Relazione Settimanale: Settimane 4 e 5

Tirocinio presso Prof. Ferretti

26 aprile 2025

Sommario

In questo documento viene presentata in modo dettagliato e approfondito l'attività svolta durante le settimane 4 e 5 del tirocinio, con focus sull'espansione del dataset, sulle tecniche di traduzione e retichettatura, sull'implementazione di euristiche per l'identificazione di pattern jailbreak e sui risultati ottenuti dal fine tuning di un modello BERT per la classificazione binaria delle risposte generate da LLM.

Indice

1 Contesto e Motivazioni

L'uso crescente di modelli di linguaggio (LLM) in applicazioni sensibili richiede la capacità di distinguere tra risposte lecite e contenuti potenzialmente pericolosi ("jailbreak"). Il dataset iniziale fornito dal Prof. Ferretti conteneva risposte etichettate automaticamente, ma presentava problemi di copertura linguistica e squilibri di classe. L'obiettivo principale di queste settimane è stato quindi:

- Ampliare il dataset per migliorarne la rappresentatività;
- Correggere etichettature errate in risposte non in lingua inglese;
- Applicare euristiche avanzate per identificare pattern testuali associati a contenuti di tipo jailbreak;
- Bilanciare le classi e costruire un set di test robusto;
- Eseguire e analizzare il fine tuning di un modello BERT per la classificazione finale.

Queste attività costituiscono la base per garantire affidabilità nella rilevazione automatica di richieste pericolose da parte di modelli di IA.

2 Espansione e Pulizia del Dataset

2.1 Analisi Iniziale del Dataset

Il dataset originale era costituito da tre file JSON contenenti risposte classificate in due categorie: jailbreak (etichette "1") e non-jailbreak (etichette "0"). Un'analisi preliminare ha evidenziato:

- **Rappresentanza linguistica limitata**: molte risposte erano in italiano, francese, spagnolo e altre lingue, non comprese dal processo di etichettatura originale;
- **Squilibrio di classe**: la classe non-jailbreak risultava prevalente, aumentando il rischio di bias nel training.

Per garantire un dataset più omogeneo e bilanciato, si è deciso di ripartire da zero dal file sorgente e di rieseguire l'intera pipeline di traduzione e retichettatura.

2.2 Processo Iterativo di Traduzione

La traduzione multipla è stata effettuata in quattro fasi:

1. Traduzione automatica da lingua origine a inglese tramite API esterne (Google Translate, DeepL);
2. Verifica di consistenza semantica: confronto tra testo originale e traduzione per rilevare possibili errori di interpretazione;
3. Traduzione inversa (back-translation) per valutare la stabilità del risultato;
4. Revisione manuale dei casi borderline, in cui la traduzione non rifletteva fedelmente il contenuto.

Questo ciclo ha permesso di correggere oltre 500 risposte che erano state originariamente etichettate come non-jailbreak solo perché non in inglese.

3 Euristiche di Pattern Detection

3.1 Osservazioni Qualitative

Analizzando un campione estratto casualmente, abbiamo riscontrato che le risposte di tipo jailbreak spesso presentavano:

- Elenchi puntati dettagliati (indicazione di passaggi sequenziali);
- Frasi con costrutti di istruzione diretta ("come costruire", "passo per passo");
- Uso di terminologia specifica (es.: "TNT", "detonatore").

Al contrario, le risposte non-jailbreak includevano frequentemente espressioni di rifiuto o disclaimer ("mi dispiace, non posso assistere...").

3.2 Implementazione Python dell'Euristica

L'euristica combina tre moduli principali:

1. Rilevamento di parole chiave sospette (lista `suspicious_words`);
2. Rilevamento di pattern testuali tramite espressioni regolari (lista `jailbreak_patterns`);
3. Identificazione di frasi di sicurezza che indicano rifiuto (lista `safety_phrases`).

Il codice completo è riportato di seguito e integra una logica conservativa: in caso di dubbio, l'istanza è etichettata come non-jailbreak.

```
import json
import re

# (Definizione dettagliata delle liste: suspicious_words,
   safety_phrases, jailbreak_patterns)
# Funzioni:
# - contains_strong_safety_phrases
# - is_safety_response
# - contains_suspicious_pattern
# - contains_suspicious_words
# - check_if_jailbreak

def check_if_jailbreak(text):
    """
    Restituisce True se il testo    probabilmente jailbreak, False
       altrimenti.
    Utilizza:
      1) conteggio di frasi di rifiuto,
      2) pattern sospetti,
      3) parole chiave.
    """
    # Implementazione dettagliata...
    return is_likely_jailbreak, matched_words

# Caricamento file JSON di partenza
```

```
with open('POTENTIAL_JAILBREA_TRANSLATE.json', 'r', encoding='utf-8
    ') as f:
    data = json.load(f)

labeled_data = []
jailbreak_count = non_jailbreak_count = 0
for item in data:
    is_jailbreak, matched_words = check_if_jailbreak(item['response
        '])
    label = '1' if is_jailbreak else '0'
    labeled_data.append({'response': item['response'], 'label':
        label})
    if is_jailbreak: jailbreak_count += 1
    else: non_jailbreak_count += 1

# Salvataggio output finale
with open('JAILBREAK_LABELED.json', 'w', encoding='utf-8') as f:
    json.dump(labeled_data, f, ensure_ascii=False, indent=2)
print(f"Totale: {len(labeled_data)}, Jailbreak: {jailbreak_count},
    Non-jailbreak: {non_jailbreak_count}")
```

Listing 1: Script Python per la retichettatura dettagliata

Dopo l'esecuzione, il dataset risultante è bilanciato con 12 000 risposte, equamente ripartite tra le due classi.

4 Creazione del Set di Test

Per il test finale, sono state selezionate 1 800 risposte nuove, non presenti nel training set, mantenendo l'equilibrio di classe:

- 900 risposte jailbreak;
- 900 risposte non-jailbreak.

Questa separazione assicura che il modello venga valutato su esempi realmente inediti, riducendo l'overfitting.

5 Fine Tuning del Modello BERT

5.1 Architettura e Setup

Abbiamo scelto il modello `bert-base-uncased` per le sue dimensioni contenute ma adeguate a task di classificazione testo. Il training è stato effettuato su Google Colab con GPU, in modalità mixed precision (FP16) per ottimizzare tempi e utilizzo della memoria.

5.2 Parametri di Training

- **Corpi di addestramento:** 3 epoche
- **Learning rate:** $2e^{-5}$ (scheduler lineare con warmup di 500 step)

- **Batch size:** 16 esempi per GPU
- **Gradient accumulation:** 2 step per aggiornamento
- **Weight decay:** 0.01
- **Valutazione:** ogni 100 step, con validazione su 2 400 esempi

5.3 Andamento della Loss e Convergenza

I valori di loss registrati evidenziano una riduzione costante:

Step	**Loss Train**	**Loss Val** height100
0.6206	0.5892	
300	0.2167	0.2345
700	0.0881	0.1123
900	0.0752	0.1010

Tabella 1: Andamento della loss durante il training

5.4 Valutazione Finale

La valutazione sui 1 773 esempi del test set ha prodotto:

- **Loss:** 0.5010
- **Accuracy:** 90.02%
- **Precision:** 90.34%
- **Recall:** 90.02%
- **F1-score:** 89.92%

Questi risultati testimoniano un'eccellente capacità discriminante del modello, con metriche bilanciate e alto grado di generalizzazione.

6 Conclusioni e Prossimi Passi

Il lavoro svolto ha permesso di:

- Realizzare un dataset ampliato e bilanciato;
- Sviluppare un sistema di retichettatura automatico basato su euristiche robuste;
- Eseguire un fine tuning efficace di BERT con performance superiori al 90% di accuratezza.

Nei prossimi step ci concentreremo su:

1. Valutazione del modello su risposte reali non etichettate, tramite API Hugging Face;

2. Analisi degli embedding multidimensionali per rilevare outlier e cluster semantici;

3. Implementazione di tecniche di data augmentation (e.g. synonym replacement) per aumentare ulteriormente la robustezza;

4. Esplorazione di modelli più recenti (es. RoBERTa, DeBERTa) e confronti prestazionali.

A Codice Completo di Fine Tuning

Il codice Python dettagliato utilizzato in Colab è riportato integralmente nelle relazioni precedenti; in questa appendice ne forniamo un estratto riassuntivo.

```
!pip install transformers datasets evaluate torch accelerate
import os, numpy as np, pandas as pd
from transformers import AutoTokenizer,
   AutoModelForSequenceClassification, TrainingArguments, Trainer
from datasets import load_dataset
import evaluate

ios.environ['WANDB_MODE'] = 'offline'

# Caricamento e preprocessing
files = {'train':'dataset_completo.json','test':'Test2.json'}
dataset = load_dataset('json',data_files=files)
# Tokenizzazione, split strict 80/20
# Definizione delle metriche
# Callback personalizzata per log

training_args = TrainingArguments(
  output_dir='output_bert',
  num_train_epochs=3,
  per_device_train_batch_size=16,
  evaluation_strategy='steps',
  eval_steps=100,
  logging_steps=100,
  learning_rate=2e-5,
  weight_decay=0.01,
  warmup_steps=500,
  fp16=True,
  gradient_accumulation_steps=2
)

trainer = Trainer(
  model=AutoModelForSequenceClassification.from_pretrained('bert-
     base-uncased',num_labels=2),
  args=training_args,
  train_dataset=train_dataset,
  eval_dataset=validation_dataset,
  compute_metrics=lambda p: evaluate.load('accuracy').compute(
     predictions=p.predictions.argmax(-1),references=p.label_ids)
```

```
)
trainer.train()
trainer.evaluate(test_dataset)
trainer.save_model('output_bert')
```

Listing 2: Estratto del codice di training BERT-base