Università degli Studi di Torino
Scuola di Scienze della Natura
Corso di Laurea Magistrale in Informatica

Tesi di Laurea Magistrale

Design, ingegnerizzazione e realizzazione di un sistema di dialogo basato su LLM nel dominio delle tecnologie assistive

RELATORE
Prof. Alessandro Mazzei

CANDIDATO
Stefano Vittorio Porta
859133

Anno Accademico 2023/2024

Dichiarazione di Originalità

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non avere plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Ringraziamenti

Todo

Abstract

Todo.

Parole chiave

«classificazionexNLU», «data annotation/augmentation», «NLGBasataSuParafrasi», «ingegnerizzazione»

Indice

1 Introduzione

1.1 Contesto generale

dare un'idea di chatbot, cos'è NovaGraphS, perché è importante, come si inserisce nel contesto delle tecnologie assistive

1.2 Motivazioni e obiettivi della tesi

1.3 Struttura del documento

2 Natural Language Understanding

(introduzione all'argomento)

2.1 Come AIML gestisce la comprensione

Negli anni '90 iniziò a guadagnare popolarità il Loebner Prize, una competizione ispirata al Test di Turing [1].
Nella competizione, chatbot e sistemi conversazionali cercavano di "ingannare" giudici umani, facendo credere loro di essere persone reali. Molti sistemi presentati alla competizione erano basati su pattern matching e rule-based, a volte integrando euristiche per la gestione di sinonimi o correzione ortografica.

Tra questi, uno dei più celebri è *ALICE* (Artificial Linguistic Internet Computer Entity), sviluppato da Richard Wallace utilizzando il linguaggio di markup AIML (Artificial Intelligence Markup Language) da lui introdotto [2], [3].
ALICE vinse per la prima volta il Loebner Prize nel 2000, e in seguito vinse altre due edizioni, nel 2001 e 2004.

2.1.1 Struttura di un chatbot AIML

Basato sull'XML [2], di base l'AIML fornisce una struttura formale per definire regole di conversazione attraverso **categorie** di *pattern* e *template*:

- `<pattern>` : la frase (o le frasi) attese in input a cui il chatbot deve reagire;
- `<template>` : la risposta (testuale o con elementi dinamici) che il chatbot fornisce quando si verifica il match del pattern.

La forma più semplice di categoria è:

```
<category>
  <pattern>CIAO</pattern>
  <template>Ciao! Come posso aiutarti oggi?</template>
</category>
```

Qui, se l'utente scrive "Ciao"[1], il sistema restituisce la risposta associata nella sezione del `<template>`.

Naturalmente questa è una regola basilare; AIML permette di definire pattern molto più complessi.
Un primo passo verso la creazione di regole più flessibili è l'uso di wildcard: associando simboli quali $*$ e – a elementi di personalizzazione (`<star/>`), il motore che esegue la configurazione AIML può gestire un certo grado di variabilità linguistica:

[1] Caratteri maiuscoli e minuscoli sono considerati uguali dal motore di riconoscimento.

```
<category>
  <pattern>MI CHIAMO *</pattern>
  <template>
    Ciao <star/>, piacere di conoscerti!
  </template>
</category>
```

In particolare, il simbolo `*` corrisponde a una wildcard che cattura qualsiasi sequenza di parole in input tra i due pattern specificati.
In questo caso, se l'utente digita "Mi chiamo Andrea", il sistema sostituisce `<star/>` con "Andrea", e risponde di conseguenza.

Spesso è necessario memorizzare informazioni fornite dall'utente per utilizzarle successivamente. A questo scopo, AIML offre i tag `<set>` e `<get>` che, rispettivamente, memorizzano e recuperano valori da variabili di contesto:

```
<category>
  <pattern>IL MIO COLORE PREFERITO È *</pattern>
  <template>
    <think>
      <set name="colore"><star/></set>
    </think>
    Ok, ricorderò che il tuo colore preferito è <star/>.
  </template>
</category>

<category>
  <pattern>QUAL È IL MIO COLORE PREFERITO</pattern>
  <template>
    Il tuo colore preferito è <get name="colore"/>.
  </template>
</category>
```

Nella prima `<category>`, il tag `<think>` fa sì che l'operazione di memorizzazione non produca output testuale per l'utente, ma aggiorni internamente la variabile `colore`.
Nel secondo blocco, si utilizza `<get name="colore"/>` per restituire all'utente il valore memorizzato.

Il tag `<condition>` permette di definire regole condizionali in base a variabili di contesto.

Se la variabile stagione (presumibilmente impostata altrove con un `<set>`) ha valore `inverno`, verrà restituito "Fa piuttosto freddo…". Un risultato simile si ottiene per `estate`, mentre per altri valori o mancanza di valore si restituisce l'ultimo *list item*.

```
<category>
  <pattern>CHE TEMPO FA</pattern>
  <template>
    <condition name="stagione">
      <li value="inverno">Fa piuttosto freddo, in questa stagione.</li>
      <li value="estate">Fa molto caldo, bevi tanta acqua!</li>
      <li>Non saprei, forse è primavera o autunno?</li>
    </condition>
  </template>
</category>
```

Il tag `<topic>` permette di raggruppare categorie che appartengono a un medesimo ambito di conversazione, per facilitare la lettura delle regole:

```
<topic name="saluti">
  <category>
    <pattern>CIAO</pattern>
    <template>
      Ciao, piacere di vederti!
    </template>
  </category>

  <category>
    <pattern>COME STAI</pattern>
    <template>
      Sto bene, grazie! E tu come stai?
    </template>
  </category>
</topic>
```

In questo modo le regole legate ai saluti sono tutte contenute all'interno di un blocco `<topic>` chiamato `saluti`.

Il tag `<srai>`[2] permette di reindirizzare l'input ad un'altra regola, che verrà processata come se fosse stata digitata dall'utente. È molto utile per riutilizzare risposte o logiche già definite:

```
<topic name="saluti">
  <category>
    <pattern>SALUTA *</pattern>
    <template>
      <srai>CIAO</srai>
    </template>
  </category>
</topic>
```

[2]Stimulus-Response Artificial Intelligence [2]

Se l'utente scrive "Saluta Andrea", la regola cattura "SALUTA *" e reindirizza il contenuto (in questo caso "CIAO") a un'altra categoria. Se esiste una categoria che gestisce il pattern "CIAO", verrà attivata la relativa risposta.

Esiste anche una versione contratta di `<srai>` chiamata `<sr>`, che è stata prevista come scorciatoia quando è necessario matchare un solo pattern. Secondo la documentazione, il tag corrisponde a `<srai><star/></srai>`.

Abbiamo già visto `<think>` in azione per evitare che il contenuto venga mostrato all'utente. In generale, `<think>` è utile quando vogliamo impostare o manipolare variabili senza generare output visibile, ad esempio:

```
<category>
  <pattern>ADESSO È *</pattern>
  <template>
    <think><set name="stagione"><star/></set></think>
    Grazie, ora so che la stagione attuale è <star/>!
  </template>
</category>
```

Il tag `<that>` permette di scrivere pattern che dipendono dalla risposta precedentemente fornita dal chatbot. È particolarmente utile per gestire contesti conversazionali più complessi:

```
  <category>
    <pattern>GRAZIE</pattern>
    <that>VA TUTTO BENE</that>
    <template>Felice di averti aiutato!</template>
  </category>
```

In questo caso la regola sarà attivata se la risposta precedente del bot era "VA TUTTO BENE" e l'utente scrive "Grazie".

Per rendere la conversazione più naturale, AIML 2.0 fornisce `<random>`, che permette di restituire una risposta fra più alternative:

```
<category>
  <pattern>COME VA</pattern>
  <template>
    <random>
      <li>Benissimo, grazie!</li>
      <li>Abbastanza bene, e tu?</li>
      <li>Non c'è male, e tu come stai?</li>
    </random>
  </template>
</category>
```

Ogni volta che l'utente scrive "Come va", il bot sceglierà casualmente una delle tre risposte elencate.

Alcune versioni di AIML supportano `<learn>`, che consente al bot di aggiungere nuove categorie "al volo" durante l'esecuzione:

```
<category>
  <pattern>TI INSEGNO *</pattern>
  <template>
    <think>
      <learn>
        <![CDATA[
          <category>
            <pattern><star/></pattern>
            <template>Ho imparato a rispondere a "<star/>"!</template>
          </category>
        ]]>
      </learn>
    </think>
    Ho imparato una nuova regola!
  </template>
</category>
```

2.1.2 Criticità e limiti di AIML

Grazie ai tag previsti dallo schema, AIML riesce a gestire conversazioni piuttosto complesse. Ciononostante, presenta comunque alcune limitazioni:

- Le strategie di wildcard e pattern matching restano prevalentemente letterali, con limitata capacità di interpretare varianti linguistiche non codificate nelle regole. Se una frase si discosta dal pattern previsto, il sistema fallisce il matching. Sono disponibili comunque alcune funzionalità per la gestione di sinonimi, semplificazione delle locuzioni e correzione ortografica (da comporre e aggiornare manualmente) che possono mitigare alcuni di questi problemi.
- La gestione del contesto (via `<that>, <topic>`, `<star>`, ecc.) è rudimentale, soprattutto se paragonata a sistemi moderni di NLU con modelli neurali che apprendono contesti ampi e riescono a tenere traccia di dettagli dal passato della conversazione.
- L'integrazione con basi di conoscenza esterne (KB, database, API) richiede estensioni o script sviluppati ad-hoc, poiché AIML di per sé non offre costrutti semantici o query integrate, e non permette di integrare script internamente alle regole [2].
- Le risposte generate sono statiche e predefinite, e non possono essere generate dinamicamente in base a dati esterni o a contesti più ampi in modo automatico (come invece avviene con LLM e modelli di generazione di linguaggio).

Nonostante questi limiti, AIML ha rappresentato un passo importante nell'evoluzione dei chatbot, offrendo un framework standardizzato e relativamente user-friendly per la creazione di agenti rule-based [3].
In alcuni ambiti ristretti (FAQ, conversazioni scriptate, assistenti vocali), costituisce ancora una soluzione valida e immediata. In domini più complessi, in cui la varietà del linguaggio e l'integrazione con dati dinamici sono essenziali, diventa indispensabile affiancare o sostituire AIML con tecniche di Natural Language Understanding basate su machine learning e deep learning.

Nelle sezioni successive sarà mostrato il percorso seguito per cercare di migliorare la comprensione degli input dell'utente, integrando tecniche di NLU basate su modelli di linguaggio neurali, e valutando le performance ottenute rispetto ad AIML.

2.2 Classificazione con LLM

Come detto poco sopra, uno dei limiti di AIML è la gestione limitata di varianti linguistiche e contesti conversazionali.

Per permettere all'AIML di generalizzare sulle richieste degli utenti, il botmaster deve dichiarare delle generalizzazioni esplicite, ad esempio utilizzando wildcard o pattern che catturano più varianti di una stessa richiesta. Questo processo richiede tempo e competenze linguistiche, oltre ad una grande attenzione per evitare ambiguità o sovrapposizioni tra regole.

Durante il mio percorso di ricerca ho deciso di seguire una strada simile a quella di AIML, ma facendo un passo indietro e ponendomi la domanda:

"Invece che cercare dei pattern nelle possibili richieste degli utenti, perchè non trovare un modello che possa generalizzare su queste richieste in modo automatico?"

Il percorso per arrivare al modello di classificazione di intenti ha richiesto i suoi tempi, ma alla fine ho ottenuto dei risultati che ritengo soddisfacenti.

I problemi principali da risolvere per poter classificare gli intenti sono due: la raccolta di dati etichettati e la scelta del modello di classificazione.

2.2.1 Dataset di training

Di base, nel mondo dell'apprendimento automatico supervisionato, per addestrare un modello di classificazione è necessario un dataset di esempi etichettati, cioè coppie di input e output che il modello deve apprendere a generalizzare.

Per la classificazione di intenti, i dataset più comuni sono quelli di chatbot e assistenti vocali, che contengono domande e richieste etichettate con l'intento che l'utente vuole esprimere.

Il dataset originario fornitomi è stato composto in seguito a una campagna di raccolta dati manuale, in cui diversi collaboratori hanno interagito con un prototipo di chatbot AIML, ponendo domande e richieste di vario tipo.

Il dataset è una collezione di circa 700 singole interazioni, metà sotto forma di domande degli utenti durante la prima fase di sperimentazione, e l'altra che coincide con ciò che il chatbot ha risposto.

Estrazione dei dati

Dovendo addestrare un modello di classificazione, ho proceduto innanzitutto con l'estrazione dei dati effettivamente a noi necessari. Un piccolo script python che adopera la libreria `pandas` [4] è stato sufficiente:

```python
import pandas as pd
from dotenv import load_dotenv

load_dotenv()

df_o = pd.read_excel('corpus/interaction-corpus.xlsx')

# Filter only the rows that have "Participant" as 'U'
df = df_o[df_o['Participant'] == 'U']
df = df[['Text']]
df = df.drop_duplicates()
df = df[df['Text'].apply(lambda x: isinstance(x, str))]
df['Text'] = df['Text'].str.strip()  # Remove trailing whitespace
texts = df['Text'].dropna()

df.to_csv("./filtered_data.csv")
```

Codice 1: Estrazione dei dati dal dataset di interazione.

Estratte le domande, si è potuto procedere con l'etichettatura.

In un primo step, ho considerato la possibilità di lasciare il compito di etichettatura delle domande ad un sistema che svolgesse il compito in automatico.
Questo permetterebbe di avere un dataset decorato, senza dover ricorrere a un'etichettatura manuale che sarebbe stata molto dispendiosa in termini di tempo e risorse, specialmente in ottica di un incremento dei dati del dataset in seguito a nuove interazioni con il chatbot.

Per fare ciò, ho rivolto la mia attenzione ai modelli di linguaggio neurali, in particolare alle Large Language Models (LLM), dal momento che sono in grado di generalizzare su una vasta gamma di task linguistici, inclusa la classificazione di intenti.

Con l'estrema disponibilità attuale di modelli pre-addestrati e API che permettono di interagire con essi, ho potuto sperimentare diverse soluzioni per l'etichettatura automatica delle domande.
In particolare, ho deciso di sperimentare con modelli di LLM open-source, dal momento che sono eseguibili localmente e permettono di mantenere i dati sensibili

all'interno dell'ambiente di lavoro, senza doverli condividere con servizi esterni.
Per utilizzarli, si sono rivelate fondamentali le API fornite da Ollama [5], un sistema per hostare localmente modelli di LLM open source (e in certi casi anche *open-weights*).

Etichettatura automatica delle domande

Per poter automatizzare l'etichettatura usando una LLM, prima di tutto ho identificato l'insieme delle possibili etichette:

```
LABELS: dict[str, str] = {
  "START": "Initial greetings or meta-questions, such as 'hi' or 'hello'.",
  "GEN_INFO": "General questions about the automaton that don't focus on specific
components or functionalities.",
  "STATE_COUNT": "Questions asking about the number of states in the automaton.",
  "FINAL_STATE": "Questions about final states of the automaton.",
  "STATE_ID": "Questions about the identity of a particular state.",
  "TRANS_DETAIL": "General questions about the transitions within the automaton.",
  "SPEC_TRANS": "Specific questions about particular transitions or arcs between
states.",
  "TRANS_BETWEEN": "Specific question about a transition between two states",
  "LOOPS": "Questions about loops or self-referencing transitions within the
automaton.",
  "GRAMMAR": "Questions about the language or grammar recognized by the automaton.",
  "INPUT_QUERY": "Questions about the input or simulation of the automaton.",
  "OUTPUT_QUERY": "Questions specifically asking about the output of the
automaton.",
  "IO_EXAMPLES": "Questions asking for examples of inputs and outputs.",
  "SHAPE_AUT": "Questions about the spatial or graphical representation of the
automaton.",
  "OTHER": "Questions not related to the automaton or off-topic questions.",
  "ERROR_STATE": "Questions related to error states or failure conditions within the
automaton.",
  "START_END_STATE": "Questions about the initial or final states of the
automaton.",
  "PATTERN_RECOG": "Questions that aim to identify patterns in the automaton's
structure or behavior.",
  "REPETITIVE_PAT": "Questions focusing on repetitive patterns, especially in
transitions.",
  "OPT_REP": "Questions about the optimal spatial or minimal representation of the
automaton.",
  "EFFICIENCY": "Questions about the efficiency or minimal representation of the
automaton."
}
```

Codice 2: Etichette possibili per le domande del dataset.

In questa mappa, ad ogni etichetta è associata una descrizione che indica alla LLM un contesto in cui collocarla, con lo scopo di assistere la LLM ad etichettare correttamente le domande togliendo il più possibile le ambiguità.
Questo genere di task è del tipo zero shot, in cui il modello non ha mai visto i dati di training e deve etichettare le domande esclusivamente in base a un contesto fornito.

Con lo scopo di assicurare un'etichettatura corretta e affidabile, ho deciso di utilizzare due modelli di LLM differenti, in modo da poter fare un majority voting tra le etichette prodotte dai due modelli:

- *Gemma 2*, sviluppato da Google Deep Mind [6];
- *llama 3.1*, sviluppato da Meta AI [7].

I modelli sono stati utilizzati nelle loro varianti da 9 miliardi di parametri per Gemma 2 (dimensione intermedia) e 8 miliardi per llama 3.1 (il più piccolo dei modelli forniti), basandomi sulle sperimentazioni che hanno mostrato un buon compromesso tra performance (intese come qualità dei risultati prodotti in seguito al prompting) e tempo di esecuzione [6], [7].

Un ulteriore modello, Qwen [8], prodotto da Alibaba, è stato utilizzato durante le sperimentazioni, ma i risultati non sono stati sufficientemente soddisfacenti da permettere un utilizzo all'interno del progetto.

Ho effettuato il prompting delle domande con i modelli di LLM utilizzando le risorse dell'hardware a mia disposizione, composto da:
- CPU AMD Ryzen 7 5800x (4.7GHz, 8 core, 16 thread, 32MB L3 cache)
- 64GB RAM DDR4 @3200MHz
- GPU Nvidia RTX 3070 Ti (8GB GDDR6, 6144 CUDA cores @1.77GHz)

Ad ogni modello è richiesto di etichettare ogni domanda. Il prompt utilizzato è stato progettato in modo da fornire un contesto chiaro e preciso, in modo da guidare la LLM verso l'etichetta corretta.
In particolare, ne sono stati utilizzati due per ogni modello, in modo da fornire un contesto più vario e permettere ai modelli di generalizzare meglio sulle domande. Ogni prompt risulta diverso dal punto di vista della composizione della richiesta, ma l'intento finale a livello semantico è lo stesso.

I prompt sono stati scelti in modo da fornire informazioni utili ai modelli per etichettare le domande, insieme ad un contesto che effettivamente faccia comprendere alla LLM quale sia l'argomento della domanda:

```
prompts = [
  # First prompt
  """You are going to be provided a series of interactions from a user regarding
questions about finite state automatons.
  Each message has to be labelled, according to the following labels:

  {labels}

  You only need to answer with the corresponding label you've identified.
  Do not explain the reasoning, do not use different terms from the labels you've
received now.
  Interaction:
  {text}
  Label:
  """,

  # Second prompt
  """You are an AI assistant trained to classify questions into the following
categories:

  {labels}

  Please classify the following question:
  {text}
  Category:
  """
]
```

Codice 3: Prompt utilizzati per l'etichettatura delle domande.

Si notino le differenze tra i due prompt: il primo è più dettagliato e fornisce una spiegazione più approfondita delle etichette, mentre il secondo è più conciso e diretto.

I tag tra parentesi graffe vengono sostituiti con i valori attualmente in uso, in modo da rendere il prompt più generico e riutilizzabile.

Segue un estratto di codice python che mostra come è stato effettuato il prompting. È inclusa una classe `Chat`, da me sviluppata, che permette di interagire con i modelli di LLM in modo più semplice, astraendo le API di ollama.

```python
from tqdm import tqdm
from chat_helper import Chat
import pandas as pd

# ollama_models = ["llama3.1:8b", "gemma:7b", "qwen:7b"]
ollama_models = ["gemma2:9b", "llama3.1:8b"]

# We are initializing a new dataframe with the same index as the original one
res_df = pd.DataFrame(index=df.index)

for model in ollama_models:
    chat = Chat(model=model)

    dataset_size = len(df)

    for p_i, prompt_version in enumerate(prompts):
        progress_bar = tqdm(
          total=dataset_size,
          desc=f"Asking {model} with prompt {p_i}", unit="rows"
        )

        for r_i, row in df.iterrows():
            text = row["Text"]

            prompt = prompts[0].replace("{text}", text)

            inferred_label = chat.interact(
              prompt,
              stream=True,
              print_output=False,
              use_in_context=False
            )
            inferred_label = inferred_label.strip().replace("'", "")

            res_df.at[r_i, f"{model} {p_i}"] = inferred_label
            progress_bar.update()

        print(progress_bar.format_dict["elapsed"])
        progress_bar.close()
```

Codice 4: Prompting delle domande con i modelli di LLM.

Ecco un esempio dei risultati dell'etichettatura del bronze dataset, in seguito al prompting con i modelli di LLM:

ID	gemma2:9b	gemma2:9b	llama3.1:8b	llama3.1:8b
0	START	START	START	START
1	GEN_INFO	GEN_INFO	GEN_INFO	GEN_INFO
2	SPEC_TRANS	SPEC_TRANS	TRANS_BETWEEN	TRANS_BETWEEN
3	SPEC_TRANS	SPEC_TRANS	TRANS_BETWEEN	TRANS_BETWEEN
4	Please provide the interaction. : START	START	START	START
…	…	…	…	…
285	OPT_REP	OPT_REP	OPT_REP	OPT_REP
286	GRAMMAR	GRAMMAR	GRAMMAR	GRAMMAR
287	REPETITIVE_PAT	REPETITIVE_PAT	REPETITIVE_PAT	REPETITIVE_PAT
288	TRANS_DETAIL	TRANS_DETAIL	TRANS_DETAIL	GEN_INFO
289	GRAMMAR	GRAMMAR	FINAL_STATE	FINAL_STATE

Tabella 1: Esempio di etichettatura delle domande del bronze dataset.

Come è possibile notare, i modelli hanno etichettato le domande in modo coerente tra di loro, ma non sempre con le etichette corrette.
In certi casi, le etichette sono state completamente sbagliate, e in altre occorrenze sono state prodotte risposte che o non sono presenti nel set di etichette fornito, o hanno ignorato il prompt fornito, fornendo risposte completamente estranee.

Come accennato, è stato adoperato un sistema di majority voting per combinare i risultati delle due LLM, in modo da ottenere un'etichettatura più affidabile:

```
from collections import Counter

def majority_vote(row: pd.Series):
    label_counts = Counter(row)
    majority_label = label_counts.most_common(1)[0][0]
    return majority_label
```

Codice 5: Funzione di majority voting per combinare le etichette.

Tuttavia, in seguito ad una prima fase di fine tuning, ho verificato che nonostante un'etichettatura valida, le classi identificate erano troppo sbilanciate, con alcune classi che contenevano un numero troppo esiguo di esempi. In più, ho realizzato che le classi scelte erano troppo generiche; questo non avrebbe permesso di identificare con precisione l'argomento della domanda.

Per questo motivo ho proceduto con una revisione delle etichette, e una successiva etichettatura manuale delle domande.

Nuove classi e etichettatura manuale

Prima di proseguire con l'etichettatura, ho provveduto a ripulire il dataset da domande non pertinenti o duplicate. Una volta fatto, ho deciso di ridurre il numero di classi, in

modo da poter avere un dataset più bilanciato e con classi più specifiche.
Avendone ridotto il numero, per ottenere un livello di granularità maggiore, ho deciso di utilizzare un sistema di etichettatura gerarchico, in modo da poter identificare con maggiore precisione l'argomento della domanda.
Ne sono risultati sono due livelli di classi:

- Le *classi principali* (o *question intent*, si veda la Tabella 2), che rappresentano l'argomento generale della domanda, per un totale di 7 classi;
- Le *classi secondarie*, che rappresentano l'argomento specifico della domanda, dipendono dalla classe principale e sono 33 in totale. A seconda della classe principale, il numero di classi secondarie varia.

Il numero ristretto di classi di domande ha permesso di creare una suddivisione più bilanciata tra le classi, e di ottenere un dataset generalmente più equilibrato.

Classe	**Scopo**	**Numero di Esempi**
transition	Domande che riguardano le transizioni tra gli stati	77
automaton	Domande che riguardano l'automa in generale	48
state	Domande che riguardano gli stati dell'automa	48
grammar	Domande che riguardano la grammatica riconosciuta dall'automa	33
theory	Domande di teoria generale sugli automi	15
start	Domande che avviano l'interazione con il sistema	6
off_topic	Domande non pertinenti al dominio che il sistema deve saper gestire	2

Tabella 2: Classi principali del dataset.

Come è possibile notare dalle tabelle che seguono, alcune classi secondarie contengono un numero esiguo di esempi, non sufficiente per una classificazione affidabile.

Sottoclassi	**Scopo**	**Numero di Esempi**
description	Descrizioni generali sull'automa	14
description_brief	Descrizione generale (breve) sull'automa	10
directionality	Domande riguardanti la direzionalità o meno dell'intero automa	1
list	Informazioni generali su nodi e archi	1
pattern	Presenza di pattern particolari nell'automa	9
representation	Rappresentazione spaziale dell'automa	13

Tabella 3: Le 6 classi secondarie del dataset per la classe primaria dell'**automa**.

Sottoclassi	Scopo	Numero di Esempi
count	Numero di transizioni	10
cycles	Domande riguardo anelli tra nodi	4
description	Descrizioni generali sugli archi	2
existence_between	Esistenza di un arco tra due nodi	12
existence_directed	Esistenza di un arco da un nodo a un altro	9
existence_from	Esistenza di un arco uscente da un nodo	18
existence_into	Esistenza di un arco entrante in un nodo	1
input	Ricezione di un input da parte di un nodo	1
label	Indicazione di quali archi hanno una certa etichetta	4
list	Elenco generico degli archi	15
self_loop	Esistenza di self-cycles	1

Tabella 4: Le 11 classi secondarie del dataset per la classe primaria delle **transizioni**.

Sottoclassi	Scopo	Numero di Esempi
count	Numero di stati	19
details	Dettagli specifici su uno stato	1
list	Elenco generale degli stati	1
start	Qual è lo stato iniziale	8
final	Esistenza di uno stato finale	7
final_count	Numero di stati finali	2
final_list	Elenco degli stati finali	3
transitions	Connessioni tra gli stati	8

Tabella 5: Le 8 classi secondarie del dataset per la classe primaria degli **stati**.

Sottoclassi	Scopo	Numero di Esempi
accepted	Grammatica accettata dall'automa	14
example_input	Input di esempio accettato dall'automa	4
regex	Regular expression corrispondente all'automa	2
simulation	Simulazione dell'automa con input dell'utente	8
symbols	Simboli accettati dalla grammatica	7
validity	Validità di un input fornito	2
variation	Richiesta di simulazione su un automa modificato	2

Tabella 6: Le 7 classi secondarie del dataset per la classe primaria della **grammatica**.

Data Augmentation

Come evidenziato nella sezione precedente, diverse classi secondarie contengono un numero esiguo di esempi, non sufficiente per una buona classificazione in seguito al fine-tuning.

Avendo solo 229 esempi, ho arricchito i dati con ulteriori domande generate automaticamente e manualmente. Ho aggiunto un totale di 525 domande, con la seguente distribuzione:

Classe	Numero di esempi aggiuntivi
transition	148
automaton	93
state	56
grammar	111
theory	100
start	17
off_topic	100

Le domande off-topic aggiuntive sono state estratte dal dataset SQUAD[3] v2 [9], [10], per avere una sufficiente varietà di domande non pertinenti.

Anche le classi secondarie hanno ricevuto alcune migliorie alla distribuzione, che rimane comunque ancora sbilanciata: (**description**: 74, **accepted**: 57, **existence_from**: 42, **count**: 40, **generic**: 39, **list**: 38, **label**: 36, **transitions**: 34, **pattern**: 27, **existence_between**: 25, **existence_directed**: 21, **final**: 21, **simulation**: 20, **variation**: 19, **greet**: 19, **representation**: 19, **states**: 18, **existence_into**: 17, **definition**: 16,

[3]Stanford Question Answering Dataset

description_brief: 16, **start**: 15, **symbols**: 14, **validity**: 14, **cycles**: 12, **details**: 12, **input**: 12, **self_loop**: 11, **example_input**: 11, **regex**: 9, **final_count**: 8, **off_topic**: 6, **final_list**: 6, **optimization**: 6, **deterministic**: 5, **reachability**: 5, **start_final**: 3, **dead**: 3, **directionality**: 2, **image**: 2).

Nonostante lo sbilanciamento, è stato possibile ottenere dei buoni risultati in seguito al fine-tuning.

L'utilizzo del dataset SQUAD ha anche introdotto un'ulteriore incremento delle performance, portando a una diminuzione dell'erronea classificazione di esempi off-topic come domande lecite. In particolare, le metriche di entropia e confidenza durante il fine tuning sono migliorate rispettivamente del 17 e del 7%.

2.2.2 Fine-tuning

Per poter utilizzare le Large Language Models (LLM) per la classificazione di intenti, ho dovuto seguire un processo di fine-tuning.

Il fine-tuning avviene verso la fine della preparazione di un modello di machine learning. In particolare, è la fase in cui si prende un modello pre-addestrato su un compito generale (o su una grande quantità di dati non etichettati) e lo si "specializza" su un compito specifico, come la classificazione di intenti, l'analisi del sentiment o il riconoscimento di entità nominate.
Si parte quindi da un modello che possiede già una buona conoscenza linguistica di base (perché allenato, ad esempio, su quantità imponenti di testo come Wikipedia, libri o pubblicazioni) e lo si ri-addestra su un dataset mirato, così da fargli apprendere le particolarità e le sfumature del nuovo scenario applicativo.

Sul piano tecnico, il processo di fine-tuning si fonda sugli stessi principi del *learning by example*: si forniscono al modello coppie di input e output (nel caso di una classificazione, l'output è la classe corretta), e si calcola la loss (ad esempio la cross-entropy tra le probabilità previste dal modello e quelle desiderate).
Tramite la retropropagazione dell'errore, i pesi del modello vengono aggiornati iterativamente, così da allineare le predizioni alle etichette reali. Il risultato è che, dopo un numero sufficiente di iterazioni (o epoche), il modello impara a predire con buona approssimazione la classe corretta anche per esempi non ancora visti.

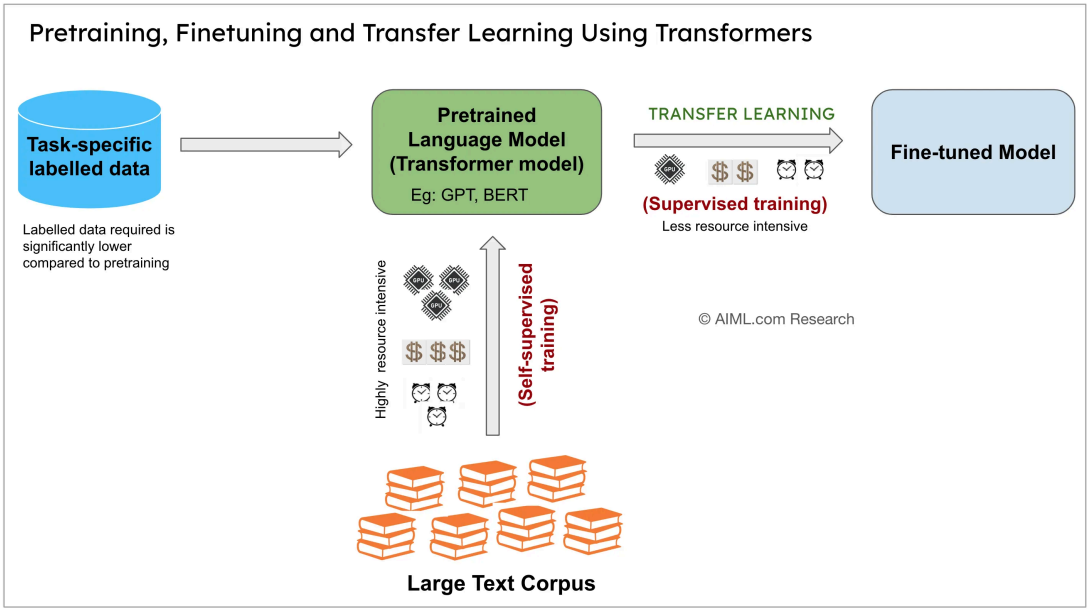

Figura 1: Processo di fine-tuning di un modello di LLM. **IMMAGINE DA SOSTITUIRE**

L'elemento distintivo del fine-tuning rispetto a un addestramento "da zero" (o from scratch) sta nel fatto che la maggior parte dei pesi del modello non parte da valori iniziali casuali, bensì da un punto in cui il modello ha già "appreso" molte regole e pattern del linguaggio. Se nel pre-addestramento ha appreso, ad esempio, la nozione di contesto, la correlazione fra parole vicine e la loro valenza semantica, durante il fine-tuning deve semplicemente specializzarsi nel riconoscere come queste informazioni si combinano per risolvere il compito target. Questo riduce drasticamente la quantità di dati e di risorse computazionali necessarie a raggiungere buone prestazioni.

Nel caso di una classificazione testuale multi-classe, si aggiunge in genere un piccolo strato di output (o head) in cima al modello pre-addestrato. La testa è una semplice rete feed-forward, spesso costituita da uno o due livelli di neuroni, che produce un vettore di dimensione pari al numero di possibili etichette. Il resto del modello rimane pressoché invariato: l'architettura interna, come i vari encoder o layer del Transformer, resta la stessa, ma i loro pesi continuano ad aggiornarsi durante il training, almeno in un contesto standard (è anche possibile, in alcuni scenari, "congelare" i primi strati e addestrare solo quelli finali, in base a considerazioni di efficienza e dimensione del dataset).

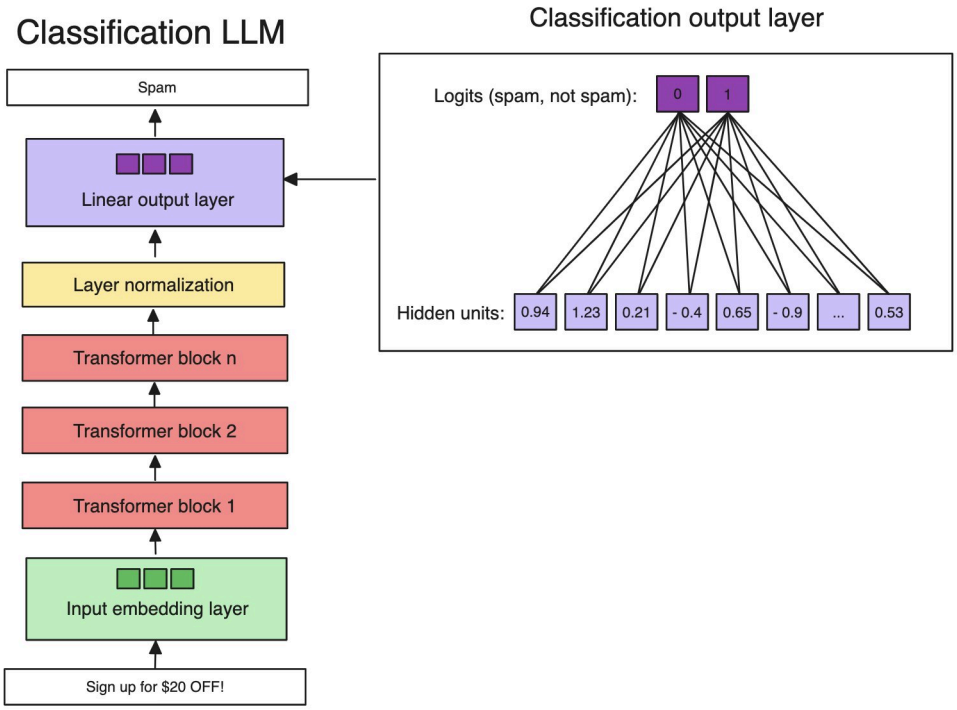

Figura 2: Struttura di un modello di classificazione basato su LLM. **IMMAGINE DA SOSTITUIRE**

2.2.3 BERT

Prima di illustrare più nel dettaglio il fine-tuning, è utile introdurre BERT[4] [11], progenitore della famiglia di modelli da me utilizzati per la sperimentazione, nonchè uno dei modelli più noti e influenti degli ultimi anni in ambito Natural Language Processing.

BERT è stato proposto nel 2018 da J. Devlin, M.-W. Chang, K. Lee, e K. Toutanova [11] come un sistema capace di apprendere rappresentazioni contestuali del testo in modo bidirezionale, basandosi sull'architettura Transformer introdotta in precedenza da A. Vaswani *et al.* [12].

L'idea portante di BERT è quella di addestrare un modello neurale a predire, data una sequenza testuale, le parole mascherate (ovvero rimosse o sostituite) e la relazione tra frasi adiacenti. Queste due tecniche di pre-addestramento vengono rispettivamente chiamate Masked Language Modeling e Next Sentence Prediction.

Nel Masked Language Modeling, BERT maschera casualmente alcune parole del testo in input e chiede al modello di indovinare quali fossero, costringendolo così a sviluppare una comprensione profonda del contesto circostante.

Nel Next Sentence Prediction, invece, il modello riceve in ingresso due frasi (A e B) e impara a classificare se B segue effettivamente A o se le due frasi appartengono a contesti disgiunti. Addestrando in parallelo su questi due compiti, BERT acquisisce

[4]Bidirectional Encoder Representations from Transformers

rappresentazioni interne che colgono sfumature sintattiche, semantiche e relazionali del linguaggio [11].

Una volta pre-addestrato su grandi corpora di testo (come Wikipedia ed estrazioni di libri), BERT può essere facilmente "specializzato" per vari task supervisionati, tra cui la classificazione di testi, l'analisi del sentiment, il question answering e, in generale, tutto ciò che riguarda la comprensione del linguaggio naturale, essendo un modello encoder. La peculiarità di BERT è che, essendo già addestrato a livello linguistico di base, necessita di meno esempi per ottenere risultati spesso notevoli su compiti altamente specializzati.

Esistono diverse varianti del modello, in termini di dimensioni e capacità. Le versioni più comuni sono `BERT-base` e `BERT-large`, differenziate per numero di livelli (encoder) e di parametri totali.
In generale, la versione `base` è più rapida e ha requisiti meno elevati in termini di memoria, mentre la versione large offre performance maggiori a fronte di tempi di calcolo e requisiti hardware superiori.

Nella libreria di Huggingface `transformers` [13], BERT è messo a disposizione come un modello pretrained, pronto per essere caricato e ulteriormente addestrato. In un contesto di classificazione di intenti, ad esempio, si può utilizzare `AutoModelForSequenceClassification` specificando il checkpoint "bert-base-uncased" (o simili).

Un esempio di codice di inizializzazione è il seguente:

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer

model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(model_name,
num_labels=num_classes)
```

`model` è in grado di elaborare sequenze di token generate dal tokenizer e, una volta fine-tuned, produce come output le probabilità di appartenere alle varie classi (o intenti) da classificare. Questa è la base su cui mi sono appoggiato per la classificazione delle domande del dataset.

2.2.4 Implementazione

In questa sezione sarà presentata la procedura di fine-tuning che ho implementato per addestrare un modello di classificazione di intenti basato su architetture Transformer.
L'intero processo sfrutta principalmente la libreria `transformers` di Huggingface [13], in combinazione con altri strumenti sempre dell'ecosistema FOSS[5] di Huggingface, come `datasets`.

[5]Free and Open Source Software, cioè Software **Libero** e Open Source

L'utilizzo di queste librerie permette di semplificare notevolmente il processo di fine-tuning, fornendo API intuitive e funzionalità di alto livello per la gestione dei dati, la creazione dei modelli e la valutazione delle performance. In questo modo è possibile addestrare un modello di classificazione di intenti in poche righe di codice, senza dover scrivere manualmente i loop di training e validation, o implementare da zero la logica di salvataggio e caricamento dei modelli, nonostante questa via sia sempre possibile.

L'obiettivo è riutilizzare un modello pre-addestrato (ad esempio BERT, DistilBERT o qualsiasi altro compatibile con AutoModelForSequenceClassification) per specializzarlo nel riconoscimento di specifiche categorie di intenti, e successivamente salvarlo per l'uso nel chatbot.

Preparazione dei dati

Un primo punto cruciale è la preparazione del dataset, gestita dalla funzione `prepare_dataset`. Qui effettuo la suddivisione stratificata tra train e validation, tokenizzo i testi tramite un `AutoTokenizer` e converto le etichette da stringhe a interi, in accordo con la mappatura definita nella classe `LabelInfo`[6].

```
def prepare_dataset(df: DataFrame,
                    tokenizer: PreTrainedTokenizer,
                    label_info: LabelInfo,
                    examples_column: str,
                    labels_column: str) -> tuple[Dataset, Dataset]:
  """
  Prepares the dataset for training and evaluation by tokenizing the text
  and encoding the labels.
  """
  def tokenize_and_label(example: dict) -> BatchEncoding:
      question = example[examples_column]
      encodings = tokenizer(question, padding="max_length", truncation=True,
max_length=128)
      label = label_info.get_id(example[labels_column])
      encodings.update({'labels': label})
      return encodings

  split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
  train_index, val_index = next(split.split(df, df[labels_column]))
  strat_train_set = df.iloc[train_index].reset_index(drop=True)
  strat_val_set = df.iloc[val_index].reset_index(drop=True)

  train_dataset = Dataset.from_pandas(strat_train_set)
  eval_dataset = Dataset.from_pandas(strat_val_set)

  train_dataset = train_dataset.map(tokenize_and_label,
remove_columns=train_dataset.column_names)
  eval_dataset = eval_dataset.map(tokenize_and_label,
remove_columns=eval_dataset.column_names)

  return train_dataset, eval_dataset
```

Codice 6: Funzione per la preparazione del dataset.

[6]Si veda l'appendice per la completa definizione.

In questo modo, ottengo due oggetti di tipo `Dataset` che rappresentano il training set e il validation set. Ciascun esempio è stato trasformato in una struttura pronta per essere gestita dal `Trainer` di Huggingface, con un campo `labels` che indica la classe corretta da apprendere.

Una volta create e preparate queste componenti (funzione di metriche, funzioni di training, dataset tokenizzato), eseguo il fine-tuning chiamando `run_fine_tuning` (presentata poco più avanti).

Metriche di valutazione

Per prima cosa, ho definito una funzione in grado di calcolare le metriche di valutazione, che permetteranno di valutare le performance del modello in fase di fine-tuning in modo automatico.

Ho scelto di considerare **accuratezza**, **precision**, **recall** e **F1** come indicatori classici di performance; in aggiunta, calcolo anche l'**entropia media** e la **confidenza media**, allo scopo di misurare rispettivamente il grado di incertezza delle previsioni e la probabilità media associata alla classe predetta. Lo snippet seguente mostra la funzione `compute_metrics`:

```
    def compute_metrics(eval_pred):
      """
      Compute evaluation metrics for the model predictions.
      """
      predictions, labels = eval_pred
      probabilities = np.exp(predictions) / np.sum(np.exp(predictions), axis=1,
keepdims=True)
      preds = np.argmax(probabilities, axis=1)

      acc = accuracy_score(labels, preds)
      precision, recall, f1, _ = precision_recall_fscore_support(labels, preds,
average='weighted', zero_division=0)

      entropies = entropy(probabilities.T)
      avg_entropy = np.mean(entropies)
      avg_confidence = np.mean(np.max(probabilities, axis=1))

      metrics = {
          'accuracy': acc,
          'precision': precision,
          'recall': recall,
          'f1': f1,
          'avg_entropy': avg_entropy,
          'avg_confidence': avg_confidence,
      }
      return metrics
```

Codice 7: Funzione per il calcolo delle metriche di valutazione.

Può essere utile soffermarci un momento a spiegare le metriche scelte:

L'**accuratezza** (o tasso di classificazione corretta) misura la proporzione di esempi classificati correttamente, senza distinzione tra le varie classi. Formalmente:

$$\text{Accuracy} = \frac{1}{N}\sum_{i=1}^{N}\{\hat{y}_i = y_i\} \tag{1}$$

dove $\{\hat{y}_i = y_i\}$ vale 1 se la previsione è corretta, 0 altrimenti. Più il valore è vicino a 1, migliore è la performance complessiva del modello.

Quando si lavora con problemi di classificazione con etichette binarie, o si valuta ciascuna classe indipendentemente, esistono alcuni conteggi che possono essere utili per valutare la qualità delle previsioni:

- i **true positives** (TP) indicano i casi in cui il modello ha predetto correttamente la classe positiva;
- i **false positives** (FP) indicano i casi previsti come positivi dal modello, ma che in realtà sono negativi;
- i **false negatives** (FN) i casi previsti negativi ma in realtà positivi.

Sulla base di queste definizioni, si introducono due metriche fondamentali:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2}$$

che indica la percentuale di esempi classificati come positivi che erano effettivamente positivi.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3}$$

stima la quota di esempi positivi che sono stati effettivamente riconosciuti come tali dal modello.

L'**F1-score** fornisce una media armonica fra Precision e Recall, combinando entrambe le metriche in un singolo indice:

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

Un F1 score alto richiede che entrambe le metriche siano elevate; se una delle due è bassa, il valore di F1 tende drasticamente a ridursi. Questo lo rende particolarmente utile in casi di class imbalance o quando è importante non trascurare né la precisione né la capacità di recuperare tutti i positivi.

L'**entropia** è una misura della disordine o incertezza di un sistema, in questo caso delle previsioni del modello.

Per un singolo esempio, se il modello produce una distribuzione di probabilità $\boldsymbol{p}_i = \big(p_{i,1}, ..., p_{i,k}\big)$ sulle k classi, è possibile calcolare l'entropia dell'esempio come

$$H(\boldsymbol{p}_i) = -\sum_{j=1}^{k} p_{i,j} \log(p_{i,j}) \tag{5}$$

.

Tale quantità esprime quanto "incerte" sono le previsioni del modello: se il modello assegna un'alta probabilità a una sola classe e bassa probabilità alle altre, l'entropia tende a essere prossima a zero (predizione più "sicura"); se distribuisce le probabilità in modo pressoché uniforme, l'entropia aumenta (maggiore incertezza).

L'entropia media su tutto il set di validazione di dimensione NN è:

$$\text{Average Entropy} = \frac{1}{N} \sum_{i=1}^{N} H(p_i) \tag{6}$$

Un valore basso di entropia media indica che, in media, le previsioni del modello sono piuttosto concentrate su una specifica classe; un valore più alto suggerisce che il modello sia spesso incerto.

Sempre definita a partire dalla distribuzione $\boldsymbol{p}_i$, la confidenza per il singolo esempio i può essere definita come la probabilità associata alla classe di output che ha la confidenza massima:

$$C(\boldsymbol{p}_i) = \max_j p_{i,j} \tag{7}$$

Maggiore è il valore di $C(\boldsymbol{p}_i)$, più il modello risulta "sicuro" di quella predizione. Analogamente, la confidenza media sul dataset si calcola come:

$$\text{Average Confidence} = \frac{1}{N} \sum_{i=1}^{N} \max_j p_{i,j} \tag{8}$$

Un valore prossimo a 1 indica che, spesso, il modello prende decisioni molto nette; un valore più basso può rivelare maggiore cautela o incertezza.

Usata congiuntamente all'entropia media, la confidenza media può fornire indicazioni interessanti su come il modello pesa le varie classi e quanto tende a "sbilanciarsi" sulle previsioni.

Addestramento

Per effettuare l'addestramento vero e proprio, ho definito anche la funzione `run_fine_tuning`, che si fa carico di gestire i parametri di training (come numero di epoche, learning rate, batch size), di configurare gli strumenti di logging e salvataggio, e di lanciare effettivamente il training tramite la classe `Trainer` della libreria `transformers`.

La classe `Trainer` semplifica notevolmente la gestione di molteplici aspetti, come la schedulazione del learning rate o la stratificazione della validazione.

```python
def run_fine_tuning(model: AutoModelForSequenceClassification,
                    tokenizer: AutoTokenizer,
                    train_dataset: Dataset,
                    eval_dataset: Dataset,
                    wandb_mode: str,
                    num_train_epochs=20) -> Trainer:
  """
  Fine-tunes a pre-trained model on the provided training dataset and evaluates it
on the evaluation dataset.
  """

  train_dataset = train_dataset.map(lambda x: {k: v.float() if isinstance(v,
torch.Tensor) and v.dtype == torch.long else v for k, v in x.items()})
  eval_dataset = eval_dataset.map(lambda x: {k: v.float() if isinstance(v,
torch.Tensor) and v.dtype == torch.long else v for k, v in x.items()})

  report_to = ["wandb"] if wandb_mode == "online" else None

  training_args = TrainingArguments(
      output_dir='./temp',  # Directory to save the model and other outputs
      num_train_epochs=num_train_epochs,  # Number of training epochs
      learning_rate=2e-5,  # Learning rate for the optimizer
      warmup_ratio=0.1,  # Warmup for the first 10% of steps
      lr_scheduler_type='linear',  # Linear scheduler
      per_device_train_batch_size=16,  # Batch size for training
      per_device_eval_batch_size=16,  # Batch size for evaluation
      save_strategy='epoch',  # Save the model at the end of each epoch
      logging_strategy='epoch',  # Log metrics at the end of each epoch
      eval_strategy='epoch',  # Evaluate the model at the end of each epoch
      logging_dir='./temp/logs',  # Directory to save the logs
      load_best_model_at_end=True,  # Load the best model at the end based on
evaluation metric
      metric_for_best_model='f1',  # Use subtopic F1-score to determine the best
model
      greater_is_better=True,  # Higher metric indicates a better model
      save_total_limit=1,  # Limit the total number of saved models
      save_only_model=True,  # Save only the model weights
      report_to=report_to,  # Report logs to Wandb if mode is "online"
  )

  trainer = Trainer(
      model=model,  # The model to be trained
      args=training_args,  # Training arguments
      train_dataset=train_dataset,  # Training dataset
      eval_dataset=eval_dataset,  # Evaluation dataset
      processing_class=tokenizer,  # Tokenizer for processing the data
      compute_metrics=compute_metrics  # Function to compute evaluation metrics
  )

  print(f"Trainer is using device: {trainer.args.device}")

  trainer.train()  # Start the training process

  return trainer
```

Il metodo mostra diverse impostazioni interessanti:

- `load_best_model_at_end=True` consente di caricare automaticamente al termine dell'addestramento i pesi del modello con il miglior valore di F1 (impostato in metric_for_best_model='f1');
- `warmup_ratio=0.1` configura un periodo iniziale di warm-up, durante il quale il learning rate cresce gradualmente prima di stabilizzarsi nella fase successiva. Questo contribuisce a rendere l'ottimizzazione più stabile ed evitare picchi di aggiornamento eccessivi nelle primissime iterazioni.
  La configurazione del warmup, assieme alla learning rate sono state scelte basandomi sull'utilissimo paper di M. Mosbach, M. Andriushchenko, e D. Klakow [14] che fornisce una guida pratica per il fine-tuning di BERT.
- `metric_for_best_model='f1'` indica che il modello migliore sarà scelto in base al valore di F1, calcolato dalla funzione `compute_metrics`. F1 torna utile in quanto è in grado di bilanciare le due metriche di precision e recall, fornendo un'indicazione complessiva delle performance del modello.

Un'ultima considerazione molto importante riguarda il parametro `report_to`, che consente di specificare a quali servizi di logging inviare i risultati del training.
Nel mio caso, ho scelto di utilizzare `wandb`[7] in modalità online, in modo da poter monitorare in tempo reale le performance del modello durante il fine-tuning.

La quasi totalità dei dati mostrati in questo documento sono stati raccolti tramite Wandb, riducendo enormemente il tempo necessario per l'analisi e la visualizzazione dei risultati: il salvataggio automatico ad ogni run e la possibilità di confrontare run diversi in un'unica dashboard sono state funzionalità fondamentali per la mia sperimentazione.

2.2.5 Valutazione e performance

2.3 Riconoscimento delle entità

2.3.1 NER e Slot-filling

2.3.2 Spacy

2.3.3 Valutazione e performance

[7] Wandb, o Weights and Biases, è un servizio di monitoraggio e logging per l'addestramento di modelli di machine learning

3 Natural Language Generation

3.1 Generazione di risposte tramite LLM

3.1.1 Parafrasi

3.1.2 Prompting

3.2 Data Retrieval

3.2.1 Retrieval tramite query

3.2.2 Retrieval basato su script

3.2.3 Retrieval automatico guidato dagli LLM

3.3 Qualità delle risposte

3.3.1 Valutazione automatica

3.3.2 Valutazione umana

4 Ingegnerizzazione

4.1 Composizione del sistema

4.2 Compilatore

4.2.1 Pipeline

4.3 Runner

Bibliografia

[1] A. M. TURING, «I.—COMPUTING MACHINERY AND INTELLIGENCE», *Mind*, vol. 59, fasc. 236, pp. 433–460, 1950, doi: 10.1093/mind/LIX.236.433.

[2] «Artificial Intelligence Markup Language». [Online]. Disponibile su: http://www.aiml.foundation/doc.html

[3] R. Wallace, «The anatomy of A.L.I.C.E», 2009, pp. 181–210. doi: 10.1007/978-1-4020-6710-5_13.

[4] *pandas - Python Data Analysis Library*. [Online]. Disponibile su: https://pandas.pydata.org/

[5] *Ollama - LLM local runner*. [Online]. Disponibile su: https://github.com/ollama/ollama

[6] G. Team *et al.*, «Gemma 2: Improving Open Language Models at a Practical Size», 2024. [Online]. Disponibile su: https://arxiv.org/abs/2408.00118

[7] A. Grattafiori *et al.*, «The Llama 3 Herd of Models». [Online]. Disponibile su: https://arxiv.org/abs/2407.21783

[8] J. Bai *et al.*, «Qwen Technical Report». [Online]. Disponibile su: https://arxiv.org/abs/2309.16609

[9] P. Rajpurkar, J. Zhang, K. Lopyrev, e P. Liang, «SQuAD: 100,000+ Questions for Machine Comprehension of Text», in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, J. Su, K. Duh, e X. Carreras, A c. di, Association for Computational Linguistics, nov. 2016, pp. 2383–2392. doi: 10.18653/v1/D16-1264.

[10] P. Rajpurkar, R. Jia, e P. Liang, «Know What You Don't Know: Unanswerable Questions for SQuAD», in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, I. Gurevych e Y. Miyao, A c. di, Association for Computational Linguistics, lug. 2018, pp. 784–789. doi: 10.18653/v1/P18-2124.

[11] J. Devlin, M.-W. Chang, K. Lee, e K. Toutanova, «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». [Online]. Disponibile su: https://arxiv.org/abs/1810.04805

[12] A. Vaswani *et al.*, «Attention Is All You Need». [Online]. Disponibile su: https://arxiv.org/abs/1706.03762

[13] T. Wolf *et al.*, «HuggingFace's Transformers: State-of-the-art Natural Language Processing». [Online]. Disponibile su: https://arxiv.org/abs/1910.03771

[14] M. Mosbach, M. Andriushchenko, e D. Klakow, «On the Stability of Fine-tuning BERT: Misconceptions, Explanations, and Strong Baselines». [Online]. Disponibile su: https://arxiv.org/abs/2006.04884