RICONOSCIMENTO DEI SENTIMENTI ESPRESSI IN UN TWEET

Utilizzando BERT

HUGGING FACE

AUTORI:
CHIARA AMALIA CAPORUSSO
MARGHERITA GALEAZZI
SIMONE SCALELLA
ZHANG YIHANG

Sommario

1 INTRODUZIONE 2
1.1 Bert 2
1.2 Trasformers 2
1.3 Dataset 3
2 Sviluppo 4
2.1 Fase di ETL 4
2.2 Realizzazione del modello 7
2.3 Addestramento 8
2.4 Test 9
3 Conclusioni 15

1 INTRODUZIONE

1.1 Bert

Bert è un framework open source di machine learning per il Natural Language Processing (NLP).
È stato progettato per aiutare i computer a comprendere il significato di un linguaggio ambiguo in un testo, utilizzando il testo circostante per stabilire il contesto.
Il framework Bert stato pre-addestrato utilizzando il testo di Wikipedia e può essere perfezionato con altri dataset. Si basa sui Transformer, un modello di deep learning in cui ogni elemento di output è collegato a ogni elemento di input e i pesi tra di essi sono calcolati dinamicamente in base alla loro connessione.
Bert è stato pre-addestrato a due task NLP diversi, ma correlati:

- Masked Language Modeling: l'obiettivo di quest'addestramento è quello di nascondere una parola in una frase e far sì che il programma preveda quale parola è stata nascosta in base al contesto della parola nascosta.
- Next Sentence Prediction: l'obiettivo di quest'addestramento è far sì che il programma preveda se due frasi date hanno una connessione logica e sequenziale o se la loro relazione è semplicemente casuale.

Per ottenere delle buone prestazioni, i modelli NLP basati su deep learning necessitano di grandi quantità di dati, ad esempio i miglioramenti maggiori si hanno quando i modelli vengono addestrati su milioni o miliardi di dati.
Sono state sviluppate varie tecniche per addestrare modelli utilizzando le enormi quantità di testo presenti su web. Questo processo ci permette di ottenere dei modelli pre-addestrati.
Questi modelli pre-addestrati per scopi generici possono poi essere affinati (processo di fine-tuning) su insiemi di dati più piccoli e specifici per le attività che ci interessano.
Questo approccio consente di migliorare notevolmente l'accuratezza rispetto all'addestramento ex novo, cioè, da capo, su insiemi di dati più piccoli e specifici per le attività.
Alcune delle applicazioni più importanti di Bert sono le seguenti:

- Matching e recupero di documenti in base a un testo o a un gruppo di parole.
- Riassunto del testo: riassunto di grandi corpora/documenti in pezzi più piccoli da consumare.
- Evidenziazione dei paragrafi con entry point cruciali quando viene posta una domanda.
- Ricerca su Google.
- Sentiment Analysis.
- Question Answering.
- Traduzione linguistica.
- Gestione delle polisemie o delle coreferenze (ovvero, parole che suonano o sembrano uguali ma hanno significati differenti)
- Word sense disambiguation.
- Natural language inference.

Si prevede che Bert avrà un grande impatto sulla ricerca vocale e sulla ricerca testuale.

1.2 Trasformers

I trasformers sono un modello di deep learning che hanno una capacità nota come **bidirezionalità**.
Questa capacità consiste nell'avere ogni elemento di output collegato a ogni elemento di input e i pesi tra di essi sono calcolati dinamicamente in base alla loro connessione.
Per gestire attività di NLP i Trasformer sono considerati un miglioramento significativo, rispetto alle reti neurali ricorrenti (RNN) e alle reti neurali convoluzionali (CNN), perché i trasformer non richiedono l'elaborazione di sequenze di dati in un ordine fisso, come invece fanno RNN e CNN.
Poiché i trasformer possono elaborare i dati in qualsiasi ordine, essi consentono l'addestramento su quantità di dati maggiori di quanto fosse possibile prima della loro esistenza. Questo, a sua volta, ha facilitato la

creazione di modelli pre-addestrati come Bert, che è stato allenato su enormi quantità di dati linguistici prima del suo rilascio.

1.3 Dataset

Per realizzare questo progetto abbiamo utilizzato il dataset Emotion, disponibile al seguente link: https://huggingface.co/datasets/emotion.
Emotion è un dataset contenente messaggi Twitter in inglese, etichettati con sei emozioni di base, che sono rabbia, paura, gioia, amore, tristezza e sorpresa. Di seguito riportiamo un'immagine del dataset.

	text	emotions
27383	i feel awful about it too because it s my job ...	sadness
110083	im alone i feel awful	sadness
140764	ive probably mentioned this before but i reall...	joy
100071	i was feeling a little low few days back	sadness
2837	i beleive that i am much more sensitive to oth...	love
18231	i find myself frustrated with christians becau...	love
10714	i am one of those people who feels like going ...	joy
35177	i feel especially pleased about this as this h...	joy
122177	i was struggling with these awful feelings and...	joy
26723	i feel so enraged but helpless at the same time	anger
41979	i said feeling a bit rebellious	anger
2046	i also feel disillusioned that someone who cla...	sadness
98659	i mean is on this stupid trip of making the gr...	joy
50434	i woke up feeling particularly vile tried to i...	anger
9280	i could feel the vile moth burrowing its way i...	anger
92846	i know its just doing its job and doesnt actua...	joy
106363	i wish you knew every word i write i write for...	sadness
23395	i feel weird knowing mine died when i wasn t a...	fear
31583	i feel assured that there is no such thing as ...	joy
8271	i feel blessed everyday for our little man and...	love
32503	i feel exhausted when i go home but i am alway...	sadness
91430	i stil can see kaibas face on every tv screen ...	sadness

Figura 1.1 - Dataset

Il dataset è composto da 416809 righe e da 2 colonne. La prima colonna rappresenta il messaggio, in ogni riga di questa colonna troviamo un messaggio di Twitter, mentre, nell'altra colonna troviamo le emotions, cioè, l'emozione che è stata associata a quel messaggio.
Le emozioni saranno associate a dei numeri durante lo sviluppo di questo progetto.
Le associazioni sono:

- 0 = Tristezza
- 1 = Gioia
- 2 = Amore
- 3 = Rabbia
- 4 = Paura
- 5 = Sorpresa

2 Sviluppo

Il nostro progetto ha come obiettivo l'utilizzo di Bert per realizzare un modello in grado di eseguire un task di text classification specifico.
Il task di text classification di cui ci siamo occupati è quello del riconoscimento dei sentimenti espressi in un dato testo, quindi, nello specifico, il nostro è un task di sentiment analysis.

2.1 Fase di ETL

Abbiamo scaricato il dataset dal sito della comunity hugging face e abbiamo deciso di convertirlo in csv per facilitare il resto dello sviluppo e per rendere a noi più comprensibile il dataset in modo tale da essere consci dei dati con cui successivamente abbiamo lavorato. Inizialmente, il dataset era stato salvato come un json, infatti, aveva l'estensione jsonl; è stato utilizzato un breve script per convertire tale file in un csv.

Le emozioni presenti nel dataset sono 6, analizzando il dataset ci siamo resi conto che c'era un piccolo sbilanciamento tra le emozioni. Le classi 4 e 5 risultavano essere più piccole rispetto alle altre. Quindi, si è deciso di addestrare un primo modello con tutte le emozioni tranne la 4 e la 5, e successivamente, si procederà ad addestrare un modello con tutte le emozioni.

Si è proceduto con l'eliminazione delle emozioni meno numerose.

```
    dfNew = []
    for i in range(0,6):
        dfNew.append(df[df.label == i])
        print("label:"+str(i))
        print(dfNew[i].count())

    df = df[(df.label == 0) | (df.label == 1) | (df.label == 2) | (df.label == 3)]
```

Figura 2.1 - Eliminazione emozioni meno presenti

Dall'immagine successiva è possibile visualizzare come le precedenti due emozioni siano state effettivamente eliminate dall'attuale dataset.

```
print(df['label'].unique())
```

```
[0 1 2 3]
```

Figura 2.2 - Emozioni rimanenti

Per migliorare l'addestramento del modello si è deciso di effettuare un'operazione di shuffle sul dataset, in maniera tale da migliorare la distribuzione degli elementi, evitando di avere grandi sequenze di messaggi con la stessa emozione.

```
## Shuffle Data
def shuffle(df, n=3, axis=0):
    df = df.copy()
    random_states = [2,42,4]
    for i in range(n):
        df = df.sample(frac=1,random_state=random_states[i]) # mischio il dataframe
    return df

new_df = shuffle(df)
new_df
```

Figura 2.3 - Mescolamento del dataset

Lo step successivo è stato quello di rimuovere le stopwords, ovvero delle parole della frase che non contribuiscono il significato. Quindi per la comprensione, da parte del modello, del testo non sono necessarie, quali ad esempio gli articoli e gli aggettivi possessivi, i caratteri speciali o le emoji.
Questo step va fatto perché essendo scritto il Tweet in linguaggio naturale esistono dei caratteri che ai fini dell'analisi con BERT non sono di alcuna utilità e quindi si decide di rimuoverli.
Si è fatto semplicemente cercando all'interno del testo quelli che venivano definiti come simboli di punteggiatura o come link o come emoji e sostituendoli con uno spazio bianco.

```
from nltk.corpus import stopwords
nltk.download('stopwords')
sw = stopwords.words('english')

def clean_text(text):

    text = text.lower()

    text = re.sub(r"[^a-zA-Z?.!,¿]+", " ", text) #Comando che permette di cambiare tutti i caratteri tranne :a-z, A-Z, ".", "?", "!", "," con uno spazio

    text = re.sub(r"http\S+", "",text) #Rimozione dei link

    html=re.compile(r'<.*?>')

    text = html.sub(r'',text) #Rimozione di tag HTML

    punctuations = '@#!?+&*[]-%.:/();$=><|{}^' + "'`" + '_'
    for p in punctuations:
        text = text.replace(p,'') #Comando che permette di rimuovere i segni definiti sopra come punteggiatura

    text = [word.lower() for word in text.split() if word.lower() not in sw]

    text = " ".join(text) #Rimozione di tutte le stopword

    emoji_pattern = re.compile("["
                           u"\U0001F600-\U0001F64F"  # emoji
                           u"\U0001F300-\U0001F5FF"
                           u"\U0001F680-\U0001F6FF"
                           u"\U0001F1E0-\U0001F1FF"
                           u"\U00002702-\U000027B0"
                           u"\U000024C2-\U0001F251"
                           "]+", flags=re.UNICODE)
    text = emoji_pattern.sub(r'', text) #Comando che rimuove quelle definite sopra come emojis

    return text
```

Figura 2.4 - Eliminazione dei caratteri inutili nella comprensione

Procederemo ora nel pre-processamento del dataset al fine di renderlo compatibile con il tipo di dato che può essere dato come input a BERT.
BERT, infatti non prende il linguaggio naturale in sé come input, ma prende invece dei tensori (array multidimensionali) formati da token (che derivano dal linguaggio naturale).
Per fare ciò abbiamo in primo creato due liste, una contenente le label (relative al sentimento) e l'altra contenente il testo del Tweet. La lista delle label ci servirà successivamente, in fase di allenamento per far capire al modello ogni Tweet a che sentimento corrisponde.

```
sentences = train_df.text
labels = list(train_df.label)
```

Figura 2.5 - Creazione della lista di Tweet e della lista di label

Prendiamo poi dal BERT tokenizer (che abbiamo importato all'inizio), e con la funzione *from_pretrained()* il modello `'bert-base-uncased'` e `do_lower_case=True` in quanto come si può vedere nella documentazione questo modello lavora con il testo in lower case.

Il tokenizer sfruttando il modello già allenato è in grado di dividere in token l'input che gli passiamo come frase. Di seguito riportiamo un esempio, che mostra la tokenizzazione del primo Tweet che compare nel nostro dataset.

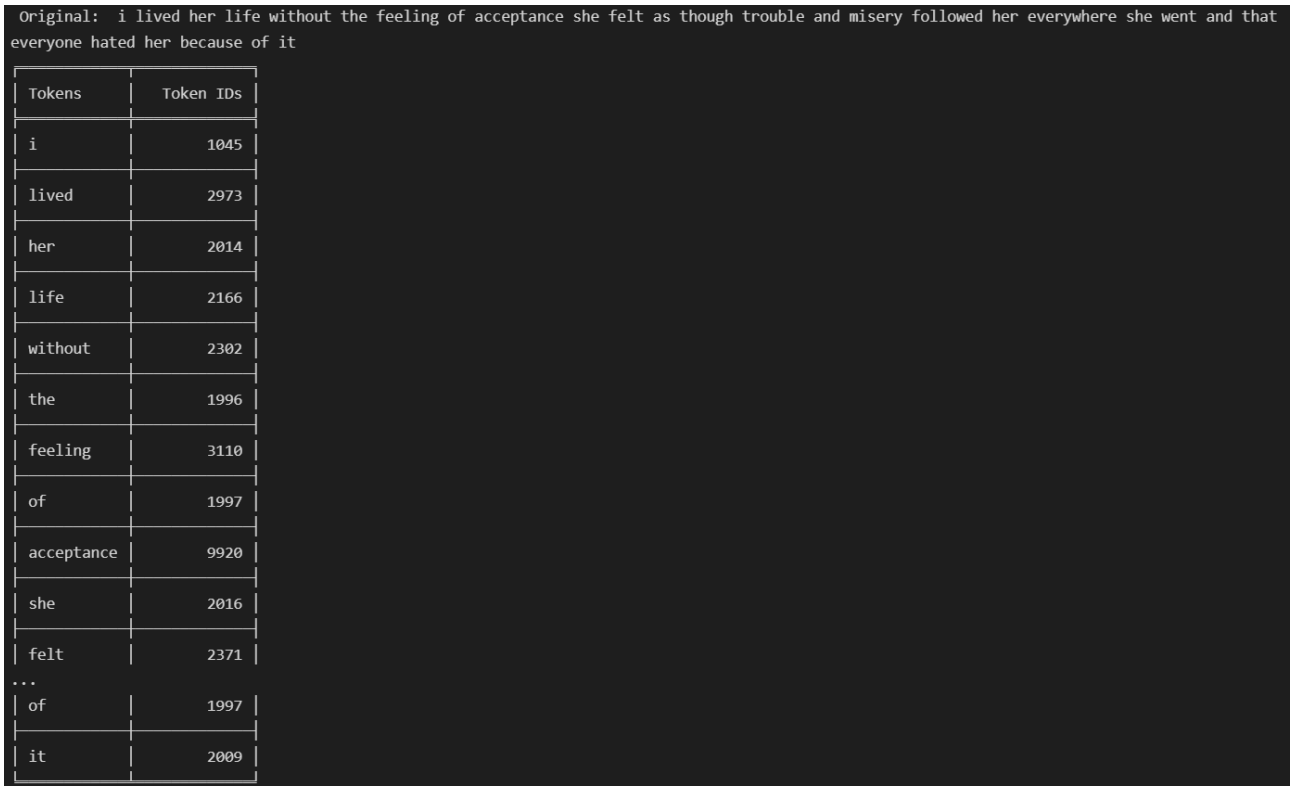

Figura 2.6 - Esempio tokenizzazione 1°Tweet

La prima riga mostra il Tweet, al quale sono state rimosse le stopword nella sua interezza, mentre nella tabella sotto viene mostrato per ogni token (parola) il token ID che gli viene associato.

Inoltre, per rendere i dati compatibili con il tipo di dati accettati in ingresso da BERT ad ogni input vengono assegnati dei token "speciali", che sono i token `[SEP]` e `[CLS]`:

- `[SEP]` è un token che se vengono date più frasi in input le separa (ID: 102);
- `[CLS]` è un token che va sempre messo all'inizio della frase e specifica che quello che si sta affrontando è un task di classificazione (ID:101).

Entrambi i token sono sempre richiesti, anche se la frase in input è una sola (in questo caso `[SEP]` verrà messo alla fine della frase).
Avendo BERT come massima lunghezza dei token 512, ovvero ogni frase di input dovrà avere al massimo 512 token (parole), dobbiamo quindi andare a vedere se le frasi del nostro dataset rispettano tale limite o meno e in caso non lo rispettassero troncarle.
Sfruttando la funzione *encode_plus()* facciamo l'encoding di tutte le frasi affinché siano quindi pronte per essere date in input al modello.
Quello che questa funzione fa è:

- Tokenizzare la frase;
- Aggiungere i token speciali (ovvero aggiunge i token `[SEP]` e `[CLS]`);
- Mappare i token con il loro ID;
- Esegue il padding (ovvero allunga aggiungendo 0) o tronca la frase affinché la sua lunghezza sia pari a max_length;
- Creare le attention mask (distinzione tra parole effettive della frase e padding).

```
MAX_LEN = 128
#Tokenizza tutte le frasi e mappa i tokens con i loro IDs
input_ids = []
attention_masks = []


for sent in sentences:
    # Quello che `encode_plus` farà:
    #   1. Tokenizza la frase
    #   2. Aggiunge il token `[CLS]` all'inizio della frase
    #   3. Aggiunge il token `[SEP]` alla fine della frase
    #   4. Mappa il token con il loro ID
    #   5. Esegue il padding o tronca la frase affinche la sua lunghezza sia pari a `max_length`
    #   6. Crea le attention masks per il token [PAD]
    encoded_dict = tokenizer.encode_plus(
                        sent,
                        add_special_tokens = True,      #Aggiunge i tokens '[CLS]' e '[SEP]'
                        max_length = MAX_LEN,           #Setta la lunghezza massima
                        pad_to_max_length = True,       #Se necessaio esegue il padding
                        return_attention_mask = True,   #Costruisce le attn. masks
                        return_tensors = 'pt',          #Restituisce un tensore di pytorch
                   )

    #Aggiunge la frase codificata alla lista degli input
    input_ids.append(encoded_dict['input_ids'])

    #E aggiunge le attention mask alla lista (semplice distinzione tra padding o meno)
    attention_masks.append(encoded_dict['attention_mask'])

#Converte la lista in un tensore
input_ids = torch.cat(input_ids, dim=0)
attention_masks = torch.cat(attention_masks, dim=0)
labels = torch.tensor(labels)
```

Figura 2.7 - Preparazione dei dati in input

Ed infine si procederà a trasformare la lista relativa agli input, la lista relativa alle attention mask e la lista relativa alle label in tensori; a questo punto i nostri dati sono pronti per essere dati in input al modello BERT.

2.2 Realizzazione del modello

Al fine di eseguire il fine tuning si è diviso il dataset in training set e validation set.

```
144,652 training samples
36,164 validation samples
```

Figura 2.8 - Dimensioni training e validation set

Come prima cosa sfruttando la funzione TensorDataset metto insieme i tre tensori, relativi agli input, alle attention mask e alle label, ottenuti con le elaborazioni viste prima.

Poi dopo aver definito la dimensione che dovranno avere il training set (80% del dataset) e il validation set (20%) tramite la funzione *random_split()* vengono creati di due dataset.

Successivamente abbiamo settato i parametri del modello. Per quanto riguarda questi ci sono dei valori “classici” da utilizzare, ma in realtà il modo migliore per selezionarli sarebbe quello di sperimentarli e visionare quale dia il miglior risultato.

Nel nostro progetto, ad esempio, in un primo momento avevamo allenato il modello su 4 epoche, ma vedendo che andava in questo modo in overfitting abbiamo deciso di ridurre il numero delle epoche a 2.

Per caricare i dati sul modello di BERT, dobbiamo utilizzare la classe DataLoader; quindi, creiamo due oggetti DataLoader a cui passeremo rispettivamente il training set e il validation set.

Ed infine procediamo nella creazione del modello, per fare ciò si andrà a riprendere il modello pre-allenato.

```
# Viene caricato il modello pre-allenato BertForSequenceClassification
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased", #Usa il modello di BERT a 12-layer con un vocabolario tutto in lower case
    num_labels = 6, #Il numero delle label
    output_attentions = False,
    output_hidden_states = False,
)

model = model.to(device)
```

Figura 2.9 - Creazione del modello

Ora dobbiamo procedere nell'allenamento e nella valutazione dei risultati ottenuti.

2.3 Addestramento

Per poter proseguire con l'operazione di fine tuning del modello, sono state realizzate altre operazioni.

```
epochs = 2

#Numero totale degli step di training, è dato da: [numero di batch] x [numero di epoche]
total_steps = len(train_dataloader) * epochs

#Viene creato lo scheduler del learning rate
scheduler = get_linear_schedule_with_warmup(optimizer,
                                            num_warmup_steps = 0,
                                            num_training_steps = total_steps)
```

Figura 2.10 - Scelta del numero di epoche e creazione dello scheduler

Siamo andati a definire il numero di epoche, che, come si può vedere nella precedente immagine, sono state poste a 2. Prima di decidere di prenderne 2 abbiamo anche provato ad allenare il modello con 4 epoche, ma si verificava una condizione di overfitting e perciò abbiamo deciso di limitare il numero di quest'ultime a due. Inoltre, in questa cella andiamo a creare lo scheduler del learning rate.

Si è definita una funzione per il calcolo dell'accuratezza.

```
#Funzione per calcolare l'accuratezza della nostra predizione (rispetto alla label)
def flat_accuracy(preds, labels):
    pred_flat = np.argmax(preds, axis=1).flatten()
    labels_flat = labels.flatten()
    return np.sum(pred_flat == labels_flat) / len(labels_flat)
```

Figura 2.11 - Funzione per il calcolo dell'accuratezza

Si è definita anche una funzione per la formattazione del tempo, che arrotonda i secondi e conferisci alla data uno specifico pattern.

```
#Funzione che formatta il tempo arrotondando i secondi e dandogli il pattern hh:mm:ss
def format_time(elapsed):
    '''
    Takes a time in seconds and returns a string hh:mm:ss
    '''
    elapsed_rounded = int(round((elapsed)))
    return str(datetime.timedelta(seconds=elapsed_rounded))
```

Figura 2.12 - Funzione per formattare il tempo

Terminata questa fase di preparazione si è proceduto con il fine tuning del modello utilizzando le varie funzioni messe a disposizione da torch e dalla libreria transformers. Di seguito riportiamo una parte del codice per il fine tuning del modello.

```
#Mette il modello in training mode (presta attenzione al fatto che NON esegue l'allenamento del modello)
model.train()
for step, batch in enumerate(train_dataloader):
    b_input_ids = batch[0].to(device)
    b_input_mask = batch[1].to(device)
    b_labels = batch[2].to(device)
    #Pulisce i gradienti precedentemente clacolati prima di eseguire un backward
    #pass (PyTorch non lo fa di default perchè nell'allenamento di RNN è conveniente accumularli)
    model.zero_grad()
    #La documentazione relativa al modello la si può trovare al link sotto:
    # https://huggingface.co/transformers/v2.2.0/model_doc/bert.html#transformers.BertForSequenceClassification
    #Ritorna un numero differente di parametri in base in base agli argomenti dati e alle flag settate.
    # Nel nostro caso ritorna la loss (perchè abbiamo dato un dataset etichettato in input)
    output = model(b_input_ids,
                   token_type_ids=None,
                   attention_mask=b_input_mask,
                   labels=b_labels)
    #Accumula la loss di allenamento su tutti i batch in modo da poter calcolare la perdita
    #media alla fine. `loss` è un tensore contenente un singolo valore;
    loss = output.loss
    logits = output.logits
    total_train_loss += loss.item()
    #Esegue un backward pass per calcolare i gradienti
    loss.backward()
    #Ritaglia la norma dei gradienti a 1.0.
    #Questo per aiutare a prevenire il problema dei "gradienti che esplodono".
```

Figura 2.13 - Parte del fine tuning del modello

Al termine dell'addestramento si è proceduto con la fase di test.

2.4 Test

In questa sezione andremo a riportare i risultati ottenuti dall'addestramento dei 4 modelli, divisi per epoche e per emozioni. Iniziamo con il modello che classifica quattro emozioni, addestrato per 4 epoche. Riportiamo le statistiche legate all'addestramento.

epoch	Training Loss	Valid. Loss	Valid. Accur.	Training Time	Validation Time
1	0.171649	0.078863	0.959892	0:13:21	0:00:57
2	0.074534	0.075348	0.959212	0:12:41	0:00:56
3	0.064564	0.087341	0.958227	0:12:40	0:00:56
4	0.055042	0.110262	0.956978	0:12:46	0:00:56

Figura 2.14 - statistiche sull'addestramento del modello

Possiamo osservare come la loss diminuisca molto dopo la prima epoca, poi sempre meno per le epoche successive, però, osserviamo che la loss di validazione aumenta con il passare delle epoche, e questo significa che si sta verificando un problema di overfitting, cioè, il modello è addestrato troppo bene sul modello di training, e quindi l'errore di training si abbassa, però riconosce bene solo quegli elementi, quindi sbaglia quelli di validazione, infatti, la loss di validazione aumenta. Anche l'accuratezza diminuisce leggermente con l'aumentare delle epoche.
I tempi di addestramento del modello sono molto lunghi, in totale l'esecuzione di tutto il progetto richiede un ora di tempo.

Figura 2.15 – errore di training e validation su quattro epoche

Andiamo a stampare l'andamento dell'errore di training e di validation. L'immagine conferma l'ipotesi precedente, all'aumentare delle epoche il modello si specializza troppo. Possiamo osservare come le due curve, dalla seconda epoca in poi iniziano a separarsi. Concludiamo che il modello deve essere nuovamente addestramento ma solo per due epoche. Riportiamo un ulteriore risultato che è la matrice di confusione del modello.

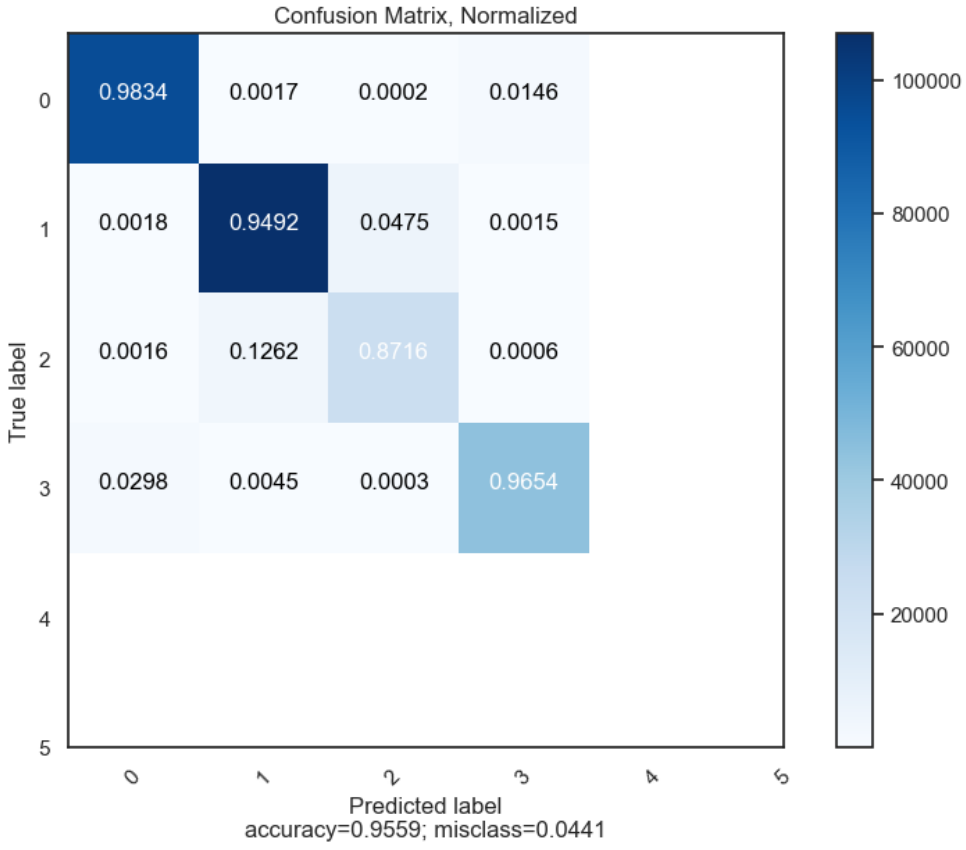

Figura 2.16 - Matrice di confusione del modello a quattro epoche

L'emozione che presenta il risultato peggiore è quella relativa all'amore, con il prossimo modello cerchiamo di verificare se riducendo le epoche otteniamo un risultato migliore.
Riportiamo le statistiche per il modello addestrato su quattro emozioni e per due epoche.

epoch	Training Loss	Valid. Loss	Valid. Accur.	Training Time	Validation Time
1	0.170610	0.071030	0.959420	0:12:35	0:00:58
2	0.068539	0.071889	0.960322	0:12:39	0:00:53

Figura 2.15 - Statistiche ottenute con due epoche

Possiamo osservare come 2 epoche sia un numero ottimo di epoche addestrare il modello, infatti l'accuratezza aumenta, entrambe le loss si abbassano e si portano su un valore molto simile.

Figura 2.18 - Errore di training e validation con due epoche

Questa immagine mostra l'andamento degli errori, con due epoche riduciamo la differenza tra l'errore di training e quello di test. Infatti, all'aumentare delle epoche aumenta questa differenza che noi vogliamo minimizzare, come si osservava, invece, nel modello precedente.
A questo punto riportiamo la matrice di confusione del modello.

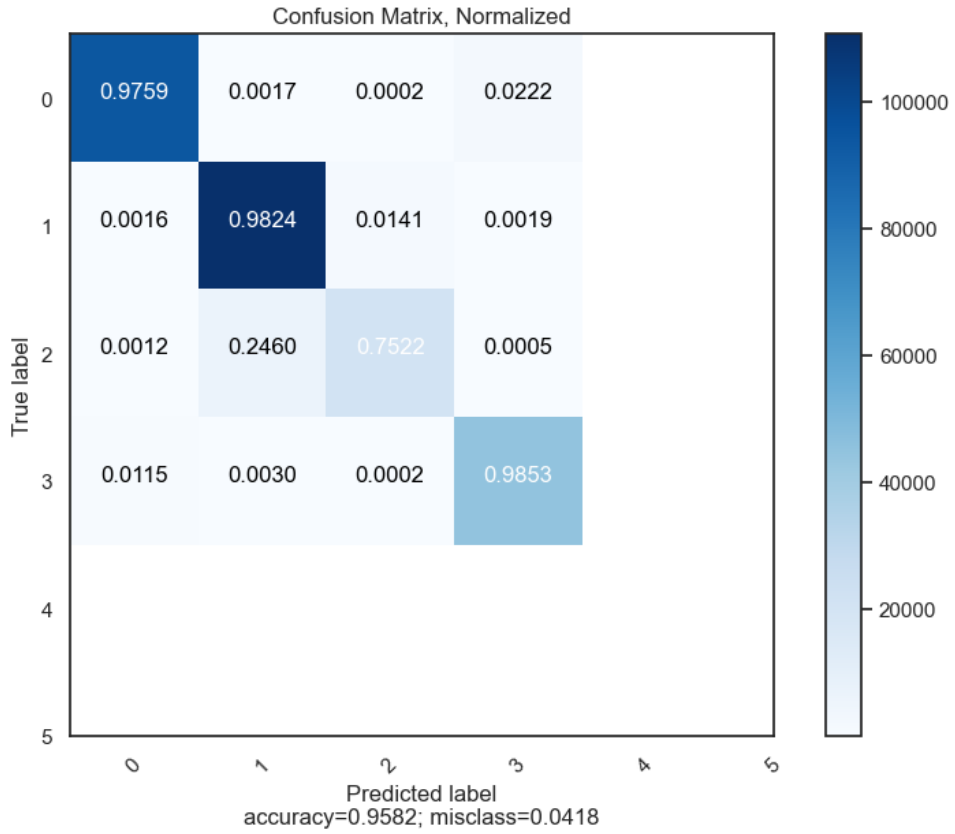

Figura 2.19 - Matrice di confusione del modello a due epoche

Dalla matrice di confusione osserviamo che due emozioni su quattro, sono state classificate meglio rispetto al modello precedente. Infatti, le emozioni 1 e 3 sono classificate meglio rispetto a prima, l'emozione 0 è stata classificata leggermente peggio rispetto a prima, purtroppo, peggiora la classificazione dell'emozione 2. Una soluzione potrebbe essere aumentare il numero di messaggi per questa emozione, oppure, fare un addestramento con tre epoche.
Riportiamo le statistiche per il modello addestrato su tutte le emozioni e per quattro epoche.

epoch	Training Loss	Valid. Loss	Valid. Accur.	Training Time	Validation Time
1	0.162915	0.101233	0.940430	0:32:22	0:02:23
2	0.093631	0.095404	0.942780	0:32:05	0:02:17
3	0.084985	0.102346	0.941176	0:31:34	0:02:19
4	0.080765	0.105687	0.938550	0:31:19	0:02:26

Figura 2.20 - Statistiche per il modello addestrato su tutte le emozioni e quattro epoche

Anche in questo caso osserviamo che all'aumentare delle epoche aumenta la differenza tra l'errore di training e quello di validation. A questo punto riportiamo il grafico con l'andamento degli errori.

Figura 2.21 - grafico andamento degli errori su quattro epoche e sei emozioni

Questa immagine conferma l'ipotesi precedente, quindi, come nel caso precedente, dalle due epoche in poi la differenza tra gli errori aumenta. Di seguito riportiamo la matrice di confusione del modello.

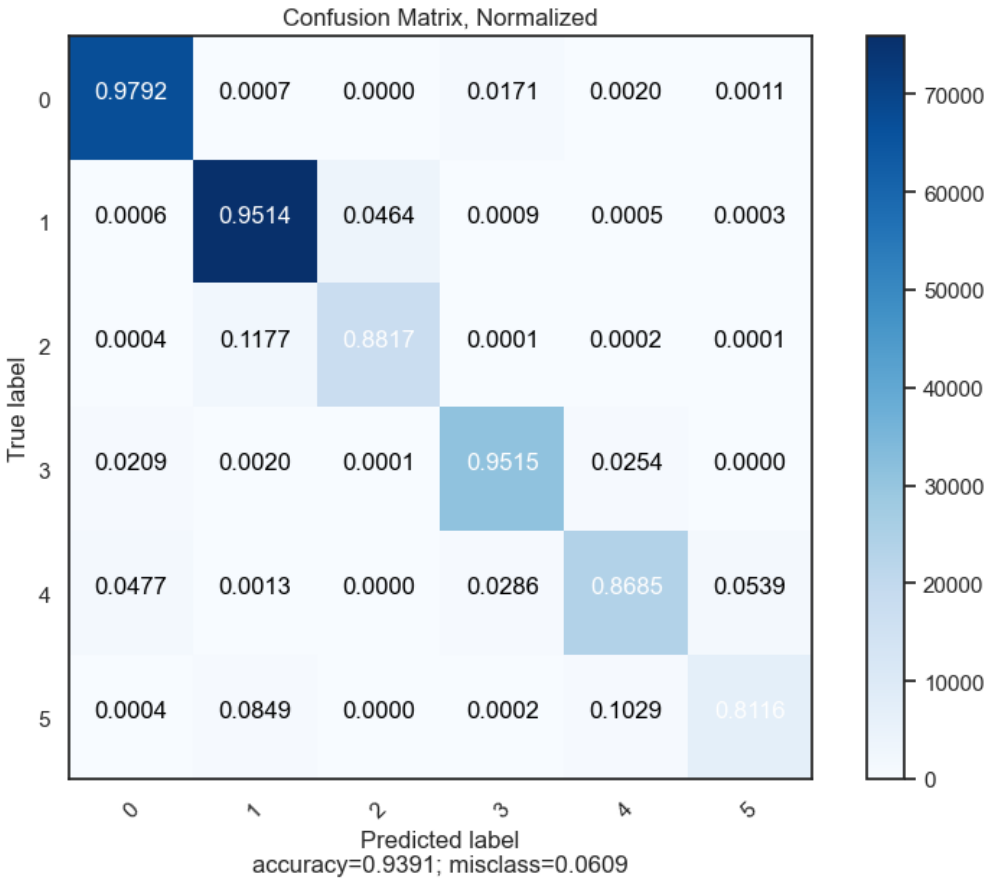

Figura 2.22 - Matrice di confusione per quattro epoche e sei emozioni

Possiamo osservare come le classi con meno dati siano quelle che vengono classificate peggio, infatti, la 5 e la 4 erano quelle con meno elementi e hanno anche il risultato peggiore. L'emozione zero, anche in questo caso è quella che viene classificata meglio.
Riportiamo le statistiche per il modello addestrato su tutte le emozioni e per due epoche.

epoch	Training Loss	Valid. Loss	Valid. Accur.	Training Time	Validation Time
1	0.163522	0.091064	0.942559	0:32:13	0:02:24
2	0.089121	0.093535	0.940817	0:32:06	0:02:18

Figura 2.23 - Statistiche del modello addestrato su tutte le emozioni e due epoche

È possibile osservare come i tempi di compilazione aumentano notevolmente all'aumentare delle dimensioni del dataset, infatti, aggiungendo due emozioni, siamo passati ad un addestramento che richiede oltre trenta minuti per epoca. Osserviamo comunque come i due valori dell'errore si portano ad un valore simile.

Figura 2.24 - Andamento errori con due epoche e tutte le emozioni

Addestrare il modello su due epoche minimizza la differenza tra errore di traning ed errore di validation. Riportiamo di seguito la matrice di confusione del modello.

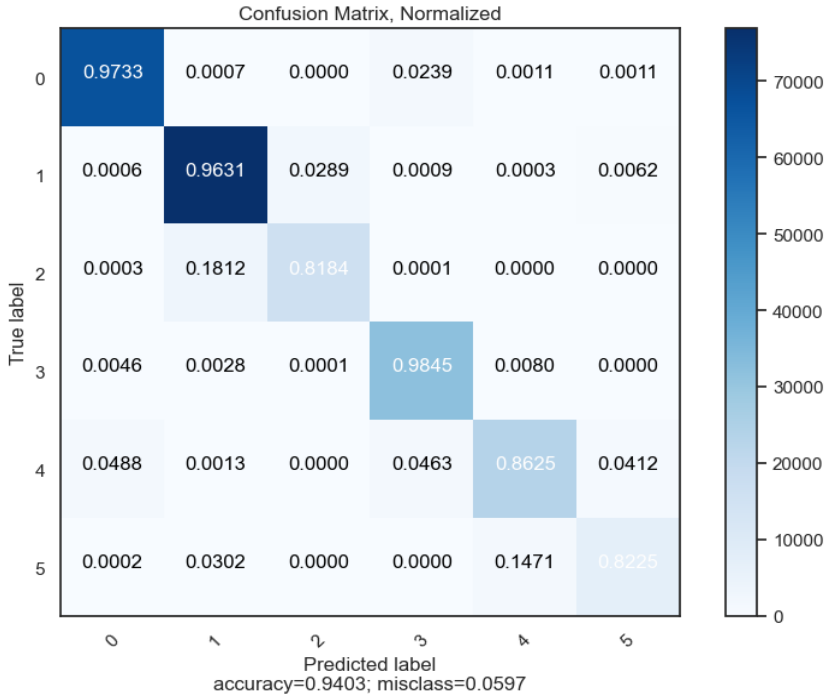

Figura 16 - Matrice di confusione per due epoche e sei emozioni

Dalla matrice è possibile osservare come le emozioni 1, 3 e 5 vengono classificate leggermente meglio rispetto al modello precedente, le emozioni 0 e 4 sono classificate quasi con lo stesso risultato, la classificazione peggiore rimane per l'emozione 2.

3 Conclusioni

Al termine di questo progetto sono stati realizzati diversi modelli che soddisfano un task di sentiment analysis. Durante lo sviluppo si è potuto osservare come la presenza di classi, leggermente sbilanciate, non causa problemi per l'addestramento del modello. Infatti, i valori ottenuti in entrambi i modelli, sia con 4 che con 6 emozioni, sono abbastanza simili.
Infine, abbiamo osservato come, dalla seconda epoca in poi aumenta la differenza tra errore di training ed errore di validation aumenta. Inoltre, riducendo il numero di epoche non si riducono le prestazioni di classificazione di tutte le classi. Di conseguenza, in base alle esigenze è possibile scegliere un numero di epoche che sia un buon compromesso tra differenza degli errori e risultati nella classificazione.
Le differenze riscontrate tra due e quattro epoche rimangono molto piccole, infatti, abbiamo, nel caso di quattro emozioni, una differenza di 0.06, e 0.02 nel caso di 6 emozioni.
Quindi, un numero di epoche maggiore di 2 ha permesso di classificare meglio l'emozione 2, però, tale valore non è stato scelto troppo distante da quello ideale.

Link al repository GitHub:
https://github.com/ChiaraAmalia/ProgettoBert