced_samples.append(genre_subset.sample(n=23198, random_state=42))
15     else:
16             print(f"Not enough samples for genre '{genre}', only {len(genre_subset)}
       ↪ available.")
17             balanced_samples.append(genre_subset.sample(n=len(genre_subset),
       ↪ random_state=42))
18
19     balanced_dataset = pd.concat(balanced_samples, ignore_index=True)
```

Come mostrato nel codice 2, abbiamo trovato questo trade-off andando a considerare solo le classi più numerose: `Rock`, `Pop`, `Hip-Hop`, `Metal`, ed effettuando poi un'operazione di *undersampling*, facendo in modo di avere 23.198 campioni per genere, pari al numero di samples del genere `Metal`, il meno numeroso tra i 4. Abbiamo in questo modo ottenuto il dataset su cui trainare il nostro modello, composto da 92.792 campioni, e perfettamente bilanciato, come mostrato in Figura 3.

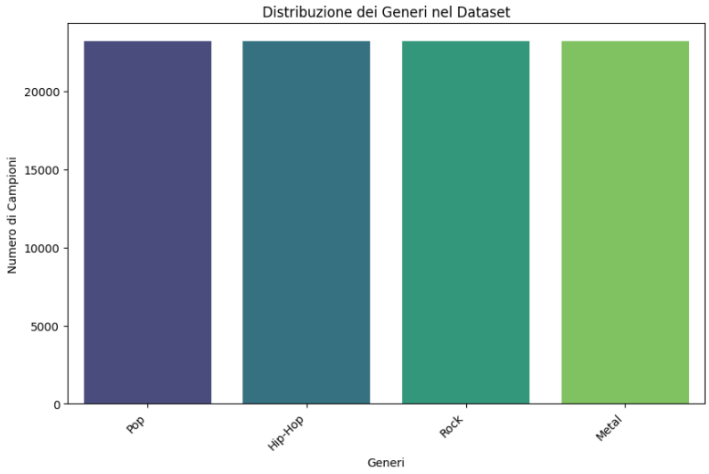

Figura 3: Distribuzione dei generi nel dataset in seguito all'operazione di undersampling.

Genre	Number of Samples
Rock	105,218
Pop	36,486
Hip-Hop	23,235
Metal	23,198
Country	14,193
Jazz	7,542
Electronic	7,349
Other	3,992
R&B	3,358
Indie	2,980
Folk	2,007

Figura 4: Numero di campioni per genere musicale in seguito alle operazioni di pulizia.

5.2 Preprocessing

La fase di **Preprocessing** nella pipeline di NLP consiste in una serie di trasformazioni applicate ai dati testuali per renderli adatti all'elaborazione da parte di modelli di machine learning o deep learning. L'obiettivo è prendere il testo grezzo e convertirlo in una rappresentazione strutturata, numerica e semanticamente rilevante, in modo che i modelli possano apprendere efficacemente le caratteristiche linguistiche. Nel nostro caso, abbiamo eseguito le seguenti operazioni:

1. **Rimozione delle Stop Words**: Si eliminano parole molto comuni e frequenti, come "the", "and", "of", che non portano valore informativo per il compito specifico, per ridurre la dimensionalità del testo e concentrare il modello sulle parole che portano più informazioni semantiche.

2. **Lemmatizzazione**: Riduce ogni parola alla sua forma base o "lemma" utilizzando la conoscenza della grammatica e del contesto, per uniformare le variazioni morfologiche delle parole, migliorando la coerenza del testo senza perdere il significato semantico.

3. **Encoding dei generi**: La colonna che contiene i generi viene trasformata in una rappresentazione numerica, convertendo le etichette categoriali in un formato comprensibile dal modello.

4. **Splittaggio del Dataset**: il dataset viene diviso in un *training set*, utilizzato per addestrare il modello, e un *validation/test set*, utilizzato per validare l'addestramento. L'obbiettivo è garantire che il modello impari in modo generalizzabile e che venga valutato su dati mai visti durante l'addestramento.

5. **Tokenizzazione**: Segmenta il testo in unità più piccole chiamate "*tokens*".

Rimozione delle stop words

Il seguente codice 3 utilizza la libreria `NLTK (Natural Language Toolkit)`, una delle principali librerie Python per il processamento del linguaggio naturale (NLP), per la rimozioni delle stop words. Viene importata la libreria `nltk` e, successivamente, viene scaricato il corpus delle stop words in inglese. Questo corpus fornisce una lista predefinita di parole comuni in diverse lingue, tra cui l'inglese. Successivamente, viene applicata una funzione alla colonna `lyrics`, che contiene i testi delle canzoni, filtrando le parole presenti nella lista. In questo modo, il codice contribuisce a migliorare la qualità dei dati testuali, semplificando la loro elaborazione per le attività di analisi. Si riduce la dimensionalità del testo e si permette al modello di concentrarsi sulle parole che portano effettivamente valore semantico.

Listing 3: Rimozione delle stop words

```
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')

# Remove stop words
stop = stopwords.words('english')
balanced_dataset['lyrics'] = balanced_dataset['lyrics'].apply(lambda x: ' '.join([
    ↪ word for word in x.split() if word not in stop]))
```

Lemmatizzazione

Nel seguente codice 4 viene utilizzata la stessa libreria `NLTK`, nello specifico il modulo `WordNetLemmatizer`, per eseguire la lemmatizzazione dei testi. Il codice inizia con il download del corpus `WordNet`, necessario per il funzionamento del lemmatizzatore, che contiene un ampio database lessicale per la lingua inglese. Viene definito poi il processo che applica il lemmatizzatore a ogni parola del testo: prima il testo viene suddiviso in parole, e poi, ciascuna parola viene ridotta alla sua forma base. Questo processo viene applicato a tutti i brani presenti nella colonna `lyrics`. Convertendo ogni parola in una forma standard, si riduce la variabilità delle parole senza perdere il loro significato.

Listing 4: Lemmatizzazione

```
nltk.download('wordnet')

# Lemmatization
lemmatizer = WordNetLemmatizer()

def lemmatize_text(text):
    return " ".join([lemmatizer.lemmatize(word) for word in text.split()])

balanced_dataset['lyrics'] = balanced_dataset['lyrics'].apply(lemmatize_text)
```

Codifica delle etichette e splittaggio del dataset

Il codice 5 esegue due operazioni cruciali per preparare i dati per l'addestramento di un modello di machine learning: la **codifica numerica** dei generi musicali e lo **splittaggio del dataset**. La prima parte del codice si concentra sulla codifica dei generi: utilizzando la classe `LabelEncoder` di `scikit-learn`, le etichette testuali dei generi musicali vengono trasformate in valori numerici. Questo passaggio è fondamentale perché i modelli di machine learning non possono elaborare direttamente i dati testuali e richiedono una rappresentazione numerica. Nella seconda parte, il dataset viene diviso in due sottoinsiemi utilizzando la funzione `train_test_split` di `scikit-learn`: il set di addestramento,

che sarà utilizzato per insegnare al modello, e il set di test, che servirà per valutarne le performance su dati mai visti prima. Il parametro `test_size=0.20` specifica che il 20% del dataset verrà utilizzato per il test, mentre l'80% rimarrà per l'addestramento. Inoltre, è da notare che l'uso di `stratify` garantisce che la divisione del dataset mantenga una distribuzione bilanciata delle etichette, evitando che il modello venga influenzato da un'eventuale predominanza di una classe, come mostrato nella tabella 1. Queste operazioni sono essenziali per preparare correttamente i dati e permettere al modello di apprendere in modo efficace e generalizzare su nuovi dati.

Y	Distribuzione dei generi nel training set	Distribuzione dei generi nel test set
0	0.249997	0.250013
1	0.250010	0.249960
2	0.249997	0.250013
3	0.249997	0.250013

Tabella 1: Distribuzione dei generi nel training e nel test set

Listing 5: Codifica delle etichette e splittaggio del dataset

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

# Encode genres (codifica i generi in valori numerici utilizzabili nei modelli di
    ↪ machinbe learning)
Y = balanced_dataset['genre']
Y = LabelEncoder().fit_transform(Y)
balanced_dataset['Y'] = Y.tolist()

# Split the dataset into training and testing balancing classes with stratify
X_train, X_test, y_train, y_test = train_test_split(
    balanced_dataset['lyrics'],
    balanced_dataset['Y'],
    test_size=0.20,
    random_state=42,
    stratify=balanced_dataset['Y']
)
```

Tokenizzazione e preparazione dei dati per l'addestramento

Il codice 6 si concentra sulla **tokenizzazione** e la preparazione del dataset per l'addestramento del modello per la classificazione dei generi musicali. Inizialmente, viene utilizzato il tokenizzatore Bert dalla libreria *Hugging Face*, nella versione "uncased" (cioè senza distinzione tra maiuscole e minuscole). Il tokenizzatore suddivide il testo in token che il modello Bert può comprendere, applicando tecniche avanzate come la gestione dei sottotoken. Successivamente, viene definita la classe `LyricsDataset`, che si occupa di preparare i dati per l'addestramento. Al suo interno, sono gestiti tre elementi fondamentali: il testo delle canzoni (`lyrics`), le etichette dei generi musicali (`labels`), e il tokenizzatore stesso. La classe definisce due metodi principali: `__len__`, che restituisce la lunghezza del dataset, e `__getitem__`, che prepara ogni campione del dataset. Per ogni elemento, `__getitem__` utilizza il tokenizzatore per trasformare il testo in un formato che il modello possa utilizzare, ovvero in sequenze di `input_ids` e `attention_mask`. `input_ids` rappresenta i token numerici, mentre `attention_mask` segnala al modello quali token sono reali e quali sono padding (aggiunti per uniformare la lunghezza delle sequenze). I dataset vengono poi preparati separatamente per il training e il test. Vengono convertiti in oggetti della classe `LyricsDataset` e poi caricati in `DataLoader` per essere utilizzati durante l'allenamento. I `DataLoader` gestiscono il batching e la randomizzazione (per il training), facilitando il flusso dei dati nel processo di addestramento del modello.

Listing 6: Tokenizzazione e preparazione dei dati

```
# Hugging Face BERT Tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
3
4  # Tokenization and Dataset preparation
5  class LyricsDataset(Dataset):
6      def __init__(self, lyrics, labels, tokenizer, max_len=128):
7          self.lyrics = lyrics
8          self.labels = labels
9          self.tokenizer = tokenizer
10         self.max_len = max_len
11
12     def __len__(self):
13         return len(self.lyrics)
14
15     def __getitem__(self, item):
16         text = self.lyrics[item]
17         label = self.labels[item]
18
19         encoding = self.tokenizer.encode_plus(
20             text,
21             add_special_tokens=True,
22             max_length=self.max_len,
23             pad_to_max_length=True,
24             truncation=True,
25             return_tensors='pt'
26         )
27
28         return {
29             'input_ids': encoding['input_ids'].flatten(),
30             'attention_mask': encoding['attention_mask'].flatten(),
31             'labels': torch.tensor(label, dtype=torch.long)
32         }
33
34 train_dataset = LyricsDataset(X_train.tolist(), y_train.tolist(), tokenizer)
35 test_dataset = LyricsDataset(X_test.tolist(), y_test.tolist(), tokenizer)
36
37 train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
38 test_loader = DataLoader(test_dataset, batch_size=16)
```

5.3 Modeling e Training

La fase di modeling e training della pipeline di NLP è cruciale per sviluppare un modello capace di apprendere dai dati e di generalizzare su nuovi esempi. In questa fase, viene selezionato un modello pre-addestrato che viene successivamente adattato al task specifico, in questo caso la classificazione dei generi musicali. Dopo aver preparato i dati, si procede con la definizione del modello, e in seguito, durante il training, il modello viene alimentato con il dataset di addestramento, che è stato precedentemente tokenizzato e convertito in sequenze di token numerici. Il training avviene attraverso l'ottimizzazione di una funzione di perdita, utilizzando l'algoritmo *AdamW* per l'aggiornamento dei pesi. Il modello impara a predire correttamente le etichette associate ai testi delle canzoni, minimizzando l'errore rispetto alle etichette reali. Il `DataLoader` gestisce l'elaborazione dei dati in batch, rendendo il training più efficiente e consentendo l'addestramento su grandi volumi di dati. Il processo di training viene inoltre monitorato per evitare problemi come l'*overfitting* e garantire che il modello possa generalizzare bene sui dati non visti.

Modeling

Il codice 7 carica un modello pre-addestrato di Bert e configura un ottimizzatore per il processo di addestramento, utilizzando la libreria `Transformers` di *Hugging Face* e *PyTorch*.
Viene utilizzata la funzione `from_pretrained` della classe `BertForSequenceClassification` per caricare una versione pre-addestrata di Bert per il compito di classificazione di sequenze. Il modello `bert-base-uncased` è una versione di Bert che non distingue tra maiuscole e minuscole, ed è ad-

destrato su una grande quantità di dati in lingua inglese. Questo modello pre-addestrato viene poi adattato al nostro task di classificazione dei generi musicali.
La seconda parte del codice riguarda la configurazione dell'ottimizzatore. Viene utilizzato l'ottimizzatore `AdamW`, una variante dell'algoritmo `Adam`, che è ampiamente utilizzato per l'addestramento di modelli di deep learning.

Listing 7: Modeling

```
# Load pre-trained BERT model for sequence classification
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels
    ↪ =4)

# Optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-5)
```

Training

Il codice 8 implementa il ciclo di addestramento del modello con il monitoraggio delle perdite (loss) e l'utilizzo della strategia di *early stopping*, che consente di fermare l'addestramento prima del tempo prestabilito se il modello non migliora ulteriormente sulle performance di validazione. L'addestramento è eseguito per un massimo di 5 epoche, fermandosi però alla quarta epoca non avendo miglioramenti significativi.
Innanzitutto, vengono definiti i parametri per l'early stopping: `patience`, che è il numero massimo di epoche senza miglioramento della perdita di validazione prima che l'addestramento venga interrotto, è impostato su 2 epoche. La variabile `best_val_loss` viene inizializzata come un numero molto grande (infinito), per tracciare la miglior perdita di validazione osservata fino a quel momento, e `epochs_without_improvement` viene usato per contare il numero di epoche consecutive senza miglioramento.
Per ogni epoca, si esegue un ciclo attraverso il dataset di addestramento (`train_loader`), dove per ogni batch di dati il modello calcola la previsione (*forward pass*), calcola la perdita e aggiorna i pesi tramite *backpropagation*. La perdita per l'epoca viene calcolata come la media delle perdite di tutti i batch e viene registrata in `train_losses`, per tenere traccia del progresso durante l'allenamento.
Dopo l'allenamento su tutti i batch del dataset di addestramento, il modello passa alla fase di valutazione. In questa fase, il modello entra in modalità `eval()` per evitare l'aggiornamento dei pesi durante il passaggio sui dati di test. Per ogni batch del dataset di test (`test_loader`), il modello calcola la perdita di validazione e le predizioni. L'*early stopping* viene applicato dopo ogni epoca di addestramento: se la perdita di validazione corrente è migliore di quella precedente, questa viene aggiornata e il contatore di epoche senza miglioramento viene azzerato. Se invece la perdita di validazione non migliora, il contatore viene incrementato. Se il numero di epoche senza miglioramento raggiunge il valore di patience, l'addestramento viene interrotto anticipatamente, risparmiando tempo computazionale e prevenendo l'*overfitting*.

Listing 8: Training

```
# Early stopping parameters
patience = 2  # Numero massimo di epoche senza miglioramento della validazione
best_val_loss = float('inf')  # Inizializziamo la miglior perdita di validazione come
    ↪ un numero molto grande
epochs_without_improvement = 0  # Conteggio delle epoche senza miglioramento

# Training loop with loss tracking
epochs = 5
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
model = model.to(device)

train_losses = []
val_losses = []

for epoch in range(epochs):
    model.train()
```

```
    epoch_train_loss = 0.0
    for batch in train_loader:
        optimizer.zero_grad()

        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        # Forward pass
        outputs = model(input_ids=input_ids, attention_mask=attention_mask, labels=
    ↪ labels)
        loss = outputs.loss
        loss.backward()

        optimizer.step()

        epoch_train_loss += loss.item()

    epoch_train_loss /= len(train_loader)
    train_losses.append(epoch_train_loss)
    print(f'Epoch {epoch + 1} - Training Loss: {epoch_train_loss:.4f}')

    # Evaluation loop with loss tracking for validation set
    model.eval()
    epoch_val_loss = 0.0
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for batch in test_loader:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            # Forward pass
            outputs = model(input_ids=input_ids, attention_mask=attention_mask,
    ↪ labels=labels)
            loss = outputs.loss
            epoch_val_loss += loss.item()

            # Get predictions
            _, preds = torch.max(outputs.logits, dim=1)

            # Store true labels and predictions
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    epoch_val_loss /= len(test_loader)
    val_losses.append(epoch_val_loss)
    print(f'Epoch {epoch + 1} - Validation Loss: {epoch_val_loss:.4f}')

    # Early stopping logic
    if epoch_val_loss < best_val_loss:
        best_val_loss = epoch_val_loss  # Update best validation loss
        epochs_without_improvement = 0  # Reset the counter if we find an improvement
    else:
        epochs_without_improvement += 1  # Increment the counter if no improvement

    # If no improvement for 'patience' epochs, stop early
    if epochs_without_improvement >= patience:
        print(f'Early stopping triggered after {epoch + 1} epochs without improvement
    ↪ .')
```

```
75          epochs_completed = epoch + 1  # Save the number of epochs completed before
        ↪ stopping
76          break
```

Durante il processo di addestramento del modello, si osserva un comportamento interessante nelle perdite di training e di validazione, che ci fornisce indizi sulle dinamiche del modello. I risultati mostrano un miglioramento consistente nelle prime epoche, ma poi si assiste a un aumento della perdita di validazione. Questi risultati sono mostrati nella tabella 2.

Epoch	**Training Loss**	**Validation Loss**
1	0.7840	0.7589
2	0.6309	0.7203
3	0.4734	0.7755
4	0.3114	0.8813

Tabella 2: Risultati del Training e della Validazione durante le Epoche

Nella prima epoca, la perdita di training è pari a 0.7840 e la perdita di validazione è 0.7589. Nella seconda epoca, si nota una diminuzione sia della perdita di training (0.6309) che di quella di validazione (0.7203), un segno positivo che indica che il modello sta iniziando a imparare dalle sue iterazioni, migliorando in entrambe le fasi, quelle di addestramento e di validazione. Tuttavia, a partire dalla terza epoca, la situazione cambia: mentre la perdita di training continua a diminuire in modo lineare (0.4734), la perdita di validazione inizia a salire, passando da 0.7755 nella terza epoca a 0.8813 nella quarta. Questo cambiamento segnala che, sebbene il modello stia diventando più preciso nel prevedere i dati di addestramento, non è in grado di generalizzare altrettanto bene sui dati di validazione. L'aumento della perdita di validazione indica che il modello potrebbe cominciare a "memorizzare" i dati di addestramento (*overfitting*), piuttosto che apprendere caratteristiche generali che si possano applicare anche a nuovi dati. A questo punto, entra in azione l'algoritmo di *early stopping* interrompendo il training in anticipo rispetto alle 5 epoche previste.
Anche se il dataset risulta bilanciato, il modello potrebbe aver trovato difficoltà a generalizzare a causa della natura complessa dei testi musicali, che spesso contengono lessico specifico e variazioni stilistiche, oltre al fatto che un genere musicale viene definito generalmente non solo dalle parole utilizzate dall'autore ma anche da tutta una combinazione di fattori musicali, culturali e sociali che contribuiscono a caratterizzare il suo stile e la sua appartenenza a una categoria specifica, ad esempio la melodia, il ritmo, gli strumenti utilizzati o gli effetti sonori. Si tratta dunque di un task particolarmente complesso.

5.4 Valutazione della performance

La fase di valutazione della performance è cruciale per comprendere come il modello si comporta sui dati di test e come generalizza rispetto ai dati sui quali non è stato addestrato. Dopo aver addestrato il modello, è fondamentale eseguire una serie di test per misurare le sue capacità predittive, soprattutto in compiti di classificazione come nel nostro caso, dove il modello è chiamato a classificare le canzoni in base al loro genere. In questa fase, vengono utilizzati dunque diversi strumenti e metriche per ottenere una visione completa delle performance del modello.
La **Matrice di Confusione** è uno degli strumenti più utilizzati per valutare le prestazioni di un classificatore, in particolare nei problemi di classificazione multi-classe come quello che stiamo trattando. Essa fornisce una rappresentazione visiva di come le predizioni del modello si confrontano con le etichette vere (*true labels*). E' una tabella che riassume i risultati delle predizioni confrontando il numero di istanze per ciascuna classe che sono state correttamente o erroneamente classificate dal modello. Ogni riga rappresenta una classe vera, e ogni colonna rappresenta le predizioni del modello.

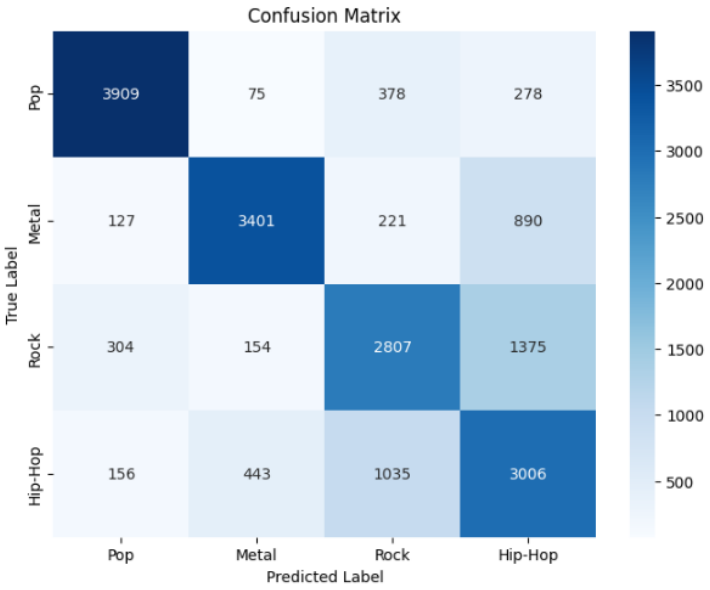

Figura 5: Matrice di confusione

Gli elementi nella diagonale principale rappresentano le istanze correttamente classificate, mentre gli altri valori indicano il numero di errori in cui una classe è stata erroneamente classificata come un'altra. Dalla nostra Matrice di Confusione, in Figura 5, è possibile notare come gli elementi all'interno della diagonale principale siano decisamente più numerosi rispetto agli altri, a indicare come la maggior parte delle predizioni del nostro modello siano corrette. Tuttavia, vediamo come il modello abbia ancora difficoltà a distinguere tra le classi `Rock` e `Hip-Pop`, probabilmente per le similitudini testuali dei brani appartenenti a questi generi musicali.

Il **classification Report** è un altro strumento utile per valutare le prestazioni del modello. Esso fornisce una serie di metriche per ciascuna classe, come la **precisione**, la **recall** e la **F1-score**, essenziali per comprendere non solo la capacità di classificare correttamente le istanze, ma anche l'equilibrio tra la capacità del modello di evitare falsi positivi e falsi negativi.

- **Precisione (Precision)**: È la proporzione di predizioni corrette di una classe rispetto a tutte le predizioni fatte per quella classe.
- **Recall**: È la proporzione di istanze di una classe che sono correttamente identificate dal modello.
- **F1-Score**: È la media armonica tra precisione e recall, ed è utile quando si ha bisogno di un singolo valore che bilanci entrambe le metriche.

Class	Precision	Recall	F1-Score	Support
Pop	0.87	0.84	0.86	4640
Metal	0.84	0.73	0.78	4639
Rock	0.63	0.60	0.62	4640
Hip-Hop	0.54	0.65	0.59	4640
Accuracy			0.71	18559
Macro avg	0.72	0.71	0.71	18559
Weighted avg	0.72	0.71	0.71	18559

Tabella 3: Classification Report per il primo modello di classificazione dei generi musicali

I risultati indicano una performance complessivamente buona, ma con alcune differenze significative tra i generi. Per il genere Pop, il modello ottiene una buona precisione (0.87) e recall (0.84), il che significa che riesce a fare previsioni corrette senza trascurare troppe canzoni di questo genere. L'F1-score elevato di 0.86 indica un buon bilanciamento tra precisione e recall. Nel caso del Metal, la precisione è ancora buona (0.84), ma il modello ha più difficoltà a identificare tutte le canzoni metal, come evidenziato dal recall di 0.73. Questo si traduce in un F1-score di 0.78, che è comunque abbastanza solido, ma inferiore a quello di Pop. Il genere Rock presenta i risultati peggiori. La precisione e il recall sono entrambi relativamente bassi (0.63 e 0.60, rispettivamente), con un F1-score di 0.62 che riflette difficoltà nel distinguere correttamente le canzoni rock dalle altre categorie. Questo potrebbe essere

dovuto a sovrapposizioni con altri generi. Per l'Hip-Hop, la precisione (0.54) è più bassa rispetto al recall (0.65). Ciò significa che il modello identifica abbastanza canzoni hip-hop, ma spesso le confonde con altri generi. L'F1-score di 0.59 indica che il modello ha bisogno di miglioramenti in questo genere per bilanciare meglio precisione e recall.
Nel complesso, il modello ha un'accuratezza globale del 71%, il che significa che il 71% delle predizioni sono corrette. Le medie macro e ponderate delle metriche (precisione, recall, F1-score) sono molto simili, suggerendo che il modello non è sbilanciato verso una singola classe, ma potrebbe essere migliorato per trattare meglio le classi con prestazioni inferiori, come Rock e Hip-Hop. Dunque, mentre il modello mostra una performance abbastanza solida, ci sono opportunità di miglioramento, specialmente per i generi con performance più basse.

6 Ottimizzazione

Dati i risultati del primo modello sviluppato, abbiamo provato ad adottare alcune tecniche per poter ottimizzare la performance. I risultati precedenti hanno mostrato come il modello soffrisse di un problema di *Overfitting* dopo solo poche epoche, e facesse ancora fatica a distinguere tra alcuni generi musicali, che mostravano risultati peggiori degli altri a livello di Precision, Recall, ed F1-Score. Per questo motivo, la prima tecnica di ottimizzazione che abbiamo adottato è stata innanzitutto l'utilizzo di più dati per trainare il nostro modello, assicurando comunque che le classi fossero bilanciate. L'abbiamo fatto integrando alla fase di undersampling anche una fase di **upsampling**, attraverso la **Data Augmentation**. In secondo luogo, con il dataset ottenuto in seguito alla data augmentation, abbiamo poi provato a migliorare ulteriormente la performance attraverso una serie di tecniche che favorissero la generalizzazione e cambiando alcuni parametri del modello.

6.1 Data Augmentation

Facciamo di nuovo riferimento alla tabella 4. Anche in questo caso, abbiamo selezionato solo i 4 generi più numerosi (Rock, Pop, Hip-Hop, Metal). Tuttavia, nella fase di undersampling, anzichè prelevare 23.198 campioni per ogni genere, abbiamo prelevato 30.000 campioni per le classi Rock e Pop, e tutti quelli disponibili per le classi Hip-Hop e Metal. A questo punto, la situazione risulta quella mostrata in Figura 6. Il dataset risulta ora sbilanciato, con 30.000 campioni per le classi Pop e Rock, e circa 23.000 campioni per le classi Hip-Hop e Metal. Per poter bilanciare i generi musicali, abbiamo dunque utilizzato la tecnica di **Data Augmentation**, attraverso **Back Translation** e **Synonym replacement**. La **Back Translation** consiste nel tradurre un testo da una lingua di origine a una lingua target e poi nuovamente indietro alla lingua di origine. Ad esempio, un testo in inglese potrebbe essere tradotto in francese e poi riportato in inglese. Questo processo può introdurre variazioni linguistiche, parafrasi e nuove strutture sintattiche, pur mantenendo il significato originale del testo. Questo metodo è particolarmente utile poiché consente di generare nuove frasi semantiche equivalenti senza alterare l'informazione sottostante. Il codice 9 implementa la funzione per effettuare la back-translation utilizzando i modelli pre-addestrati della famiglia *Helsinki-NLP/Opus-MT* per eseguire le traduzioni. La tecnica di **Synonym Replacement** consiste invece nel sostituire alcune parole del testo originale con i loro sinonimi. Questo approccio permette di creare varianti del

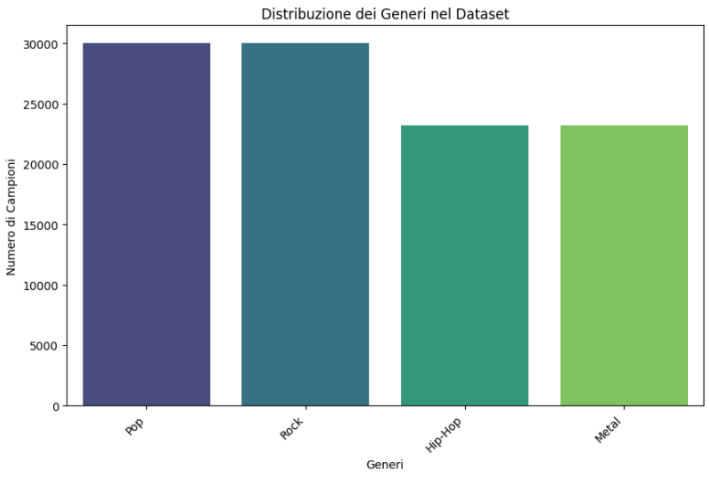

Figura 6: Distribuzione dei generi in seguito a undersampling nella fase di ottimizzazione

testo originale che mantengono lo stesso significato. Nel codice 10 è rappresentata la funzione che implementa la sostituzione con i sinonimi, nella quale per ogni parola, si utilizzano i *synset* forniti da *WordNet*.

Listing 9: Back Translation

```
# Funzione per la back-translation con batch
def back_translate_batch(texts, src_lang='en', tgt_lang='fr', batch_size=16):
    # Modello per la traduzione nella lingua target
    model_name = f"Helsinki-NLP/opus-mt-{src_lang}-{tgt_lang}"
    model = MarianMTModel.from_pretrained(model_name).to(device)
    tokenizer = MarianTokenizer.from_pretrained(model_name)

    # Modello per la traduzione di ritorno
    model_back = MarianMTModel.from_pretrained(f"Helsinki-NLP/opus-mt-{tgt_lang}-{
    ↪ src_lang}").to(device)
    tokenizer_back = MarianTokenizer.from_pretrained(f"Helsinki-NLP/opus-mt-{tgt_lang
    ↪ }-{src_lang}")

    back_translated_texts = []

    # Processa i testi in batch
    for i in range(0, len(texts), batch_size):
        batch = texts[i:i + batch_size]

        # Traduzione batch nella lingua target
        inputs = tokenizer(batch, return_tensors="pt", padding=True, truncation=True)
    ↪ .to(device)
        outputs = model.generate(**inputs, max_length=256)
        translated_texts = [tokenizer.decode(output, skip_special_tokens=True) for
    ↪ output in outputs]

        # Traduzione batch di ritorno nella lingua originale
        back_inputs = tokenizer_back(translated_texts, return_tensors="pt", padding=
    ↪ True, truncation=True).to(device)
        back_outputs = model_back.generate(**back_inputs, max_length=256)
        back_translated_texts.extend([tokenizer_back.decode(output,
    ↪ skip_special_tokens=True) for output in back_outputs])

    return back_translated_texts
```

Listing 10: Synonym Replacement

```
# Funzione per sostituire parole con sinonimi
def synonym_augmentation(text):
    words = text.split()
    augmented_text = []
    for word in words:
        synonyms = set()
        for syn in wn.synsets(word):
            for lemma in syn.lemmas():
                synonyms.add(lemma.name())

        # Sostituisci con un sinonimo se disponibile
        if len(synonyms) > 1:
            synonyms.discard(word)
            augmented_text.append(random.choice(list(synonyms)))
        else:
            augmented_text.append(word)

    return " ".join(augmented_text)
```

Entrambe le tecniche si concentrano sull'aumento della diversità linguistica del dataset, e abbiamo pensato che una combinazione di entrambe potesse essere una buona scelta in quanto, mentre la prima è utile per generare variazioni a livello di frase o struttura, la seconda opera principalmente a livello di parole. Nel codice 11 abbiamo la funzione principale che implementa la Data Augmentation, facendo in modo di avere per ogni classe 30.000 campioni.

Listing 11: Data Augmentation

```
# Funzione principale per la data augmentation
def generate_augmented_data(df, genres_to_augment, target_per_class, batch_size=16):
    augmented_data = []

    for genre, target in target_per_class.items():
        genre_data = df[df['genre'] == genre]
        current_count = len(genre_data)
        samples_needed = max(0, target - current_count)

        print(f"Generando {samples_needed} campioni per il genere: {genre}")

        if samples_needed > 0:
            # Estrai testi originali da augmentare
            original_texts = genre_data['lyrics'].sample(n=samples_needed, replace=
    ↪ True).tolist()

            # Applica la back-translation in batch
            back_translated_texts = back_translate_batch(original_texts, batch_size=
    ↪ batch_size)

            # Applica la sostituzione di sinonimi
            augmented_texts = [synonym_augmentation(text) for text in
    ↪ back_translated_texts]

            # Aggiungi i campioni augmentati alla lista
            augmented_data.extend([[text, genre] for text in augmented_texts])

    # Ritorna un DataFrame con i nuovi campioni
    augmented_df = pd.DataFrame(augmented_data, columns=['lyrics', 'genre'])
    return augmented_df


# Distribuzione desiderata (target per classe)
target_per_class = {
    'Pop': 30000,
    'Rock': 30000,
    'Hip-Hop': 30000,
    'Metal': 30000,
}

# Generi da augmentare
genres_to_augment = [genre for genre in target_per_class.keys() if len(
    ↪ balanced_dataset[balanced_dataset['genre'] == genre]) < target_per_class[genre
    ↪ ]]

# Genera nuovi campioni
augmented_df = generate_augmented_data(balanced_dataset, genres_to_augment,
    ↪ target_per_class)

# Unisci i nuovi dati al dataset originale
augmented_dataset = pd.concat([balanced_dataset, augmented_df])
```

Abbiamo dunque eseguito lo stesso training del caso precedente con questo nuovo dataset aumentato di 120.000 campioni. I risultati sono rappresentati di seguito nella Matrice di Confusione in Figura 7 e

nel Classification Report in Tabella 4. La situazione risulta essere pressochè uguale al caso precedente, con una buona performance generale del modello ma un maggior numero di errori in corrispondenza dei generi Rock e Hip-Hop. Anche dal classification report, viene evidenziato come l'uso del dataset aumentato abbia migliorato le performance del modello di classificazione dei generi musicali in modo significativo per alcune classi, sebbene vi siano anche alcune aree in cui i risultati non siano cambiati drasticamente. Per il genere Pop la precisione è aumentata dal 0.87 al 0.90, con una stabilizzazione del valore di F1-Score a 0.86. Questo indica che il modello è diventato più preciso nell'identificare correttamente i brani Pop. Per il Rock il miglioramento più evidente si osserva nella Recall, da 0.60 a 0.75, con un conseguente incremento del F1-Score da 0.62 a 0.66. Questo suggerisce che il modello ha migliorato la capacità di identificare brani Rock, riducendo il numero di falsi negativi. Miglioramenti minimi invece si hanno invece per i generi Metal e Hip Hop. Per quanto riguarda le metriche globali, la precisione è aumentata a 0.73, che riflette un miglioramento complessivo della performance tra le classi. L'aumento del dataset ha permesso dunque al modello di migliorare significativamente le prestazioni su generi come Pop e Rock, probabilmente grazie a un arricchimento delle variazioni linguistiche o testuali dei dati. Tuttavia, il miglioramento per i generi Metal e Hip-Hop è stato meno pronunciato, evidenziando possibili limiti nell'augmentation utilizzata o una maggiore complessità nel distinguere questi generi dai loro concorrenti. Il grafico in Figura 8 inoltre mostra l'andamento della *training loss* (in blu) e della *validation loss* (in arancione) durante tre epoche di addestramento di un modello. La *training loss* diminuisce costantemente, indicando che il modello sta migliorando la sua capacità di adattarsi ai dati di training. Tuttavia, la *validation loss* mostra un comportamento opposto: rimane quasi costante durante la prima epoca e successivamente inizia ad aumentare. Questo suggerisce che il modello potrebbe iniziare a sovradattarsi ai dati di training, perdendo la capacità di generalizzare sui dati di validazione. Il miglioramento continuo della training loss e l'aumento della validation loss indicano un problema di *overfitting*. Dopo la prima epoca, il modello non riesce a migliorare la sua performance sui dati di validazione.

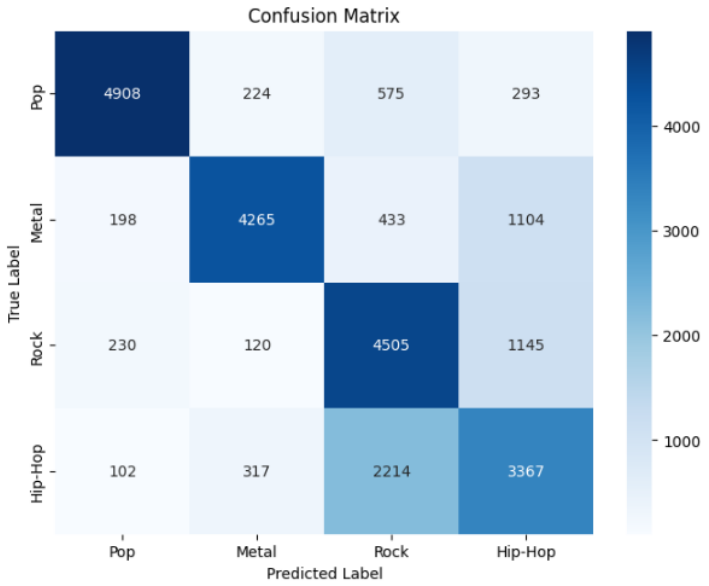

Figura 7: Matrice di confusione ottenuta con il dataset aumentato

Class	Precision	Recall	F1-Score	Support
Pop	0.90	0.82	0.86	6000
Metal	0.87	0.71	0.78	6000
Rock	0.58	0.75	0.66	6000
Hip-Hop	0.57	0.56	0.57	6000
Accuracy			0.71	24000
Macro avg	0.73	0.71	0.72	24000
Weighted avg	0.73	0.71	0.72	24000

Tabella 4: Classification Report per il primo modello di classificazione dei generi musicali ottenuto con i dataset aumentato

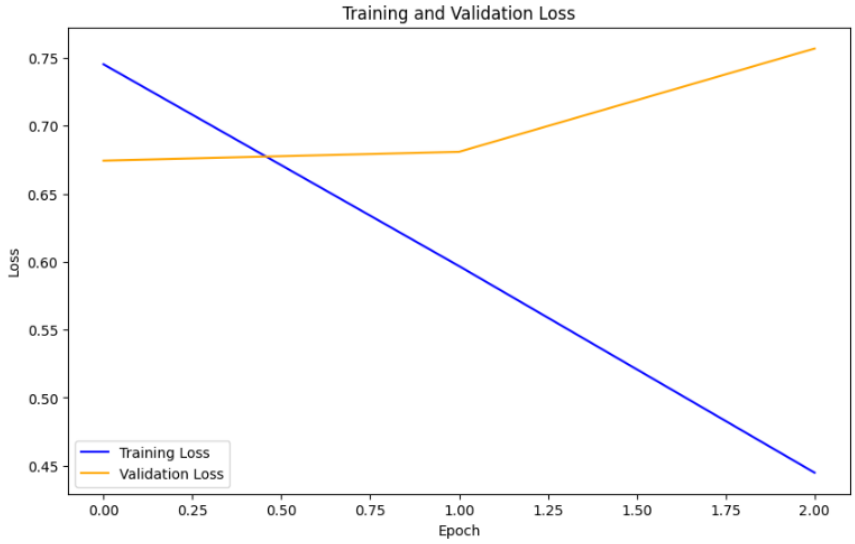

Figura 8: Andamento di training e validation loss durante l'addestramento

6.2 Ulteriori tecniche di ottimizzazione

Per tentare di mitigare i problemi riscontrati in precedenza, favorire la generalizzazione e migliorare i risultati del modello, abbiamo adottato in seguito le seguenti tecniche:

1. **Aumento della lunghezza massima dei token**
2. **Riduzione del learning rate**
3. **Dropout**
4. **Focal Loss**

L'aumento della lunghezza massima dei token, consente al modello di catturare una porzione più ampia del contenuto originale. In molti casi, tra i quali l'analisi di testi lunghi, come i testi delle canzoni, limitare i token a una lunghezza di 128 significa che una parte significativa del testo viene troncata, riducendo la quantità di informazioni disponibili per il modello. Abbiamo dunque incrementato la lunghezza `max_len` a 512, permettendo al modello di considerare una sequenza più lunga del testo durante l'addestramento e l'inferenza. Abbiamo inoltre ridotto il valore del *learning rate* da 3e-5 a 2e-5, permettendo aggiornamenti più piccoli ai pesi del modello, consentendogli di esplorare meglio il paesaggio della funzione di perdita e di avvicinarsi a un minimo locale o globale con maggiore precisione. Il *dropout* è invece una tecnica di regolarizzazione utilizzata nei modelli di deep learning per ridurre il rischio di overfitting e migliorare la capacità di generalizzazione del modello. Consiste nel disattivare (o "spegnere") casualmente una percentuale di neuroni durante il training, impedendo al modello di diventare eccessivamente dipendente da specifiche connessioni o unità. Applicando questa tecnica, è possibile ottenere modelli più affidabili e che si comportano meglio sui dati di test. Il codice 12 mostra come abbiamo implementato il dropout nel nostro modello.

Listing 12: Dropout

```
config = BertConfig.from_pretrained(
    'bert-base-uncased',
    num_labels=4,  # Numero di classi
    hidden_dropout_prob=0.2,  # Aumenta il dropout
    attention_probs_dropout_prob=0.2  # Dropout per l'attenzione
)
```

Per poter affrontare inoltre il problema degli "esempi difficili", abbiamo poi utilizzato una *Focal Loss*, illustrata nel codice 13, che modifica la *Cross-Entropy Loss* per assegnare un peso maggiore a questi ultimi, e ridurre il peso degli esempi facili. Utilizza per questo due parametri:

- **Alpha** (α): controlla il peso della perdita per una classe specifica.
- **Gamma** (γ): regola l'intensità dell'effetto di focalizzazione, aumentando il peso degli esempi difficili.

Listing 13: Focal Loss

```
class FocalLoss(nn.Module):
    def __init__(self, alpha=1, gamma=2, reduction='mean'):
        super(FocalLoss, self).__init__()
        self.alpha = alpha
        self.gamma = gamma
        self.reduction = reduction

    def forward(self, inputs, targets):
        ce_loss = F.cross_entropy(inputs, targets, reduction='none')
        pt = torch.exp(-ce_loss)
        focal_loss = self.alpha * (1 - pt) ** self.gamma * ce_loss

        if self.reduction == 'mean':
            return focal_loss.mean()
        elif self.reduction == 'sum':
            return focal_loss.sum()
        else:
            return focal_loss
```

I risultati ottenuti sono mostrati nella Matrice di Confusione in Figura 9 e nel Classification Report in tabella 5.

Class	Precision	Recall	F1-Score	Support
Pop	0.91	0.82	0.86	6000
Metal	0.79	0.80	0.80	6000
Rock	0.62	0.67	0.65	6000
Hip-Hop	0.58	0.58	0.58	6000
Accuracy			0.72	24000
Macro avg	0.73	0.72	0.72	24000
Weighted avg	0.73	0.72	0.72	24000

Tabella 5: Classification Report ottenuto in seguito a ottimizzazione

Il confronto tra i tre modelli di classificazione evidenzia che, seppur con differenze marginali, l'ultimo modello ottimizzato risulta complessivamente il migliore. Questo modello raggiunge la più alta accuratezza (0.72) rispetto ai precedenti (entrambi con 0.71) e mostra un bilanciamento più efficace tra precisione e richiamo nelle classi principali, in particolare Metal, che ottiene il miglior F1-Score (0.80). Inoltre, la classe Pop mantiene prestazioni elevate e stabili (F1-Score 0.86) con un incremento nella precisione. Sebbene il modello ottimizzato non superi il secondo modello nella gestione della classe Rock (F1-Score di 0.65 contro 0.66), né migliori in modo significativo le prestazioni su Hip-Hop, il suo comportamento più uniforme e la migliore gestione complessiva delle classi principali lo rendono la scelta preferibile per l'implementazione.

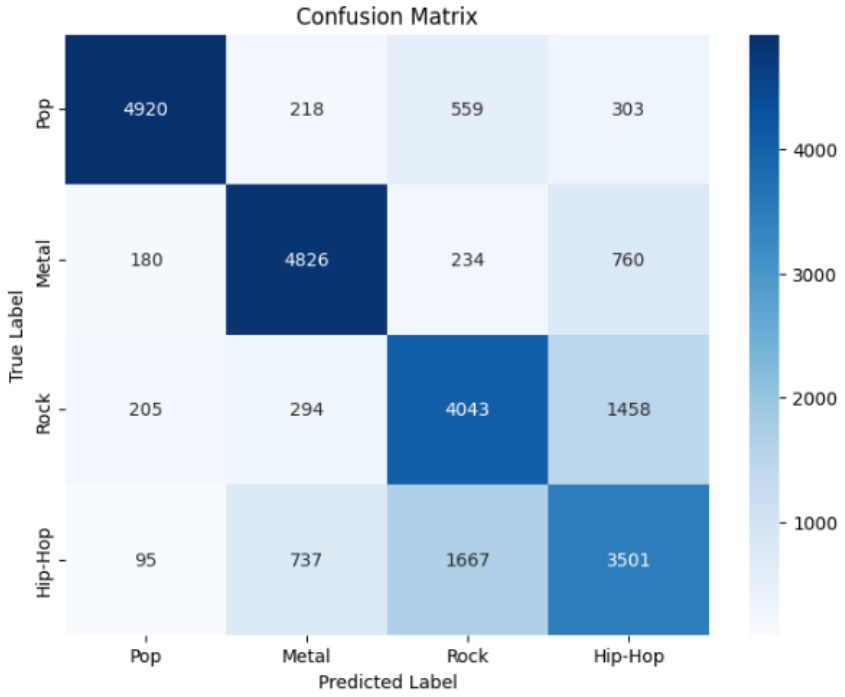

Figura 9: Matrice di Confusione del modello ottimizzato

7 Risultati e Lavori Futuri

Il presente progetto ha esplorato l'utilizzo di BERT per il task di classificazione dei generi musicali a partire dai testi delle canzoni. Attraverso la costruzione di un'intera pipeline, comprensiva di cleaning, preprocessing, modellazione e valutazione, è stato possibile raggiungere buoni risultati, con un'accuratezza del modello pari a 72-73%, anche grazie all'utilizzo di varie tecniche di ottimizzazione. Il risultato ottenuto può essere attribuito alla complessità intrinseca del task. La classificazione dei generi musicali basandosi esclusivamente sul testo delle canzoni presenta diverse sfide: i testi musicali sono spesso ambigui, caratterizzati da metafore, contaminazioni culturali e linguistiche, e non sempre presentano pattern linguistici univoci associabili a specifici generi. Inoltre, l'eventuale sovrapposizione stilistica tra generi e la soggettività delle etichette utilizzate nel dataset rappresentano ulteriori fattori critici.
Per il futuro, si propongono alcune direzioni di ricerca per migliorare la performance del modello. E' consigliabile utilizzare un dataset più grande, che consenta di rappresentare meglio le peculiarità di ciascun genere musicale e ridurre il rischio di generalizzazioni limitate. L'utilizzo di un hardware più adeguato consentirebbe di gestire modelli più complessi e un numero maggiore di campioni senza sacrificare l'efficienza computazionale. Si suggerisce, inoltre, di sperimentare ulteriormente con diverse tecniche di data augmentation, e per evitare fenomeni di overfitting, potrebbero essere implementate anche tecniche di regolarizzazione aggiuntive. Infine, un'accurata procedura di hyperparameter tuning potrebbe contribuire a identificare le combinazioni ottimali di parametri per massimizzare le prestazioni del modello.

8 Note finali

Per lo sviluppo del nostro progetto (vedi [ZM24]), abbiamo adottato un approccio flessibile per garantire l'accessibilità e l'adattabilità alle esigenze di tutti gli utenti. In particolare, abbiamo strutturato sia un codice eseguibile in ambiente locale che uno eseguibile su *Google Colab*. L'esecuzione locale è pensata per utenti che dispongono di una GPU adeguata al training di modelli di deep learning, in grado di accelerare significativamente i processi di allenamento. Tra le GPU consigliate per un utilizzo locale troviamo modelli della serie *NVIDIA*, come la *RTX 3060*, *RTX 3070*, *RTX 3080* o superiori, tutte dotate di un'elevata capacità di calcolo compatibili con le librerie *CUDA* necessarie per il deep

learning. Tuttavia, per coloro che non dispongono di risorse hardware di questo tipo, come nel nostro caso, abbiamo eseguito i nostri esperimenti su *Google Colab*, sfruttando le GPU disponibili sulla piattaforma. Questa scelta ci ha permesso di condurre test e allenamenti in modo efficiente, mantenendo al contempo una compatibilità universale del progetto.

Bibliografia

[ZM24] Antonini Antonio Zazzarini Micol, Fiorani Andrea. Liricsfinder. https://github.com/125ade/LiricsFinder.git, 2024. Repository GitHub del progetto.