
III PROGETTO DATA SCIENCE







III PROGETTO DATA SCIENCE

ANALISI SU UN DATASET CON L'USO DI BERT

TESINA DI:
DATA SCIENCE

ESAME PER IL CORSO TENUTO DAL PROF. DOMENICO URSINO, DURANTE
L'ANNO ACCADEMICO 2024-2025
9 CFU

AUTORI:
GABRIEL PIERCECCHI
TOSCA PIERRO

PROFESSORI:
DOMENICO URSINO
MICHELE MARCHETTI

ANNO ACCADEMICO 2024-2025

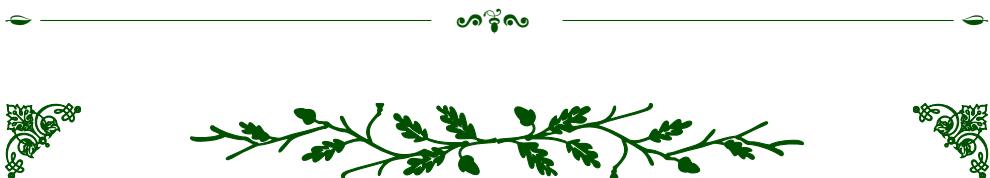




Indice

1 BERT	1
1.1 Introduzione a BERT	1
 ANALISI	 2
2 Analisi del dataset	3
2.1 ETL	3
2.1.1 Modifica campi NaN	4
2.1.2 Visualizzazione del dataset	4
2.1.3 Pulizia del codice	5
2.2 Realizzazione del modello	7
3 Risultati	9
3.1 Allenamento	9
3.2 Risultati	20
3.2.1 Train & Validation	20
3.2.2 Test	21





1. BERT

In questa tesina, verrà analizzato un dataset riguardante la Letteratura inglese <https://www.kaggle.com/datasets/abdelrahmanekhaldi/english-poem-dataset>, con l'obiettivo di applicare metodi di analisi del linguaggio naturale, in particolare l'uso del modello BERT (Bidirectional Encoder Representations from Transformers).



1.1. Introduzione a BERT

Il campo dell'elaborazione del linguaggio naturale (NLP) ha compiuto notevoli progressi con l'introduzione di modelli basati su reti neurali profonde. Tra questi, **BERT** (Bidirectional Encoder Representations from Transformers) si distingue per la capacità di apprendere rappresentazioni contestuali bidirezionali a partire da testo non etichettato, considerando simultaneamente sia il contesto a sinistra che a destra di ogni parola.

BERT è un modello contestuale, in grado di generare rappresentazioni delle parole tenendo conto dell'intera frase in cui sono inserite. Il suo sviluppo si è basato su ricerche e approcci innovativi, tra cui Semi-supervised Sequence Learning, Generative Pre-Training, ELMo, OpenAI Transformer, ULMFit e il Transformer. A differenza di questi modelli, che sono unidirezionali o solo parzialmente bidirezionali, BERT adotta un'architettura completamente bidirezionale.

L'innovazione principale di BERT risiede proprio nell'addestramento bidirezionale del modello di linguaggio, che consente di comprendere in modo più profondo il contesto e la struttura del linguaggio rispetto ai modelli tradizionali. Grazie a questa caratteristica, BERT riesce a considerare simultaneamente sia i token precedenti che quelli successivi, migliorando significativamente l'accuratezza nelle attività di NLP.



ANALISI





2. Analisi del dataset

La prima parte dell'analisi si concentra su una dettagliata esplorazione del dataset. Di seguito, per una migliore comprensione, vengono riportate le sue colonne con allegata descrizione:

- **Titolo:** contiene i titoli delle poesie incluse nel dataset.
- **Poesia:** memorizza il testo delle poesie estratto dal sito web della Poetry Foundation.
- **Poeta:** elenca i poeti che hanno scritto le poesie.
- **Genere:** rappresenta la classificazione emotiva assegnata a ciascuna poesia in base al contenuto del testo.



2.1. ETL

Una volta scaricato il dataset in formato .CSV e caricato su *Jupyter Notebook*, sono state eseguite operazioni di pulizia e visualizzazione dei dati utilizzando le librerie pandas, matplotlib e seaborn.

```
1 drive.mount('/content/drive', force_remount=True)
2 file_path = '/content/drive/MyDrive/PoemDataset.csv'
3 poem = pd.read_csv(file_path)
4 poem.head()
```

Le quali hanno restituito come output le prime 5 istanze del dataset (Figura 1).



	Title	Poem	Poet	Genre
0	Search	Wandered tonight through a cityas ruined as a ...	Hester Knibbe	Fear
1	A Poem for the Cruel Majority	The cruel majority emerges!Hail to the cruel m...	Jerome Rothenberg	Anger
2	"Do Not Embrace Your Mind's New Negro Friend"	Do not embrace your mind's new negro friendOr ...	William Meredith	Anger
3	The Greatest Love	She is sixty. She livesthe greatest love of he...	Anna Swir	Love
4	Bilingual/Bilingüe	My father liked them separate, one there,one h...	Rhina P. Espaillat	Anger

Figura 1: Output che verifica il corretto caricamento del file Poem.csv

2.1.1 Modifica campi NaN

Confermato il corretto caricamento del dataset, si è proceduto alla gestione delle istanze contenenti campi *Nan*. Per prima cosa si è cercato di capire se ci fossero campi vuoti e, in caso di risposta affermativa, quali fossero.

```
1 poem.isnull().sum()
```

Successivamente si è passati ad eliminare le righe contenenti i valori nulli.

```
1 poem = poem.dropna()
```

2.1.2 Visualizzazione del dataset

In seguito alla pulizia del dataset, sono state mostrate le varie emozioni contenute nella tabella *Genre*

```
Counter({'Fear': 3131,
        'Anger': 1259,
        'Love': 696,
        'Sadness': 1541,
        'Joy': 2521,
        'Surprise': 852})
```

rappresentate poi visivamente in un grafico nella Figura 2.



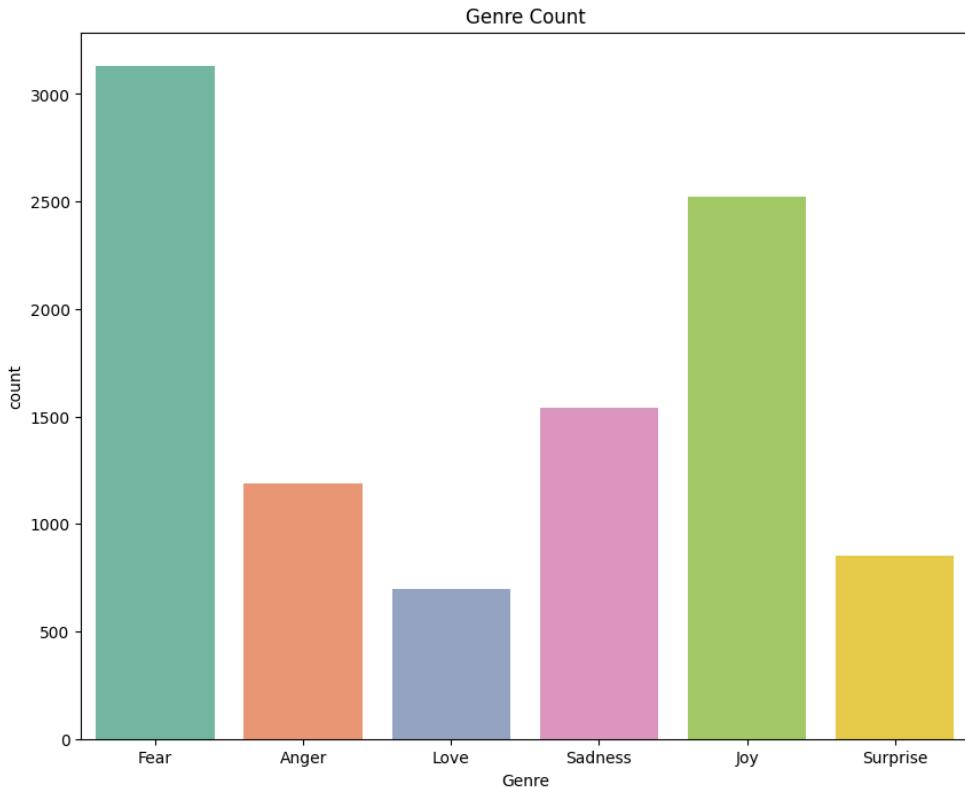


Figura 2: Grafico che mostra la distribuzione dei sentimenti

2.1.3 Pulizia del codice

```
1 sw = stopwords.words('english')
2
3 def clean_poem_text(text):
4     if isinstance(text, str):
5         text = text.lower()
6
7     text = re.sub(r"[^a-zA-Z?!. ,]", " ", text)
```



```
8
9     punctuations = '@#!?+&* [] -%.:;/() ;$=><|{}^,.' +
10    " ``~" + '_'
11
12    for p in punctuations:
13        text = text.replace(p, ' ')
14
15    text = [word for word in text.split() if word
16        not in sw]
17
18    text = " ".join(text)
19
20    return text
21 else:
22     return ""
23 poem['cleaned_poem'] = poem['Poem'].apply(
24     clean_poem_text)
25
26 print("Poema originale:", poem['Poem'].head())
27 print("Poema pulito:", poem['cleaned_poem'].head())
```

Questo codice ha permesso di pulire il testo delle poesie contenute. Nel dettaglio, la funzione `clean_poem_text` esegue i seguenti passaggi:

- **Controllo del tipo di dato:** Verifica che il testo sia una stringa prima di applicare ulteriori operazioni.
- **Conversione in minuscolo:** Tutto il testo viene trasformato in minuscolo per uniformarlo.
- **Rimozione di caratteri non alfanumerici:** Rimuove tutti i caratteri che non sono lettere dell'alfabeto o segni di punteggiatura utili (come punti e virgolette).
- **Rimozione della punteggiatura:** Vengono eliminati caratteri speciali come simboli e segni di punteggiatura non necessari.
- **Rimozione delle stopwords:** Le parole comuni (stopwords) vengono rimosse per evitare che influenzino l'analisi del testo.

- **Tokenizzazione e riformattazione:** Il testo viene suddiviso in parole e poi ricomposto in una stringa pulita, senza le parole rimosse.

2.2. Realizzazione del modello

```
1 poem_shuffled = poem.sample(frac=1, random_state=42).  
2     reset_index(drop=True)  
3  
4 split_idx = int(len(poem_shuffled) * 0.8)  
5 print("Total length of dataset:", len(poem_shuffled))  
6  
7 train_poem = poem_shuffled.iloc[:split_idx, :]  
8 test_poem = poem_shuffled.iloc[split_idx:, :]  
9  
10 print('Training set length:', len(train_poem))  
11 print("Training set distribution:")  
12 print(train_poem.groupby(['Genre'])['Poem'].count())  
13  
14 print('Test set length:', len(test_poem))  
15  
16 print("Test set distribution:")  
17 print(test_poem.groupby(['Genre'])['Poem'].count())
```

Il codice del listato 2.2 è servito per mescolare il dataset in modo da garantire che non ci fossero bias nel processo di separazione.

La funzione utilizzata per mescolare casualmente tutte le righe del dataset è `sample(frac=1, random_state=42)`, la quale utilizza un seme fisso (`random_state=42`) per rendere il processo riproducibile.

Successivamente, è stato calcolato l'indice che rappresenta l'80% della lunghezza totale del dataset, utilizzato per separare i dati in un training set e un test set. Il training set comprende le righe dalla prima fino all'indice calcolato, mentre il test set è composto dalle righe successive.



Il codice poi ha calcolato e stampato la lunghezza del training set per confermare la quantità di dati destinata a questa parte del processo.

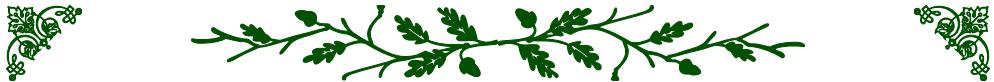
Inoltre, è stata eseguita un'analisi sulla distribuzione dei generi (mostrata in Figura 3) all'interno del training set, utilizzando il metodo `groupby`, per assicurarsi che i dati fossero distribuiti in modo adeguato tra le diverse categorie.

Lo stesso tipo di analisi è stata fatta anche sul test set, per verificare che la distribuzione delle categorie nel set di test rispecchiasse quella del training set.

```
Training set distribution:  
Genre  
Fear      2003  
Joy       1613  
Sadness   986  
Anger     759  
Surprise  545  
Love      446  
Name: count, dtype: int64  
  
Validation set distribution:  
Genre  
Fear      501  
Joy       404  
Sadness   247  
Anger     190  
Surprise  136  
Love      111  
Name: count, dtype: int64  
  
Test set distribution:  
Genre  
Fear      626  
Joy       504  
Sadness   308  
Anger     238  
Surprise  171  
Love      139  
Name: count, dtype: int64
```

Figura 3: Distribuzione del Genre





3. Risultati

In questo capitolo verrà introdotta la costruzione del modello di addestramento basato su BERT, descrivendo i passaggi necessari per la preparazione dei dati.



3.1. Allenamento

```
1 device = torch.device("cuda:0" if torch.cuda.  
    is_available() else "cpu")  
2 device
```

Listato 3.1: Configurazione del device in PyTorch

Per garantire un'elaborazione efficiente, il codice costruito stabilisce prima su quale dispositivo eseguire il modello, scegliendo automaticamente tra GPU e CPU in base alla disponibilità di una scheda grafica compatibile con CUDA (Listato 3.1).

Successivamente (Listato 3.2), viene utilizzato il tokenizer `BertTokenizer` per processare le poesie contenute nel dataset. Il tokenizer viene inizializzato con il modello pre-addestrato `bert-base-uncased`, impostando la conversione in minuscolo per uniformare il testo. Eventuali valori mancanti nella colonna dei testi sono stati sostituiti con stringhe vuote per evitare errori durante l'elaborazione.

Dopo questa fase di pre-processing, il codice ha estratto i testi delle poesie e le etichette dei generi, convertendoli in liste: per ogni poesia, calcola la lunghezza in token, includendo i token speciali di BERT, e determina la lunghezza massima tra tutti i testi.

Infine, il dataset è stato tokenizzato con un limite massimo di 200 token per ogni poesia e le lunghezze dei token per ogni testo sono state salvate, preparando così i dati per l'addestramento del modello.

```
1 tokenizer = BertTokenizer.from_pretrained('bert-base-  
    uncased', do_lower_case=True)
```

```

2
3 train_poem['Poem'] = train_poem['Poem'].fillna('')
4
5 sentences = train_poem['Poem'].tolist()
6 labels = poem['Genre'].tolist()
7
8 token_lengths = [len(tokenizer.encode(sentence,
9     add_special_tokens=True)) for sentence in
10    sentences]
11
12 max_length = max(token_lengths)
13
14 for txt in poem['Poem']:
15     tokens = tokenizer.encode(txt, max_length=200)
16     token_lens.append(len(tokens))
17
18 encoded_inputs = tokenizer.batch_encode_plus(
19     sentences,
20     padding=True,
21     truncation=True,
22     max_length=200,
23     return_tensors="pt"
24 )
25
26 print(encoded_inputs)

```

Listato 3.2: Tokenizzazione

Proseguendo, è stata eseguita la tokenizzazione delle frasi del dataset utilizzando il `tokenizer` di BERT, gestendo il padding, il troncamento e l'impostazione della lunghezza massima dei token per ogni frase. In particolare, la funzione `batch_encode_plus` è stata utilizzata per tokenizzare l'intero dataset in un unico passaggio: le frasi vengono troncate a una lunghezza massima di 200 token e viene applicato il padding per allinearle alla stessa lunghezza. I risultati sono restituiti come tensori PyTorch, pronti per l'addestramento del modello.

Per visualizzare la tokenizzazione di una poesia, è stata creata una tabella di output (Figura 4). In essa, i token vengono prima organizzati, seguiti dagli ID dei token e, infine,

dalla maschera di attenzione, rappresentata in un array NumPy.

Successivamente, viene stampato il testo originale della poesia e una tabella dettagliata che mostra ciascun token, il suo ID corrispondente e il valore della maschera di attenzione, che indica se un token è rilevante per il modello.

Tokenizzazione dettagliata:		
Tokens	Token IDs	Attention Mask
[CLS]	101	1
-	1517	1
after	2044	1
gwen	11697	1
##do	3527	1
##lyn	9644	1
brooks	8379	1
##no	3630	1
matter	3043	1
the	1996	1
pull	4139	1
toward	2646	1
brink	20911	1
.	1012	1

Figura 4: Tabella token

```

1 label_encoder = LabelEncoder()
2
3 label_encoder.fit(poem['Genre'])
4
5 class PoemDataset(Dataset):
6     def __init__(self, poem, tokenizer, max_length):
7

```



```
8     self.sentences = poem['Poem'].tolist()
9     self.labels = poem['Genre'].tolist()
10    self.tokenizer = tokenizer
11    self.max_length = max_length
12
13    def __len__(self):
14        return len(self.sentences)
15
16    def __getitem__(self, idx):
17        sentence = self.sentences[idx]
18        label = self.labels[idx]
19
20        label = label_encoder.transform([label])[0]
21
22        encoded = self.tokenizer(
23            sentence,
24            padding='max_length',
25            truncation=True,
26            max_length=self.max_length,
27            return_tensors='pt'
28        )
29        input_ids = encoded['input_ids'].squeeze(0)
30        attention_mask = encoded['attention_mask'].
31        squeeze(0)
32
33        (input_ids, attention_mask, label)
34        return input_ids, attention_mask, label
35
36 batch_size = 16
37
38 train_dataset = PoemDataset(train_poem, tokenizer,
39                             max_length)
40 val_dataset = PoemDataset(val_poem, tokenizer,
41                           max_length)
42
43 train_dataloader = DataLoader(
```

```
41     train_dataset,
42     sampler=RandomSampler(train_dataset),
43     batch_size=batch_size
44 )
45
46 val_dataloader = DataLoader(
47     val_dataset,
48     sampler=SequentialSampler(val_dataset),
49     batch_size=batch_size
50 )
51
52 def count_labels(dataloader):
53     """
54         Scorre ogni batch nel dataloader, estraе le
55         etichette (terzo elemento della tupla)
56         e conta quante volte appare ogni etichetta.
57     """
58     label_counts = {}
59     total_samples = 0
60     for batch in dataloader:
61         labels = batch[2]
62         for label in labels:
63             if isinstance(label, torch.Tensor):
64                 label = label.item()
65
66                 label_counts[label] = label_counts.get(
67 label, 0) + 1
68
69             total_samples += len(labels)
70     return label_counts, total_samples
71
72 train_label_counts, train_total = count_labels(
73     train_dataloader)
74 print("\nDistribuzione nel Training set:")
75 for label, count in train_label_counts.items():
76     percentage = (count / train_total) * 100
```



```

74     print(f"Etichetta {label}: {count} campioni ({percentage:.2f}%)")
75
76 val_label_counts, val_total = count_labels(
    val_dataloader)
77 print("\nDistribuzione nel Validation set:")
78 for label, count in val_label_counts.items():
    percentage = (count / val_total) * 100
    print(f"Etichetta {label}: {count} campioni ({percentage:.2f}%)")

```

Listato 3.3: Definizione del dataset e gestione delle etichette in PyTorch

Si è proseguito, attraverso il codice del listato 3.3, preparando il dataset per l'addestramento e convertendo i generi letterari in valori numerici mediante `LabelEncoder`.

Dopo aver appreso le etichette presenti nel dataset, è stata definita una classe personalizzata `PoemDataset`, che gestisce la tokenizzazione delle poesie tramite un tokenizer di Hugging Face e associa ogni poesia al suo genere codificato. I dati vengono organizzati in due dataset distinti, uno per l'addestramento e uno per la validazione, e caricati in batch tramite i `DataLoader`, che permettono una gestione efficiente durante il training del modello.

Per garantire una distribuzione bilanciata delle etichette, il codice include una funzione che analizza i dataset e calcola il numero di poesie per ciascun genere, fornendo una panoramica della distribuzione delle classi. Questo processo assicura che il modello riceva dati ben strutturati e pronti per l'addestramento, migliorando l'efficacia della classificazione.

```

1 EPOCHS = 4
2 optimizer = AdamW(model.parameters(), lr=1e-5,
3                     correct_bias=False, weight_decay=0.01)
4 total_steps = len(train_dataloader) * EPOCHS
5
6 scheduler = get_linear_schedule_with_warmup(
7     optimizer,
8     num_warmup_steps=0,
9     num_training_steps=total_steps
)

```



```
10 loss_fn = nn.CrossEntropyLoss().to(device)
11
```

Listato 3.4: Configurazione dell'addestramento

Nel listato 3.4 viene mostrato l'addestramento del modello con l'algoritmo AdamW, utilizzando una schedule lineare con riscaldamento (`get_linear_schedule_with_warmup`). Il numero di epoche è impostato su 4, con una funzione di perdita `CrossEntropyLoss` che è adatta per compiti di classificazione multiclasse.

Sono state definite due funzioni principali:

- `train_epoch` (Listato 3.5), che gestisce l'addestramento di un'epoca, calcola la perdita, esegue il backpropagation e aggiorna i pesi del modello.

```
1   from tqdm import tqdm
2
3   def train_epoch(
4       model,
5       data_loader,
6       loss_fn,
7       optimizer,
8       device,
9       scheduler,
10      n_examples
11  ):
12      model = model.train()
13
14      losses = []
15      correct_predictions = 0
16
17      for batch in tqdm(data_loader, desc="Training",
18                         leave=False):
19          input_ids = batch[0].to(device)
20          attention_mask = batch[1].to(device)
21          targets = batch[2].to(device)
22          targets = batch[2].to(device).long()
23          outputs = model(
```



```
23             input_ids=input_ids,
24             attention_mask=attention_mask
25         )
26
27         _, preds = torch.max(outputs, dim=1)
28         loss = loss_fn(outputs, targets)
29
30         correct_predictions += torch.sum(
31     preds == targets)
32
33         losses.append(loss.item())
34
35         loss.backward()
36         nn.utils.clip_grad_norm_(model.
37     parameters(), max_norm=1.0)
38         optimizer.step()
39         scheduler.step()
40         optimizer.zero_grad()
41
42
43     return correct_predictions.double() /
44 n_examples, np.mean(losses)
```

Listato 3.5: train_epoch

- `eval_model` (Listato 3.6), che valuta il modello sui dati di validazione, calcolando le perdite e la precisione delle previsioni senza aggiornare i pesi.

```
1 def eval_model(model, data_loader, loss_fn,
2 device, n_examples):
3     model = model.eval()
4
5     losses = []
6     correct_predictions = 0
7
8     with torch.no_grad():
9         for batch in tqdm(data_loader, desc="Evaluating",
10                           leave=False):
```



```
9         input_ids = batch[0].to(device)
10        attention_mask = batch[1].to(
11            device)
12
13        targets = batch[2].to(device)
14
15        outputs = model(
16            input_ids=input_ids,
17            attention_mask=attention_mask
18        )
19        _, preds = torch.max(outputs, dim
20                            =1)
21
22        loss = loss_fn(outputs, targets)
23
24        correct_predictions += torch.sum(
25            preds == targets)
26        losses.append(loss.item())
27
28    return correct_predictions.double() /
29    n_examples, np.mean(losses)
```

Listato 3.6: eval_model

Durante l'addestramento (Listato 3.7), sono stati monitorati i progressi usando `tqdm`, una barra di progresso per visualizzare lo stato dell'addestramento e della valutazione. Le metriche principali sono la precisione e la perdita media per ciascun batch.

```
1 history = defaultdict(list)
2 best_accuracy = 0
3 epochs_no_improve = 0
4 best_loss = float("inf")
5 patience = 2
6
7
8 for epoch in range(EPOCHS):
```

```
9     print(f'Epoch {epoch + 1}/{EPOCHS}')
10    print('-' * 10)
11
12    train_acc, train_loss = train_epoch(
13        model,
14        train_dataloader,
15        loss_fn,
16        optimizer,
17        device,
18        scheduler,
19        len(train_poem)
20    )
21
22    print(f'Train loss: {train_loss:.4f} | Train
23          accuracy: {train_acc:.4f}')
24
25    val_acc, val_loss = eval_model(
26        model,
27        val_dataloader,
28        loss_fn,
29        device,
30        len(val_poem)
31    )
32
33    print(f'Val loss: {val_loss:.4f} | Val accuracy:
34          {val_acc:.4f}')
35    print()
36
37    history['train_acc'].append(train_acc)
38    history['train_loss'].append(train_loss)
39    history['val_acc'].append(val_acc)
40    history['val_loss'].append(val_loss)
41
42    if val_loss < best_loss:
43        best_loss = val_loss
44        epochs_no_improve = 0
```

```
43     torch.save(model.state_dict(), '  
best_model_state.bin')  
44 else:  
45     epochs_no_improve += 1  
46     if epochs_no_improve >= patience:  
47         print("Early stopping attivato!")  
48         break  
49  
50  
51 print("Training complete!")
```

Listato 3.7: Addestramento

Arrivati a questo punto, è stato possibile implementare il ciclo di addestramento per il modello di classificazione dei sentimenti, includendo meccanismi di monitoraggio delle prestazioni e *early stopping*. Nel dettaglio, sono stati monitorati vari parametri durante l'addestramento, tra cui la perdita e la precisione per i dati di addestramento e di validazione.

Ad ogni epoca, il modello veniva addestrato tramite la funzione `train_epoch`, che calcolava la perdita e l'accuratezza sui dati di addestramento, per poi essere valutato sui dati di validazione con la funzione `eval_model`, determinando così la perdita e l'accuratezza di validazione. I risultati di ciascuna epoca sono stati salvati in un dizionario `history`, che tiene traccia dell'accuratezza e della perdita per entrambi i set di dati.

Un meccanismo di *early stopping* è stato implementato per interrompere l'addestramento nel caso in cui non si verificasse un miglioramento della perdita di validazione per un certo numero di epoche consecutive, definito dalla variabile `patience`.

In particolare, se non si osservava un miglioramento per un numero di epoche superiore al valore di `patience`, l'addestramento veniva interrotto prematuramente.

L'andamento dell'addestramento è mostrato nella Figura 5.





```

Epoch 1/4
-----
Train loss: 2.7241 | Train accuracy: 0.3293
Val loss: 1.5318 | Val accuracy: 0.4676

Epoch 2/4
-----
Train loss: 1.3267 | Train accuracy: 0.5520
Val loss: 1.1235 | Val accuracy: 0.6098

Epoch 3/4
-----
Train loss: 0.8391 | Train accuracy: 0.7391
Val loss: 0.9702 | Val accuracy: 0.6828

Epoch 4/4
-----
Train loss: 0.5932 | Train accuracy: 0.8369
Val loss: 0.9515 | Val accuracy: 0.6910

Training complete!

```

Figura 5: Addestramento

3.2. Risultati

3.2.1 Train & Validation

La Figura 6 mostra due grafici affiancati che rappresentano l'andamento dell'errore nel training e nella validation.

Nel grafico a sinistra, "Model Loss", sono presenti due curve distinte, che rappresentano rispettivamente l'errore sul set di addestramento e quello sul set di validazione. L'errore diminuisce progressivamente con l'aumentare delle epoche, con la curva relativa all'addestramento che scende più rapidamente rispetto a quella della validazione. Questo suggerisce che il modello sta apprendendo dai dati di addestramento, sebbene la riduzione dell'errore di validazione sembri rallentare nelle ultime epoche.

Nel grafico a destra, "Model Accuracy", sono presenti due curve: una per l'accuratezza di addestramento e una per quella di validazione. Entrambe le metriche migliorano con il progredire dell'addestramento, con la curva dell'accuratezza di addestramento che cresce più rapidamente rispetto a quella della validazione. Tuttavia, nelle ultime epoche si nota una minore crescita nell'accuratezza di validazione, suggerendo che il miglioramento potrebbe iniziare a stabilizzarsi.



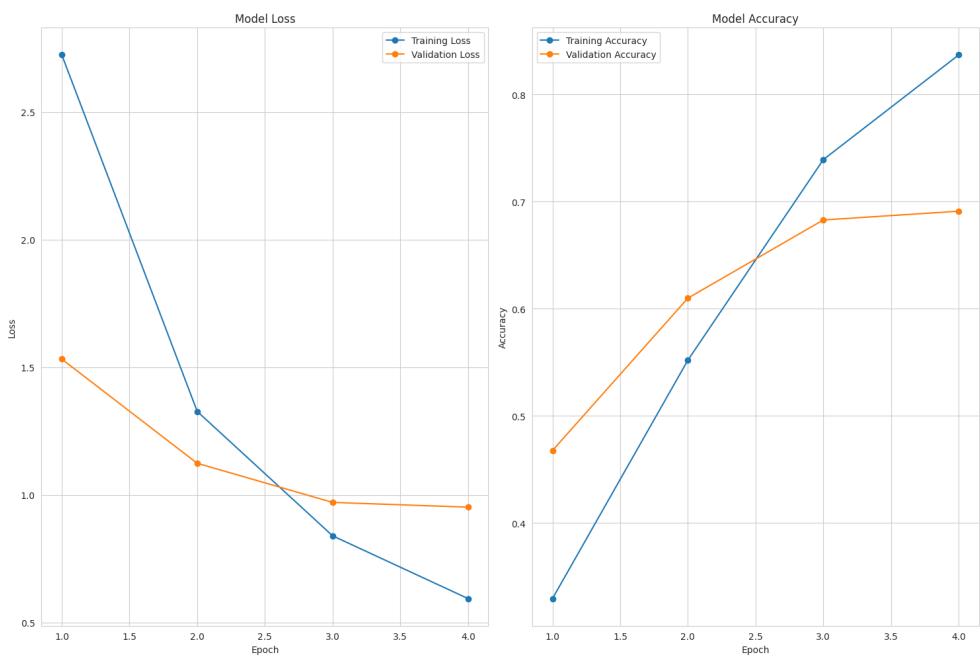


Figura 6: Risultati training e validation

3.2.2 Test

Successivamente, il modello è stato testato utilizzando la funzione di valutazione, che calcola l'accuratezza sulle poesie fornite. Il valore ottenuto è 0.69, ossia il 69% di accuratezza.



Figura 7: Confusion Matrix

Fatto ciò, per una maggiore comprensione, si mostrano i risultati nella Figura 7, che presenta una matrice di confusione in cui vengono illustrate le prestazioni del modello. È possibile notare che il modello apprende molto bene alcune emozioni, in particolare "Anger" (Rabbia) e "Love" (Amore). Tuttavia, la capacità di identificare altre emozioni, come "Sadness" (Tristezza) e "Surprise" (Sorpresa), risulta più limitata.

In conclusione, analizzando la Curva di ROC (Figura 8), è possibile notare che le curve colorate corrispondono a specifiche emozioni, ciascuna con il relativo valore di AUC, il quale varia tra 0.88 e 0.96, indicando un buon livello di separabilità delle classi.

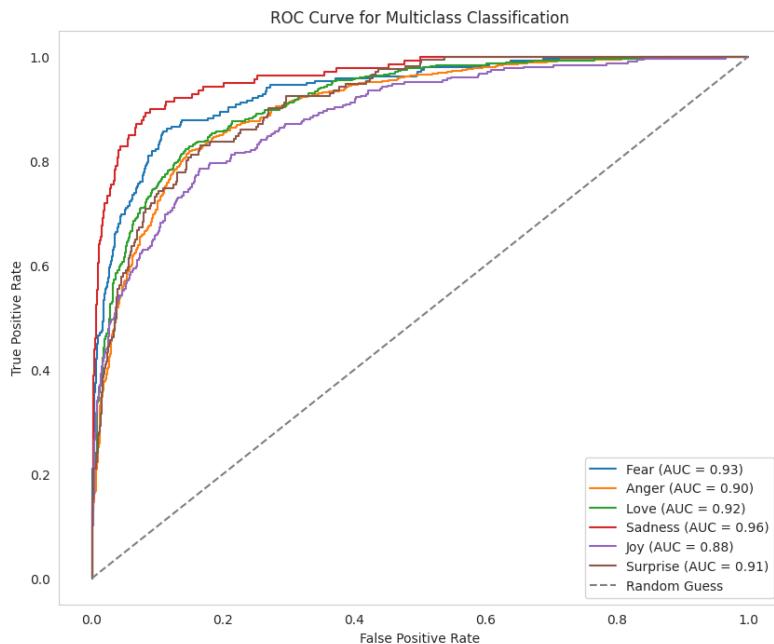


Figura 8: Curva di ROC

