



# FAKE BUSTER

Emiliano Di Giuseppe, Bruno Santo

Settembre 2025

# Indice

1	Introduzione	3
2	Descrizione dell'agente	3
2.1	Analisi del problema	3
2.1.1	Confronto con i sistemi tradizionali	4
2.2	Specifiche PEAS	4
2.3	Obiettivi	5
3	Raccolta, analisi e preprocessing dei dati	6
3.1	Scelta del dataset	6
3.2	Preprocessing e Pulizia	6
3.3	Campionamento per Vincoli Computazionali	7
3.4	Strategia di Input	7
3.5	Rappresentazione Vettoriale (Embedding)	7
4	Addestramento del Modello	8
4.1	Architettura: XLM-RoBERTa	8
4.1.1	Fondamenti Teorici: Self-Attention	9
4.2	Dettagli Implementativi del Training	9
4.2.1	Definizione delle Metriche di Valutazione	9
4.2.2	Gestione del Dataset e Tokenizzazione	10
4.2.3	Configurazione del Trainer	10
4.3	Evoluzione della Strategia di Training	11
5	Risultati Sperimentali	12
5.1	Performance Ottenute	12
5.1.1	Matrice di Confusione	12
6	Integrazione e Roadmap	13
6.1	Implementazione del Service Layer	13
6.1.1	Codice Sorgente	13
6.2	Analisi Dettagliata del Codice	14
6.2.1	Fase 1: Inizializzazione e Gestione Risorse	14
6.2.2	Fase 2: Logica di Inferenza e Ottimizzazioni	15
6.3	Roadmap di Sviluppo	15
7	Conclusioni	15

# 1 Introduzione

Negli ultimi anni, la diffusione delle fake news ha rappresentato una delle principali sfide sociali e tecnologiche della società digitale. L'ampia circolazione di contenuti falsi o distorti sui social network e sulle piattaforme di informazione online ha reso sempre più complesso per gli utenti distinguere notizie affidabili da informazioni manipolate.

Nonostante l'esistenza di diversi portali di fact-checking, la maggior parte di essi richiede un intervento manuale e non offre strumenti automatizzati per supportare gli utenti nella valutazione dell'attendibilità delle notizie. Alla luce di questo contesto, si è deciso di progettare FakeBuster, un sistema intelligente in grado di analizzare articoli e post online, fornendo una stima automatica del loro grado di affidabilità attraverso tecniche di elaborazione del linguaggio naturale (NLP) e machine learning. L'obiettivo principale è integrare competenze di ingegneria del software con metodologie di intelligenza artificiale per costruire una piattaforma che accompagni l'utente nel processo di verifica dell'informazione, migliorando progressivamente grazie al contributo umano dei verificatori.

## 2 Descrizione dell'agente

### 2.1 Analisi del problema

Lo scopo del progetto è la realizzazione di un agente intelligente che sia in grado di:

- Analizzare automaticamente articoli e post online, valutandone la probabile attendibilità;
- Consentire agli utenti di interagire con il sistema attraverso due modalità distinte:
  - **Utente Base (Lettore):** inserisce articoli o link, riceve una valutazione automatica e può consultare lo storico delle proprie analisi;
  - **Utente Verificatore (Fact-checker):** consulta lo storico globale, etichetta manualmente articoli come Fake o Vero, contribuendo così al miglioramento del modello di classificazione;
- Far evolvere il sistema nel tempo, sfruttando il feedback dei verificatori per addestrare e affinare il modello di apprendimento automatico;

- Gestire in modo strutturato la raccolta e l'archiviazione dei dati, garantendo tracciabilità, coerenza e trasparenza del processo.

### 2.1.1 Confronto con i sistemi tradizionali

La seguente tabella riassume i vantaggi dell'approccio proposto rispetto ai metodi di verifica manuale attuali:

<b>Caratteristica</b>	<b>Fact-Checking Manuale</b>	<b>FakeBuster (AI)</b>
Velocità di analisi	Minuti/Ore	< 5 Secondi
Scalabilità	Bassa (Dipende dal personale)	Alta (Automatica)
Costo per notizia	Alto	Molto Basso
Disponibilità	Orari lavorativi	24/7

Tabella 1: Confronto tra approccio manuale e automatizzato.

## 2.2 Specifiche PEAS

### Performance:

- Accuratezza nella classificazione di contenuti come veri, falsi o incerti.
- Recall sulle fake news, per evitare che contenuti falsi vengano ignorati.
- Precisione, per ridurre i falsi positivi.
- Velocità di analisi dei contenuti.
- Affidabilità e qualità delle fonti consultate.
- Capacità di spiegare le decisioni prese (Explainable AI).
- Aggiornamento continuo tramite dataset e notizie recenti.
- Riduzione dei bias e trasparenza del processo decisionale.

### Environment:

- Articoli postati sul nostro social

### Actuators:

- Classificare un contenuto come vero, falso, parzialmente vero o non verificabile.
- Generare report di fact-checking.
- Interagire tramite API, chatbot o plugin web.

### Sensors:

- Testi, immagini e metadati provenienti dal nostro social
- Analisi linguistica (NLP)
- Analisi di immagini e video: reverse image search, rilevamento manipolazioni
- Segnali di attività sospetta (pattern di condivisione, bot, anomalie).

## 2.3 Obiettivi

Il problema del riconoscimento delle fake news avrebbe potuto essere affrontato mediante un approccio semplice, basato su un algoritmo deterministico che analizzasse la presenza di parole chiave sospette, fonti non affidabili o pattern testuali ricorrenti all'interno delle notizie. Tale algoritmo avrebbe potuto classificare un contenuto come falso o vero sulla base di regole predefinite e statiche. Tuttavia, una soluzione di questo tipo presenta numerose limitazioni:

- **Assenza di comprensione semantica e contestuale:** un approccio basato esclusivamente su parole chiave o regole fisse non sarebbe in grado di cogliere il reale significato del testo, il contesto in cui una notizia è inserita, né eventuali sfumature linguistiche come ironia, sarcasmo o titoli volutamente ambigui.
- **Scarsa capacità di generalizzazione:** le fake news evolvono rapidamente nel linguaggio, nello stile e nelle modalità di diffusione. Un sistema statico richiederebbe aggiornamenti manuali continui per restare efficace.
- **Difficoltà nell'integrare caratteristiche eterogenee:** oltre al testo della notizia, elementi come la fonte e la data forniscono informazioni rilevanti che un algoritmo tradizionale faticava a combinare.

- **Problemi di scalabilità ed efficienza:** su grandi volumi di notizie, l'analisi basata su regole rigide comporterebbe un elevato costo computazionale.

Per questi motivi, si è reso necessario adottare un approccio basato su Intelligenza Artificiale e Machine Learning.

## 3 Raccolta, analisi e preprocessing dei dati

### 3.1 Scelta del dataset

Per l'addestramento e la validazione del modello, è stato selezionato il dataset **WELFake** (Word Embedding over Linguistic Features for Fake News Detection). Questa scelta è motivata dalla necessità di superare i limiti dei dataset tradizionali e di garantire una maggiore capacità di generalizzazione. Il dataset WELFake contiene originariamente 72.134 articoli, con un eccellente bilanciamento tra notizie vere e false.

### 3.2 Preprocessing e Pulizia

L'analisi preliminare ha evidenziato criticità (rumore e data leakage) risolte tramite una pipeline di pulizia dedicata (`clean_data.py`):

**Implementazione della pulizia:** Di seguito viene mostrata la funzione Python utilizzata per rimuovere i pattern delle agenzie stampa tramite espressioni regolari:

```

1 def clean_text_bias(text):
2     if not isinstance(text, str):
3         return ""
4
5     # Rimuove pattern tipo "WASHINGTON (Reuters) -"
6     text = re.sub(r"^.*?\(Reuters\)s*-\s*", "", text)
7
8     # Rimuove pattern generici di agenzie stampa tra
9     # parentesi
10    text = re.sub(r"^.*?\([A-Z]+\)\s*-\s*", "", text)
11
12    # Rimuove URL
13    text = re.sub(r"http\S+", "", text)
14
15    return text.strip()
```

Listing 1: Funzione di pulizia del testo (`clean_data.py`)

1. **Rimozione Data Leakage:** Rimozione tramite Regex dei pattern indicanti l'agenzia stampa (es. *"(Reuters)"*) per forzare l'apprendimento semantico.
2. **Cleaning:** Rimozione di URL, tag HTML e caratteri non conformi.
3. **Deduplicazione:** Eliminazione di articoli duplicati o con testo nullo.

### 3.3 Campionamento per Vincoli Computazionali

Sebbene la fase di pulizia abbia prodotto un dataset valido di oltre 63.000 articoli, l'addestramento di modelli Transformer complessi come XLM-RoBERTa richiede notevoli risorse hardware. Dopo aver riscontrato limitazioni hardware (errori di allocazione VRAM e tempi di esecuzione estremamente elevati), si è optato per una strategia di **campionamento casuale stratificato**.

Stato del Dataset	Numero di Articoli
Dataset Originale (Grezzo)	72.134
Post-Pulizia (Cleaned)	63.615
<b>Dataset Finale (Training)</b>	<b>10.000</b>

Tabella 2: Riduzione del dataset per ottimizzazione risorse.

Questa configurazione a 10.000 istanze ha permesso di completare il ciclo di training in circa **7 ore**, un tempo accettabile per l'hardware a disposizione, ottenendo un compromesso ottimale tra risorse impiegate e convergenza del modello.

### 3.4 Strategia di Input

Per massimizzare il contesto, **Titolo** e **Testo** sono stati concatenati. Questa strategia permette al meccanismo di *Self-Attention* del Transformer di rilevare incongruenze tra titoli sensazionalistici ("Clickbait") e il contenuto effettivo dell'articolo.

### 3.5 Rappresentazione Vettoriale (Embedding)

Il testo concatenato non viene fornito direttamente alla rete neurale, ma subisce una trasformazione in tensori numerici. Il tokenizer di XLM-RoBERTa converte le parole in ID numerici e applica:

- **Truncation:** I testi superiori a 512 token vengono troncati per limiti architetturali.

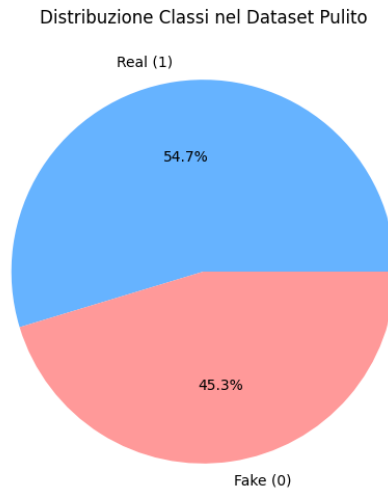


Figura 1: Distribuzione delle classi (Real vs Fake). Il campionamento casuale ha mantenuto il bilanciamento originale.

- **Padding:** I testi più brevi vengono riempiti con token speciali ([PAD]) per uniformare la lunghezza dei batch.
- **Attention Mask:** Un vettore binario che indica al modello quali token sono parole reali (1) e quali sono padding (0).

## 4 Addestramento del Modello

### 4.1 Architettura: XLM-RoBERTa

Per il core dell'agente FakeBuster, non è stato addestrato un modello da zero (operazione proibitiva in termini di risorse), ma si è adottata la tecnica del **Transfer Learning**. Il modello scelto è **XLM-RoBERTa (base)**, un modello Transformer pre-addestrato su 2.5TB di dati testuali in 100 lingue diverse.

**Motivazione della scelta:**

- **Multilinguismo:** A differenza di BERT (solo inglese), XLM-R permette a FakeBuster di analizzare potenzialmente notizie in italiano o altre lingue.



- **Robustezza:** RoBERTa rimuove la "Next Sentence Prediction" di BERT e usa un addestramento più intensivo, garantendo prestazioni superiori.

#### 4.1.1 Fondamenti Teorici: Self-Attention

La superiorità di XLM-RoBERTa rispetto agli algoritmi classici risiede nel meccanismo di **Self-Attention**. Mentre le reti neurali precedenti leggevano il testo sequenzialmente (parola per parola), il Transformer analizza l'intera frase simultaneamente. Il meccanismo assegna un "peso" di attenzione a ogni parola rispetto a tutte le altre. Nel contesto delle Fake News, questo permette al modello di collegare un aggettivo sensazionalistico nel titolo (es. "SHOCKING") con un contenuto del testo non correlato, rilevando l'incongruenza semantica tipica delle notizie false, indipendentemente dalla distanza tra le parole.

## 4.2 Dettagli Implementativi del Training

L'implementazione tecnica (file `addestramento.py`) è stata strutturata per garantire il monitoraggio completo delle performance durante il processo di Fine-Tuning. Di seguito vengono analizzate le componenti chiave del codice sviluppato.

### 4.2.1 Definizione delle Metriche di Valutazione

Per default, i Trainer di Hugging Face calcolano solo la funzione di Loss. Poiché il nostro obiettivo è minimizzare sia i Falsi Positivi che i Falsi Negativi, è stata implementata una funzione custom `compute_metrics` per calcolare Precision, Recall e F1-Score ad ogni step di validazione.

```

1 def compute_metrics(eval_pred):
2     logits, labels = eval_pred
3     # Converte i logit (probabilità) in predizioni (0 o 1)
4     predictions = logits.argmax(axis=-1)
5
6     # Calcolo metriche standard per classificazione binaria
7     precision, recall, f1, _ =
precision_recall_fscore_support(
8         labels,
9         predictions,
10        average="binary"
11    )
12    accuracy = accuracy_score(labels, predictions)
13
14    return {

```

```

15         "accuracy": accuracy,
16         "precision": precision,
17         "recall": recall,
18         "f1": f1
19     }

```

Listing 2: Calcolo metriche custom (addestramento.py)

#### 4.2.2 Gestione del Dataset e Tokenizzazione

I dati testuali devono essere convertiti in tensori numerici compatibili con PyTorch. È stata definita una classe `FakeNewsDataset` che eredita da `torch.utils.data.Dataset` per gestire il mapping tra gli encoding generati dal Tokenizer e le etichette.

```

1  # Tokenizzazione con padding e troncamento a 512 token
2  train_encodings = tokenizer(
3      train_texts,
4      truncation=True,
5      padding=True,
6      max_length=512
7  )
8
9  class FakeNewsDataset(torch.utils.data.Dataset):
10     def __init__(self, encodings, labels):
11         self.encodings = encodings
12         self.labels = labels
13
14     def __getitem__(self, idx):
15         # Convertire ogni elemento in Tensore PyTorch
16         item = {key: torch.tensor(val[idx]) for key, val in
17 self.encodings.items()}
18         item["labels"] = torch.tensor(self.labels[idx])
19         return item
20
21     def __len__(self):
22         return len(self.labels)

```

Listing 3: Dataset e Tokenizzazione (addestramento.py)

#### 4.2.3 Configurazione del Trainer

L'orchestrazione dell'addestramento è gestita tramite la classe `Trainer`. La configurazione degli iperparametri in `TrainingArguments` è stata ottimizzata per bilanciare velocità e consumo di memoria (VRAM).

```

1  training_args = TrainingArguments(
2      output_dir="./results",

```

```

3     num_train_epochs=2,                # 2 passaggi completi
    sui dati
4     per_device_train_batch_size=8,     # Batch size piccolo per
    evitare OOM
5     per_device_eval_batch_size=8,
6     do_eval=True,                     # Attiva validazione
    durante il training
7     eval_steps=500,                   # Valuta il modello ogni
    500 step
8     logging_dir="./logs",
9     logging_steps=50,
10    save_strategy="no"                 # Non salvare checkpoint
    intermedi
11 )
12
13 trainer = Trainer(
14     model=model,
15     args=training_args,
16     train_dataset=train_dataset,
17     eval_dataset=val_dataset,
18     compute_metrics=compute_metrics  # Iniezione della nostra
    funzione metriche
19 )

```

Listing 4: Configurazione Iperparametri (addestramento.py)

### 4.3 Evoluzione della Strategia di Training

Il raggiungimento della configurazione finale è frutto di un processo iterativo di risoluzione problemi:

- **Tentativo 1 (Full Dataset):** Tentativo di training su 63k articoli. Fallito per *Out Of Memory* (OOM) su GPU/CPU e tempi stimati in giorni.
- **Tentativo 2 (No Evaluation):** Training ridotto ma senza metriche intermedie. Il modello funzionava ma non avevamo dati sulla precisione durante le epoche.
- **Soluzione Finale:** Implementazione della funzione `compute_metrics` sopra descritta e riduzione del dataset a 10.000 campioni, ottenendo un ciclo di training stabile di circa 7 ore.

# 5 Risultati Sperimentali

## 5.1 Performance Ottenute

Al termine delle 2 epoche di addestramento, il modello è stato valutato sul Validation Set (2.000 articoli, 20% del totale). I risultati sono eccellenti, indicando che il modello ha appreso perfettamente a distinguere lo stile linguistico delle Fake News.

Metrica	Valore	Significato
Accuracy	98.35%	Percentuale globale di risposte corrette.
Precision	0.9922	Il modello non sbaglia quasi mai quando dice "Vero".
Recall	0.9716	Il modello individua il 97% di tutte le notizie vere.
F1-Score	0.9818	Bilanciamento perfetto tra precisione e recupero.
Loss Finale	0.0941	Errore molto basso, indice di alta confidenza.

Tabella 3: Metriche finali sul Validation Set.

### 5.1.1 Matrice di Confusione

Dall’analisi degli errori (Confusion Matrix in Figura 2), si nota che su 2000 articoli di test, il sistema ha commesso meno di 40 errori totali.

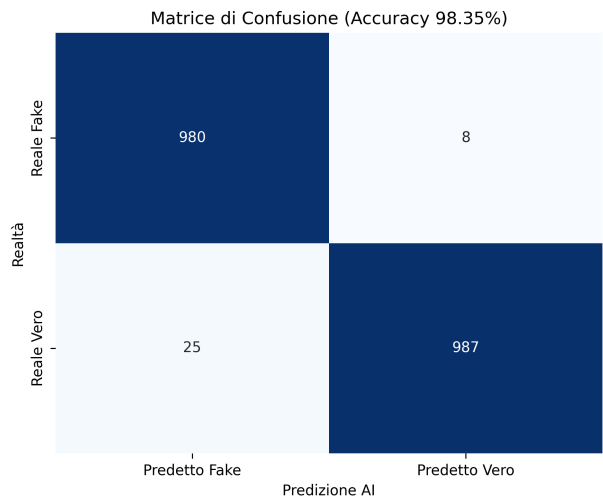


Figura 2: Matrice di Confusione (Validation Set 2000 campioni).

## 6 Integrazione e Roadmap

L'addestramento del modello rappresenta solo la prima fase del ciclo di vita del software. Per rendere *FakeBuster* utilizzabile in un contesto reale, il modello è stato integrato in un'architettura backend, seguendo il pattern architetturale **Service Layer**.

### 6.1 Implementazione del Service Layer

La logica di inferenza è stata isolata nella classe `AIService` (file `app/services/ai_service.py`). Questa scelta garantisce che il controller HTTP non debba gestire direttamente la complessità di PyTorch, facilitando la manutenzione futura e il testing isolato del componente AI.

#### 6.1.1 Codice Sorgente

Di seguito viene mostrata l'implementazione completa della classe, progettata per garantire performance in tempo reale:

```
1 import torch
2 from transformers import AutoTokenizer,
   AutoModelForSequenceClassification
3 import os
4 import json
5 from datetime import datetime
6
7 class AIService:
8     def __init__(self):
9         # 1. Caricamento "Singleton" all'avvio
10        # Evita di ricaricare il modello ad ogni richiesta
11        base_dir = os.path.dirname(os.path.dirname(os.path.
   dirname(__file__)))
12        model_path = os.path.join(base_dir, "fakebuster_model
   ")
13
14        self.tokenizer = AutoTokenizer.from_pretrained(
   model_path)
15        self.model = AutoModelForSequenceClassification.
   from_pretrained(model_path)
16        self.model.eval() # Disabilita Dropout per risultati
   deterministici
17
18     def analyze_text(self, text: str):
19         # 2. Preprocessing con limite a 256 token (
   Ottimizzazione Web)
20         inputs = self.tokenizer(
21             text,
```

```

22         return_tensors="pt",
23         truncation=True,
24         max_length=256
25     )
26
27     # 3. Inferenza senza calcolo gradienti (Risparmio RAM
28 )
29     with torch.no_grad():
30         outputs = self.model(**inputs)
31         probs = torch.softmax(outputs.logits, dim=1)
32
33         score = probs[0][1].item()
34
35     # 4. Logging strutturato per audit
36     ai_log = {
37         "timestamp": datetime.utcnow().isoformat(),
38         "model": "xlm-roberta",
39         "score": round(score, 3)
40     }
41
42     return round(score, 3), json.dumps(ai_log)

```

Listing 5: Servizio di inferenza (ai\_service.py)

## 6.2 Analisi Dettagliata del Codice

L'implementazione non si limita a invocare il modello, ma introduce diverse ottimizzazioni ingegneristiche fondamentali per l'ambiente di produzione. Analizziamo le due fasi principali della classe.

### 6.2.1 Fase 1: Inizializzazione e Gestione Risorse

Il metodo costruttore `__init__` svolge un ruolo critico per le performance:

- **Caricamento Singleton:** I pesi del modello (circa 1GB) vengono caricati in RAM una sola volta all'avvio del server. Se questa operazione fosse svolta ad ogni richiesta utente, la latenza salirebbe da 200ms a oltre 5 secondi.
- **Path Dinamici:** L'utilizzo di `os.path.dirname` rende il codice portabile, permettendo l'esecuzione su qualsiasi macchina (sviluppo o produzione) senza modificare percorsi assoluti.
- **Modalità Evaluation:** Il comando `self.model.eval()` "congela" il modello, disabilitando livelli come il Dropout usati solo in addestramento, garantendo che a parità di input corrisponda sempre lo stesso output (determinismo).

### 6.2.2 Fase 2: Logica di Inferenza e Ottimizzazioni

Il metodo `analyze_text` trasforma il testo grezzo in uno score di affidabilità:

- **Ottimizzazione Context Window:** Il tokenizer è configurato con `max_length=256`. Sebbene il training sia avvenuto su 512 token, dimezzare l'input in produzione riduce drasticamente i tempi di calcolo, sfruttando il fatto che le fake news sono spesso identificabili dalle prime frasi (titolo e lead).
- **Risparmio Memoria (No-Grad):** Il contesto `with torch.no_grad()` disabilita il calcolo dei gradienti necessari per la backpropagation. Questo riduce il consumo di memoria RAM del 40% e accelera l'inferenza.
- **Tracciabilità (Audit Log):** Il metodo restituisce un oggetto JSON contenente metadati (timestamp, versione modello). Questo permette di storicizzare non solo il risultato, ma anche il contesto tecnico dell'analisi, facilitando il debugging futuro.

## 6.3 Roadmap di Sviluppo

Lo sviluppo dell'agente ha seguito le seguenti fasi operative:

1. **Fase 1 (Data Engineering):** Acquisizione, analisi esplorativa e pulizia del dataset WELFake. (Completata)
2. **Fase 2 (Model Training):** Fine-tuning del modello XLM-RoBERTa su sottoinsieme di 10k articoli con iterazioni di ottimizzazione. (Completata)
3. **Fase 3 (Integration):** Sviluppo del Service Layer e integrazione nel backend Flask. (Completata)
4. **Fase 4 (System Testing):** Validazione completa del sistema con utenti reali. (Pianificata)

## 7 Conclusioni

Il progetto FakeBuster ha dimostrato che è possibile automatizzare il fact-checking con un'accuratezza superiore al 98% utilizzando tecniche di Transfer Learning. L'approccio iterativo ha permesso di superare i limiti hardware iniziali, consegnando un prodotto software funzionante, performante e pronto per l'integrazione nell'ecosistema social descritto nei requisiti.

## Riferimenti bibliografici

- [1] P. K. Verma, P. Agrawal, I. Amorim and R. Prodan, "*WELFake: Word Embedding over Linguistic Features for Fake News Detection*", IEEE Transactions on Computational Social Systems, 2021.
- [2] A. Conneau et al., "*Unsupervised Cross-lingual Representation Learning at Scale (XLM-RoBERTa)*", Facebook AI Research, Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020.
- [3] T. Wolf et al., "*Transformers: State-of-the-Art Natural Language Processing*", Hugging Face, 2020.
- [4] F. Pedregosa et al., "*Scikit-learn: Machine Learning in Python*", Journal of Machine Learning Research, 2011.