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

CO-RELATORI
Pier Felice Balestrucci
Michael Oliverio

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

NLU mediante classificazione, data annotation, data augmentation, data retrieval, NLG basata su LLM, software engineering

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

L'implementazione di chatbot basati su AIML (Artificial Intelligence Markup Language) ha rappresentato un primo passo nella formalizzazione delle interazioni uomo-macchina, fornendo una struttura rule-based che permette di rispondere a input testuali tramite pattern di corrispondenza. Questo approccio, sebbene efficace in una grande varietà di contesti, mostra diversi limiti quando si tratta di gestire variabilità linguistica, contesto e scalabilità delle regole.

In questo capitolo, analizzeremo innanzitutto il funzionamento di AIML, illustrandone la sintassi e le proprietà attraverso esempi pratici. Questo ci permetterà di evidenziare le principali criticità del paradigma rule-based, che si riveleranno essere la rigidità nella definizione delle regole e la necessità di una manutenzione manuale delle conoscenze.

Alla luce di queste limitazioni, ci sposteremo verso un approccio più flessibile e adattabile, basato su sistemi neurali per la classificazione degli intenti. In particolare, esploreremo come tali modelli possano essere strutturati in modo da emulare un comportamento simile a un albero decisionale, capace di generalizzare le richieste degli utenti senza la necessità di specificare esplicitamente ogni possibile variazione.

Infine, introdurremo brevemente anche il task della Named Entity Recognition (NER) come componente fondamentale per migliorare la comprensione dei messaggi, permettendo di estrarre informazioni strutturate dagli input e affinare ulteriormente il processo decisionale del chatbot.

2.1 Come AIML gestisce la comprensione

Negli anni '90 iniziò a guadagnare popolarità il Loebner Prize [1], una competizione ispirata al Test di Turing [2].
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 [3], [4].
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 [3], 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:

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

Snippet 1: Esempio basilare di una categoria AIML.

In questo caso, 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.

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.

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

Snippet 2: Esempio di utilizzo di wildcard in AIML.

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:

```aiml
<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>
```

Snippet 3: Esempio di utilizzo dei tag `<set>` e `<get>` in AIML.

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

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>
```

Snippet 4: Esempio di utilizzo del tag `<condition>` in AIML.

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>
```

Snippet 5: Esempio di utilizzo del tag `<topic>` in AIML.

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:

[2] Stimulus-Response Artificial Intelligence [3]

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

Snippet 6: Esempio di utilizzo del tag `<srai>` in AIML.

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:

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

Snippet 7: Esempio di utilizzo del tag `<think>` in AIML.

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

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

Snippet 8: Esempio di utilizzo del tag `<that>` in AIML.

In questo caso la regola sarà attivata se la risposta precedente del bot era "VA TUTTO BENE" e l'utente risponde in modo affermativo.

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

```aiml
<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>
```

Snippet 9: Esempio di utilizzo del tag `<random>` in AIML.

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:

```aiml
<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>
```

Snippet 10: Esempio di utilizzo del tag `<learn>` in AIML.

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 [3].

- 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 [4].
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[3] 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 su cui il modello deve imparare 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.

[3] Lo sviluppatore delle regole AIML per un certo progetto

Il dataset è una collezione di circa 700 singole interazioni "botta e risposta" prodotte dagli utenti durante la prima fase di sperimentazione. Metà sono domande, l'altra metà coincide con ciò che il chatbot ha risposto. Sono anche presenti ulteriori metriche e valutazioni qualitative delle interazioni, che però non sono state utilizzate per l'addestramento del modello di classificazione.

Estrazione dei dati

Dovendo addestrare un modello di classificazione, ho provveduto innanzitutto ad estrarre i dati effettivamente a noi necessari. Un piccolo script python che adopera la libreria `pandas` [5] 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")
```

Script 1: Estrazione dei dati dal dataset di interazione.

Estratte le domande, ho 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 ai 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'enorme 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 [6], un sistema per hostare localmente modelli di LLM open source (e in certi casi anche *open-weights*).

2.2.2 Etichettatura automatica del dataset

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."
}
```

Snippet 11: 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 [7];
- *llama 3.1*, sviluppato da Meta AI [8].

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 [7], [8].

Un ulteriore modello, Qwen [9], 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:
  """
]
```

Snippet 12: 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 generico e riutilizzabile.

Segue un estratto di codice python che mostra come è stato effettuato il prompting. Viene importata 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()
```

Script 2: 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
```

Snippet 13: 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, portando a una classificazione poco affidabile. In più, ho realizzato che le classi scelte erano troppo generiche: questo problema 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.

2.2.3 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. Il risultato è stato un dataset con due livelli di classi: le *classi principali* e le *classi secondarie*, che ci permetteranno di classificare le domande come se ci trovassimo in un albero decisionale [10].
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**.

2.2.4 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 scritte manualmente e anche generate artificialmente.

Le domande artificiali sono state prodotte in grandi quantità adoperando diversi modelli disponibili online e locali, tra cui:

- ChatGPT `4o`, `o1` e `o3-mini` [11];
- Llama3.1 [8];
- DeepSeek R1 [12], [13].

Ad ogni modello è stato presentato un insieme di domande con lo stesso topic principale o secondario, assieme al contesto in cui vengono poste e ad una richiesta di produzione di ulteriori domande simili semanticamente. Per maggiore convenienza, è stato richiesto ai modelli di rispondere fornendo le nuove domande formattate in markdown [14].

Dato il grosso volume di risposte, per verificare l'adesione dei modelli alle richieste è stato effettuato un controllo a campione, che non ha evidenziato particolari problematiche nella precisione di nessuno dei modelli.

In totale sono stati aggiunti 851 nuovi quesiti, con la seguente distribuzione:

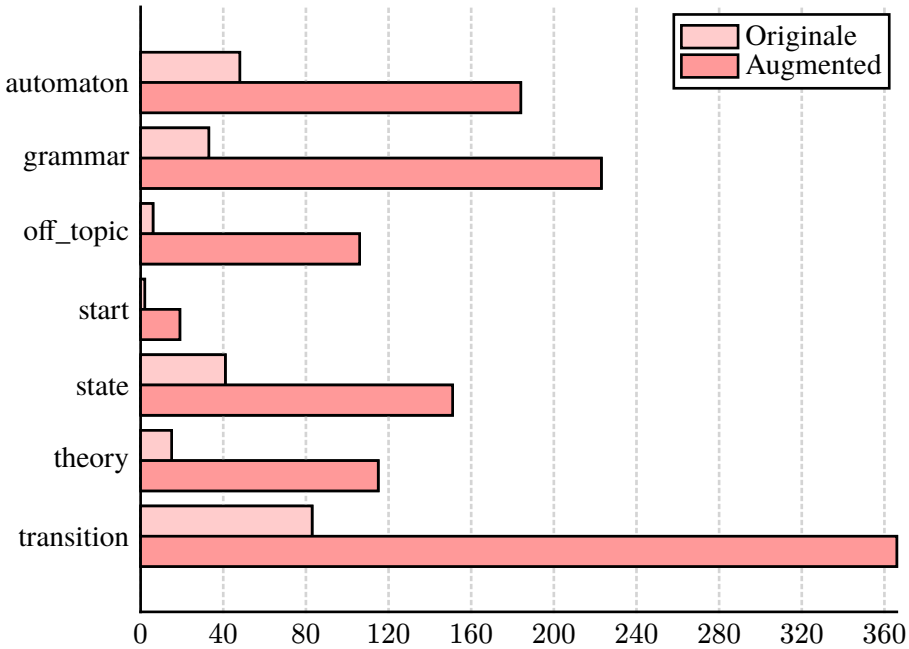


Grafico 1: Distribuzione delle domande originali e generate artificialmente per ogni classe principale.

Le domande off-topic aggiuntive sono state estratte dal dataset SQUAD[4] v2 [15], [16], per avere una sufficiente varietà di domande non pertinenti.

Anche le classi secondarie hanno ricevuto alcune migliorie alla distribuzione, che rimane comunque ancora sbilanciata, com'è possibile vedere nel Grafico 2:

[4]Stanford Question Answering Dataset

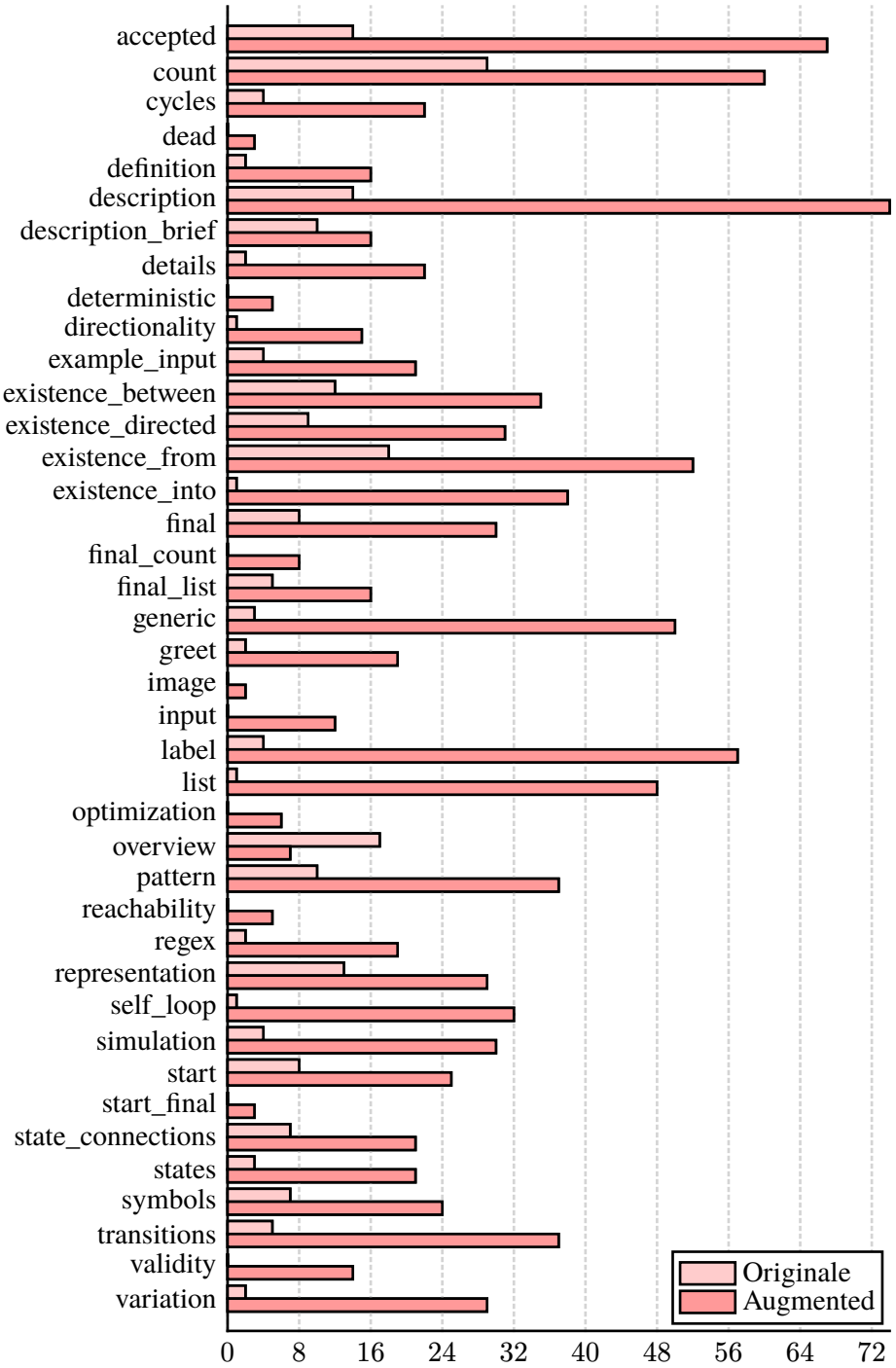


Grafico 2: Distribuzione delle domande originali e generate artificialmente per ogni classe secondaria.

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%.

2.2.5 Fine-tuning

Per poter utilizzare i 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 addestra ulteriormente su un dataset mirato, così da fargli apprendere le particolarità e le sfumature del nuovo scenario applicativo, senza dover ripartire da zero.

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 *backpropagation* 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.

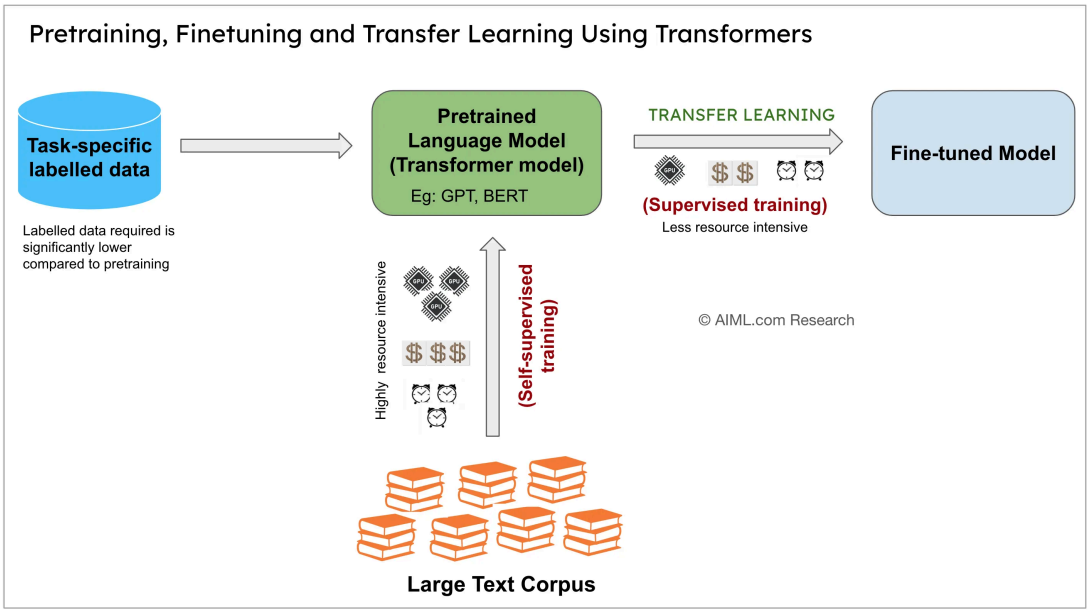


Figura 1: Processo di fine-tuning di un modello di LLM. **IMMAGINE DA SOSTITUIRE**

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).

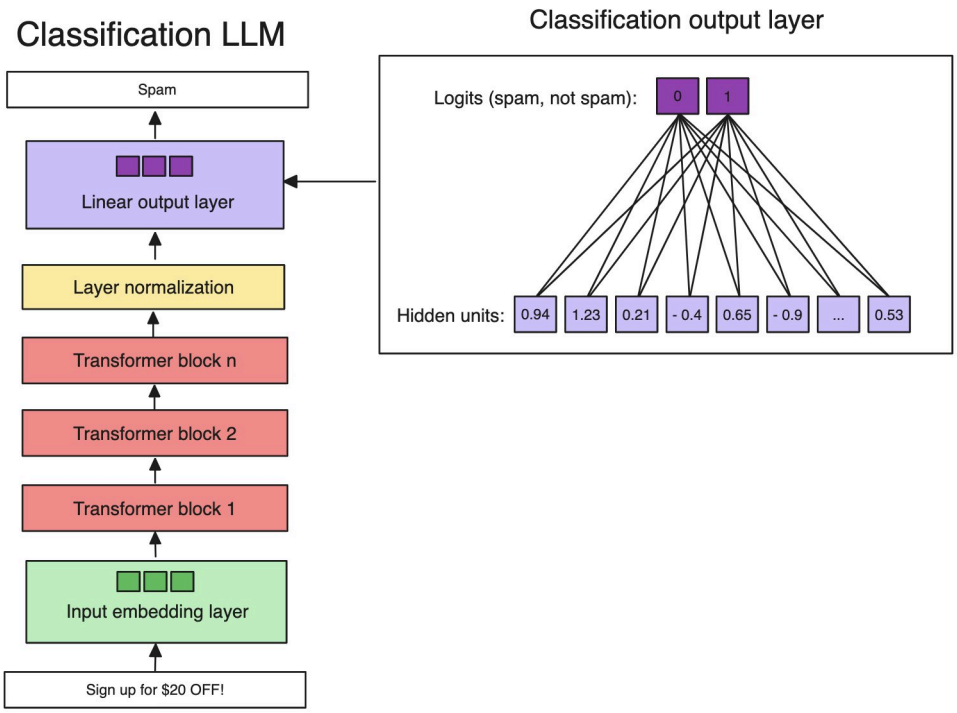


Figura 2: Struttura di un modello di classificazione basato su LLM. **IMMAGINE DA SOSTITUIRE**

2.2.6 BERT

Prima di illustrare più nel dettaglio il fine-tuning, è utile introdurre BERT[5] [17], progenitore della famiglia di modelli da me utilizzati per la sperimentazione, nonchè uno dei modelli più noti e influenti degli ultimi anni nell'ambito del Natural Language Processing.

BERT è stato proposto nel 2018 da J. Devlin, M.-W. Chang, K. Lee, e K. Toutanova [17] 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.* [18].

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.

[5] Bidirectional Encoder Representations from Transformers

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 rappresentazioni interne che colgono sfumature sintattiche, semantiche e relazionali del linguaggio [17].

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` [19], 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)
```

Snippet 14: Inizializzazione di un modello BERT per la classificazione di intenti.

`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.

2.2.7 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 [19],

in combinazione con altri strumenti sempre dell'ecosistema FOSS[6] di Huggingface, come `datasets`.

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 utilizzare un modello pre-addestrato (ad esempio BERT, DistilBERT o qualsiasi altro compatibile con `AutoModelForSequenceClassification`) con lo scopo di 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`[7].

```
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
```

Funzione 1: Funzione per la preparazione del dataset.

[6] Free and Open Source Software, cioè Software **Libero** e Open Source [20], [21]

[7] 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
```

Funzione 2: 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** [22] 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

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

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 N 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.

Il metodo espone diversi parametri significativi:

- `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 [23] 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 fare affidamento a **Weights and Biases**[8] in modalità online, in modo da poter monitorare in tempo reale le performance del modello durante il fine-tuning.

```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.
    """
    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 by 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
```

Funzione 3: Funzione per l'addestramento del modello.

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.

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

Modelli e architettura utilizzate

Tutti i modelli che ho utilizzato per la sperimentazione sono basati su BERT, o ELECTRA [24], entrambi fondati sull'architettura encoder [17].

In particolare, dal repository di Huggingface dedicato ai modelli di classificazione ho deciso di utilizzare:

- `google-bert/bert-base-uncased`, versione da 110 milioni di parametri [25]. Si tratta del modello originale di BERT ideato da Google [17];
- `distilbert/distilbert-base-uncased` [26], versione distillata [27] di BERT, con circa il 40% in meno di parametri [28]. Il modello è il risultato di una operazione dove si addestra un modello più piccolo ad imitare al meglio l'originale;
- `google/mobilebert-uncased` [29], versione di BERT ingegnerizzata con lo scopo di essere eseguibile su dispositivi mobili. Ha un totale di 25 milioni di parametri [30].
- `google/electra-small-discriminator` [31], da 14 milioni di parametri. Questo modello è stato addestrato utilizzando tecniche simili a quelle utilizzate per addestrare le GAN[9] [24], [32]

Tutti i modelli utilizzati sono direttamente adoperabili per i nostri scopi essendo modelli encoder: dato un certo input produrranno una rappresentazione vettoriale o matriciale. Il risultato è successivamente classificabile da una rete feed-forward, restituendo così come risultato la classe più probabile (si veda la Sezione 2.2.5).

Sono state effettuate anche delle sperimentazioni con una variante della normale architettura, dove su un unico encoder vengono addestrati due modelli separati di classificazione, per riconoscere con un'unica esecuzione del modello entrambe le classi della domanda presentata.

L'idea, già utilizzata anche in altri ambiti per il Transfer Learning [33] o direttamente su BERT [34], [35] può permettere di ridurre notevolmente il costo e i tempi di addestramento, oltre ai requisiti di memoria. Infatti, avendo la quasi totalità dei pesi concentrati nei layer del transformer, lo strato finale di classificazione risulta molto "sottile", e richiede una percentuale minima rispetto al resto del modello.

Nel mio caso sfortunatamente l'architettura a doppia testa di classificazione non si è rivelato migliore, con performance in media inferiori del 20% rispetto al miglior modello addestrato finora. Nonostante le performance peggiori, l'utilizzo di un modello del genere può essere considerato in contesti soggetti da forti limiti hardware, come su dispositivi mobili, edge o low-end.

L'intera implementazione fa nuovamente fondamento sull'enorme flessibilità della libreria `transformers`. È stato sufficiente infatti soltanto aggiungere le due classification heads ed estendere il metodo `forward` che si occupa della predizione:

[9] Generative Adversarial Networks, modelli addestrati in coppia, dove uno impara a svolgere un certo compito generativo, e l'altro a riconoscere se un certo esempio presentato è generato o meno.

```python
from torch import nn as nn
from transformers import BertPreTrainedModel, BertModel


class BertForHierarchicalClassification(BertPreTrainedModel):
    def __init__(self, config, num_main_topics, num_subtopics):
        super().__init__(config)
        self.bert = BertModel(config)
        self.classifier_main = nn.Linear(config.hidden_size, num_main_topics)
        self.classifier_sub = nn.Linear(config.hidden_size, num_subtopics)
        self.init_weights()

    def forward(self, input_ids, attention_mask, labels_main=None, labels_sub=None):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = outputs.pooler_output
        logits_main = self.classifier_main(pooled_output)
        logits_sub = self.classifier_sub(pooled_output)

        loss = None
        if labels_main is not None and labels_sub is not None:
            loss_fct = nn.CrossEntropyLoss()
            loss_main = loss_fct(logits_main, labels_main)
            loss_sub = loss_fct(logits_sub, labels_sub)
            loss = loss_main + loss_sub  # Adjust weighting if needed

        return {'loss': loss, 'logits_main': logits_main, 'logits_sub': logits_sub}
```

Classe 1: Estensione di un modello BERT per la classificazione gerarchica in-model.

2.2.8 Valutazione e performance

Come spiegato a pag. 21, per compiere l'addestramento dei modelli è stato essenziale sfruttare metriche di valutazione adeguate, in grado di fornire un quadro completo delle performance del modello.

Iniziamo quindi valutando i risultati dell'addestramento sulla classe principale del dataset:

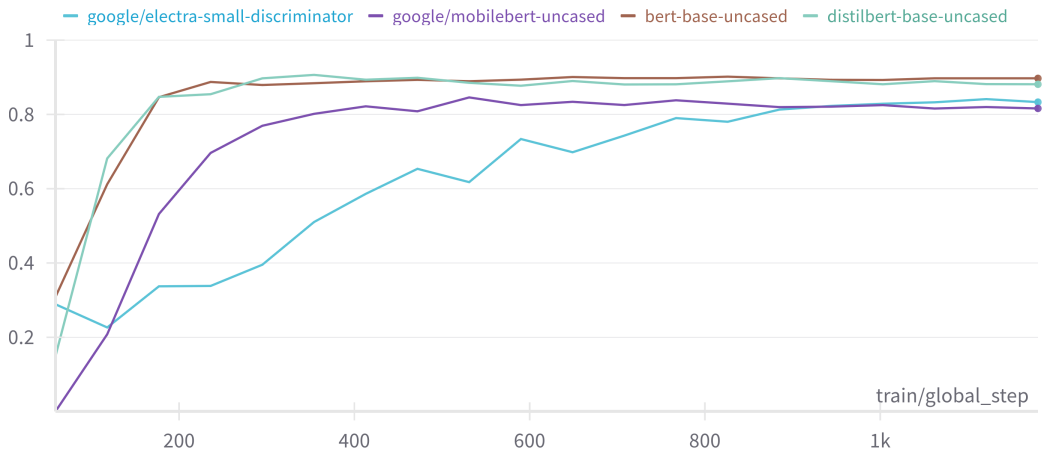


Grafico 3: Confronto delle performance di F1 tra i modelli addestrati.

Come possiamo osservare, le performance crescono man mano che procediamo con il processo di fine tuning, ma si stabilizzano dopo aver visto circa i tre quarti del dataset. I modelli `bert` e `distilbert` terminano l'addestramento con performance pressochè identiche (la differenza

è dello 0.01%), mentre i modelli `mobilebert` e `electra` differiscono di circa l'8% rispetto a `bert`.

Le differenze di performance sono sempre da confrontare considerando anche il tempo di addestramento e la complessità del modello: `electra` ad esempio, pur avendo performance leggermente inferiori, è stato addestrato in meno della metà del tempo rispetto a `bert`.

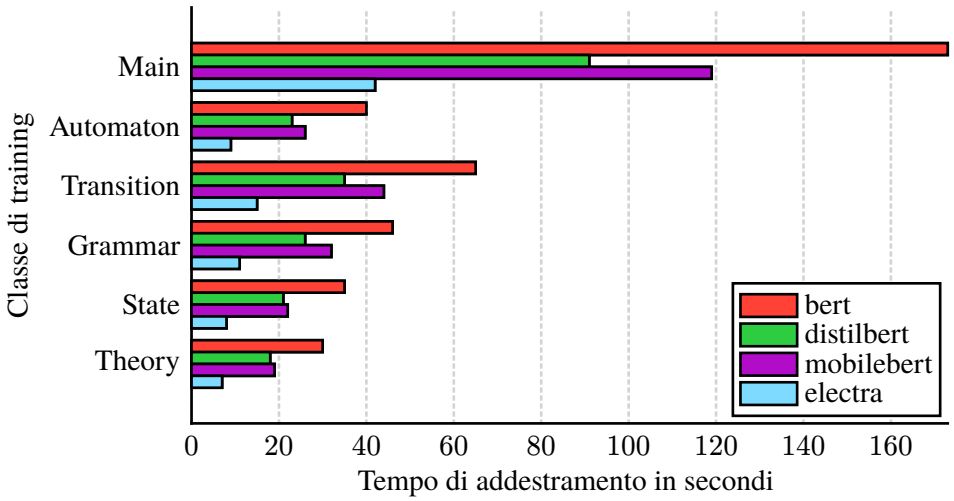


Grafico 4: Confronto dei tempi di addestramento per ciascuna classe di training.

Questo salto nei tempi di addestramento così brusco in realtà porta dei peggioramenti: le sue performance su un test separato mostra risultati peggiori ridotte rispetto agli altri modelli, come possiamo constatare nel Grafico 5. Questo ci ricorda come la scelta del modello non debba essere fatta solo in base alle performance ottenute durante l'addestramento, ma che queste devono essere sempre confermate verificando con un test set separato.

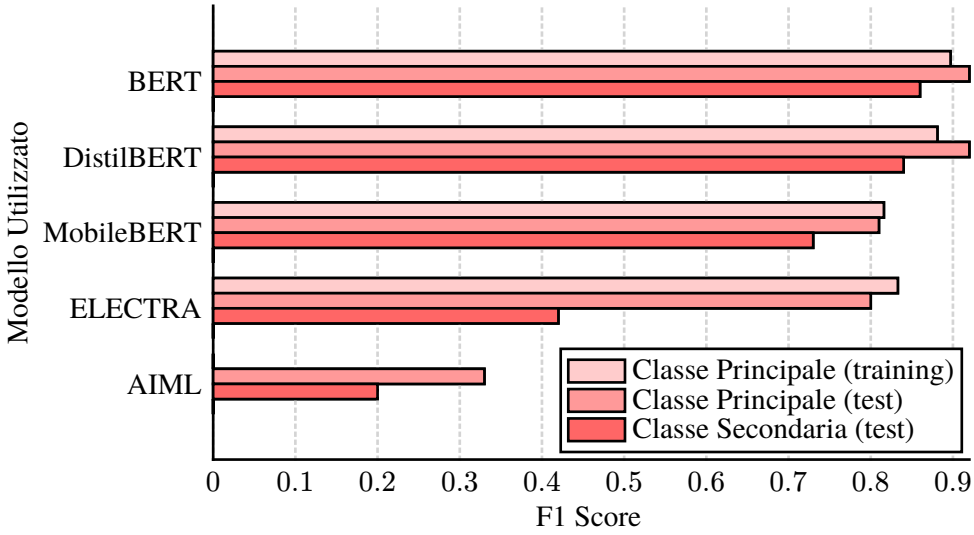


Grafico 5: Confronto delle performance di F1 tra i modelli addestrati.
È anche mostrato il valore di F1 per la classe principale ottenuto durante il fine-tuning dei modelli.

Tutte le valutazioni sono effettuate utilizzando un ulteriore dataset di test, separato dal dataset di training e di validazione, per evitare overfitting e garantire una valutazione imparziale. È composto da 468 ulteriori domande, distribuite in modo da assicurare una verifica sufficiente

su tutte le classi di intenti secondarie, cruciali per la corretta classificazione e per fornire effettivamente risposte utili agli utenti.

Utilizzeremo le performance di AIML come baseline di riferimento per il confronto con gli altri modelli neurali. In seguito alla comparazione delle performance mediante la metrica F1 tra i vari modelli vista in Grafico 5, d'ora in avanti ci concentreremo sulle metriche ottenute con `bert-base-uncased`, il modello più performante tra quelli addestrati.

Per poterlo fare, sfrutteremo le matrici di confusione per valutare le performance dei modelli, in particolare per osservare come si comportano in presenza di classi sbilanciate o di domande ambigue.[10] Siamo interessati a capire se il modello riesce a classificare correttamente domande mai viste; con la matrice di confusione ci aspetteremo di vedere una diagonale principale molto più marcata rispetto agli altri elementi, indicando che il modello è in grado di classificare correttamente la maggior parte delle domande.

Saranno anche presentate le tabelle di valutazione per ciascuna classe, in modo da poter osservare le performance di ciascun modello in modo più dettagliato.

Le metriche presentate nelle tabelle seguenti sono state prodotte utilizzando la funzione `classification_report` della libreria `scikit-learn` [36], che calcola precision, recall e F1 score per ciascuna classe, oltre all'accuracy complessiva.

Il modello AIML, nonostante sia costituito da un numero non indifferente di regole e pattern (103), ha performance mediamente basse, con un F1 score medio del 33% rispetto al 92% di BERT (Tabella 7).

	Performance AIML			**Performance BERT**			
	Precision	Recall	F1-score	Precision	Recall	F1-score	**Esempi**
Automaton	0.52	0.19	0.27	0.93	0.93	0.93	75
Grammar	0.90	0.13	0.23	0.78	0.83	0.81	70
Off-Topic	0.26	0.82	0.39	1.00	0.96	0.98	100
Start	1.00	0.53	0.69	1.00	0.90	0.95	40
State	0.17	0.19	0.18	0.96	1.00	0.98	43
Theory	0.00	0.00	0.00	0.57	0.57	0.57	30
Transition	0.67	0.28	0.40	0.97	0.99	0.98	110
Accuracy			0.35			0.92	468
Macro avg	0.44	0.27	0.27	0.89	0.88	0.88	468
Weighted avg	0.53	0.35	0.33	0.92	0.92	0.92	468

Tabella 7: Risultati delle metriche principali di valutazione per la classificazione del main intent, sia con AIML che con BERT.

Possiamo anche vedere come, dove questo non sia in grado di classificare una certa domanda, finisca col classificarla come off-topic, indicando una certa difficoltà nel riconoscere domande in realtà valide per il nostro dominio (Grafico 6).

[10] Una matrice di confusione è una tabella che mostra il numero di predizioni corrette e incorrette fatte dal modello, confrontando le predizioni con le etichette reali.

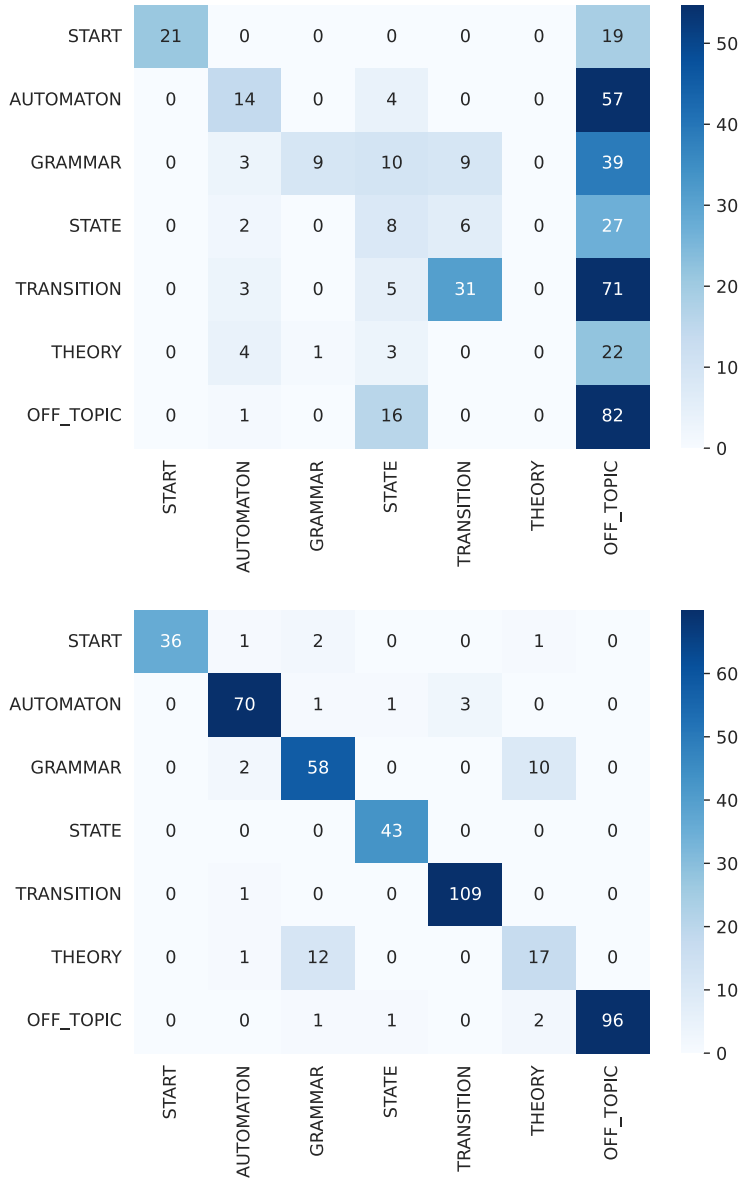


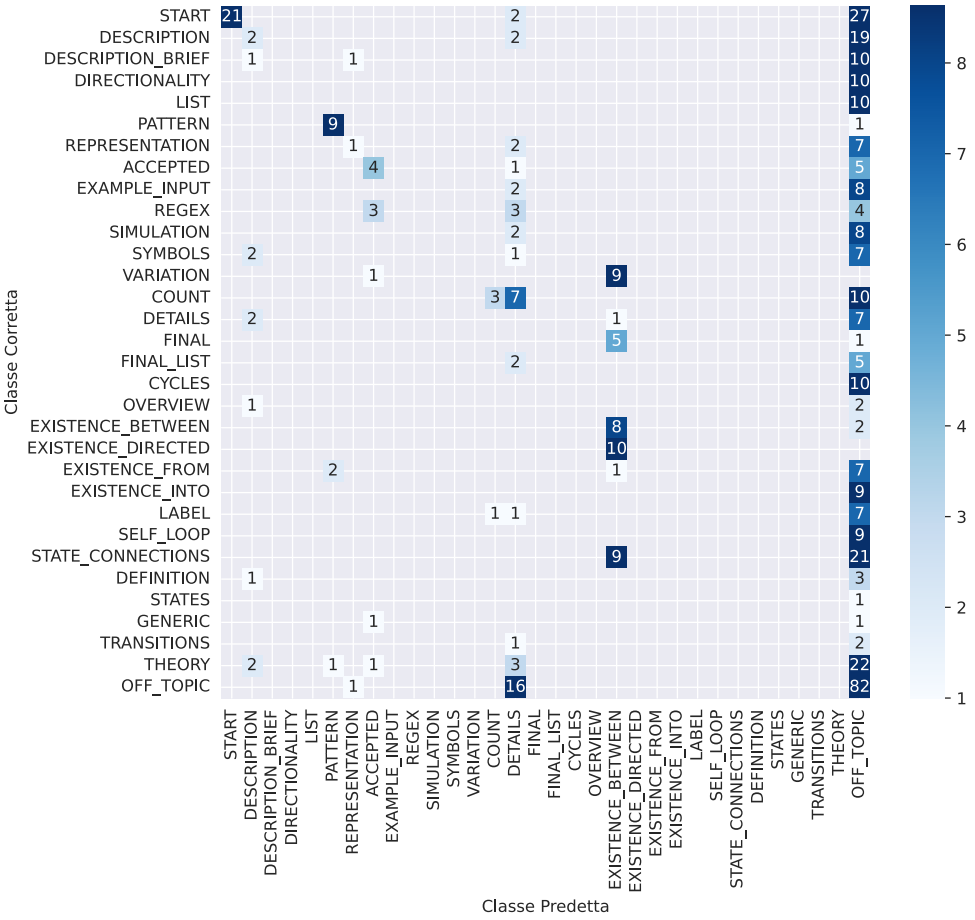
Grafico 6: Matrici di confusione per la classe principale classificata con AIML (sopra) e BERT (sotto). Seguendo una certa riga (classe) possiamo vedere per ogni colonna (classe predetta) quanti esempi sono stati classificati correttamente e quanti no. La diagonale invece indica il numero di esempi classificati senza errori.

Le stesse osservazioni sono applicabili anche alle classi di intenti secondarie (Tabella 8), dove AIML mostra un F1 score medio del 20%, rispetto all'86% di BERT.

Le matrici di confusione confermano le tendenze già presentate dai due modelli per la classe principale, con BERT che mostra una diagonale principale molto marcata (Figura 3). AIML effettivamente riesce a classificare correttamente pochi elementi delle varie classi, mentre BERT mostra una maggiore capacità di generalizzazione.

	Performance AIML			Performance BERT			
	Precision	Recall	F1	Precision	Recall	F1	Esempi
ACCEPTED	0.40	0.40	0.40	0.83	1.00	0.91	10
COUNT	0.75	0.15	0.25	0.95	1.00	0.98	20
CYCLES	0.00	0.00	0.00	1.00	1.00	1.00	10
DEFINITION	0.00	0.00	0.00	0.00	0.00	0.00	4
DESCRIPTION	0.18	0.09	0.12	0.51	0.78	0.62	23
DESCRIPTION_BRIEF	0.00	0.00	0.00	0.60	0.25	0.35	12
DETAILS	0.00	0.00	0.00	1.00	1.00	1.00	10
DIRECTIONALITY	0.00	0.00	0.00	1.00	0.80	0.89	10
EXAMPLE_INPUT	0.00	0.00	0.00	1.00	1.00	1.00	10
EXISTENCE_BETWEEN	0.19	0.80	0.30	0.67	0.60	0.63	10
EXISTENCE_DIRECTED	0.00	0.00	0.00	0.75	0.60	0.67	10
EXISTENCE_FROM	0.00	0.00	0.00	0.80	0.80	0.80	10
EXISTENCE_INTO	0.00	0.00	0.00	0.82	1.00	0.90	9
FINAL	0.00	0.00	0.00	1.00	0.83	0.91	6
FINAL_LIST	0.00	0.00	0.00	0.75	0.86	0.80	7
GENERIC	0.00	0.00	0.00	0.00	0.00	0.00	2
LABEL	0.00	0.00	0.00	0.00	0.00	0.00	9
LIST	0.00	0.00	0.00	1.00	1.00	1.00	10
OFF_TOPIC	0.26	0.82	0.39	1.00	0.96	0.98	100
OVERVIEW	0.00	0.00	0.00	0.50	0.67	0.57	3
PATTERN	0.75	0.90	0.82	1.00	1.00	1.00	10
REGEX	0.00	0.00	0.00	1.00	1.00	1.00	10
REPRESENTATION	0.33	0.10	0.15	1.00	0.70	0.82	10
SELF_LOOP	0.00	0.00	0.00	0.90	1.00	0.95	9
SIMULATION	0.00	0.00	0.00	0.91	1.00	0.95	10
START	1.00	0.42	0.59	0.98	0.92	0.95	50
STATES	0.00	0.00	0.00	0.00	0.00	0.00	1
STATE_CONNECTIONS	0.00	0.00	0.00	1.00	1.00	1.00	30
SYMBOLS	0.00	0.00	0.00	0.70	0.70	0.70	10
THEORY	0.00	0.00	0.00	0.57	0.57	0.57	30
TRANSITIONS	0.00	0.00	0.00	0.00	0.00	0.00	3
VARIATION	0.00	0.00	0.00	0.91	1.00	0.95	10
Accuracy			0.28			0.86	468
Macro avg	0.11	0.11	0.09	0.73	0.73	0.72	468
Weighted avg	0.24	0.28	0.20	0.87	0.86	0.86	468

Tabella 8: Risultati delle metriche di valutazione per la classificazione delle classi secondarie con AIML e BERT.



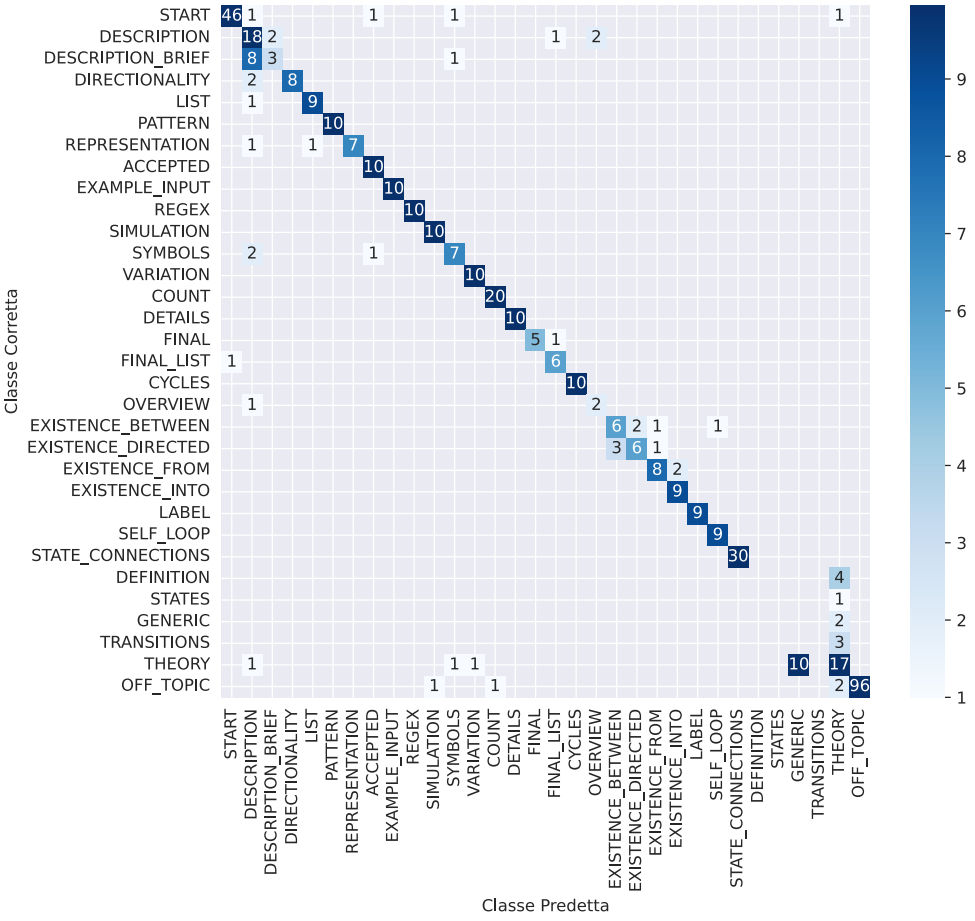


Figura 3: Matrici di confusione per le classi secondarie classificate con AIML (sopra) e BERT (sotto).

Per finire, vediamo anche alcuni esempi del test set etichettati da AIML e Bert. Questi esempi sono stati scelti in modo da mostrare come i due modelli si comportano in un'ipotetica situazione reale:

	Ground Truth		AIML		BERT	
Domanda	Main	Sub	Main	Sub	Main	Sub
Would you display a full list of all components, including nodes and transitions, in the finite state automaton?	AUT	LST	OT	OT	**AUT**	**LST**
Could you summarize the automaton for me?	AUT	DB	OT	OT	**AUT**	**DB**
Could you give me a brief summary of the automaton?	AUT	DB	OT	OT	**AUT**	**DB**
Can you point out the start state in this automaton?	STT	STRT	OT	OT	**STT**	**STRT**
Is a systematic pattern evident in the automata arcs?	AUT	PTT	**AUT**	**PTT**	**AUT**	**PTT**
Can you point out a repetitive pattern among the arcs?	AUT	PTT	**AUT**	**PTT**	**AUT**	**PTT**
I'd like to hear an explanation of how this automaton's states are formed and related.	AUT	DESC	OT	OT	AUT	REP
Would you mind describing the automaton's configuration and links?	AUT	DESC	OT	OT	AUT	REP

In generale, i risultati ottenuti con BERT sono molto soddisfacenti, con performance nettamente superiori rispetto ad AIML. Questo conferma l'efficacia dei modelli neurali per la classificazione di intenti in un contesto di chatbot, e dimostra come l'uso di modelli preaddestrati come BERT possa portare a risultati molto migliori rispetto a soluzioni rule-based.

Non mancano tuttavia degli esempi in cui BERT non riesce a classificare correttamente la domanda, ma in generale il modello mostra una capacità di generalizzazione molto più elevata rispetto ad AIML. Per migliorare le prestazioni, una buona strategia potrebbe essere quella di raccogliere più dati etichettati, in modo da poter addestrare il modello su un dataset più ampio e variegato, cercando di incrementare la copertura delle classi meno frequenti.

2.3 Riconoscimento delle entità

Negli anni Novanta, parallelamente agli studi sull'Intelligenza Artificiale per la realizzazione di sistemi conversazionali rule-based come AIML, si sviluppavano anche nuovi compiti di Natural Language Processing (NLP) orientati all'estrazione di informazioni dal testo in modo più strutturato. Uno dei compiti chiave in questo processo è il Named Entity Recognition (NER), o riconoscimento delle entità nominate.

Nato inizialmente nell'ambito di competizioni e conferenze come le **Message Understanding Conferences** [37], il NER si propose come task cruciale per identificare all'interno di un testo i riferimenti a persone, organizzazioni, luoghi, date e altre categorie, assegnando a ciascuna entità un'etichetta appropriata. Se AIML, per certi versi, si concentra su *che cosa l'utente vuole*

(**intent classification**), il NER si focalizza su *chi o che cosa è menzionato* all'interno di un messaggio o di un documento.

Un semplice esempio può essere la frase "Mario Rossi ieri è stato a Roma per un incontro con Telecom Italia.". Un sistema NER ideale dovrebbe riconoscere:

- Mario Rossi come una **persona** (`PERSON`);
- Roma come un **luogo** (`LOC`);
- Telecom Italia come un'**organizzazione** (`ORG`).

In un chatbot avanzato, questa funzionalità è particolarmente importante perché consente di trasformare testi destrutturati (come messaggi, email o query di ricerca) in informazioni utilizzabili da moduli di analisi successivi.

Consideriamo la possibilità che l'utente chieda a un chatbot che si occupa di automi a stati finiti "Qual è la differenza tra un **automa deterministico** e un **automa non deterministico**?": la NER dovrebbe individuare correttamente "automa deterministico" e "automa non deterministico" come entità rilevanti (anche se meno "classiche" rispetto a persona/luogo/ organizzazione).
In questo modo, il chatbot sa di dover recuperare informazioni specifiche su queste due tipologie di automi, assegnandole poi al modulo incaricato di rispondere alla domanda.

2.3.1 Approcci e metodologie nel NER

La ricerca sul riconoscimento delle entità (Named Entity Recognition) ha attraversato diverse fasi, ognuna caratterizzata da metodologie specifiche e da un livello di "intelligenza" sempre crescente [38]. Inizialmente, i sistemi si basavano su regole statiche o elenchi di entità predefiniti, mentre negli ultimi anni si è passati a tecniche di machine learning via via più complesse, fino ad arrivare ai più recenti modelli neurali basati su architetture di tipo Transformer.

Metodi rule-based o a dizionario

Nella prima fase, molti sistemi NER si affidavano a liste di entità note (chiamate "gazetteer") e a regole linguistiche (pattern o espressioni regolari) per individuare nomi di persone, luoghi, organizzazioni e così via. L'idea di fondo era piuttosto semplice: se una parola compariva in un elenco di nomi propri oppure coincideva con un pattern di stringa (ad esempio iniziale maiuscola, presenza di determinati suffissi), allora veniva etichettata come entità.

Questi approcci erano relativamente facili da implementare e sufficientemente efficaci in un contesto ben definito, purché gli elenchi fossero tenuti costantemente aggiornati. Tuttavia, mostravano rapidamente i loro limiti nel momento in cui si presentavano nomi o entità nuove non inclusi nei dizionari, oppure quando si operava in un dominio estremamente vasto (es. social media) o molto specialistico. In tali casi, l'aggiornamento continuo dei gazetteer e la gestione manuale delle regole si rivelavano complessi e poco scalabili.

Metodi statistici (CRF, HMM, SVM)

Successivamente, con la diffusione del machine learning, si sono affermati approcci statistici in grado di automatizzare gran parte del processo di individuazione e classificazione delle entità. Tra i metodi più noti, spiccano i Conditional Random Fields [39], gli Hidden Markov Models [40] e le Support Vector Machines [41]. Questi algoritmi imparano a riconoscere le entità partendo da un dataset annotato, ossia un corpus di testi in cui ogni parola è già etichettata

come “entità” o “non entità” (con eventuali sotto-categorie quali PERSON, LOCATION, ORGANIZATION, ecc.).

Il vantaggio principale di questo approccio è che i modelli statistici non dipendono più soltanto da elenchi o regole scritte dall’uomo: essi apprendono le regolarità linguistiche e i pattern lessicali (per esempio, la probabilità che un termine che inizia con la maiuscola sia un nome proprio di persona) direttamente dai dati. Per molti anni, questi metodi hanno rappresentato lo stato dell’arte del NER, garantendo performance elevate a fronte di un’adeguata disponibilità di dati annotati.

Modelli neurali

Negli ultimi anni, la scena del NER è stata rivoluzionata dall’avvento di reti neurali, inizialmente di tipo ricorrente (come RNN [42] o LSTM [43], [44]) e, più di recente, di tipo Transformer [18] (ad es. BERT [17], RoBERTa, GPT). L’adozione di embedding per le parole e di meccanismi di attenzione (self-attention) ha permesso di superare molte limitazioni dei metodi precedenti, poiché queste architetture sono in grado di:

- gestire contesti testuali più lunghi in modo efficace;
- catturare la struttura sintattica e il significato semantico delle frasi;
- fornire rappresentazioni linguistiche approfondite in grado di distinguere omonimi e contesti diversi.

In questo modo, anche in domini molto specifici (come quello degli automi a stati finiti, in cui esistono entità specialistiche come “automa deterministico” o “automa non deterministico”), i modelli neurali si sono dimostrati capaci di riconoscere e catalogare con maggiore accuratezza le entità rilevanti [45]. Questa evoluzione ha portato a un vero e proprio salto di qualità nelle prestazioni del NER, consentendo al sistema di operare su testi complessi e ricchi di sfumature senza richiedere il continuo intervento di programmatori o linguisti per aggiornare le regole manuali.

2.3.2 Slot Filling

Mentre il NER si concentra su dove compaiono le entità nel testo e su che tipo di entità si tratti (persona, luogo, organizzazione, ecc.), lo slot-filling rappresenta un’operazione più specifica e spesso orientata al dominio [46], [47]. In altre parole:

- Il NER produce una segmentazione e un’etichettatura generica:
  Mario Rossi → `PERSON`, Roma → `LOC`, Telecom Italia → `ORG`.
- Lo slot-filling prende queste entità (o altre componenti di testo) e le associa a ruoli predefiniti, tipici di una determinata applicazione. Ad esempio, per un chatbot di viaggi potremmo avere:
  - città_di_partenza = “Milano”
  - città_di_arrivo = “Roma”
  - data_viaggio = “2025-03-07”

In alcuni sistemi, il compito di “trovare gli slot” e “riempirli” è integrato in un singolo modello (joint model di intent classification e slot-filling [48]). In altri casi, come in pipeline più complesse, si preferisce separare il passaggio di NER dal passaggio di mapping di dominio (slot-filling).

Facciamo un esempio di conversazione per un assistente virtuale di prenotazione ristoranti:

1. L'utente scrive: "Voglio prenotare un tavolo per stasera da Gianni".
2. Il NER riconosce nel testo:
   - "Gianni" come entità di tipo `PER` (potrebbe essere ambiguo, ma in contesto gastronomico potrebbe anche essere un `LOC` se "Da Gianni" è il nome del ristorante).
   - "stasera" come `TIME`.
3. Lo slot-filling contestualizza:
   - nome_ristorante = "Da Gianni"
   - data_prenotazione = "2025-03-02 20:00" (se "stasera" è mappato a una data specifica e magari un orario predefinito)
   - richiesta_utente = "prenotazione"

Da un punto di vista implementativo, potremmo anche definire uno slot "ristorante" e uno slot "orario", che vengono riempiti con i valori estratti. Il NER fornisce la base per capire dove si trovano le informazioni nel testo, mentre lo slot-filling si assicura di collocarle correttamente nei campi del database o nei parametri del servizio di prenotazione.

2.3.3 Annotazione dei dati con Doccano

Prima di procedere all'addestramento del modello di Named Entity Recognition, è stato necessario produrre un dataset adeguatamente etichettato. A questo scopo, ho impiegato Doccano [49], uno strumento web open-source pensato per facilitare il processo di annotazione di testi. L'interfaccia di Doccano consente di selezionare frammenti di testo (ad esempio, termini rilevanti in un dominio specifico) e assegnare loro delle etichette, generando in output un file JSONL pronto per la fase di training.

Una serie di record del file JSONL prodotto da Doccano potrebbe avere il seguente formato:

```
[{
    "id": 632,
    "text": "What is the output when the automaton processes '1010'?",
    "label": [[49,53,"input"]]
},
{
    "id": 634,
    "text": "Does the automaton accept strings where the number of '0's equals the number
of '1's?",
    "label": [[55,56,"input"], [81,82,"input"]]
},
{
    "id": 635,
    "text": "What is the effect on the accepted language if we remove state q1?",
    "label": [[63,65,"node"]]
}]
```

Snippet 15: Esempio di record JSONL prodotto da Doccano per l'annotazione dei dati di NER.

Nel dettaglio, vediamo che:

- Il campo `text` contiene la stringa completa del messaggio o della domanda.
- Il campo `label` indica le etichette come tuple, ciascuna composta da:
  1. La posizione iniziale del frammento etichettato (inclusa).
  2. La posizione finale del frammento etichettato (esclusa).
  3. L'etichetta stessa (ad esempio, `input` o `node`).

La fase di annotazione è stata svolta manualmente, con particolare attenzione alla coerenza e alla completezza delle etichette. Doccano ha permesso di semplificare il lavoro, consentendo di visualizzare i testi e le etichette in modo chiaro e di aggiungere nuove annotazioni con pochi clic, senza la necessità di scrivere codice o utilizzare strumenti esterni.

In seguito all'etichettatura sono risultate tre classi di entità:

- `input` : per i frammenti di testo che contengono input o sequenze di simboli. Ad esempio, nella frase "Does it only accept 1s and 0s?" ci aspetteremmo di individuare due entità di tipo `input` : `[20,21,"input"],[27,28,"input"]` ;
- `node` : per i frammenti di testo che contengono nodi o stati dell'automa. Ad esempio, nella frase "Is there a transition between q2 and q0?" ci aspetteremmo di individuare due entità di tipo `node` : `[30,32,"node"],[37,39,"node"]` .
- `language` : per i frammenti di testo che contengono informazioni sulla lingua accettata dall'automa. Ad esempio, nella frase "Does the automaton accept strings over the alphabet {0,1}?" ci aspetteremmo di individuare un'entità di tipo `language` : `[53,58,"language"]` .

2.3.4 Implementazione con spaCy

spaCy è una libreria open-source in Python progettata per l'elaborazione del linguaggio naturale. Offre una serie di strumenti avanzati per l'analisi e la comprensione di testi, tra cui tokenizzazione, lemmatizzazione, part-of-speech tagging e, essenziale per il nostro progetto, la Named Entity Recognition, per la quale ha un motore altamente performante.

Un tipico flusso di lavoro con spaCy prevede la creazione di un "modello" (o il caricamento di un modello pre-addestrato) corrispondente a una determinata lingua, il passaggio di testo a questo modello per il processamento (tokenizzazione, tagging, ecc.) e, se necessario, la personalizzazione o l'addestramento ulteriore delle varie componenti. Proprio quest'ultimo aspetto - l'addestramento di un modello di Named Entity Recognition - è al centro di questa sezione.

Caricamento dati in formato Doccano JSONL

La prima componente fondamentale è la classe `NERData` , che si occupa di caricare e rappresentare i dati etichettati in formato Doccano JSONL. Doccano produce un file in cui ogni riga corrisponde a un esempio di testo con le relative annotazioni (indici di inizio/fine e nome dell'etichetta). Bisogna notare come il file non sia un vero e proprio JSON, ma una sequenza di righe JSON, ciascuna contenente un singolo esempio.

Per questo motivo al momento dell'importazione è essenziale leggere il file riga per riga e caricare ogni esempio separatamente:

```python
class NERData:
    """
    A class to represent NER data in Doccano JSONL format.
    """

    def __init__(self, line: str):
        data = json.loads(line.strip())

        self.text: str = data['text']
        self.labels: list[SpacyEntity] = data.get('label', [])  # (start, end, label)
        self.entity_labels = list({label for (_, _, label) in self.labels})

    @staticmethod
    def load_jsonl_data(file_path: Path) -> list['NERData']:
        """
        Convert Doccano JSONL format to spaCy training data format.
        """
        with file_path.open('r', encoding='utf-8') as f:
            return [NERData(line) for line in f]

    def make_example(self, nlp):
        doc = nlp.make_doc(self.text)
        annotations = {"entities": self.labels}
        return Example.from_dict(doc, annotations)
```

Classe 2: La classe `NERData` permette di gestire in modo semplice i dati etichettati in formato Doccano JSONL.

L'idea alla base segue design pattern comuni per la gestione dei dati:

- La classe si occupa di rappresentare un singolo esempio, con il testo e le relative annotazioni;
- Il metodo `load_jsonl_data` si occupa di caricare tutti gli esempi da un file JSONL e restituirli come una lista di oggetti `NERData`. Per poterlo fare, viene sfruttato il costruttore della classe in modo da rendere il codice più modulare e manutenibile possibile;
- Il metodo d'istanza `make_example` converte un esempio in un formato compatibile con spaCy [50], [51], in modo da poter essere utilizzato per l'addestramento del modello.

Pre-elaborazione dei dati

Una volta caricati i dati, il passo successivo consiste nell'analisi delle etichette e nella suddivisione in train set e validation set. Per questo scopo, si utilizzano due funzioni:

```python
def prepare_multilabel_data(entities: list[NERData]) -> tuple[ndarray, list[str]]:
    """
    Prepare multilabel data for NER entities, converting them into
    a binary matrix format using MultiLabelBinarizer.
    """
    binarizer = MultiLabelBinarizer()
    all_labels = [e.entity_labels for e in entities]
    binarizer.fit(all_labels)

    label_matrix = binarizer.transform(all_labels)
    return label_matrix, binarizer.classes_
```

Funzione 4: La funzione `prepare_multilabel_data` si occupa di preparare i dati etichettati per l'addestramento del modello NER.

Qui si sfrutta un `MultiLabelBinarizer` dal modulo `sklearn.preprocessing` per convertire le etichette multiclasse in un formato binario, in modo da poterle utilizzare per l'addestramento

di un modello di classificazione. Questo passaggio è essenziale per poter addestrare un modello di NER, che deve essere in grado di riconoscere più entità contemporaneamente.

```
def stratified_split(entities: list[NERData],
                     label_matrix,
                     val_size=0.2,
                     random_state=42) -> tuple[list[NERData], list[NERData]]:
  """
  Perform a stratified split on multilabel data using
  MultilabelStratifiedShuffleSplit.
  """
  msss = MultilabelStratifiedShuffleSplit(n_splits=1,
                                          test_size=val_size,
                                          random_state=random_state)

  train_indices, test_indices = next(msss.split(np.zeros(len(label_matrix)),
label_matrix))

  train_data = [entities[i] for i in train_indices]
  test_data = [entities[i] for i in test_indices]

  return train_data, test_data
```

Funzione 5: La funzione `stratified_split` si occupa di dividere i dati in training set e validation set in modo stratificato.

Grazie a `MultilabelStratifiedShuffleSplit` dal modulo di estensione di scikit-learn `iterstrat.ml_stratifiers`, è possibile dividere i dati in modo stratificato, garantendo che le proporzioni delle etichette siano mantenute sia nel training set che nel validation set. Questo è particolarmente importante quando si lavora con dataset multiclasse, in cui alcune etichette possono essere sottorappresentate e rischierebbero di non essere presenti in uno dei due set.

Addestramento del modello

Vediamo ora a step come la funzione `train_spacy` è implementata. Questa funzione si occupa di addestrare un modello di Named Entity Recognition con spaCy, partendo dai dati preparati e suddivisi in precedenza.

1. Creazione del modello spaCy a partire da zero: definiamo una nuova pipeline vuota, a cui viene aggiunto esclusivamente il componente per la NER. Registriamo anche tutte le etichette possibili nella pipeline:

   ```
   nlp = spacy.blank(language)
   ner = nlp.add_pipe('ner', last=True)

   for label in label_list:
     ner.add_label(label)
   ```

   Normalmente, nel caso in cui si dovesse addestrare una pipeline più complessa, spaCy offre la possibilità di descriverla in un file di configurazione. Dal momento che la NER sarà preparata per un sistema più grande, è preferibile cercare di ridurre al minimo i file intermedi di configurazione proprio per permettere un controllo più centralizzato.
2. Training loop. Per ogni iterazione:
   1. Si mescola il training set che viene poi diviso in piccoli batch:

```
random.shuffle(train_data)
batches = minibatch(items=train_data,size=compounding(4.0, 32.0, 1.001))
```

Il parametro del compounding permette di incrementare gradualmente la dimensione dei batch, partendo da un valore minimo fino a un valore massimo, in modo da bilanciare la varianza e la stabilità dell'addestramento

2. Si itera sui batch e si aggiorna il modello. Usando la funzione `make_example` definita in precedenza, si convertono gli esempi in un formato compatibile con spaCy e si aggiornano i pesi del modello tramite la funzione `update` (che userà internamente la discesa del gradiente):

```
for batch in batches:
  examples = [el.make_example(nlp) for el in batch]
  nlp.update(examples, drop=0.5, sgd=optimizer, losses=losses)
```

3. Valutazione. Alla fine di ogni epoca, si valuta il modello sul validation set e si calcolano le metriche di interesse (precision, recall, F1 score):

```
with nlp.use_params(optimizer.averages):
  examples = [el.make_example(nlp) for el in val_data]
  scores = nlp.evaluate(examples)
```

Come si può notare, l'addestramento di un modello di NER con spaCy richiede poche righe di codice, grazie alla semplicità e alla flessibilità della libreria. Inoltre, la modularità delle componenti (tra cui `NERData`, `prepare_multilabel_data`, `stratified_split`) permette di mantenere il codice pulito e facilmente estendibile, adattandolo a nuovi dataset o a nuove esigenze.

2.3.5 Valutazione e performance

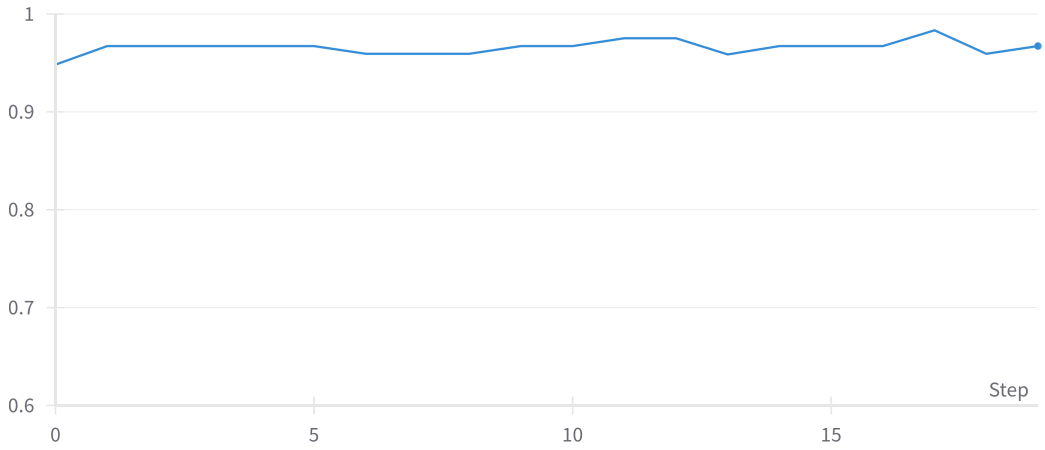


Grafico 7: Performance di F1 del modello di NER durante l'addestramento.

Anche nel caso della Named Entity Recognition la metrica di riferimento è l'F1 score, che tiene conto sia della precisione che del recall del modello. Nel grafico sopra, possiamo vedere come

l'F1 score del modello si stabilisca quasi fin dall'inizio oltre il 90%, confermando la bontà del training set e la capacità del modello di generalizzare correttamente le entità riconosciute.

Prese singolarmente, le tre tipologie di entità del training set (`input`, `node`, `language`) mostrano performance molto simili, con un F1 score medio intorno al 92%. Questo indica che il modello è in grado di riconoscere con precisione e recall elevati le entità di interesse, indipendentemente dalla loro categoria.

3 Natural Language Generation

Nel contesto di un sistema di dialogo, la generazione del linguaggio naturale (NLG) è una delle componenti fondamentali affinchè il sistema possa effettivamente comunicare con gli utilizzatori in modo efficace.

Una volta che abbiamo compreso l'intenzione dell'utente, dobbiamo generare una risposta che sia **coerente con la richiesta** e che **fornisca un qualche valore** all'interlocutore, senza scordare che quest'ultima deve essere comprensibile. Il processo può essere svolto in diversi modi, a seconda delle esigenze e delle capacità del sistema stesso.

La Sezione 3 punterà ad illustrare le tecniche di generazione del linguaggio naturale ricercate per il sistema, mentre la Sezione 4 si occuperà di mostrare più nel dettaglio come queste siano rese disponibili per i botmaster.

Nelle sezioni seguenti vedremo che il processo è divisibile in due fasi principali:

1. Il recupero dei dati: senza informazioni, il sistema non può generare risposte significative. Il recupero dei dati è quindi il primo passo per poter generare risposte coerenti e pertinenti.
2. La generazione delle risposte: una volta che il sistema ha a disposizione i dati necessari, può procedere con la generazione del testo che verrà presentato all'utente. Possono essere utilizzate diverse tecniche, tra cui prompting o parafrasi, per generare risposte di qualità.

3.1 Data Retrieval

Vi possono essere casi in cui il sistema non ha bisogno di recuperare dati o dettagli per poter rispondere; in questi casi, la risposta può essere generata direttamente dal sistema stesso, senza bisogno di informazioni aggiuntive. Questa situazione è tipica quando il sistema deve rispondere a domande prestabilite o statiche, come ad esempio quelle relative a informazioni generali o a domande di cortesia.

Possiamo pensare ad esempio a interazioni basilari da inizio conversazione, le cui risposte sono fisse e non necessitano di alcun tipo di elaborazione, come:

- Semplici saluti: "Ciao!", "Potresti aiutarmi?";
- Informazioni generali: "Qual è il tuo nome?", "Come posso contattare il servizio clienti?";
- Interazioni di cortesia: "Grazie!", "A presto!".

In questi casi, è evidente che il sistema non abbia bisogno di recuperare ulteriori dati o dettagli **potenzialmente variabili a seconda del contesto** per poter rispondere, ma può generare la risposta direttamente.

Se invece il sistema deve fornire informazioni specifiche o personalizzate, allora è necessario che sia in grado di recuperare i dati necessari per generare la risposta. Questo processo può avvenire in diversi modi, a seconda delle esigenze del sistema, della complessità delle informazioni richieste e del modo in cui queste sono strutturate e salvate.

3.1.1 Basi di conoscenza strutturate

Le basi di conoscenza costituiscono una delle fonti più affidabili da cui un sistema di dialogo può attingere informazioni. La loro natura ordinata, con tabelle, relazioni e campi ben definiti, assicura una gestione dei dati che riduce la possibilità di incongruenze o duplicazioni. Allo

stesso tempo, la costruzione e la manutenzione di uno schema ben progettato richiedono un certo impegno iniziale, poiché bisogna prevedere in anticipo quali tipi di informazioni saranno necessari all'interno del sistema.

In uno scenario di dialogo, il processo di risposta idealmente seguirebbe due passi. Consideriamo ad esempio la domanda "Vorrei informazioni sul mio ordine numero 25565":

1. Prima di tutto il sistema riconoscerebbe il genere di richiesta posta dall'utente. Ipotizzando una classificazione simile a quella discussa nella Sezione 2.2 (quindi con uno o più livelli di classificazione dell'intent), riusciremmo a comprendere che la richiesta è legata al recupero di informazioni su un ordine.
2. Una volta identificato l'intent, mediante un modello di NER, il sistema estrarrebbe il valore di `orderId` dalla frase dell'utente, per poi interrogare una base di dati, tramite un prepared statement SQL [52], per recuperare i dettagli dell'ordine richiesto. Un esempio di query per il recupero di dettagli di un ordine in un sistema e-commerce potrebbe essere il seguente:

```sql
SELECT customer_name, order_date, total_amount
FROM orders
WHERE order_id = :orderId
  # Aggiungiamo un vincolo per evitare accessi non autorizzati
  AND customer_id = :customerId;
```

Query 1: Esempio di query SQL per il recupero di dettagli di un ordine in un sistema e-commerce.

L'esecuzione di questa interrogazione restituisce al modulo di generazione del linguaggio naturale i dettagli necessari a comporre una risposta personalizzata.
Un ulteriore vantaggio di questo approccio risiede nella possibilità di definire in anticipo diversi vincoli e relazioni che facilitano la coerenza dei dati.

In alternativa alla struttura difficilmente variabile (una sfida comunque superabile![11]) in produzione di un database relazionale, un database NoSQL può risultare altrettanto efficace. Nel dominio della *Knowledge Representation* sono stati definiti alcuni linguaggi il cui scopo è permettere di rappresentare conoscenze strutturate e complesse, dalle quali è anche possibile inferire nuove informazioni.

Alla base vi è RDF (Resource Description Framework), un modello di dati che permette di rappresentare informazioni in forma di triple `<soggetto, predicato, oggetto>`, e il suo linguaggio di interrogazione SPARQL [53].

Le annotazioni in formato Turtle ad esempio ci permettono di rappresentare un Grafo RDF testualmente (rendendolo molto facilmente intellegibile da un lettore), come nel seguente esempio:

[11] https://softwareengineering.stackexchange.com/questions/235785/how-to-handle-unexpected-schema-changes-to-production-database

```
BASE <http://example.org/>
# Definiamo degli IRI, ovvero identificatori di risorse che abbreviano URL più lunghi
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rel: <http://www.perceive.net/schemas/relationship/>

<#green-goblin>
  rel:enemyOf <#spiderman> ;
  a foaf:Person ;
  foaf:name "Green Goblin", "Goblin"@it .

<#spiderman>
  rel:enemyOf <#green-goblin> ;
  a foaf:Person ;
  foaf:name "Spiderman", "Uomo Ragno"@it .
```

Query 2: Esempio di annotazione in formato Turtle per il film d'animazione _Gli Incredibili_ e il suo regista.

Questo modello è alla base di molte knowledge base, come DBpedia [54] e Wikidata [55], che raccolgono informazioni strutturate su una vasta gamma di argomenti. Le basi di conoscenza sono normalmente codificate su file, in formati come RDF-XML [56] o Turtle [57].

Anche in questo caso, dovendo rispondere a una richiesta come "Chi ha diretto il film d'animazione *Gli Incredibili*?[12]", una volta determinato l'intent ed estratta la named entity del film, possiamo recuperare in modo preciso le informazioni necessarie con una veloce query:

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?director
WHERE {
  dbr:The_Incredibles dbo:director ?director .
}
```

Query 3: Esempio di query SPARQL per il recupero del regista del film Inception.

Ricevuta la risposta (`dbr:Brad_Bird` [13]), interagendo coi campi `dbp:name` e `dbo:thumbnail` dell'entità, il sistema potrà rapidamente comporre una risposta completa (se usassimo un template, ne risulterebbe "Il film Gli Incredibili è stato diretto da Brad Bird") e arricchirla con un'immagine del regista.

3.1.2 Corpora non testuali

A differenza delle knowledgebase strutturate, i corpora testuali non strutturati offrono la possibilità di attingere a un bacino molto più ampio e flessibile di informazioni, ma richiedono tecniche di recupero dati più complesse per poter restituire risultati pertinenti. Questi corpora possono includere documenti di varia natura, come articoli, pagine web, FAQ, manuali e qualsiasi altro contenuto scritto che non segua uno schema predeterminato.

I metodi tradizionali di Information Retrieval si basano solitamente su indici inversi o modelli a spazio vettoriale, come TF-IDF[14] [58] o BM25 [59], che confrontano la query dell'utente con i

[12] http://dbpedia.org/resource/The_Incredibles
[13] https://dbpedia.org/page/Brad_Bird
[14] Term Frequency-Inverse Document Frequency

termini presenti nel corpus. Sebbene questi approcci siano ancora efficaci in molti scenari, con l'evoluzione dei modelli neurali è possibile sfruttare reti specializzate che codificano frasi e documenti in uno spazio di embedding semantico. Un esempio diffuso è l'utilizzo di Sentence-BERT [60], che permette di generare vettori numerici rappresentativi del significato di un testo e, di conseguenza, di calcolare la similarità fra query e documenti in modo più accurato rispetto alle semplici corrispondenze di parole chiave.

Per illustrare questo principio, si consideri il seguente snippet di codice Python che usa la libreria sentence-transformers:

```python
from sentence_transformers import SentenceTransformer, util

# Carichiamo un modello pre-addestrato
model = SentenceTransformer('all-MiniLM-L6-v2')

# Supponiamo di avere un piccolo corpus di documenti
corpus = [
    "Questo è un documento sul funzionamento dei sistemi di dialogo.",
    "Ecco un articolo sui vantaggi dei knowledge graph.",
    "Una breve introduzione all'Information Retrieval."
]
corpus_embeddings = model.encode(corpus, convert_to_tensor=True)

# Definiamo la query dell'utente
query = "Cosa sono i sistemi di dialogo?"
query_embedding = model.encode(query, convert_to_tensor=True)

# Calcoliamo la similarità tra la query e i documenti
scores = util.pytorch_cos_sim(query_embedding, corpus_embeddings)[0]
best_score_idx = scores.argmax().item()

print("Testo più simile:", corpus[best_score_idx])
```

Script 3: Esempio di utilizzo di Sentence-BERT per trovare il documento più simile a una query.

In questo esempio, dopo aver calcolato gli embeddings[15] dei documenti del corpus e della query utente, calcoliamo la **cosine similarity** tra di essi e identifichiamo l'indice del documento più affine dal punto di vista semantico. La differenza sostanziale rispetto a metodi tradizionali consiste nel fatto che l'uso di embeddings semantici permette di riconoscere relazioni di significato **anche quando il lessico non coincide esattamente**.

Integrando un simile modulo di retrieval in un sistema di dialogo, è possibile estendere la copertura informativa ben oltre i limiti di una base di dati strutturata, sebbene ciò comporti un aumento della complessità. Le performance dipendono, infatti, dalla qualità del modello neurale e dalla quantità di risorse computazionali disponibili per l'indicizzazione e la ricerca.

Il ricorso a corpora testuali non strutturati è particolarmente utile nei sistemi open-domain, dove l'utente potrebbe porre quesiti su una gamma di argomenti molto ampia. L'ampiezza della base informativa fornisce un potenziale enorme, a condizione di implementare strategie di filtraggio e ranking dei risultati che riducano il rischio di rumore o di risposte poco rilevanti. In tal senso, molte pipeline di retrieval prevedono una fase di re-ranking [61], nella quale uno o

[15] Strutture che codificano il testo numericamente per permettere di effettuare operazioni matematiche, come la **cosine similarity**

più modelli ricalcolano la pertinenza dei documenti più promettenti prima di fornire la risposta definitiva all'utente.

3.1.3 API e servizi esterni

L'accesso a dati provenienti da fonti esterne in tempo reale rappresenta un altro tassello fondamentale per i sistemi di dialogo moderni. Integrare API e servizi di terze parti permette, ad esempio, di fornire aggiornamenti meteorologici, visualizzare informazioni sul traffico, ottenere prezzi di mercato o eseguire prenotazioni, arricchendo notevolmente le capacità del sistema.

A differenza delle basi di dati documentali o dei corpora statici, le API espongono endpoint che possono variare da semplici chiamate REST, fino a interfacce più complesse che richiedono autenticazione e parametri di configurazione.

Tipicamente il motore di dialogo intercetta la domanda dell'utente e, riconoscendo la necessità di dati esterni, effettuerà una chiamata API verso il servizio più appropriato. Nel caso di servizi che restituiscono risposte JSON, si può utilizzare un linguaggio di query come JMESPath [62] per filtrare i risultati e isolare solo i campi più rilevanti.

JMESPath è un linguaggio espressivo e leggero, progettato per filtrare, cercare e trasformare dati in formato JSON. A differenza di query tradizionali con linguaggi come SQL, JMESPath è progettato per operare esclusivamente su strutture JSON e consente di navigare in maniera semplice all'interno di oggetti annidati, liste e chiavi complesse. Grazie alla sua sintassi intuitiva, risulta particolarmente utile quando si devono gestire risposte provenienti da API REST, servizi esterni o qualunque altra fonte che fornisca dati in formato JSON.

Per illustrare in concreto JMESPath, consideriamo la seguente struttura di dati (immaginiamo sia la risposta di un servizio esterno che fornisce informazioni su articoli tecnologici):

Un esempio basilare in Python potrebbe presentarsi così:

```python
import requests
import jmespath

response = requests.get("https://api.example.com/v1/articles?category=tech")
if response.status_code == 200:
    data = response.json()
    # Utilizziamo JMESPath per estrarre una parte specifica del JSON
    titles = jmespath.search("items[].title", data)
    print("Trovati i seguenti titoli:", titles)
else:
    print("Impossibile contattare l'API.")
```

Script 4: Esempio di chiamata a un'API di news e filtraggio dei titoli tramite JMESPath.

Qui, dopo avere inviato la richiesta a un'ipotetica API di news, si analizza il contenuto JSON ricevuto e lo si filtra mediante un'espressione JMESPath, in modo da estrarre tutti i titoli degli articoli in un singolo passaggio. Il sistema di dialogo può quindi usare queste informazioni per formulare una risposta, magari raggruppando i titoli più rilevanti o generando una breve sintesi.

Oltre a restituire dati in forma di testo, alcune API consentono di eseguire azioni che hanno un effetto sul mondo esterno, come prenotare un ristorante o inviare un ordine. Ciò comporta un aumento delle responsabilità da parte del sistema, il quale deve gestire correttamente eventuali errori o limiti di utilizzo, come soglie massime di richieste al minuto o specifiche politiche di caching. Al tempo stesso, la capacità di interagire dinamicamente con risorse esterne rende il sistema di dialogo molto più potente e utile, in particolare in contesti in cui la tempestività dell'informazione è fondamentale.

Un aspetto delicato consiste nella gestione delle possibili contraddizioni tra i dati forniti da un servizio e quelli custoditi internamente al sistema. Se, ad esempio, il database aziendale riporta una certa disponibilità di un prodotto, mentre la verifica tramite API rivela esaurimento scorte, è necessario definire delle regole di priorità o di riconciliazione, così da garantire la coerenza e l'affidabilità delle risposte. All'interno del flusso di gestione delle chiamate a servizi esterni, questi conflitti vanno rilevati tempestivamente, per poi essere gestiti nel modulo di generazione, magari informando l'utente dell'aggiornamento in tempo reale.

La combinazione di dati provenienti da API e basi di dati locali apre scenari particolarmente ricchi, in cui il sistema di dialogo può rispondere sia a domande generiche attingendo a corpora esterni, sia a interrogativi specifici all'azienda o all'utente, accedendo a informazioni riservate. Con un'architettura ben progettata, si ottiene così un ecosistema integrato che favorisce interazioni più naturali e, allo stesso tempo, garantisce il controllo sulla qualità e l'accuratezza delle risposte.

3.1.4 Retrieval automatico

3.2 Generazione di risposte tramite LLM

3.2.1 Parafrasi

3.2.2 Prompting

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] «The Loebner Prize». [Online]. Disponibile su: https://www.ocf.berkeley.edu/~arihuang/academic/research/loebner.html

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

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

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

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

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

[7] G. Team *et al.*, «Gemma 2: Improving Open Language Models at a Practical Size», 2024. doi: 10.48550/arXiv.2408.00118.

[8] A. Grattafiori *et al.*, «The Llama 3 Herd of Models», 2024. doi: 10.48550/arXiv.2407.21783.

[9] J. Bai *et al.*, «Qwen Technical Report», 2023. doi: 10.48550/arXiv.2309.16609.

[10] C. N. Silla e A. A. Freitas, «A survey of hierarchical classification across different application domains», *Data Mining and Knowledge Discovery*, vol. 22, fasc. 1, pp. 31–72, 2011, doi: 10.1007/s10618-010-0175-9.

[11] T. B. Brown *et al.*, «Language Models are Few-Shot Learners». 2020. doi: 10.48550/arXiv.2005.14165.

[12] DeepSeek-AI *et al.*, «DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning». 2025. doi: 10.48550/arXiv.2501.12948.

[13] DeepSeek-AI *et al.*, «DeepSeek-V3 Technical Report». 2025. doi: 10.48550/arXiv.2412.19437.

[14] John Gruber, «Markdown». [Online]. Disponibile su: https://daringfireball.net/projects/markdown/

[15] 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.

[16] 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.

[17] J. Devlin, M.-W. Chang, K. Lee, e K. Toutanova, «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». 2019. doi: 10.48550/arXiv.1810.04805.

[18] A. Vaswani *et al.*, «Attention Is All You Need». 2023. doi: 10.48550/arXiv.1706.03762.

[19] T. Wolf *et al.*, «HuggingFace's Transformers: State-of-the-art Natural Language Processing», 2020. doi: 10.48550/arXiv.1910.03771.

[20] «GNU General Public License». [Online]. Disponibile su: http://www.gnu.org/licenses/gpl.html

[21] R. Stallman, *Free software free society: selected essays of Richard M. Stallman*, 3rd ed. Free Software Foundation, 2015. [Online]. Disponibile su: https://www.gnu.org/doc/fsfs3-hardcover.pdf

[22] J. Opitz, «A Closer Look at Classification Evaluation Metrics and a Critical Reflection of Common Evaluation Practice», *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 820–836, 2024, doi: 10.1162/tacl_a_00675.

[23] M. Mosbach, M. Andriushchenko, e D. Klakow, «On the Stability of Fine-tuning BERT: Misconceptions, Explanations, and Strong Baselines», 2021. doi: 10.48550/arXiv.2006.04884.

[24] K. Clark, M.-T. Luong, Q. V. Le, e C. D. Manning, «ELECTRA: Pre-training text encoders as discriminators rather than generators». [Online]. Disponibile su: https://openreview.net/pdf?id=r1xMH1BtvB

[25] *Bert uncased model by Google*. [Online]. Disponibile su: https://huggingface.co/google-bert/bert-base-uncased

[26] *Distilbert base model*. [Online]. Disponibile su: https://huggingface.co/distilbert/distilbert-base-uncased

[27] G. Hinton, O. Vinyals, e J. Dean, «Distilling the Knowledge in a Neural Network», 2015. doi: 10.48550/arXiv.1503.02531.

[28] V. Sanh, L. Debut, J. Chaumond, e T. Wolf, «DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter», 2020. doi: 10.48550/arXiv.1910.01108.

[29] *MobileBERT on Huggingface*. [Online]. Disponibile su: https://huggingface.co/google/mobilebert-uncased

[30] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, e D. Zhou, «MobileBERT: a Compact Task-Agnostic BERT for Resource-Limited Devices», 2020. doi: 10.48550/arXiv.2004.02984.

[31] *Electra on Huggingface*. [Online]. Disponibile su: https://huggingface.co/google/electra-small-discriminator

[32] I. J. Goodfellow *et al.*, «Generative Adversarial Networks». 2014. doi: 10.48550/arXiv.1406.2661.

[33] R. Caruana, «Multitask Learning», *Machine Learning*, vol. 28, fasc. 1, pp. 41–75, 1997, doi: 10.1023/A:1007379606734.

[34] Sanjaye Elayattu, «Multi-task Fine-tuning with BERT». [Online]. Disponibile su: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1234/final-reports/final-report-169425270.pdf

[35] Jiacheng Hu e Jack Hung, «Multitasking with BERT». [Online]. Disponibile su: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1234/final-reports/final-report-169989122.pdf

[36] F. Pedregosa *et al.*, «Scikit-learn: Machine Learning in Python», 2018. doi: 10.48550/arXiv.1201.0490.

[37] R. Grishman e B. Sundheim, «Message Understanding Conference-6: a brief history», in *Proceedings of the 16th Conference on Computational Linguistics - Volume 1*, in COLING '96. Association for Computational Linguistics, 1996, pp. 466–471. doi: 10.3115/992628.992709.

[38] M. Munnangi, «A Brief History of Named Entity Recognition», 2024. doi: 10.48550/arXiv.2411.05057.

[39] J. D. Lafferty, A. McCallum, e F. C. N. Pereira, «Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data», in *Proceedings of the Eighteenth International Conference on Machine Learning*, in ICML '01. Morgan Kaufmann Publishers Inc., 2001, pp. 282–289. doi: 10.5555/645530.655813.

[40] A. McCallum, D. Freitag, e F. C. N. Pereira, «Maximum Entropy Markov Models for Information Extraction and Segmentation», in *Proceedings of the Seventeenth International Conference on*

Machine Learning, in ICML '00. Morgan Kaufmann Publishers Inc., 2000, pp. 591–598. doi: 10.5555/645529.658277.

[41] C. Cortes e V. Vapnik, «Support-vector networks», *Machine Learning*, vol. 20, fasc. 3, pp. 273–297, set. 1995, doi: 10.1007/BF00994018.

[42] R. M. Schmidt, «Recurrent Neural Networks (RNNs): A gentle Introduction and Overview». [Online]. Disponibile su: https://arxiv.org/abs/1912.05911

[43] S. Hochreiter e J. Schmidhuber, «Long Short-Term Memory», *Neural Computation*, vol. 9, fasc. 8, pp. 1735–1780, 1997, doi: 10.1162/neco.1997.9.8.1735.

[44] Christopher Olah, «Understanding LSTM Networks». [Online]. Disponibile su: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[45] M. Polignano, M. de Gemmis, e G. Semeraro, «Comparing transformer-based NER approaches for analysing textual medical diagnoses», F. G., V. G. 6. P. University of Padova Department of Information Engineering, F. N., V. G. 6. P. University of Padova Department of Information Engineering, J. A., I. Z. 1. R. A. M. C. 5. University of Montpellier LIRMM, M. M., U. 1. C. University of Copenhagen Department of Computer Science, P. F., e F. 9.-1. V. Vienna University of Technology (TU) Institute of Information Systems Engineering, A c. di, CEUR-WS, 2021, pp. 818–833. [Online]. Disponibile su: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85113468144&partnerID=40&md5=415546a367cea8c30e30881072935304

[46] R. C. Schank e R. P. Abelson, «Scripts, plans, and knowledge», in *International Joint Conference on Artificial Intelligence*, 1975. [Online]. Disponibile su: https://api.semanticscholar.org/CorpusID:264593435

[47] P. F. Brown *et al.*, «A statistical approach to machine translation», *Comput. Linguist.*, vol. 16, fasc. 2, pp. 79–85, giu. 1990, doi: 10.5555/92858.92860.

[48] S. Louvan e B. Magnini, «Exploring Named Entity Recognition As an Auxiliary Task for Slot Filling in Conversational Language Understanding», in *Proceedings of the 2018 EMNLP Workshop SCAI: The 2nd International Workshop on Search-Oriented Conversational AI*, A. Chuklin, J. Dalton, J. Kiseleva, A. Borisov, e M. Burtsev, A c. di, Association for Computational Linguistics, ott. 2018, pp. 74–80. doi: 10.18653/v1/W18-5711.

[49] H. Nakayama, T. Kubo, J. Kamura, Y. Taniguchi, e X. Liang, «doccano: Text Annotation Tool for Human». [Online]. Disponibile su: https://github.com/doccano/doccano

[50] M. Honnibal e I. Montani, «spaCy: Industrial-strength Natural Language Processing in Python». [Online]. Disponibile su: https://spacy.io/

[51] M. Honnibal e I. Montani, «Introducing spaCy». [Online]. Disponibile su: https://explosion.ai/blog/introducing-spacy

[52] «SQL Injection Prevention Cheat Sheet». [Online]. Disponibile su: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[53] «SPARQL Query Language for RDF». [Online]. Disponibile su: https://www.w3.org/TR/sparql11-query/

[54] «DBpedia». [Online]. Disponibile su: https://wiki.dbpedia.org/

[55] «Wikidata». [Online]. Disponibile su: https://www.wikidata.org/

[56] W3C, «RDF 1.1 XML Syntax», feb. 2014. [Online]. Disponibile su: https://www.w3.org/TR/rdf-syntax-grammar/

[57] W3C, «RDF 1.1 Turtle», feb. 2025. [Online]. Disponibile su: https://w3c.github.io/rdf-turtle/spec/

[58] A. Rajaraman e J. D. Ullman, «Data Mining», in *Mining of Massive Datasets*, Cambridge University Press, 2011, pp. 1–17.

[59] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, e M. Gatford, «Okapi at TREC-3.», 1994.

[60] N. Reimers e I. Gurevych, «Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks». [Online]. Disponibile su: https://arxiv.org/abs/1908.10084

[61] Z. Wang, P. Ng, R. Nallapati, e B. Xiang, «Retrieval, Re-ranking and Multi-task Learning for Knowledge-Base Question Answering», in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, P. Merlo, J. Tiedemann, e R. Tsarfaty, A c. di, Association for Computational Linguistics, apr. 2021, pp. 347–357. doi: 10.18653/v1/2021.eacl-main.26.

[62] James Saryerwinnie, «JMESPath is a query language for JSON». [Online]. Disponibile su: https://jmespath.org/