Università degli Studi di Perugia
Dipartimento di Matematica e Informatica

Corso di Laurea Specialistica in Informatica

Basi di Dati Avanzate e Data Mining

Studenti:
Alex Dominici

Professore:
Prof. Valentina Poggioni

Anno Accademico 2019–2020

Introduzione

Il problema preso in considerazione per il progetto riguarda lo svolgimento di una task appartenente alla challenge di Evalita 2018. La task in questione è ITAmoji dove vengono forniti 250.000 tweets in italiano per allenare il sistema di previsione emoji: ogni tweet comprenderà una e una sola emoji, eventualmente ripetuta. Quindi si tratta di un problema di apprendimento supervisionato poichè le etichette sono disponibili, nonché di tipo NLP (Natural Language Processing).

I capitoli successivi mostrano come risolvere questo problema, dove prima di tutto è stato recuperato il dataset di training contenenti i tweets etichettati con una delle 25 emoji presenti, inoltre è stata fatta una profonda analisi e pulizia di quest'ultimi. Successivamente è stata identificata una tecnica che permetta di catturare il significato delle parole per poi progettare diversi modelli di ANN e allenarli con tali tweets. Infine una volta scelta l'architettura per la classificazione vengono valutati i modelli prodotti.

Emoji	Label
❤️	red heart
😂	face with tears of joy
💋	kiss mark
😋	face savoring food
🌹	rose
☀️	sun
😍	smiling face with heart eyes
😘	face blowing a kiss
💙	blue heart
😊	smiling face with smiling eyes
😀	grinning face
😉	winking face
😁	beaming face with smiling eyes
✨	sparkles
🤣	rolling on the floor laughing
👍	thumbs up
😎	smiling face with sunglasses
💪	flexed biceps
🤔	thinking face
💕	two hearts
😭	loudly crying face
🔝	top arrow
😅	grinning face with sweat
😜	winking face with tongue
😱	face screaming in fear

Figura 1: Lista delle classi

Capitolo 1

Linguaggio e Librerie utilizzate

Il linguaggio di programmazione utilizzato per lo svolgimento del progetto è stato Python, il quale è un linguaggio di alto livello e orientato a oggetti, tipicamente utilizzato a sviluppare applicazioni distribuite, scripting e sistemi di testing. In questo progetto è stato proprio utilizzato per lo sviluppo dei modelli di apprendimento attraverso le seguenti librerie.

1.1 Pandas

Libreria che consente di utilizzare diversi strumenti per l'analisi e la gestione dei dati scritti in python. Essa è open source e viene fornita con molteplici strutture dati, come Series e Dataframes, che ci aiutano non solo a rappresentare i dati in modo efficiente, ma anche a manipolarli in vari modi. Inoltre permette di recuperare facilmente i dati da diverse fonti come database SQL, CSV O file JSON ed è in grado di organizzare ed etichettare i dati tramite dei metodi intelligenti di allineamento e indicizzazione che ne consentono una semplice lettura.

Quando i dati sono stati recuperati e memorizzati, vengono fornite diverse funzioni per pulirli, recuperare i valori mancanti, analizzarli, trasformarli e un set di operazioni statistiche per eseguire semplici analisi.

Pandas è dunque una libreria molto conosciuta ed usata per il recupero e per il pre-processing dei dati in modo da usarli successivamente in altre librerie come Scikit-learn o Tensorflow.

1.2 NumPy

Numpy ovvero Numeric Python, risulta essere il pacchetto fondamentale per il calcolo scientifico con Python ed è una delle più grandi librerie di calcolo matematico e scientifico. Una delle funzionalità più importanti di NumPy è la sua interfaccia Array. Questa interfaccia può essere utilizzata per fornire strutture dati per il calcolo e l'implementazione di vettori e matrici multi-dimensionali. Inoltre questa libreria fornisce diverse funzioni matematiche.

1.3 Scikit-learn

Anche questa è una libreria open source scritta in Python ed è in grado di interagire con altre libreririe come Pandas, NumPy e Keras. Essa principalmente permette di utilizzare diversi algoritmi (ad esempio di classificazione, di regressione o di clustering) che consentono di costruire modelli per l'apprendimento sia supervisionanto che non. Inoltre concede la possibilità di costruire la matrice di confusione attraverso funzioni e metodi per la suddivisione dei dati e per il calcolo delle performance dei modelli.

1.4 Keras

Libreria scritta in Python che fornisce un modo semplice per esprimere l'apprendimento automatico e la costruzione di reti neurali.

Oltre alla facilità di apprendimento e di costruzione dei modelli, essa ha il vantaggio di essere adottata da un gran numero di utenti, di essere integrata con cinque diversi motori di back-end (come TensorFlow, CNTK, Theano, MXNet e PlaidML), di avere un supporto per GPU multiple e la capacità di training distribuito.

Keras dunque non esegue le proprie operazioni di basso livello, come ad esempio i prodotti tra tensori ma si affida ad una libreria specializzata per farlo, quindi viene permesso il collegamento a diversi back-end, come citato precedentemente, in modo da gestire il problema modularmente e impedendo che l'implementazione si leghi ad una singola libreria. Infine essa fornisce alcune utilità per l'elaborazione di set di dati, per la compilazione di modelli, per la valutazione dei risultati e per la visualizzazione di grafici.

Capitolo 2

Procedimento risolutivo

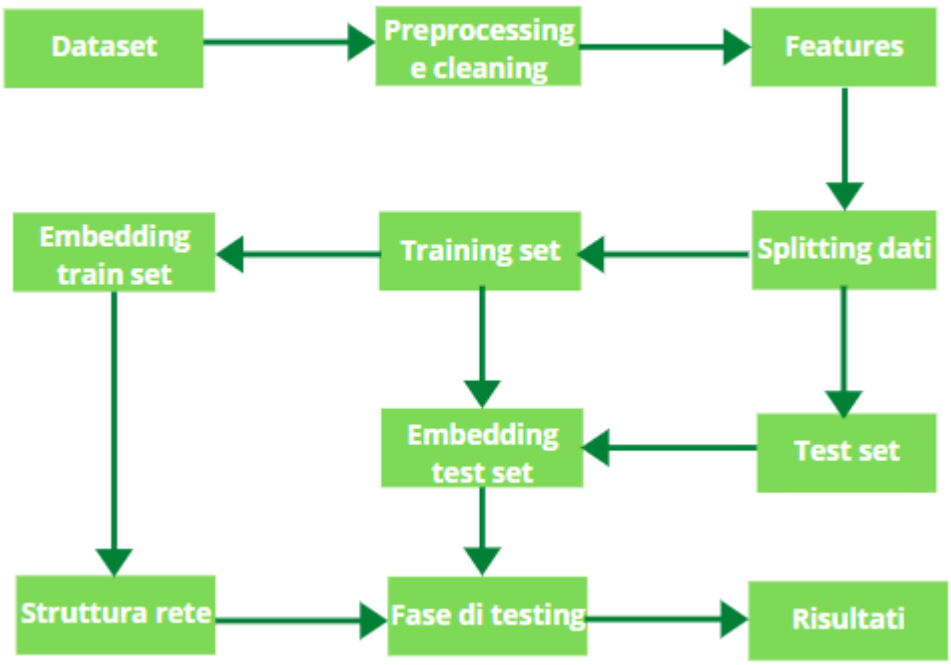


Figura 2.1: Fasi principali della soluzione

In questo capitolo verranno illustrate le fasi principali per raggiungere l'obiettivo proposto, partendo dall'ottenimento del Dataset dal sito di Evalita, passando per il pre-processing e il cleaning accurato dei dati. Successivamente vengono calcolate le features sul dataset e poi vengono splittati i dati in training e test set. Infine per entrambi i set di dati viene trasformato il testo in formato numerico in modo tale che possano essere usati per allenare i modelli creati per classificare le frasi. Tramite il test set invece è stata svolta la fase di testing valutandone anche i relativi risultati.

2.1 Visualizzazione del dataset

Il dataset recuperato dal sito di Evalita è stato distribuito in formato Json e per poterlo visualizzare e lavorarci è stata utilizzata la citata libreria Pandas. Come mostrato in Figura 2.2 il dataset contiene 250,000 tweets in lingua italiana e questi vengono etichettati in base all'emoji, tuttavia per poterle utilizzare queste classi sono state tradotte in numeri interi come evidenziato dalla colonna "label_int" .

Inoltre ai fini del progetto sono state mantenute solo le colonne 'uid', 'text_no_emoji' e 'label', dato che l'Id dei tweets non risulta essere utile per la classificazione.

	text_no_emoji	label	uid	label_int
0	#Noiaaa#goro#aspettandolasera @ Porto di Goro ...	red_heart	3115912511	0
1	e niente, nonostante i casini e gli impegni di...	red_heart	423498157	0
2	#Faccebuffe #friends #friendship #saturdaynigh...	red_heart	488427533	0
3	Un nuovo post è ora online su http://t.co/h6pK...	red_heart	3041411470	0
4	@vogliosoloriker video e magari iscriverti?Ho ...	red_heart	1693227686	0
...	...	...	...	...
249995	A chi non sceglie mai A chi non rischia mai A ...	rose	1137209850	24
249996	Io sono CHARLIE http://t.co/h93GZPJTds #dress...	rose	2535236773	24
249997	#10febbraio Voglio ricordare gli innocenti inf...	rose	2290924705	24
249998	@rominamanguel @animalesoficial @weareblackmil...	rose	1317304885	24
249999	@Edi_hermida Belleza neneeeee	rose	1317304885	24

250000 rows × 4 columns

Figura 2.2: Dataset di training

2.2 Bilanciamento del dataset

Vediamo adesso come sono state ripartite queste classi all'interno del dataset appena ottenuto.

Possiamo intuire facilmente dalla figura 2.3 la presenza di uno sbilanciamento visto che le prime tre classi contengono metà dei tweets. Tale distribuzione non omogenea potrebbe portare i futuri modelli a classificare i tweets solo per queste tre classi e dunque a influenzarli negativamente.

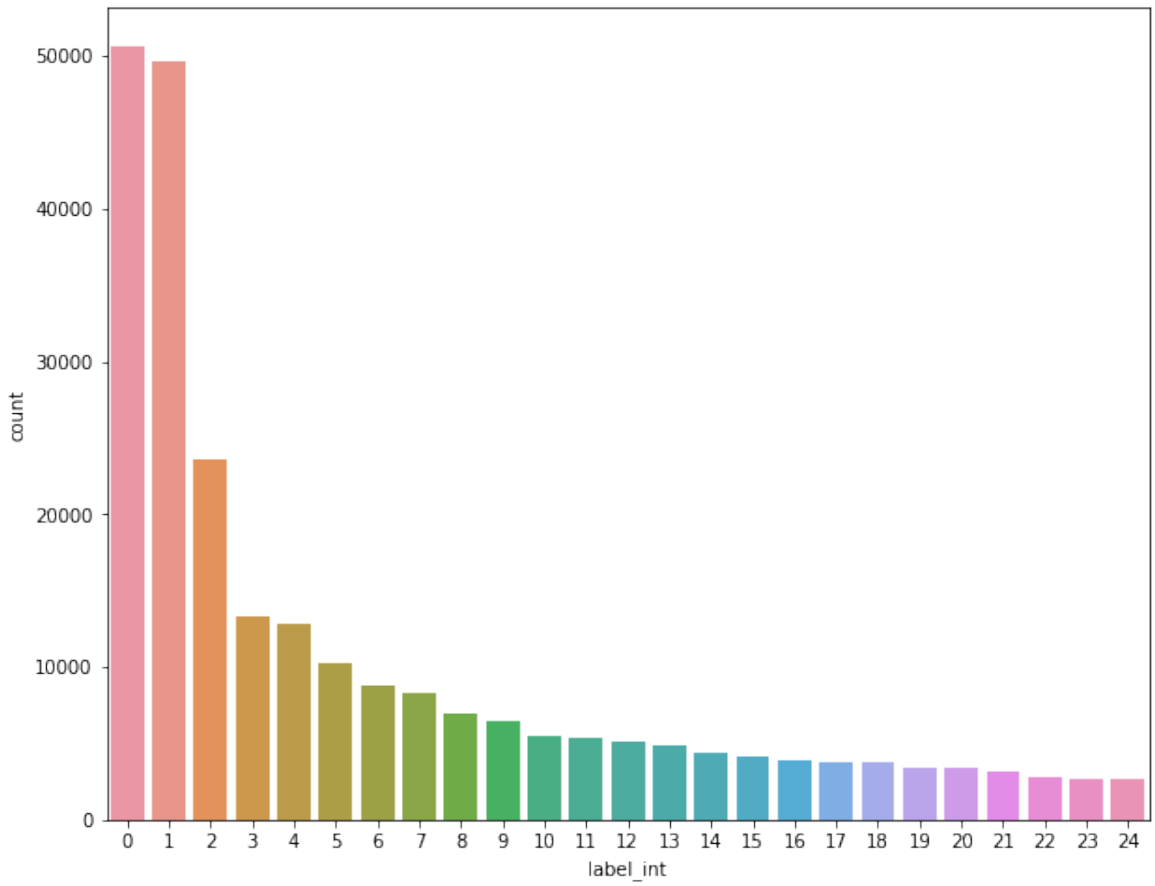


Figura 2.3: Ripartizione calssi

Dunque al fine di bilanciare il dataset è stato eseguito sia l'up-sampling

relativo alle classi etichettate dalla 24 alla 7 in modo tale da poter incrementare il loro numero. Sia il down-sampling relativo alle classi etichettate dalla 0 alla 2 per poter decrementare il loro numero. Tutta via l'upsampling viene fatto fino ad un massimo del 20% dei dati già esistenti, poichè se maggiore influenzerebbe troppo l'intero dataset.

Per poter eseguire questo bilanciamento è stata utilizzata inizialmente la funzione "Resample" fornita dalla libreria, già citata, "Scikit-learn" la quale prendendo in input i testi etichettati restituisce una sequenza di copie ricampionate. Il numero di samples generati viene indicato come parametro nella funzione, in modo da poter individuare il valore corretto senza influenzare negativamente i dati.

Tuttavia questo primo approccio ha permesso di aumentare i dati ma ha anche introdotto copie di tweets già esistenti, dunque è stato utilizzata una seconda tecnica, sempre appartentente alla libreria "Scikit-learn", per eseguire up-sampling e down-sampling. Per poter applicare questo metodo è necessario a priori splittare il dataset in train e test set, tokenizzare il testo dei tweets e aggiungere il padding per far avere la stessa lunghezza a tutti. Tutto ciò è importante poichè il ribilanciamento deve avvenire sul train test e i relativi dati passati devono essere trasformati in formato numerico per poter essere trattati.

A questo punto per quanto riguarda il sottocampionamento viene importato "RandomUnderSampler" che non è altro che un modo semplice e veloce per bilanciare i dati selezionando casualmente un sottoinsieme di dati per le classi.

```
from imblearn.under_sampling import RandomUnderSampler
undersample = RandomUnderSampler(sampling_strategy={2:10800, 1: 11000, 0: 11000})
X_over, y_over = undersample.fit_resample(X_train, y_train)
```

Figura 2.4: Applicazione under-sampling

Per quanto riguarda l'up-sampling viene invece importato "SMOTE" che sta per "Synthetic Minority Oversampling Technique" ed è un metodo popolare per sovracampionare le classi di minoranza.

La classe di minoranza viene sovracampionata prendendo ogni sample di tale classe e introducendo esempi sintetici lungo la linea di segmenti che uniscono tutte le k classi di minoranza più vicine. Questi esempi sintentici sono generati prendendo la differenza tra il vettore del sample in questione e quello più vicino ad esso, moltiplicandolo per un valore casuale compreso tra 0 e 1 e aggiungerlo nuovamente al vettore in esame.

```
from imblearn.over_sampling import SMOTE
sm = SMOTE(sampling_strategy={24: 3000, 23: 3000, 22: 3000, 21: 3700, 20: 3900, 19
X_res, y_res = sm.fit_resample(X_over, y_over)
```

Figura 2.5: Applicazione up-sampling

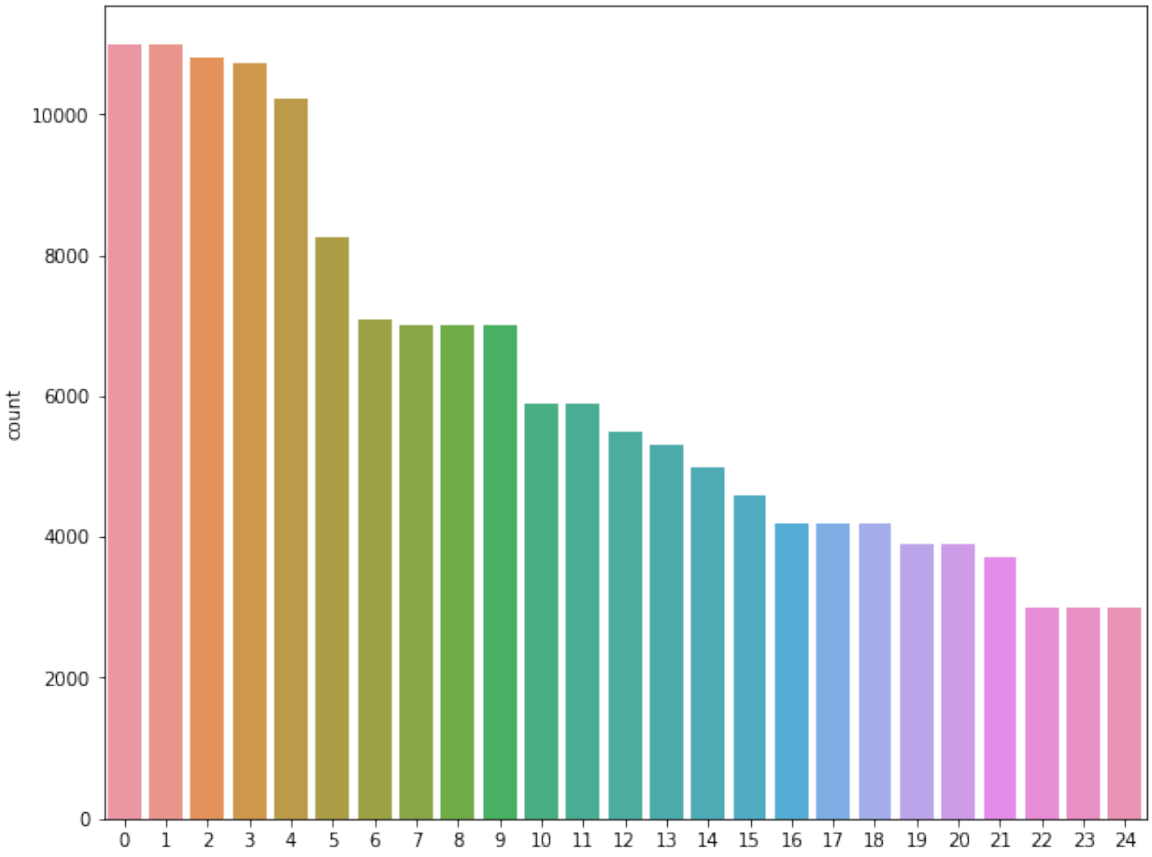


Figura 2.6: Ribilanciamento dataset

2.3 Pre-processing e cleaning dei dati

Per poter fare una buona classificazione, prima di trasformare il testo in formato numerico, ho eseguito un pre-processing per ogni tweet del data set, in modo da filtrarlo e ripulirlo dalle stopwords, da link "http" e tag "@", i quali non sono utili ai fini della classificazione. Il pre-processing dei dati è quindi il processo di conversione di quest'ultimi in qualcosa che un computer possa capire.

Per la rimozione delle stopwords, cioè parole comunemente usate ma che devono essere ignorate, viene adottata la libreria NLTK la quale fornisce file in diverse lingue contenenti per l'appunto tali parole da filtrare. Inoltre ho rimosso tutti i caratteri speciali, i quali non sono stati ritenuti importanti per la classificazione.

L'ultimo punto del pre-processing riguarda lo Stemming, il quale risulta essere solitamente utile per la pulizia del testo. Esso è un processo di riduzione di una parola alla sua radice di parole, tuttavia questo processo non è stato più applicato poichè le prestazioni dei modelli peggioravano con la sua presenza.

Vediamo adesso alcuni esempi di come sono diventati i tweets dopo il loro pre-processing:

Tuttavia, dopo la pulizia, alcune righe del dataframe sono diventate vuote, dunque eliminate per evitare di influenzare il futuro modello negativamente.

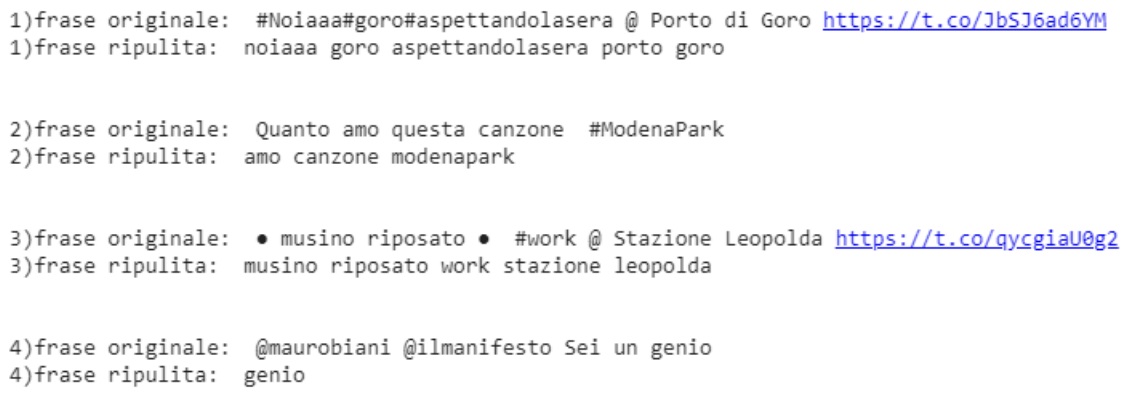


Figura 2.7: Esempio pulizia tweets

2.4 Ottenimento delle features

Un'ulteriore studio nel dataset è stato rivolto alla raccolta delle caratteristiche, che forniscono informazioni aggiuntive per il training dei modelli.

Piuttosto che affidarmi esclusivamente al testo del tweet di destinazione, ho sfruttato ulteriori funzionalità basate sull'utente per migliorare le prestazioni. Il task presenta diverse varianti del volto sorridente ("smiling_face_with heart_eyes" o "smiling_face_with_smiling_eyes") e tre diversi emojis del cuore, rendendo impossibile anche per una persona determinare quello più adatto solo sulla base di un tweet.

Quindi una delle caratteristiche che ho deiciso di considerare è stata la precedente distribuzione di emoji per un autore, per questo viene mantenuta la colonna "uid" nel dataset di partenza. L'ipotesi è che la scelta di un particolare emoji viene guidata principalmente dalle preferenze personali dell'utente, semplificate per l'appunto dalle precedenti scelte.

A tal fine, viene raccolta la emoji-history per ogni utente utilizzando le etichette nel set di train per calcolare le distribuzioni delle emoji, ottenendo

così vettori di dimensione 25. Infine viene concatenato questo risultato con il dataset totale per poi eseguire lo splitting in train e test set.

label_int	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
uid																									
11847	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12243	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31253	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
42173	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
63533	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
992562998073004032	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
992798072991252480	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
992854744933781504	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
993001424823955456	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
993040625602834432	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

43671 rows × 25 columns

Figura 2.8: History emoji per ogni utente nel dataset

	text_pulito	label	label_int	uid	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	noiaaa goro aspettandolasera porto goro	red_heart	0	3115912511	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	wee va mui bien page taggata veramente fantast...	two_hearts	20	3115912511	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
2	niente nonostante casini impegni tutte amiche ...	red_heart	0	423498157	24	21	10	0	0	0	0	0	4	0	0	0	0	0	6	3	0	0	16	2	2	1	0	1	0
3	niente immagino ragazza ricoverata li trova so...	red_heart	0	423498157	24	21	10	0	0	0	0	0	4	0	0	0	0	0	6	3	0	0	16	2	2	1	0	1	0
4	sempre pacco regalo	red_heart	0	423498157	24	21	10	0	0	0	0	0	4	0	0	0	0	0	6	3	0	0	16	2	2	1	0	1	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
249401	buon compleanno	rose	24	926755904485240832	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
249402	buongiorno te	rose	24	907220181549711360	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
249403	auguri tutte donne 8marzo festadelladonna	rose	24	934075873619308544	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
249404	buongiorno annalisa auguri felice festa donna	rose	24	841289941	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
249405	infatti buonanotte annalisa	rose	24	841289941	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2

249406 rows × 29 columns

Figura 2.9: Merge del dataset con l'history emoji

2.5 Suddivisione dei dati in training e test set

Il punto successivo da affrontare è quello della suddivisione in training e test set. Per fare ciò è stato essenziale riordinare il testo ripulito, i vettori delle features e le etichette che identificano l'emoji.

Utilizzando la funzione "train_test_split" viene assegnato l'80% dei dati al training set e il restante 20% al test set, inoltre viene impostato il parametro "RANDOM_STATE", il quale permette di eseguire uno shuffle dei dati e di evitare che vengano passati come input al modello prima tutti i dati di una classe e poi tutti i dati di un'altra, imparando così a riconoscere sempre prima una e poi l'altra.

Una volta ottenuti i due dataframe, i valori venogno recuperati direttamente da essi tramite la funzione "values", così da avere separati il testo ripulito, le features e le etichette. Questo passo risulta essere necessario poichè i dati da fornire come input al futuro modello saranno di due tipi, i primi verranno passati al livello di Embedding di Keras mentre i secondi verranno

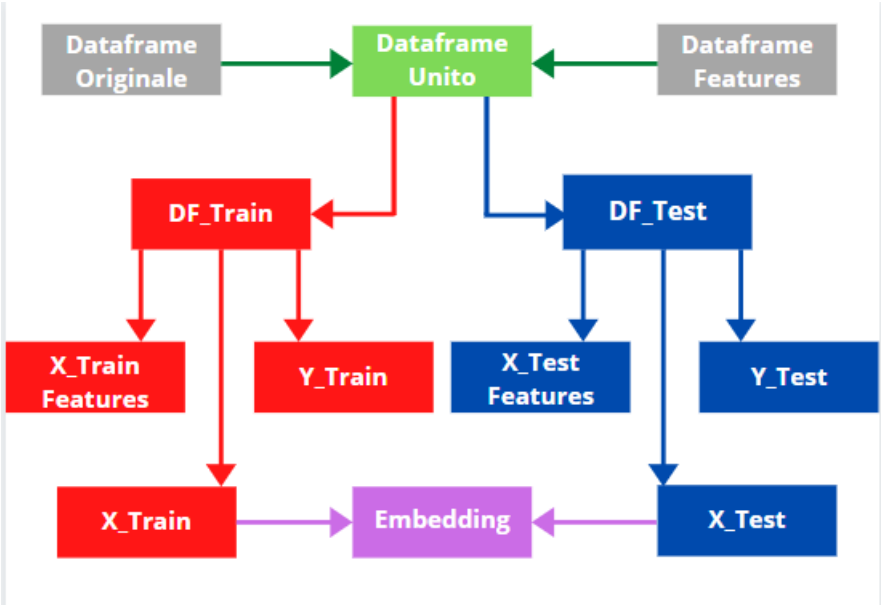


Figura 2.10: Suddivisione dei dati in train e test set

passati ad uno specifico livello di Input avente un numero di neuroni pari al numero delle features calcolate. Tutta via per poter passare il testo ripulito al livello di Embedding è necessario eseguire prima la trasformazione in formato numerico.

2.6 Word Embedding

Per poter effettivamente ottenere delle informazioni usabili dal testo è necessario eseguire una trasformazione in formato numerico in modo tale da passarle poi alle reti neurali.

Il word embedding è un tipo di rappresentazione distribuita che consente alle parole che hanno stesso significato di avere una rappresentazione simile. Dunque vengono creati vettori di numeri reali in uno spazio vettoriale predefinito, questi rappresentano delle parole o delle frasi attraverso l'utilizzo di strumenti e tecniche di apprendimento.

L'aspetto più interessante di questo approccio è che a differenza di metodi come "one-hot embeddings" che rappresentano solo le parole attraverso vettori di grandi dimensioni, questi altri codificano il significato e il ruolo della parola nel vettore mantenendo così le informazioni sulle relazioni delle diverse parole. Dunque tramite il word embedding possiamo sviluppare modelli che addestrino i vettori stessi tramite le reti neurali partendo da un insieme di testi selezionati e organizzati per facilitare le analisi. In questo caso in particolare ho sfruttato un Word Embeddings pre-allenato che ha permesso di incrementare le prestazioni dei modelli.

Dunque per eseguire la trasformazione di ogni tweet ripulito in un vettore di numeri interi è necessario creare ed allenare un vocabolario con tutte le parole del dataset, le quali vengono associate ad un'indice cioè ad un numero. Per fare ciò viene utilizzata la classe Tokenizer di Keras ed utilizzato il metodo "fit_on_texts".

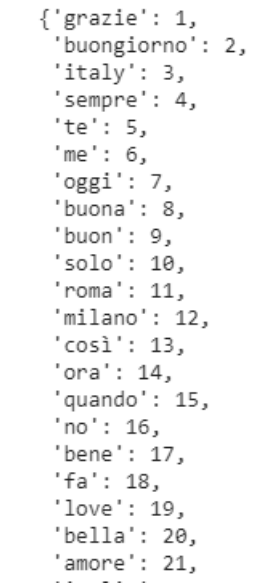


Figura 2.11: Esempio creazione vocabolario

Una volta ottenuto il vocabolario viene calcolata la sua dimensione che è pari a 152759 parole e viene ricercata la frase con massima lunghezza che è di 25 parole.

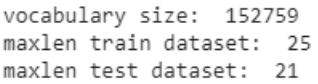


Figura 2.12: Dimensione vocabolario e frase con massima lunghezza

Questo viene fatto poichè per poter dare in input i dati ottenuti ad una rete neurale le frasi devono essere tutte della stessa lunghezza, quindi per

riportarle tutte alla stessa lunghezza uso la tecnica del post_zero_padding. Quando trovo una frase che è più piccola aggiungo degli zeri tramite la funzione pad_sequences specificando dove si trovano le frasi, dove aggiungere gli 0 e la lunghezza massima che sarà di 25 parole

```
Frase:
noiaaa goro aspettandolasera porto goro


Frase in testo numerico:
[61193, 32232, 61194, 212, 32232]


Frase in testo numerico con Post_0_padding:
[61193, 32232, 61194, 212, 32232, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figura 2.13: Esempio Embedding di una frase del dataset

Come possiamo vedere in Figura è stata eseguita la trasformazione della frase in testo nuemerico. La sua lunghezza prima del padding risulta essere pari a 5 parole, successivamente con l'aggiunta del padding viene raggiunta la lunghezza massima di 25 parole.

2.7 Carico il word embedding pre-addestrato

L'Embedder in questione è stato messo a disposizione da "Fasttext" e al suo interno contiene vettori di dimensioni pari a 100, i quali sono stati addestrati su 48 milioni di tweet italiani.

Una volta scaricato tale word embedding viene caricato e vengono associate le parole alla loro rappresentazione vettoriale.

```
with open("/content/drive/My Drive/embeddings.vec", "r") as ins:
  array = []
  for line in ins:
    array.append(line)

array = array[1:]
word_embedder = dict()
for el in array:
  word_array = el.split(sep=" ")
  values = []
  for value in word_array[1:-1]:
    values.append(float(value))
  word_embedder[word_array[0]]= values

print('vettori trovati: ',len(word_embedder))
print('vettore per la parola "buongiorno":')
print(word_embedder["buongiorno"])
```

```
vettori trovati:  1173127
vettore per la parola "buongiorno":
[-0.42796, -0.27427, -0.26669, 0.20342, 0.075293, 0.40008, -0.24121, -0.12603, -0.3
```

Figura 2.14: Caricamento embedding pre-addestrato

Successivamente, viene creata l'embedding matrix corrispondente che possiamo usare nell' Embedding layer del futuro modello. Questa embedding matrix è una semplice matrice NumPy in cui all'indice "i" abbiamo il vettore pre-addestrato per la parola del nostro vocabolario. Dunque il risultato è una matrice di pesi solo per le parole che vedremo durante l'allenamento.

```python
def embedding_matrix_process(vocab_size, embedding_words):
  embedding_matrix = np.zeros((vocab_size,100))
  for word, index in tokenizer.word_index.items():
    embedding_vector = embedding_words.get(word)
    if embedding_vector is not None:
      embedding_matrix[index]= embedding_vector
  return embedding_matrix
  pesi = embedding_matrix_process(len_voc, word_embedder)
```

Figura 2.15: Embendding matrix

```python
print('parola nel dizionario con indice 2: ',list(tokenizer.word_in
[list(tokenizer.word_index.values()).index(2)])
print('vettore riferito alla parola "buongiorno": ')
print(pesi[2])
```

```
parola nel dizionario con indice 2:  buongiorno
vettore riferito alla parola "buongiorno":
[-0.42796   -0.27427   -0.26669    0.20342    0.075293   0.40008
 -0.24121   -0.12603   -0.31635    0.42902   -0.22592   -0.53956
  0.56822   -0.30026   -0.18446   -0.088367  -0.11481    0.027399
 -0.34573    0.23588    0.063126  -0.082184  -0.36246   -0.14178
  0.19619   -0.04596   -0.19408    0.19945   -0.090471   0.51436
  0.38475    0.11413   -0.28427   -0.071697   0.09936   -0.26537
 -0.34247   -0.40442   -0.074626   0.24123    0.045693  -0.3081
 -0.39334    0.44426    0.17339   -0.38771    0.39896    0.67075
  0.25187   -0.053918  -0.24634    0.49793    0.40529    0.0024158
  0.10269    0.26622   -0.1466     0.18717    0.21863    0.2537
  0.33323    0.01371    0.23325   -0.0092188  0.11024    0.65207
  0.36633    0.16951   -0.11397   -0.096092  -0.86539   -0.22932
 -0.13951   -0.31495    0.271      0.010508   0.20947    0.09442
  0.37677    0.015595   0.22908    0.11139    0.7105     0.30741
 -0.65215    0.40109    0.61314    0.43583    0.88781    0.33354
  0.50502    0.20713    0.14646    0.090021  -0.28759    0.013162
  0.069668  -0.31478    0.57253   -0.25457  ]
```

Figura 2.16: Esempio funzionamento embedding matrix

2.8 Struttura delle reti

Sono stati prodotti diversi modelli nella costruzione della Artificial Neural Network finale, cambiando di volta in volta la selezione delle variabili e l'apprendimento dei parametri in base:

- a quali input verranno utilizzati,
- al numero di hidden layers che ha la rete,
- al numero di nodi presenti in ogni hidden layers.

I primi modelli creati risultano essere tutti Fully Connected Networks, cioè che ogni neurone è collegato con tutti quelli presenti nel layer successivo, mentre andando avanti con lo studio si è passati a reti più complesse come le BiLSTM, le quali non sono altro che reti con un livello bidirezionale aventi come parametro un layer LSTM (Long short-term memory).

Questa tipologia di reti sono in grado di catturare le relazioni temporali tra le parole che compaiono in una frase, quindi prendendo in considerazione una parola è in grado di metterla in relazione sia con quelle precedenti che con le successive.

Inoltre per quanto riguarda i primi due modelli, questi sono caratterizzati dall'avere un solo livello di input dedicato al Embedding del testo, mentre dal terzo modello si aggiunge un livello di input dedicato alle Features calcolate in precedenza.

Il layer di Embedding viene utilizzato specificando come paramtri:

- la dimensione dell'input ovvero quella del vocabolario,
- la dimensione dell'output ovvero quella dello spazio vettoriale in cui verranno incorporate le parole, dove ognuna di esse verrà rappresentata da un vettore avente questa dimensione, cioè 100 visto che è stato utilizzato un word embedding pre-addestrato avente tale dimensione,
- Infine la lunghezza delle nostre frasi che in questo caso sarà 25 ovvero la frase più lunga che abbiamo nel dataset.

La differenza sostanziale per questo embedding layer è che i pesi non verranno inizializzati casualmente ma verranno utilizzati quelli ottenuti dalla precedente embedding matrix, imparando così un embedding per tutte le parole passate come vettore numerico. L'output di questo livello sarà l'ottenimento di una matrice, con valori normalizzati tra 1 e -1, che contiene un vettore per ogni parola nel testo. Infine, non vogliamo aggiornare i pesi delle parole apprese in questo modello, quindi imposteremo l'attributo addestrabile per il modello su False.

2.8.1 Modello base

Come è possibile vedere dall'immagine sottostante, viene sviluppato il modello sequenziale offerto da Keras composto principalmente dall'embedder non allenabile e dagli altri livelli fully-connected di tipo Dense.

Per poter applicare quanto detto vengono passate le frasi in formato numerico come input all'embedder pre-allenato generando così una matrice per ogni frase. Dato che si vuole collegare la rete direttamente al risultato di questo livello, è necessario utilizzare un layer di tipo Flatten() che permetterà di compattare tale matrice 2D in un vettore 1D.

Dopo aver eseguito questa trasformazione dei dati possiamo passare al livello MLP, formato da tre ulteriori livelli di tipo Dense(). I primi due sono detti "hidden" ed hanno rispettivamente 256 e 128 neuroni con funzione di attivazione "Relu" mentre l'ultimo, detto "Output", ha 25 neuroni con funzione di attivazione "Softmax", quest'ultimo fornisce una distribuzione di probabilità rispetto alle classi menzionate.

L'allenamento vede il ripetersi di 10 epoche con un batch_size pari a 128. L'ottimizatore utilizzato è stato Adam, come funzione di loss si è usata la "binary_crossentropy" e come metrica per il calcolo delle performance si è utilizzata l'Accuracy. Tale allenamento ha visto come risultato un'accuratezza pari al 35%.

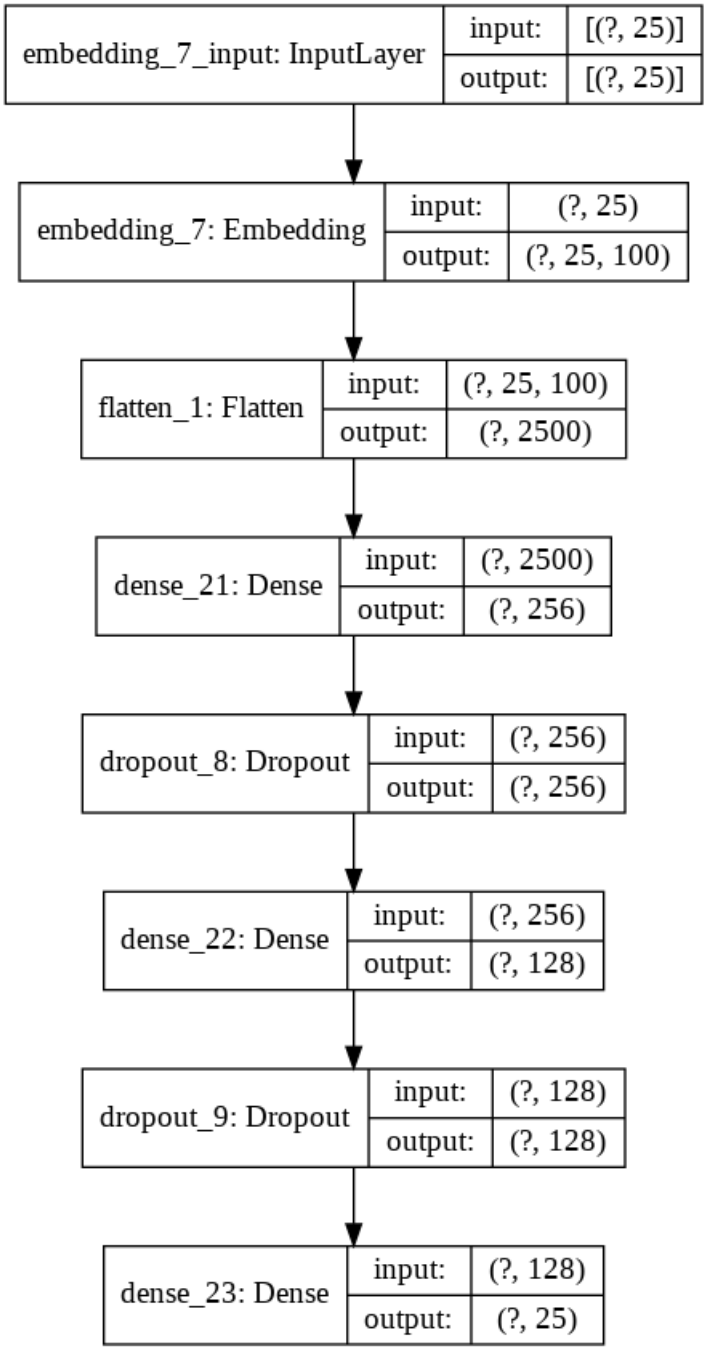


Figura 2.17: Struttura modello base

```
Epoch 1/10
1559/1559 [==============================] - 5s 3ms/step - loss: 2.4022 - accuracy: 0.3359 - val_loss: 2.2892 - val_accuracy: 0.3532
Epoch 2/10
1559/1559 [==============================] - 5s 3ms/step - loss: 2.2840 - accuracy: 0.3574 - val_loss: 2.2673 - val_accuracy: 0.3600
Epoch 3/10
1559/1559 [==============================] - 5s 3ms/step - loss: 2.2280 - accuracy: 0.3682 - val_loss: 2.2683 - val_accuracy: 0.3623
Epoch 4/10
1559/1559 [==============================] - 5s 3ms/step - loss: 2.1775 - accuracy: 0.3775 - val_loss: 2.2667 - val_accuracy: 0.3624
Epoch 5/10
1559/1559 [==============================] - 5s 3ms/step - loss: 2.1291 - accuracy: 0.3870 - val_loss: 2.2773 - val_accuracy: 0.3602
Epoch 6/10
1559/1559 [==============================] - 5s 3ms/step - loss: 2.0834 - accuracy: 0.3973 - val_loss: 2.3015 - val_accuracy: 0.3595
Epoch 7/10
1559/1559 [==============================] - 5s 3ms/step - loss: 2.0395 - accuracy: 0.4058 - val_loss: 2.3197 - val_accuracy: 0.3583
Epoch 8/10
1559/1559 [==============================] - 5s 3ms/step - loss: 2.0028 - accuracy: 0.4149 - val_loss: 2.3419 - val_accuracy: 0.3568
Epoch 9/10
1559/1559 [==============================] - 5s 3ms/step - loss: 1.9648 - accuracy: 0.4234 - val_loss: 2.3709 - val_accuracy: 0.3534
Epoch 10/10
1559/1559 [==============================] - 5s 3ms/step - loss: 1.9338 - accuracy: 0.4298 - val_loss: 2.3995 - val_accuracy: 0.3525
```

Figura 2.18: Addestramento modello base

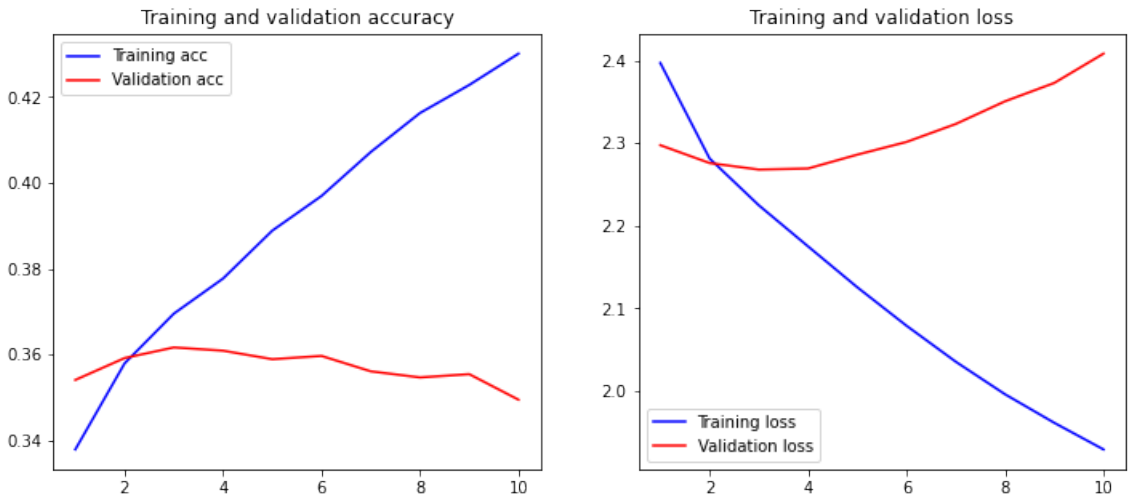


Figura 2.19: Training and validation accuracy/loss

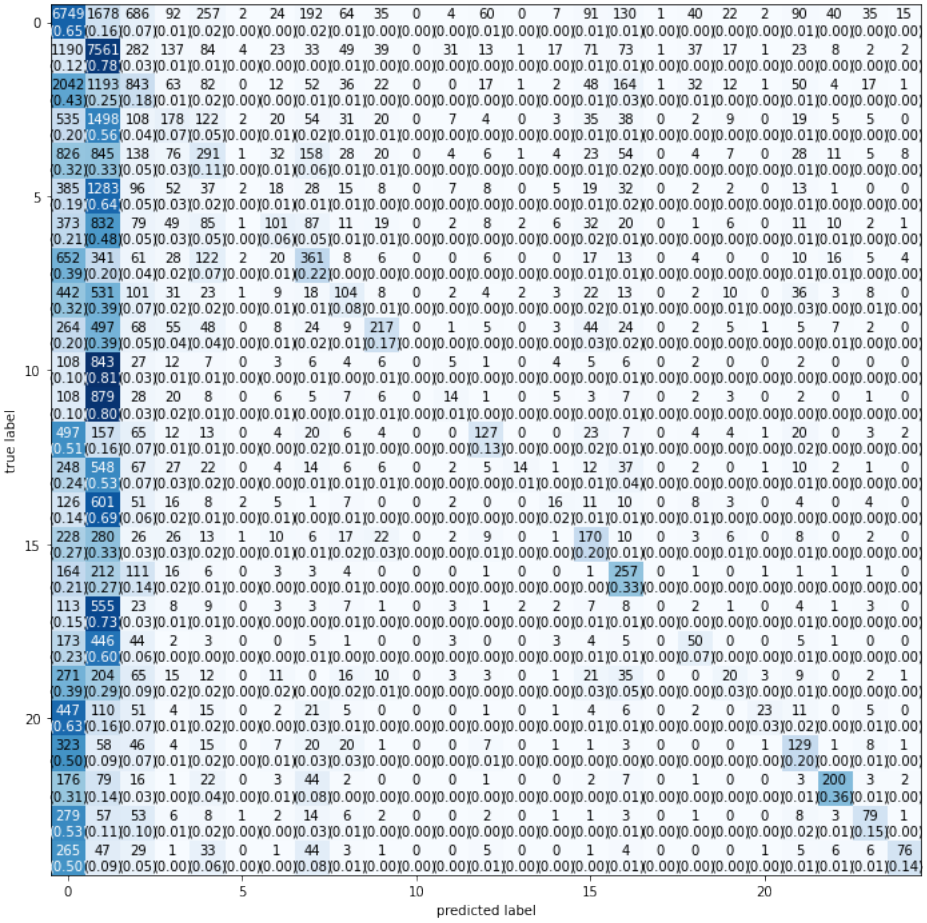


Figura 2.20: Matrice di confusione del modello base

2.8.2 Modello base: dataset bilanciato

Tuttavia avendo visto che le precedenti prestazioni non sono così elevate, sono state supposte diverse ipotesi per migliorarle. Analizzando la matrice di confusione riportata sopra si è notato che il modello prevedeva qualsiasi tweet come appartenente o alla prima o alla seconda classe, questo è dovuto probabilmente alla grandissima presenza di quest'ultime due classi rispetto tutte le altre. Dunque dopo aver bilanciato il dataset si è provato a riallenare lo stesso modello. L'allenamento del modello avviene sempre su 10 epoche

```
Epoch 1/10
1212/1212 [==============================] - 7s 5ms/step - loss: 2.8569 - accuracy: 0.1628 - val_loss: 2.5273 - val_accuracy: 0.2877
Epoch 2/10
1212/1212 [==============================] - 6s 5ms/step - loss: 2.7247 - accuracy: 0.1977 - val_loss: 2.5129 - val_accuracy: 0.2937
Epoch 3/10
1212/1212 [==============================] - 6s 5ms/step - loss: 2.6603 - accuracy: 0.2126 - val_loss: 2.4888 - val_accuracy: 0.2881
Epoch 4/10
1212/1212 [==============================] - 7s 6ms/step - loss: 2.5996 - accuracy: 0.2281 - val_loss: 2.4693 - val_accuracy: 0.3013
Epoch 5/10
1212/1212 [==============================] - 8s 6ms/step - loss: 2.5377 - accuracy: 0.2442 - val_loss: 2.5328 - val_accuracy: 0.2665
Epoch 6/10
1212/1212 [==============================] - 6s 5ms/step - loss: 2.4780 - accuracy: 0.2609 - val_loss: 2.5314 - val_accuracy: 0.2714
Epoch 7/10
1212/1212 [==============================] - 6s 5ms/step - loss: 2.4223 - accuracy: 0.2731 - val_loss: 2.5709 - val_accuracy: 0.2580
Epoch 8/10
1212/1212 [==============================] - 6s 5ms/step - loss: 2.3699 - accuracy: 0.2863 - val_loss: 2.5805 - val_accuracy: 0.2600
Epoch 9/10
1212/1212 [==============================] - 6s 5ms/step - loss: 2.3229 - accuracy: 0.2980 - val_loss: 2.6263 - val_accuracy: 0.2542
Epoch 10/10
1212/1212 [==============================] - 6s 5ms/step - loss: 2.2801 - accuracy: 0.3092 - val_loss: 2.6460 - val_accuracy: 0.2504
```

Figura 2.21: Addestramento modello base dopo il bilanciamento

con un batch_size pari a 128. L'ottimizatore utilizzato è sempre l'Adam, come funzione di loss si è usata la "binary_crossentropy" e come metrica per il calcolo delle performance si è utilizzata l'Accuracy.

Tuttavia come è possibile vedere già dall'allenamento il modello invece di portare ad un miglioramento risulta essere peggiorato. A questo punto è stato pensato di sfruttare la distribuzione delle emoji usate da ogni utente autore di un tweet, aggiungendo alla rete neurale le features calcolate precedentemente.

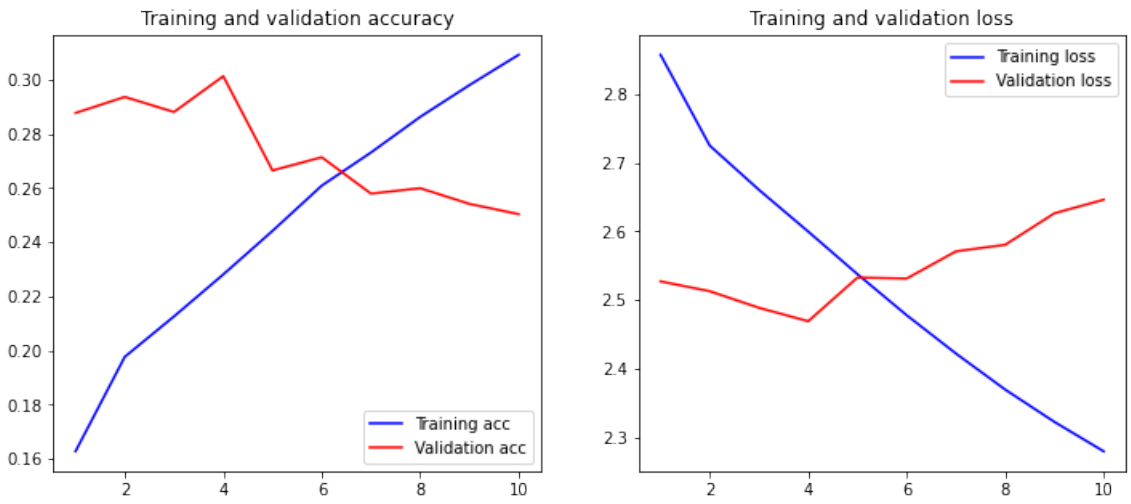


Figura 2.22: Training and validation accuracy/loss

2.8.3 Modello base: livello di input per le Features

In questo modello, come detto in precedenza, viene aggiunto un secondo layer di input, il quale è caratterizzato dal numero di neuroni pari al numero di features calcolate e prese in input per ogni frase.

A questo punto i due livelli di input, cioè quello dove viene inserito l'embedding della frase e quello dove vengono inserite le features, sono concatenati insieme per poi passare l'output ai livelli di tipo Dense() come in precedenza. L'allenamento di tale modello vede il ripetersi anche un questo caso di 10 epoche con un batch_size pari a 128. L'ottimizatore utilizzato è stato Adam, come funzione di loss si è usata la "binary_crossentropy" e come metrica per il calcolo delle performance si è utilizzata l'Accuracy.

Come è possibile vedere da quest'ultimo allenamento, l'accuratezza del modello è stata incrementata passando dal 35% del modello base fino a raggiungere il 58%.

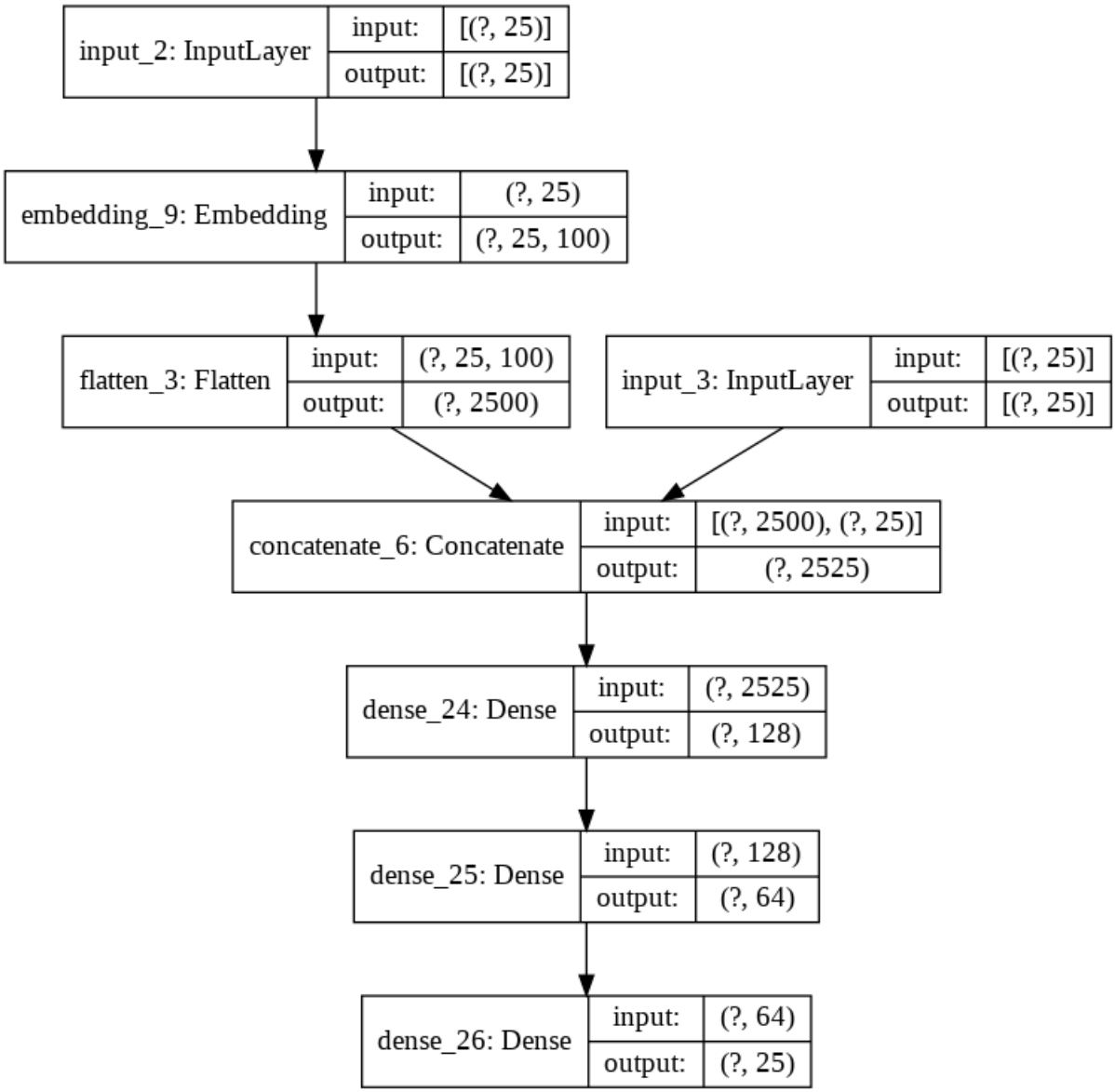


Figura 2.23: Struttura modello base con le features

```
Epoch 1/10
1559/1559 [==============================] - 6s 4ms/step - loss: 0.1044 - accuracy: 0.5063 - val_loss: 0.0893 - val_accuracy: 0.5670
Epoch 2/10
1559/1559 [==============================] - 6s 4ms/step - loss: 0.0810 - accuracy: 0.5888 - val_loss: 0.0771 - val_accuracy: 0.5905
Epoch 3/10
1559/1559 [==============================] - 5s 3ms/step - loss: 0.0718 - accuracy: 0.6147 - val_loss: 0.0749 - val_accuracy: 0.5953
Epoch 4/10
1559/1559 [==============================] - 6s 4ms/step - loss: 0.0680 - accuracy: 0.6320 - val_loss: 0.0754 - val_accuracy: 0.5962
Epoch 5/10
1559/1559 [==============================] - 6s 4ms/step - loss: 0.0651 - accuracy: 0.6454 - val_loss: 0.0757 - val_accuracy: 0.5949
Epoch 6/10
1559/1559 [==============================] - 5s 3ms/step - loss: 0.0632 - accuracy: 0.6577 - val_loss: 0.0771 - val_accuracy: 0.5925
Epoch 7/10
1559/1559 [==============================] - 5s 3ms/step - loss: 0.0609 - accuracy: 0.6700 - val_loss: 0.0790 - val_accuracy: 0.5899
Epoch 8/10
1559/1559 [==============================] - 5s 3ms/step - loss: 0.0605 - accuracy: 0.6805 - val_loss: 0.0804 - val_accuracy: 0.5899
Epoch 9/10
1559/1559 [==============================] - 5s 3ms/step - loss: 0.0577 - accuracy: 0.6901 - val_loss: 0.0820 - val_accuracy: 0.5836
Epoch 10/10
1559/1559 [==============================] - 5s 3ms/step - loss: 0.0567 - accuracy: 0.6989 - val_loss: 0.0844 - val_accuracy: 0.5822
```

Figura 2.24: Allenamento modello base con le features

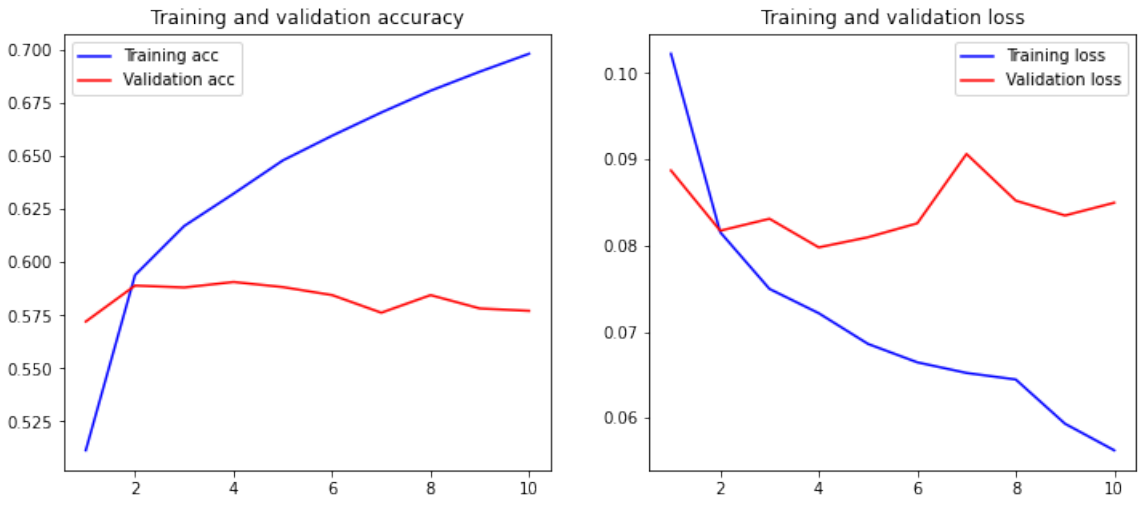


Figura 2.25: Training and validation accuracy/loss

2.8.4 Modello BiLSTM

Questo modello viene sviluppato componendo principalmente l'embedder non allenabile con una BiLSTM, alla quale viene applicato un recurrent dropout pari a 0,2. Anche in questo caso la frase in forma numerica viene passata in input all'embedder che ne estrapola la codifica con il significato intrinseco. Sarà la BiLSTM che cercherà di intuire poi il significato generale.

Come per il precedente modello viene aggiunto un secondo layer di input, il quale è caratterizzato dal numero di neuroni pari al numero di features calcolate e prese in input per ogni frase. A questo punto i due livelli di input, cioè quello dove viene inserito l'embedding della frase e quello dove vengono inserite le features, sono concatenati insieme per poi passare l'output ai livelli di tipo Dense() come in precedenza.

L'allenamento di tale modello vede il ripetersi di 10 epoche con un batch_size pari a 128. L'ottimizatore utilizzato è stato Adam, come funzione di loss si è usata la "binary_crossentropy" e come metrica per il calcolo delle performance si è utilizzata l'Accuracy.

Come è possibile vedere dal seguente allenamento, l'accuratezza del modello è stata incrementata nuovamente passando dal 35% del modello base fino a raggiungere il 54%, tuttavia risulta essere ancora inferiore rispetto la precedente, la quale è stata calcolata con un modello fully connected, quindi provo a modificare leggermente quest'ultimo modello per incrementare ancora le prestazioni.

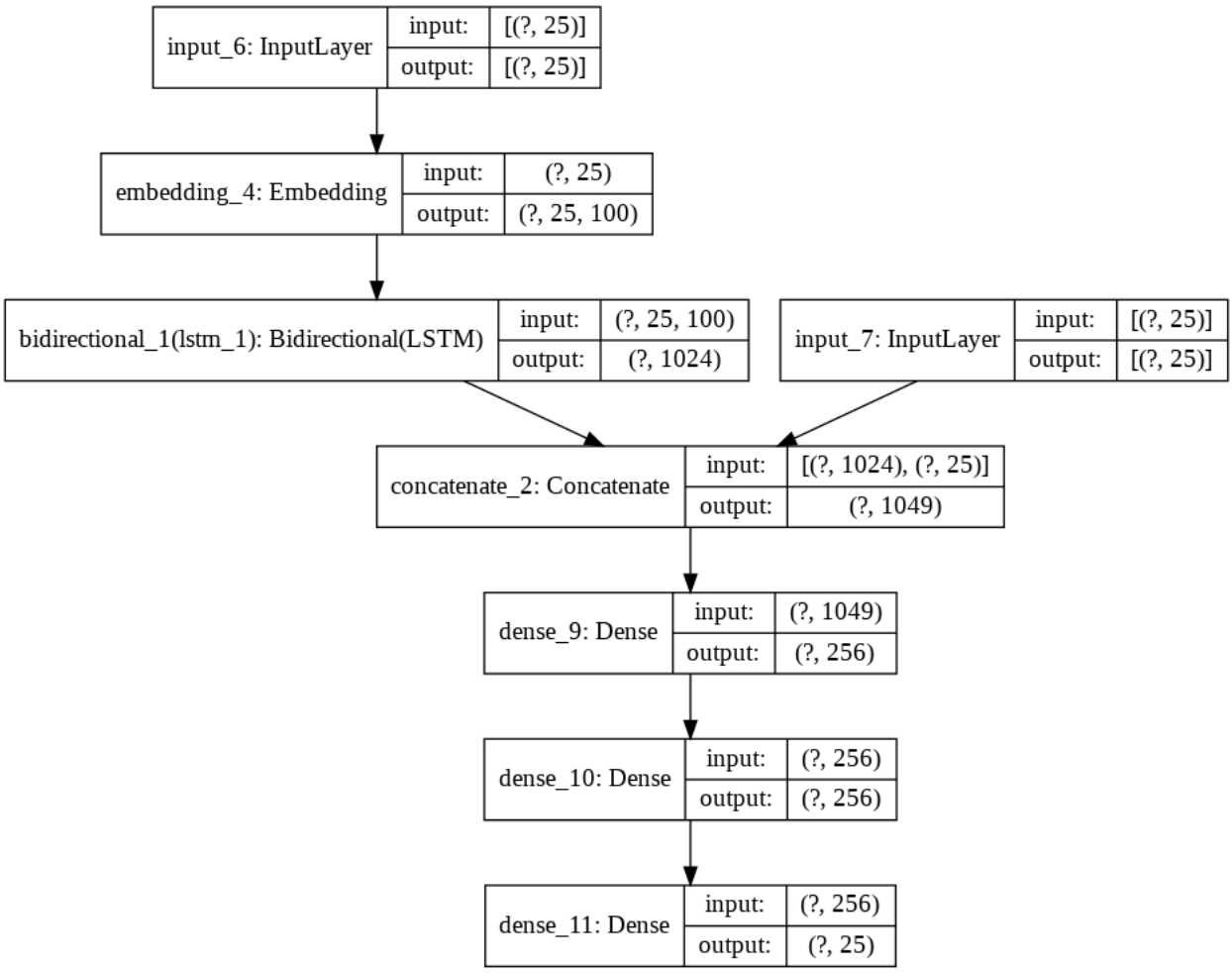


Figura 2.26: Struttura modello BiLSTM

```
Epoch 1/10
1559/1559 [==============================] - 710s 455ms/step - loss: 0.0940 - accuracy: 0.5519 - val_loss: 0.0811 - val_accuracy: 0.5977
Epoch 2/10
1559/1559 [==============================] - 705s 452ms/step - loss: 0.0729 - accuracy: 0.6252 - val_loss: 0.0753 - val_accuracy: 0.6056
Epoch 3/10
1559/1559 [==============================] - 707s 454ms/step - loss: 0.0642 - accuracy: 0.6742 - val_loss: 0.0782 - val_accuracy: 0.6011
Epoch 4/10
1559/1559 [==============================] - 710s 455ms/step - loss: 0.0539 - accuracy: 0.7252 - val_loss: 0.0810 - val_accuracy: 0.5940
Epoch 5/10
1559/1559 [==============================] - 705s 452ms/step - loss: 0.0428 - accuracy: 0.7718 - val_loss: 0.0930 - val_accuracy: 0.5844
Epoch 6/10
1559/1559 [==============================] - 703s 451ms/step - loss: 0.0363 - accuracy: 0.8057 - val_loss: 0.1054 - val_accuracy: 0.5751
Epoch 7/10
1559/1559 [==============================] - 708s 454ms/step - loss: 0.0312 - accuracy: 0.8328 - val_loss: 0.1262 - val_accuracy: 0.5657
Epoch 8/10
1559/1559 [==============================] - 703s 451ms/step - loss: 0.0274 - accuracy: 0.8534 - val_loss: 0.1377 - val_accuracy: 0.5626
Epoch 9/10
1559/1559 [==============================] - 713s 457ms/step - loss: 0.0243 - accuracy: 0.8693 - val_loss: 0.1495 - val_accuracy: 0.5627
Epoch 10/10
1559/1559 [==============================] - 704s 452ms/step - loss: 0.0219 - accuracy: 0.8837 - val_loss: 0.1647 - val_accuracy: 0.5487
```

Figura 2.27: Allenamento modello BiLSTM

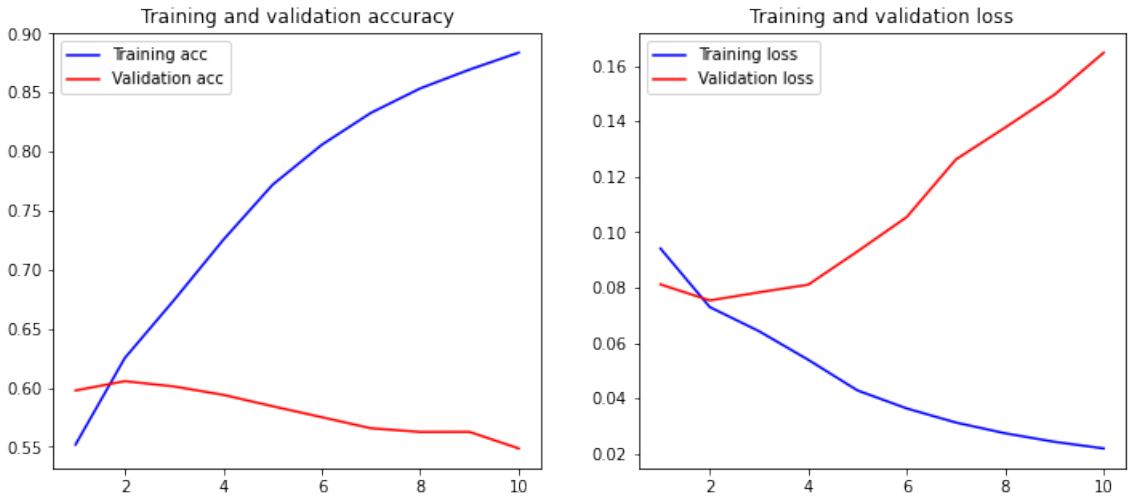


Figura 2.28: Training and validation accuracy/loss

2.8.5 Modello BiLSTM finale

Questo ultimo modello strutturalmente parlando risulta essere molto simile al precedente, la principale differenza risiede nel aver eliminato i due livelli di tipo Dense() dopo la concatenazione dei due livelli di input e nel averli aggiunti dopo il secondo livello di input dedicato alle features.

Inoltre sono stati aggiunti e modificati dei parametri ai singoli livelli, come ad esempio il parametro "regularizers" che è stato aggiunto sia al livello di Embedding che ai livelli di tipo Dense. Questo parametro permette alla rete neurale di apprendere caratteristiche sparse, ridurre l'overfitting e migliorare l'abilità del modello di generalizzare. Inoltre in questo modello la funzione di attivazione utilizzata per il livelli di tipo Dense è stata la "Tanh" poichè la "Relu" ha il problema che trasforma tutti i valori negativi immediatamente pari zero, il che riduce la capacità del modello di adattarsi o addestrarsi correttamente tramite i dati.

L'allenamento di tale modello vede il ripetersi di 10 epoche con un batch_size pari a 128. L'ottimizatore utilizzato è stato Adam, come funzione di loss si è usata la "binary_crossentropy" e come metrica per il calcolo delle performance si è utilizzata l'Accuracy.

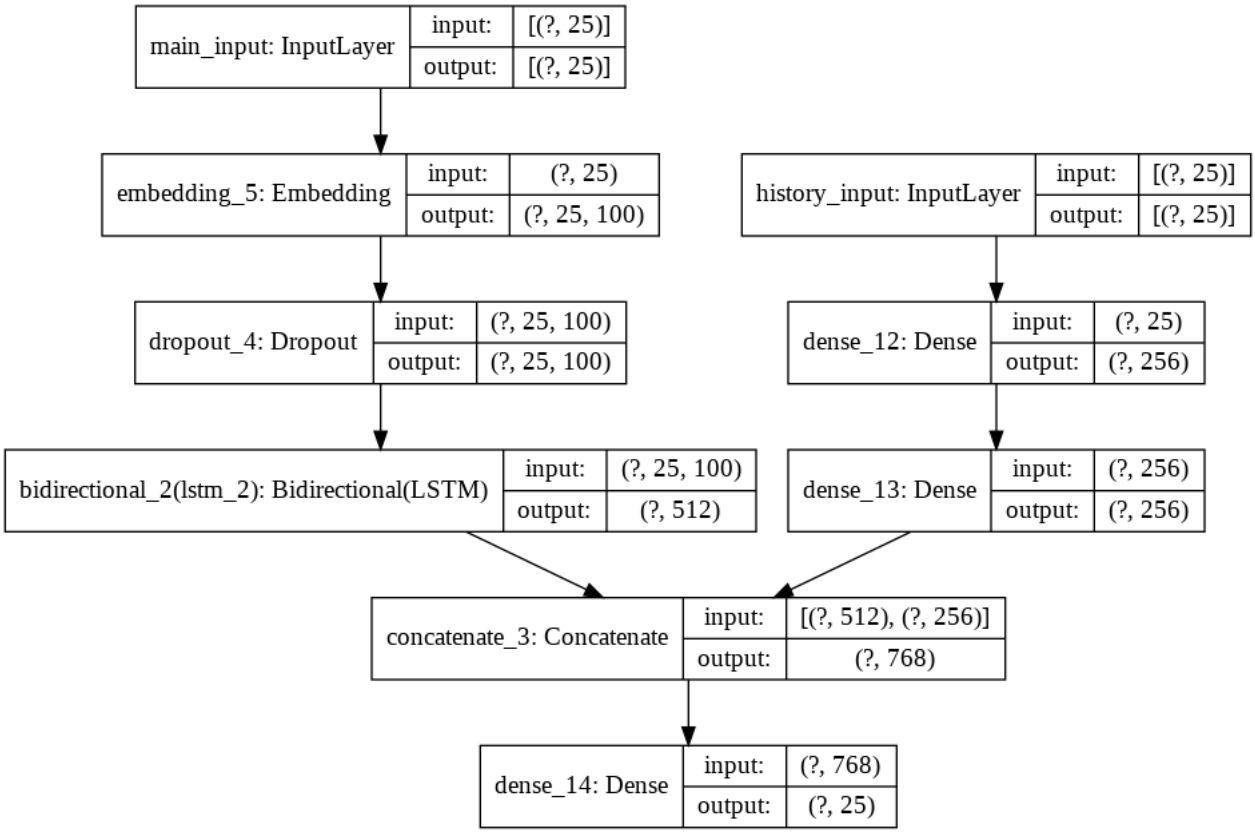


Figura 2.29: Struttura modello BiLSTM Finale

```
Epoch 1/10
1559/1559 [==============================] - 73s 47ms/step - loss: 2.9874 - accuracy: 0.5885 - val_loss: 2.8445 - val_accuracy: 0.6066
Epoch 2/10
1559/1559 [==============================] - 72s 46ms/step - loss: 2.8245 - accuracy: 0.6086 - val_loss: 2.8028 - val_accuracy: 0.6135
Epoch 3/10
1559/1559 [==============================] - 72s 46ms/step - loss: 2.7929 - accuracy: 0.6133 - val_loss: 2.7865 - val_accuracy: 0.6167
Epoch 4/10
1559/1559 [==============================] - 72s 46ms/step - loss: 2.7762 - accuracy: 0.6178 - val_loss: 2.7795 - val_accuracy: 0.6182
Epoch 5/10
1559/1559 [==============================] - 71s 46ms/step - loss: 2.7660 - accuracy: 0.6193 - val_loss: 2.7760 - val_accuracy: 0.6195
Epoch 6/10
1559/1559 [==============================] - 72s 46ms/step - loss: 2.7576 - accuracy: 0.6214 - val_loss: 2.7717 - val_accuracy: 0.6197
Epoch 7/10
1559/1559 [==============================] - 72s 46ms/step - loss: 2.7518 - accuracy: 0.6230 - val_loss: 2.7682 - val_accuracy: 0.6197
Epoch 8/10
1559/1559 [==============================] - 72s 46ms/step - loss: 2.7465 - accuracy: 0.6238 - val_loss: 2.7667 - val_accuracy: 0.6199
Epoch 9/10
1559/1559 [==============================] - 72s 46ms/step - loss: 2.7424 - accuracy: 0.6249 - val_loss: 2.7651 - val_accuracy: 0.6207
Epoch 10/10
1559/1559 [==============================] - 72s 46ms/step - loss: 2.7388 - accuracy: 0.6265 - val_loss: 2.7651 - val_accuracy: 0.6216
```

Figura 2.30: Allenamento modello BiLSTM Finale

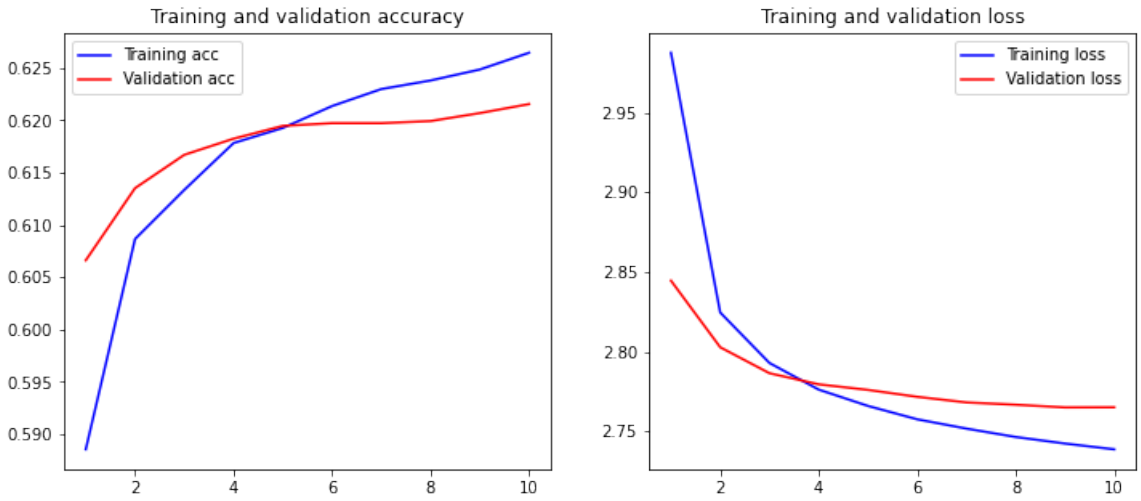


Figura 2.31: Training and validation accuracy/loss

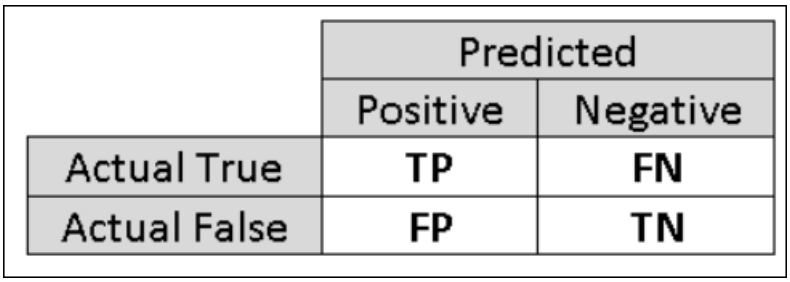
Come è possibile vedere da quest'ultimo allenamento, l'accuratezza del modello è stata incrementata nuovamente passando dal 54% del modello BiLSTM di partenza, fino a raggiungere il 62%.

Capitolo 3

Valutazione dei modelli prodotti

3.1 Metriche utilizzate

Per poter valutare i modelli prodotti è stata utilizzata la matrice di confusione.



Questa immagine mostra un esempio di matrice binaria dove ogni colonna rappresenta i valori predetti, mentre ogni riga rappresenta i valori reali. L'elemento sulla riga i e sulla colonna j è il numero di casi in cui il classificatore

ha classificato la classe "vera" i come classe j. Attraverso questa matrice è osservabile se vi è "confusione" nella classificazione di diverse classi.

Tutti questi valori, (TP, FN, FP, TN) consentono di calcolare delle metriche di performance:

$$Precision = \frac{TP}{TP + FP}$$

La precisione è la capacità di un classificatore di non assegnare un'istanza positiva che è in realtà negativa. Per ogni classe è definito come il rapporto tra veri positivi e la somma di veri e falsi positivi. Detto in un altro modo, "per tutte le istanze classificate come positive, quale percentuale era corretta?".

$$Recall = \frac{TP}{TP + FN}$$

La recall è l'abilità di un classificatore di trovare tutte le istanze positive. Per ogni classe è definito come il rapporto tra i veri positivi e la somma dei veri positivi e dei falsi negativi. Detto in altri termini, "per tutte le istanze che erano effettivamente positive, quale percentuale è stata classificata correttamente?".

$$F - score = \frac{2\ x\ Recall\ x\ Precision}{Recall + Precision}$$

L'F-score è una media armonica ponderata delle metriche Precision e Recall in modo tale che il punteggio migliore sia 1 e il peggiore sia 0.

$$Accuracy = \frac{TP + TN}{TN + FP + FN + TP}$$

L'accuracy indica l'accuratezza del modello come dice il nome. Pertanto, la migliore accuratezza è 1, mentre la peggiore è 0.

Infine un altro valore che viene riportato nelle tabelle dei risultati è il Support, cioè il numero di campioni analizzati per ogni classe.

3.1.1 Analisi modello base

Partendo dalla matrice di confusione, vediamo i risultati ottenuti dal modello base che prende in input il testo in formato numerico.

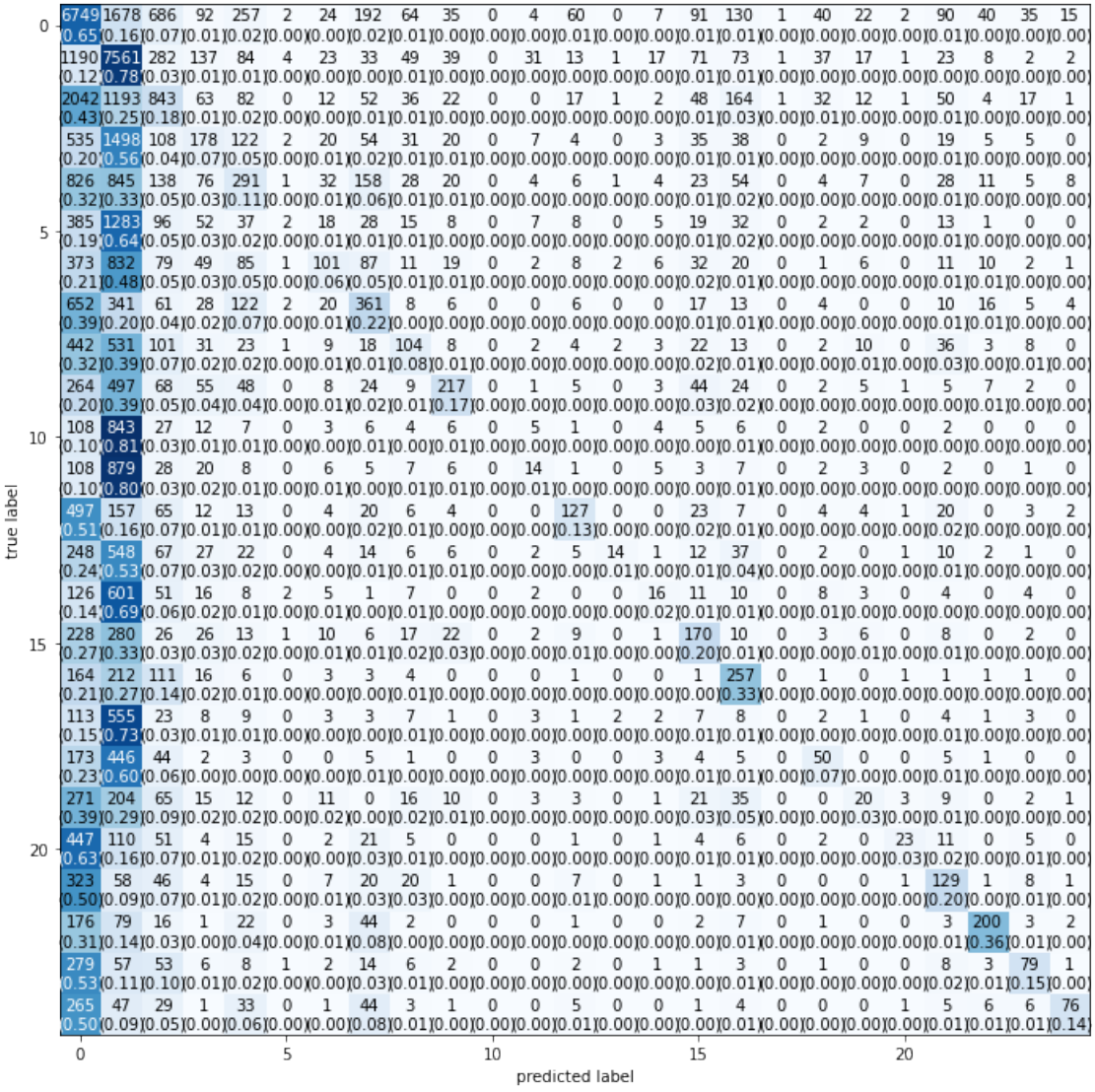


Figura 3.1: Matrice di confusione modello base

	precision	recall	f1-score	support
0	0.40	0.65	0.49	10316
1	0.35	0.78	0.49	9699
2	0.27	0.18	0.21	4695
3	0.19	0.07	0.10	2695
4	0.22	0.11	0.15	2570
5	0.11	0.00	0.00	2013
6	0.31	0.06	0.10	1738
7	0.30	0.22	0.25	1676
8	0.22	0.08	0.11	1373
9	0.48	0.17	0.25	1289
10	0.00	0.00	0.00	1041
11	0.15	0.01	0.02	1105
12	0.43	0.13	0.20	969
13	0.61	0.01	0.03	1029
14	0.19	0.02	0.03	875
15	0.25	0.20	0.23	840
16	0.27	0.33	0.29	783
17	0.00	0.00	0.00	756
18	0.25	0.07	0.11	745
19	0.16	0.03	0.05	702
20	0.64	0.03	0.06	708
21	0.25	0.20	0.22	646
22	0.62	0.36	0.45	562
23	0.40	0.15	0.22	527
24	0.67	0.14	0.24	528
accuracy			0.35	49880
macro avg	0.31	0.16	0.17	49880
eighted avg	0.31	0.35	0.28	49880

Figura 3.2: Metriche modello base

Confrontiamo tale risultato con quello ottenuto dal bilanciamento del dataset. Come possiamo vedere già dalla matrice di confusione abbiamo un'apparente miglioramento nella classificazione generale dei tweets, tutta via andando a confrontare le metriche dei due modelli si evince che abbiamo avuto si un guadagno nelle classi che erano precedentemente meno presenti, ma a costo di una grande perdita nelle classi che erano più presenti, infatti

l'accuracy del modello risulta essere peggiorata passando dal 35% al 25%.

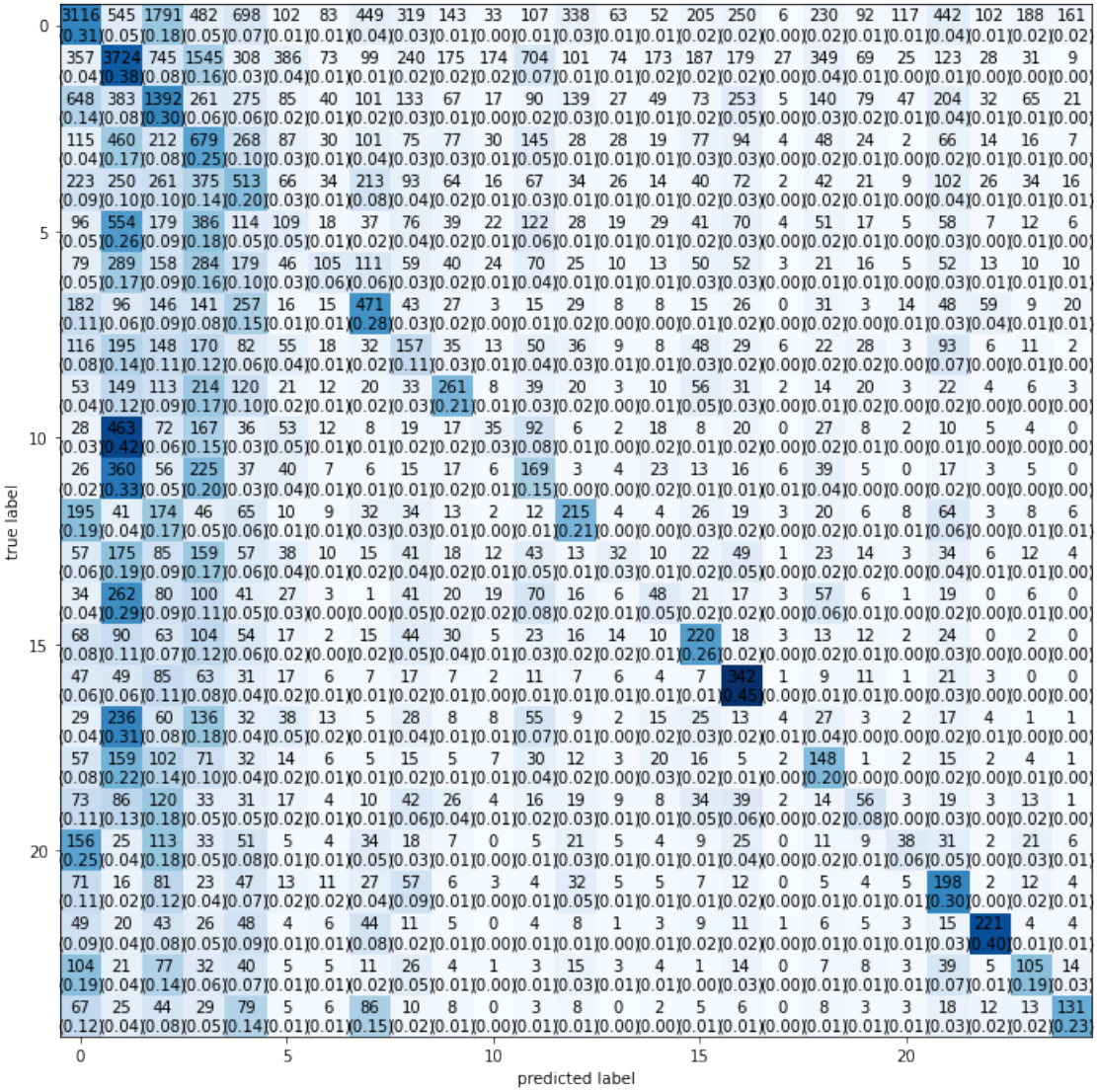


Figura 3.3: Matrice di confusione modello base dopo il bilanciamento

	precision	recall	f1-score	support
0	0.52	0.31	0.39	10114
1	0.43	0.38	0.40	9905
2	0.22	0.30	0.25	4626
3	0.12	0.25	0.16	2706
4	0.15	0.20	0.17	2613
5	0.09	0.05	0.06	2099
6	0.20	0.06	0.09	1724
7	0.24	0.28	0.26	1682
8	0.10	0.11	0.10	1372
9	0.23	0.21	0.22	1237
10	0.08	0.03	0.04	1112
11	0.09	0.15	0.11	1098
12	0.18	0.21	0.20	1019
13	0.09	0.03	0.05	933
14	0.09	0.05	0.07	898
15	0.18	0.26	0.21	849
16	0.21	0.45	0.28	754
17	0.05	0.01	0.01	771
18	0.11	0.20	0.14	734
19	0.11	0.08	0.09	682
20	0.12	0.06	0.08	633
21	0.11	0.30	0.16	650
22	0.39	0.40	0.40	551
23	0.18	0.19	0.18	547
24	0.31	0.23	0.26	571
accuracy			0.25	49880
macro avg	0.18	0.19	0.18	49880
weighted avg	0.28	0.25	0.26	49880

Figura 3.4: Metriche modello base dopo il bilanciamento

Passiamo dunque a eseguire un confronto con il modello al quale sono state aggiunte le features.

Difatti già dalla matrice di confusione si vede un netto miglioramento, anche se per alcune delle classi aventi un numero minore di tweet il classificatore ancora sbaglia facilmente. Dalle metriche possiamo dire che si ha un miglioramento delle prestazioni su ogni fronte.

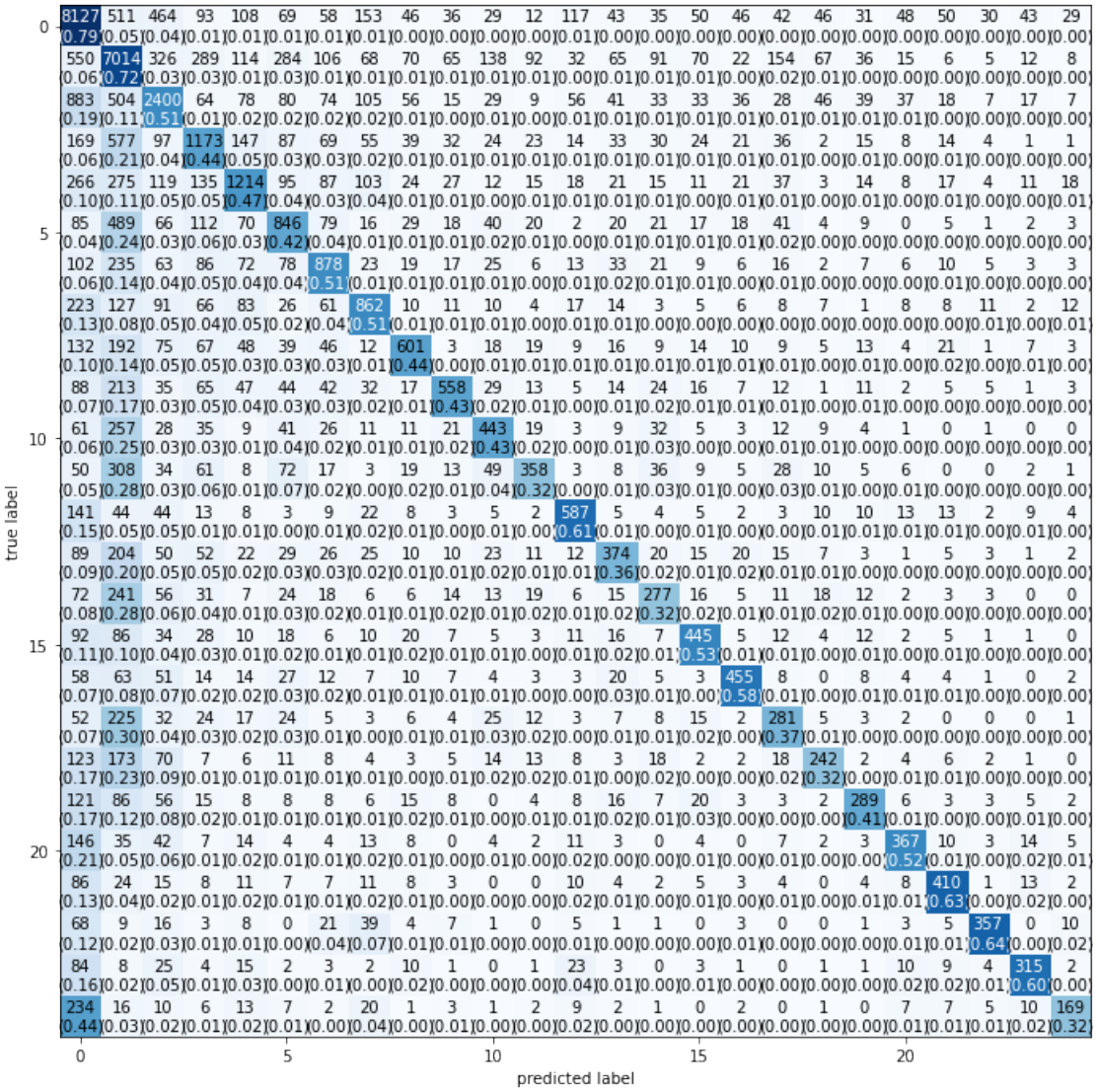


Figura 3.5: Matrice di confusione modello base con features

	precision	recall	f1-score	support
0	0.67	0.79	0.73	10316
1	0.59	0.72	0.65	9699
2	0.56	0.51	0.53	4695
3	0.48	0.44	0.46	2695
4	0.56	0.47	0.51	2570
5	0.44	0.42	0.43	2013
6	0.53	0.51	0.51	1738
7	0.54	0.51	0.52	1676
8	0.57	0.44	0.50	1373
9	0.63	0.43	0.51	1289
10	0.47	0.43	0.45	1041
11	0.54	0.32	0.41	1105
12	0.60	0.61	0.60	969
13	0.48	0.36	0.41	1029
14	0.40	0.32	0.35	875
15	0.56	0.53	0.54	840
16	0.65	0.58	0.61	783
17	0.36	0.37	0.36	756
18	0.49	0.32	0.39	745
19	0.54	0.41	0.47	702
20	0.64	0.52	0.57	708
21	0.65	0.63	0.64	646
22	0.78	0.64	0.70	562
23	0.67	0.60	0.63	527
24	0.59	0.32	0.41	528
accuracy			0.58	49880
macro avg	0.56	0.49	0.52	49880
weighted avg	0.58	0.58	0.57	49880

Figura 3.6: Metriche modello base con features

3.1.2 Analisi modello BiLSTM

Infine vediamo i risultati dei due modelli che fanno uso della BiLSTM, effettuando un confronto con i precedenti.

Per quanto riguarda questo primo modello, la matrice di confusione ci permette di affermare che abbiamo un'ulteriore miglioramento sulle classi aventi un numero minore di sample, tuttavia come ci confermano le metriche si ha una lieve perdità sulle classi aventi un numero più alto di sample. In linea di massima in questo caso l'accuratezza ha un peggioramento rispetto

al modello che utilizza tutti livelli fully connected, passando da un 58% ad un 55%.

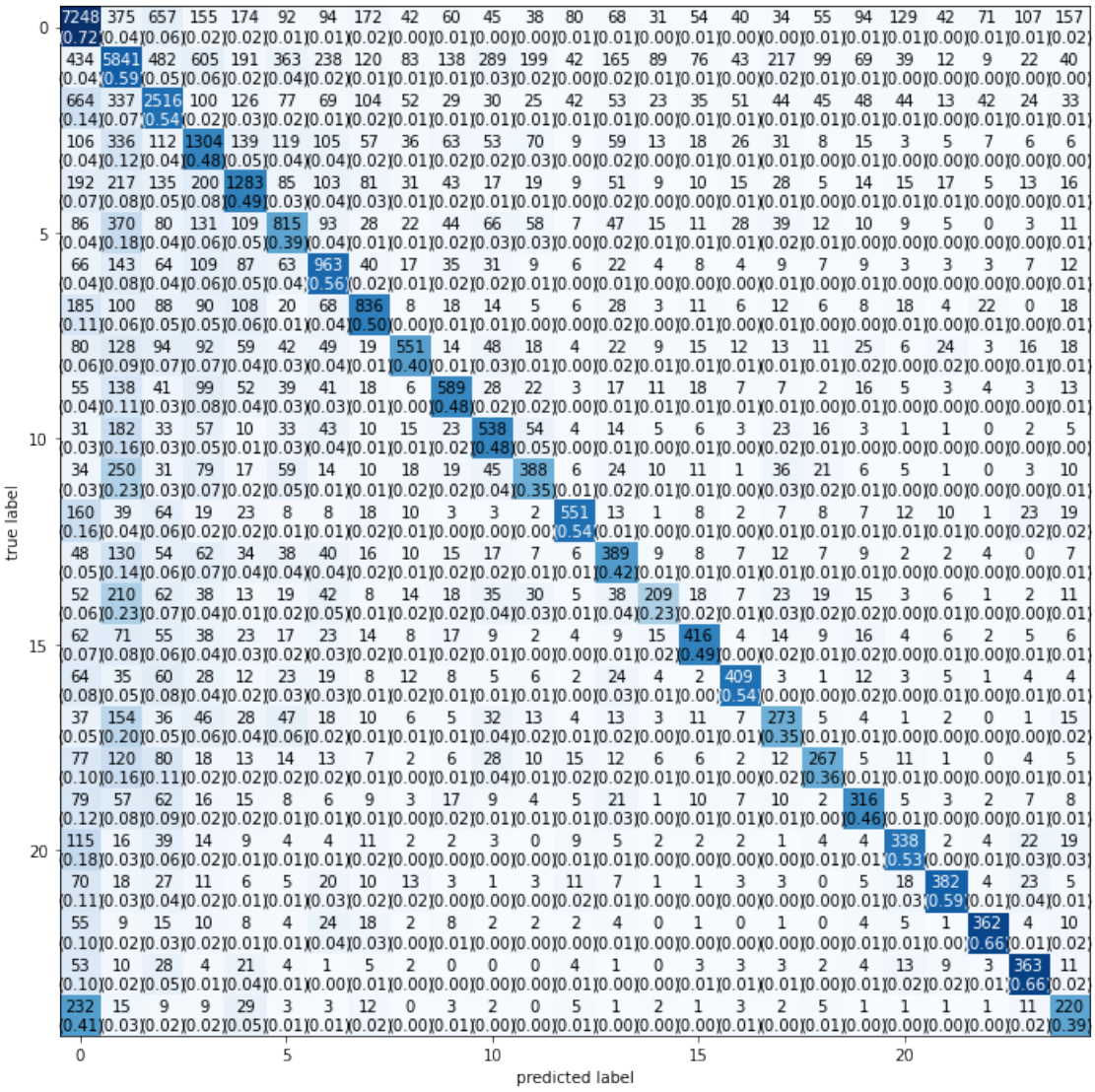


Figura 3.7: Matrice di confusione Modello BiLSTM

	precision	recall	f1-score	support
0	0.70	0.72	0.71	10114
1	0.63	0.59	0.61	9905
2	0.51	0.54	0.53	4626
3	0.39	0.48	0.43	2706
4	0.50	0.49	0.49	2613
5	0.41	0.39	0.40	2099
6	0.46	0.56	0.50	1724
7	0.51	0.50	0.50	1682
8	0.57	0.40	0.47	1372
9	0.50	0.48	0.49	1237
10	0.40	0.48	0.44	1112
11	0.39	0.35	0.37	1098
12	0.66	0.54	0.59	1019
13	0.35	0.42	0.38	933
14	0.44	0.23	0.30	898
15	0.55	0.49	0.52	849
16	0.59	0.54	0.57	754
17	0.32	0.35	0.34	771
18	0.43	0.36	0.40	734
19	0.44	0.46	0.45	682
20	0.49	0.53	0.51	633
21	0.68	0.59	0.63	650
22	0.66	0.66	0.66	551
23	0.54	0.66	0.59	547
24	0.32	0.39	0.35	571
accuracy			0.55	49880
macro avg	0.50	0.49	0.49	49880
weighted avg	0.55	0.55	0.55	49880

Figura 3.8: Metriche modello BiLSTM

Per quanto riguarda l'ultimo modello che sfrutta la BiLSTM, possiamo vedere che la matrice di confusione risulta essere simile alla precedente ma confrontando le metriche viene evidenziato un miglioramento per tutte classi su ogni fronte, che porta ad avere un'accuratezza finale pari al 62%.

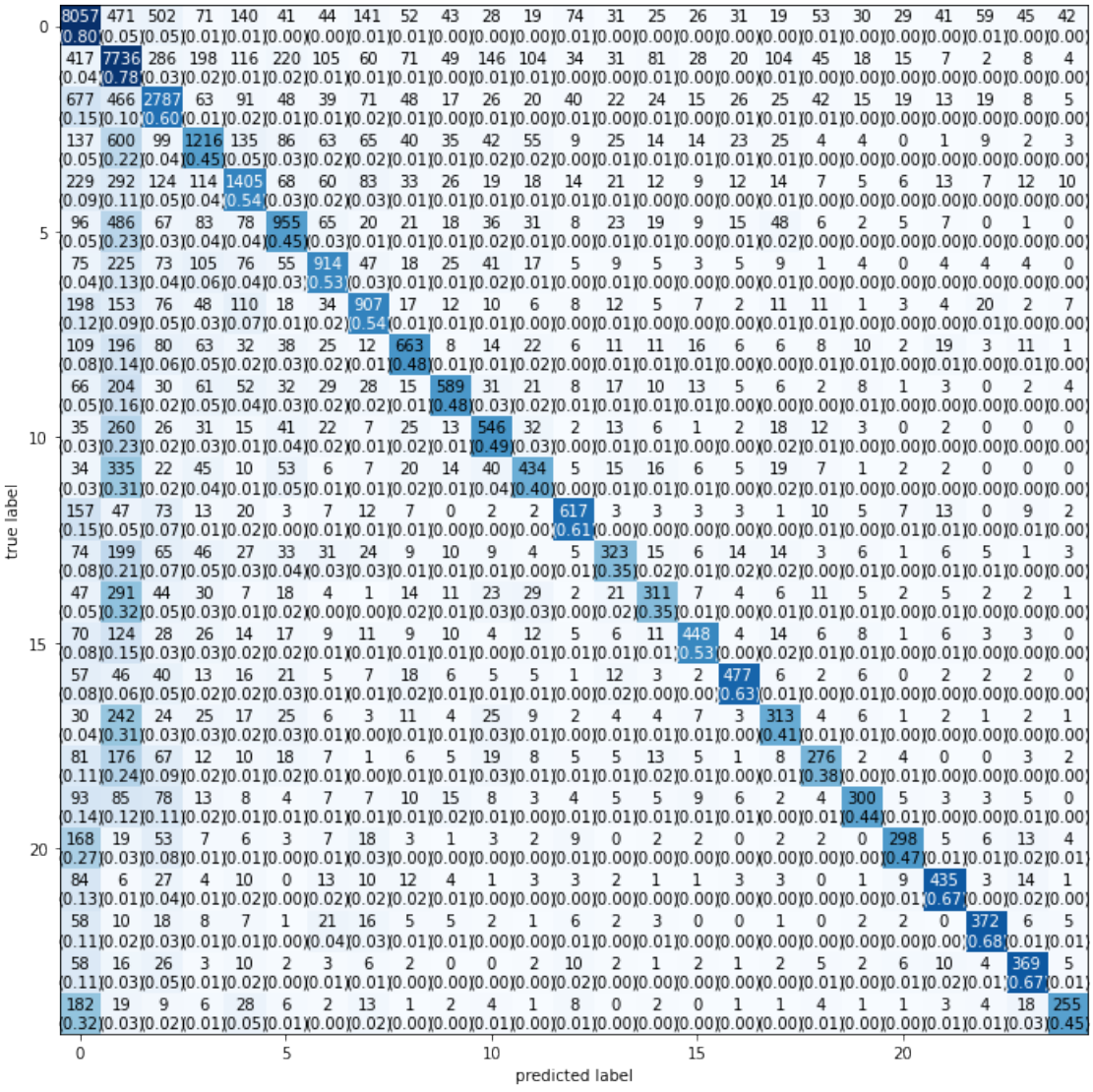


Figura 3.9: Matrice di confusione Modello BiLSTM finale

	precision	recall	f1-score	support
0	0.71	0.80	0.75	10114
1	0.61	0.78	0.68	9905
2	0.59	0.60	0.60	4626
3	0.53	0.45	0.49	2706
4	0.58	0.54	0.56	2613
5	0.53	0.45	0.49	2099
6	0.60	0.53	0.56	1724
7	0.58	0.54	0.56	1682
8	0.59	0.48	0.53	1372
9	0.64	0.48	0.55	1237
10	0.50	0.49	0.50	1112
11	0.50	0.40	0.44	1098
12	0.69	0.61	0.65	1019
13	0.53	0.35	0.42	933
14	0.52	0.35	0.41	898
15	0.70	0.53	0.60	849
16	0.71	0.63	0.67	754
17	0.46	0.41	0.43	771
18	0.53	0.38	0.44	734
19	0.67	0.44	0.53	682
20	0.71	0.47	0.57	633
21	0.72	0.67	0.69	650
22	0.70	0.68	0.69	551
23	0.68	0.67	0.68	547
24	0.72	0.45	0.55	571
accuracy			0.62	49880
macro avg	0.61	0.53	0.56	49880
weighted avg	0.62	0.62	0.61	49880

Figura 3.10: Metriche modello BiLSTM finale

Conclusioni

L'obbiettivo di questo progetto è stato quello di riuscire a carpire e identificare i contenuti delle frasi e dei tweets in modo tale da creare un modello in grado di classificare automaticamente tali tweet scritti dagli utenti rispetto una delle 25 emoji.

Un primo approccio al problema è stato quello di eseguire il bilanciamento del dataset tramite l'up-sampling e il down-sampling dei tweets, questo però non ha portato ad un miglioramento vero e proprio del task.

Non tutti i modelli prodotti hanno mostrato delle buone prestazioni, tuttavia una svolta al problema è stata l'aggiunta delle features che riguardavano l'history emoji di ogni singolo utente. Questo ha permesso il miglioramento delle prestazioni poichè il task presentava diverse varianti per la stessa tipologia di emoji, rendendo quindi impossibile anche per una persona la risoluzione del quesito. Inoltre l'aggiunta dei livelli BiLSTM ha permesso di incrementare leggermente le performance in modo da raggiungere buone prestazioni.

Possiamo dunque affermare che l'obiettivo del progetto è stato raggiunto.