

Question Answer Matching

Introduzione

L'obiettivo di questo progetto è eseguire il fine-tuning di due modelli pre-allenati sfruttando un dataset di Hugging Face. Il dataset scelto è composto da due features:

- "text": è di tipo stringa. Un record è composto da una domanda e da un'affermazione
- "label": è di tipo integer. Assume il valore 0 se l'affermazione non fornisce informazioni pertinenti e sufficienti alla domanda posta, mentre assume il valore 1 in caso contrario.

Esempio:

Sentence: The cat lived with Jeremy, along with his orange and black cat. Question: Where did Jeremy find the cat?"	0
"Sentence: Two of her friends were going to the nearby beach to do some swimming and enjoy the sun. Question: Where did Jenny and her friends lay down on the towels to enjoy the sun?"	1

Lo scopo del fine-tuning da effettuare è dunque quello di fare in modo che il modello allenato riesca a determinare se l'asserzione fornisce informazioni inerenti alla domanda a cui è collegata.

Lo sviluppo di questo task potrebbe rivelarsi utile nell'ambito dell'NLP e trovare applicazione in diversi ambiti, ad esempio nel campo dello studio delle risposte automatiche o della verifica della coerenza domanda/risposta.

Il dataset è consultabile al link: <https://huggingface.co/datasets/mattymchen/natural-instruction-050>, mentre il codice da cui sono stati estratti gli script per questa relazione è consultabile al link: https://colab.research.google.com/drive/1jK_RIA2XvyLFogFubG_ruNWHOajeLO6?usp=sharing

1. Data exploration

dataset

```
DatasetDict({  
  test: Dataset({  
    features: ['text', 'label'],  
    num_rows: 5912  
  })  
})
```

Dopo aver installato le librerie necessarie al caricamento del dataset, ne è stata rappresentata la struttura e la composizione

```
print("Dimensioni del dataset:", df.shape)
print("\nInformazioni:", df.info())
print("\nStatistiche:", df.describe())
```

```
Dimensioni del dataset: (5912, 2)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5912 entries, 0 to 5911
Data columns (total 2 columns):
#   Column  Non-Null Count  Dtype
---  -
0    text    5912 non-null     object
1    label    5912 non-null     int64
dtypes: int64(1), object(1)
memory usage: 92.5+ KB
```

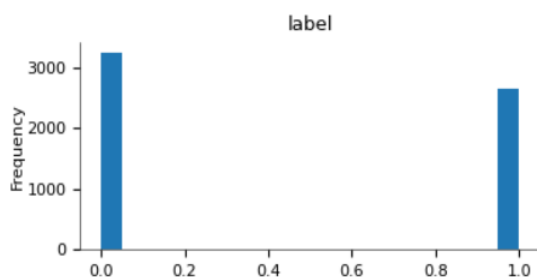
Informazioni: None

```
Statistiche:          label
count  5912.000000
mean    0.450271
std     0.497563
min     0.000000
25%     0.000000
50%     0.000000
75%     1.000000
max     1.000000
```

```
class_counts = df['label'].value_counts()
print("Distribuzione label:", (class_counts))
```

```
Distribuzione label: 0    3250
1    2662
Name: label, dtype: int64
```

Distributions



Grazie all'utilizzo di *pandas*, è stata effettuata un'analisi più approfondita del dataset: esso risulta composto da 5912 record per la prima colonna mentre i valori assunti dalla seconda colonna possono essere 2. Sono stati descritti anche alcuni dati statistici.

La distribuzione delle label del dataset appare abbastanza bilanciata, con 3250 record per la label "0" e 2662 per la label "1".

```
null_value = df.isnull().sum()
null_value
```

```
text          0
label         0
lunghezza_testo  0
dtype: int64
```

Non sono presenti valori null/None nel dataset.

2. Split Dataset

Il dataset è stato suddiviso: è stato costituito un training set da utilizzare per allenare il modello, un validation set per validare i risultati ottenuti sul training set e un test set per verificare il funzionamento del modello.

```
train_test_datasets = dataset['test'].train_test_split(0.2)
```

La suddivisione è stata così effettuata:

```
val_test_datasets = train_test_datasets['test'].train_test_split(0.4)
```

-training set: 80% del dataset

```
raw_datasets
```

-validation set: 12% del dataset

```
DatasetDict({
  train: Dataset({
    features: ['text', 'label'],
    num_rows: 4729
  })
  val: Dataset({
    features: ['text', 'label'],
    num_rows: 709
  })
  test: Dataset({
    features: ['text', 'label'],
    num_rows: 474
  })
})
```

-test set: 8% del dataset

Dunque, la suddivisione ha portato alla composizione del dataset in:

- Train set: 4729 record

- Val set: 709 record

- Test set: 474 record

Prima di passare alla fase di processing dei dati, è possibile effettuare ulteriori controlli per verificare che le classi di label siano effettivamente bilanciate in tutti i set ottenuti e che non siano presenti valori None.

```
import numpy as np
```

```
train_labels = np.array(raw_datasets['train']['label'])
val_labels = np.array(raw_datasets['val']['label'])
test_labels = np.array(raw_datasets['test']['label'])
```

```
print(np.unique(train_labels, return_counts = True))
print(np.unique(val_labels, return_counts = True))
print(np.unique(test_labels, return_counts = True))
```

```
(array([0, 1]), array([2591, 2138]))
(array([0, 1]), array([405, 304]))
(array([0, 1]), array([254, 220]))
```

```
def isNotNone(example):
    if example['text']!=None and example['label']!=None:
        return True
    else:
        return False
```

```
raw_datasets = raw_datasets.filter(isNotNone)
```

Tramite la libreria *NumPy*, utilizzata in questo caso per la manipolazione di array, viene esaminata la distribuzione delle label tramite il conteggio dei valori unici. Il risultato conferma ciò che è emerso dalla [data exploration effettuata con pandas](#).

Con questa funzione, abbiamo verificato che non sono presenti valori None all'interno delle features Text e Label.

Filter: 100%  4729/4729 [00:00<00:00, 15994.45 examples/s]

Filter: 100%  709/709 [00:00<00:00, 11511.80 examples/s]

Filter: 100%  474/474 [00:00<00:00, 8618.06 examples/s]

3. Data Processing

Per effettuare il finetuning verrà utilizzato il modello prealllenato “*bert-base-uncased*”: la scelta deriva dal fatto che per il task da portare a termine è sufficiente avere a disposizione un modello di tipo encoder, e *bert* è il migliore tra questi: esso, infatti, è stato allenato su un vasto corpus di testi e questo lo rende adatto a molti compiti di NLP, come la classificazione.

```
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

```
def tokenize_function(example):
    return tokenizer(example["text"], truncation=True)
```

```
tokenized_datasets = raw_datasets.map(tokenize_function, batched=True, remove_columns=(['text']))
```

Map: 100%  4729/4729 [00:00<00:00, 5813.32 examples/s]

Map: 100%  709/709 [00:00<00:00, 5677.64 examples/s]

Map: 100%  474/474 [00:00<00:00, 2991.63 examples/s]

```
tokenized_datasets
```

```
DatasetDict({
  train: Dataset({
    features: ['label', 'input_ids', 'token_type_ids', 'attention_mask'],
    num_rows: 4729
  })
  val: Dataset({
    features: ['label', 'input_ids', 'token_type_ids', 'attention_mask'],
    num_rows: 709
  })
  test: Dataset({
    features: ['label', 'input_ids', 'token_type_ids', 'attention_mask'],
    num_rows: 474
  })
})
```

E' stata definita una funzione per tokenizzare il *raw_datasets*

Il risultato della tokenizzazione è stato istanziato nella variabile *tokenized_datasets*, in cui compaiono gli *input_ids*, i *token_type_ids* e gli *attention_mask* utili per l'allenamento del modello.

4. Fine-tuning

Per eseguire il fine-tuning sul dataset, è stato importato l'*AutoModelForSequenceClassification*. Le label sono state impostate a 2, dato che nel dataset possono assumere soltanto due valori.

```
from transformers import AutoModelForSequenceClassification
```

```
model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
```

Downloading model.safetensors: 100%  440M/440M [00:04<00:00, 133MB/s]

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are ne You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

```

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)

def compute_metrics(eval_preds):
    metric = evaluate.load("accuracy")
    logits, labels = eval_preds
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)

from transformers import Trainer, TrainingArguments
import evaluate
import numpy as np

training_args = TrainingArguments(
    "test-trainer",
    evaluation_strategy="steps",
    eval_steps = 500,
    num_train_epochs = 10,
    logging_strategy = "steps",
    logging_steps=500,
    report_to = "wandb"
)


device = 'cuda'
trainer = Trainer(
    model.to(device),
    training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["val"],
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)

```

E' stata definita una funzione *compute_metrics* per calcolare l'accuratezza del modello durante la fase di valutazione; essa prende in input le previsioni (*logits*) del modello e le label corrispondenti; con la funzione *argmax()* si estrae l'indice con il valore massimo lungo l'asse delle label e infine si calcola l'accuratezza confrontando le previsioni con le label di riferimento.

Vengono poi definiti i *TrainingArguments* che saranno poi passati al *Trainer* istanziato.

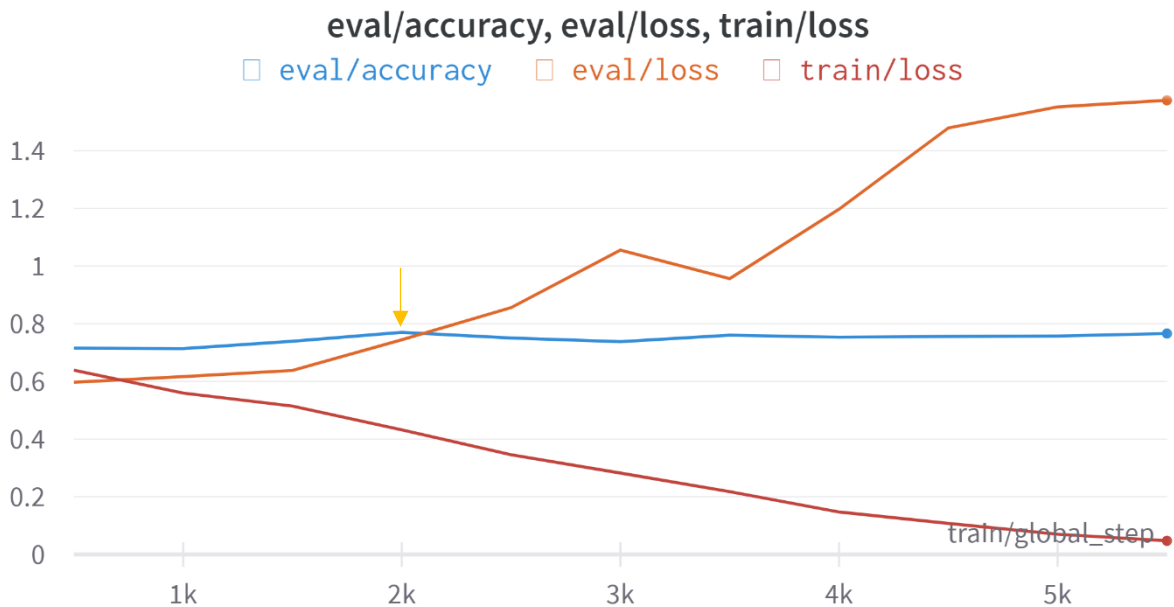
Di seguito, i risultati dell'allenamento effettuato:

You're using a BertTokenizerFast tokenizer. Please note that with a fast tokenizer,
 [5920/5920 12:42, Epoch 10/10]

Step	Training Loss	Validation Loss	Accuracy
500	0.639100	0.597006	0.715092
1000	0.559400	0.616663	0.713681
1500	0.514500	0.637924	0.739069
2000	0.431800	0.744456	0.770099
2500	0.345800	0.855726	0.750353
3000	0.282100	1.055199	0.737659
3500	0.217600	0.956433	0.760226
4000	0.147200	1.196957	0.753173
4500	0.107500	1.478819	0.755994
5000	0.070100	1.551827	0.757405
5500	0.047200	1.574681	0.765867

La prima colonna indica gli step (i passi) compiuti durante l'addestramento del modello. La seconda la Training Loss che indica la discrepanza tra le previsioni del modello e le label del dataset, mentre la Validation Loss viene misurata sul set di dati ('val') non visionato durante l'allenamento, per comprendere come il modello performa con dati precedentemente non presi in esame.

Di seguito il grafico dell'allenamento:



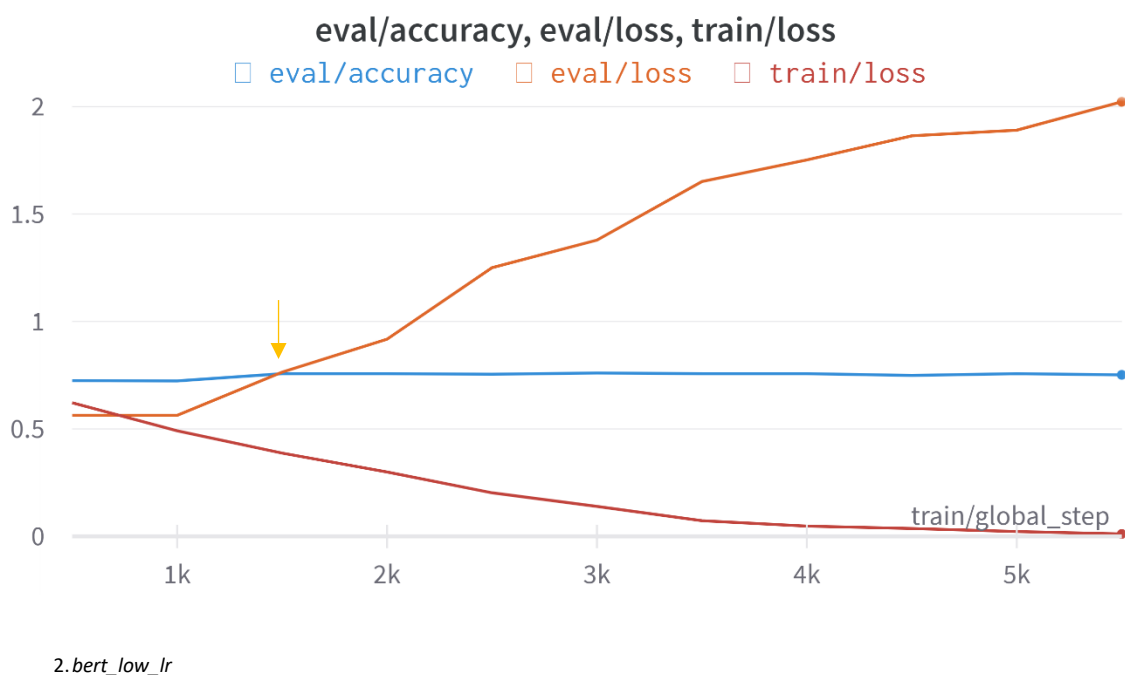
1.bert_def_lr

In giallo, viene segnato il punto in cui l'allenamento sembra ottenere i risultati migliori. Allo step 2000, l'accuracy tocca il picco del 77% e lo scarto tra la training loss e la validation loss è ancora relativamente basso (continuerà ad aumentare con gli step successivi), quindi in questa fase l'apprendimento del modello può essere considerato più robusto.

Per provare a migliorare le prestazioni, è stato effettuato un altro allenamento cambiando l'iperparametro del *learning rate*, che è stato impostato a $2e-5$, allo scopo di permettere al modello di effettuare un allenamento più accurato:

[5920/5920 12:36, Epoch 10/10]

Step	Training Loss	Validation Loss	Accuracy
500	0.621900	0.563347	0.724965
1000	0.491400	0.563424	0.723554
1500	0.387600	0.764233	0.757405
2000	0.299700	0.917772	0.757405
2500	0.202700	1.250437	0.754584
3000	0.139200	1.379079	0.760226
3500	0.072800	1.651678	0.757405
4000	0.047400	1.752236	0.757405
4500	0.036400	1.864545	0.748942
5000	0.022500	1.890869	0.757405
5500	0.011200	2.022990	0.751763



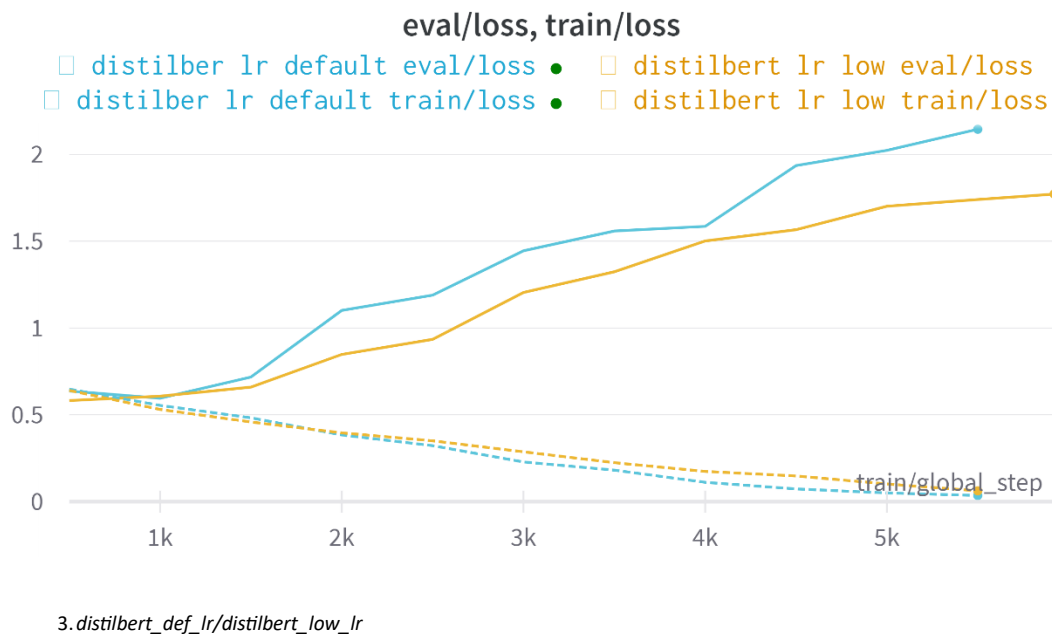
Con questo secondo allenamento non è stato raggiunto nessun risultato significativo rispetto all'allenamento precedente, anzi è stato registrato un lieve calo dell'accuracy. Le prestazioni migliori sono state ottenute allo step 1500, in cui l'accuracy è del 76%: anche se successivamente è stata ottenuta un'accuracy dello stesso valore o poco maggiore, è in questo step che si registra uno scarto minore tra Validation Loss e Training Loss. Il fatto che l'abbassamento del *learning rate* non ha portato a risultati soddisfacenti potrebbe essere determinato dalla configurazione del modello o dall'influenza di altri iperparametri.

Per effettuare un'ulteriore prova, è stato effettuato il fine-tuning con un altro modello, il **distilbert-base-uncased**. E' stato scelto questo modello perché costituisce una versione più compatta di BERT e richiede risorse computazionali minori, pur mantenendo una buona efficienza. Sono stati effettuati due allenamenti riproducendo gli stessi iperparametri utilizzati per il modello *bert*.

A sinistra, l'allenamento effettuato con il *learning rate* di default; a destra, con il *learning rate* a $2e-5$:

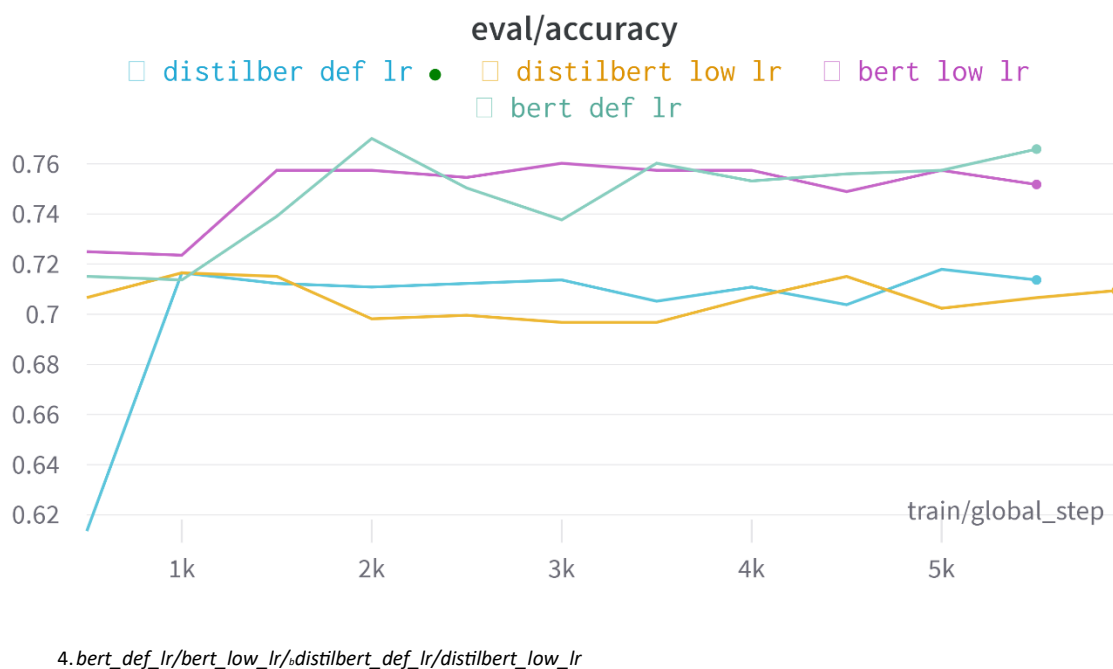
[5920/5920 07:13, Epoch 10/10]				[5920/5920 06:59, Epoch 10/10]			
Step	Training Loss	Validation Loss	Accuracy	Step	Training Loss	Validation Loss	Accuracy
500	0.648000	0.637399	0.613540	500	0.640500	0.582041	0.706629
1000	0.554400	0.595653	0.716502	1000	0.531000	0.607019	0.716502
1500	0.482800	0.717466	0.712271	1500	0.458600	0.659321	0.715092
2000	0.383900	1.101370	0.710860	2000	0.396100	0.847823	0.698166
2500	0.322300	1.189292	0.712271	2500	0.350300	0.934655	0.699577
3000	0.228300	1.444509	0.713681	3000	0.286800	1.204765	0.696756
3500	0.181300	1.558530	0.705219	3500	0.224300	1.323844	0.696756
4000	0.111200	1.585336	0.710860	4000	0.173900	1.501137	0.706629
4500	0.073100	1.935574	0.703808	4500	0.147800	1.566200	0.715092
5000	0.049900	2.023398	0.717913	5000	0.101700	1.701193	0.702398
5500	0.035200	2.144830	0.713681	5500	0.062000	1.739903	0.706629

Con il modello *distilbert* si ottengono prestazioni più basse rispetto al finetuning effettuato sul modello *bert*. Infatti, il livello di accuracy più alta che si raggiunge è del 72% e si tocca già allo step 1000: a questo step, entrambi gli allenamenti mostrano uno scarto molto basso tra Training Loss e Validation Loss. Questo scarto aumenterà gradualmente in entrambi gli allenamenti ma nel train effettuato con il learning rate di default sarà soggetto a una crescita maggiore, come mostra il grafico:



Questo potrebbe significare che il modello con il learning rate di $5e-5$ potrebbe essere maggiormente soggetto all'overfitting.

A scopo esemplificativo e conclusivo, nel grafico che segue è possibile visualizzare l'andamento dell'accuracy dei modelli fin qui allenati:



Possiamo concludere che, tenendo in considerazione le valutazioni precedentemente effettuate riguardo i rapporti tra Train Loss e Validation Loss, il modello più performante sul task oggetto della relazione è il *bert_base_uncased*, con un accuracy del 77%.

Per visualizzare alcune previsioni sul set di validazione, possiamo estrarre un esempio come verifica:

```
predictions = trainer.predict(tokenized_datasets["val"])
```

Nel caso che segue, la label predetta dal modello è uguale alla label di riferimento del dataset.

```
print(predictions.predictions.shape, predictions.label_ids.shape)
```

```
(709, 2) (709,)
```

```
import random
i = random.randint(0, 709)
example = tokenized_datasets['val'][i]['input_ids']
example_label = tokenized_datasets['val'][i]['label']
```

```
tokenizer.decode(example)
```

```
'[CLS] sentence : the dragons went to the river and started eating berries, and fruits. question : why did the men go to the village? [SEP]'
```

```
example_label, predictions.predictions[i].argmax()
```

```
(0, 0)
```

Infine, possono essere valutate le prestazioni del modello sul test set

```
trainer.evaluate(eval_dataset=tokenized_datasets['test'])
```

```
[60/60 00:01]
{'eval_loss': 2.138193130493164,
 'eval_accuracy': 0.7531645569620253,
 'eval_runtime': 1.8192,
 'eval_samples_per_second': 260.553,
 'eval_steps_per_second': 32.981,
 'epoch': 10.0}
#
```

Il modello finale allenato è stato infine caricato su hugging face (<https://huggingface.co/AgneseSpinella/Question-Answer-Matching>) e i report riguardanti gli allenamenti sono reperibili su <https://wandb.ai/site>.

Considerazioni finali

Lo scopo del finetuning effettuato sui modelli presi in considerazione era finalizzato a verificare quanto il modello potesse essere in grado di verificare la pertinenza di risposte a domande

specifiche. Il task è stato affrontato attraverso quattro addestramenti: il primo è stato quello più proficuo, raggiungendo un'accuracy del 77%.

In generale, i risultati ottenuti mettono in luce una mancanza di robustezza e solidità da parte dei modelli allenati: infatti, non si registra mai un calo della validation loss e tutti i modelli appaiono in overfitting.

Possibili sviluppi futuri di questo progetto potrebbero riguardare un'ulteriore sperimentazione per ottimizzare le prestazioni del modello, magari attraverso l'esplorazione di altre configurazioni iperparametriche; ad esempio, potrebbe essere valutata l'aggiunta di un drop out che eviti l'overfitting. Sarebbe interessante anche provare a utilizzare altri tipi di modello, o effettuare l'allenamento su dataset più complessi.