Relazione informale lavoro svolto

2024-12-31

Abstract In questo documento cercherò di riassumere tutto il lavoro svolto finora per la tesi, oltre al lavoro che vi propongo di svolgere per poter effettivamente considerare la tesi come conclusa.

1. Compilatore programmatico

All'inizio della tesi si era parlato di un compilatore che, forniti i dati dell'automa, fosse in grado di produrre il file AIML per le interazioni.

- Scritto in kotlin
- Utilizza un file contenente l'automa rappresentato col linguaggio graphviz
- Genera un file AIML che contiene tutte le regole e le possibili domande riguardanti l'automa

1.1. Pro e contro

1.1.1. Pro

- Facilità di utilizzo
- Velocità di esecuzione
- Linguaggio memory safe

1.1.2. Contro

- Le regole sono molto stringenti sulla forma delle domande che possono essere fatte, e per poter permettere una corretta risposta è necessario che l'utente conosca la forma delle domande che può fare, e che le rispetti
- Non è possibile fare domande complesse
- Le risposte sono molto limitate e ripetitive

2. Ricerca successiva per migliorare i contro del compilatore

Problemi da risolvere:

1. Le regole sono molto stringenti sulla forma delle domande che possono essere fatte
2. Le risposte sono molto limitate e ripetitive

Finora mi sono concentrato sulla generazione delle regole AIML, ma non ho ancora pensato a come migliorare le risposte. La direzione che ho preso si basa sulla costruzione di alcuni modelli fondati su reti neurali che permettano di riconoscere con maggiore flessibilità le domande poste dagli utenti.

Il piano è di dividere la gestione di una domanda (lato riconoscimento) in due step:

1. Riconoscimento dell'argomento della domanda
2. Riconoscimento dei complementi della domanda (se presenti), come il riferimento a nodi o input dell'automa o degli archi. Questi saranno poi usati per riempire degli slot.

2.1. Modelli di reti neurali per il riconoscimento di domande

Prima dell'utilizzo di modelli di reti neurali ho investigato ancora sulla possibilità di utilizzare dei modelli statistici costruiti automaticamente. Questo principalmente per la velocità di esecuzione e la footprint ridotta rispetto a modelli di reti neurali. Spacy era un'opzione che ho effettivamente esplorato per un po' di tempo, tuttavia dopo aver scoperto BERT [1] ho deciso di orientarmi verso l'utilizzo di modelli di reti neurali basati sui Transformer, basandomi sulla loro promessa di prestazioni migliori rispetto ai modelli classici.

BERT è preaddestrato su un corpus di testo molto grande; esistono diverse versioni che idealmente dovrebbero essere utilizzate in base alla quantità di risorse disponibili, considerando anche il task che si vuole svolgere.

È stato essenziale un primo periodo di studio e sperimentazione con l'architettura di BERT. Durante questa prima fase ho utilizzato indicazioni per il fine tuning di modelli presenti su alcuni paper, tra cui S. Gururangan *et al.* [2]

Tuttavia in seguito ho deciso di utilizzare la libreria `Transformers` di HuggingFace [3], che permette di utilizzare modelli preaddestrati e fornisce ottime API per l'addestramento, il fine tuning e in generale per l'interazione con modelli di reti neurali basati sui Transformer.

Oltre a BERT «base» ho anche utilizzato DistilBERT [4], una versione distillata di BERT che riesce a mantenere buone performance con una footprint ridotta.

Durante il lavoro di ricerca mi sono concentrato sul perfezionamento anche dei parametri di fine-tuning; per questo oltre a una serie di tentativi sperimentali in autonomia, ho usato come base di partenza il lavoro di altri ricercatori [5] che hanno perfezionato i parametri di fine-tuning di BERT per la classificazione di testo, per evitare problemi di overfitting e/o vanishing/exploding gradient che portano potenzialmente a catastrophic forgetting

2.1.1. Fine tuning di BERT

BERT è un modello preaddestrato e supervisionato, quindi per poter effettuare il fine tuning è necessario avere un dataset etichettato.

Sono partito dalle circa 200 domande prodotte durante i vari test con gli utenti svolti in precedenza.

2.1.1.1. Preparazione del dataset

I dati iniziali sono una collezione delle interazioni degli utenti con il sistema quando le interazioni erano state costruite a mano. Per poter utilizzare questi dati per il fine tuning di BERT è stato necessario:

1. Estrarre tutte le domande escludendo le risposte del sistema
2. Etichettare le domande con l'argomento a cui si riferiscono

L'estrazione è stata piuttosto rapida, dopo aver scritto un semplice script python che usa pandas.

```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")
```

Estratte le domande, ho proceduto con l'etichettatura.

Inizialmente ho pensato di etichettare automaticamente le domande utilizzando un modello di LLM locale. Ho individuato le utilissime API fornite da Ollama [6], un sistema per hostare localmente modelli di LLM open.

2.1.1.2. Etichettatura automatica delle domande

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

```python
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."
}
```

Ad ogni etichetta è associata una descrizione che aiuta a capire a cosa si riferisce, specialmente per permettere alla LLM di etichettare correttamente le domande togliendo il più possibile le ambiguità.

Per avere una maggiore sicurezza nella correttezza delle etichette, ho preferito utilizzare più modelli di LLM:

- Gemma 2 [7], sviluppato da Google Deep Mind
- llama 3.1 [8], sviluppato da Meta AI

Ogni modello ha ricevuto più prompt diversi con le stesse domande, in modo da poter poi effettuare un majority voting per stabilire l'etichetta finale.

I modelli sono stati utilizzati nelle loro varianti da 9 miliardi di parametri per gemma (intermedio) e 8 miliardi per llama3.1 (più piccolo), in seguito ad alcune veloci sperimentazioni che hanno mostrato un buon compromesso tra performance (intese come qualità dei risultati prodotti in seguito al prompting) e tempo di esecuzione.

Ho provato un ulteriore modello, Qwen [9], prodotto da Alibaba, ma i risultati non sono stati soddisfacenti.

Utilizzando le risorse hardware a disposizione, ho effettuato il prompting delle domande con i modelli di LLM. Segue un esempio di codice python che mostra come è stato effettuato il prompting. È inclusa una classe `Chat`, da me sviluppata, che permette di interagire con i modelli di LLM in modo più semplice, astraendo le API di ollama.

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

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

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

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

    dataset_size = len(df)

    for p_i, prompt_version in enumerate(prompts):
        progress_bar = tqdm(
          total=dataset_size,
```

```
            desc=f"Asking {model} with prompt {p_i}", unit="rows"
        )

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

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

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

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

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

Ecco un esempio dei risultati dell'etichettatura del bronze dataset:

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

Combinare i risultati tramite majority voting è stato essenziale, in quanto i modelli di LLM non sono perfetti e alcune volte sono state prodotte risposte completamente estranee rispetto alle etichette fornite:

```
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
```

In seguito ad una prima fase di fine tuning tuttavia, ho verificato che nonostante un'etichettatura valida, le classi identificate erano troppo sbilanciate, con alcune classi che contenevano pochissimi esempi. In più, le etichette erano troppo generiche e non permettevano 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.

2.1.1.3. Nuove classi e etichettatura manuale

Prima di tutto ho effettuato una ulteriore passata di revisione delle domande, escludendo quelle non pertinenti o incomprensibili. Durante questa fase ho compreso che non sarebbe stato sufficiente utilizzare un solo «livello» di etichette, ma che sarebbe stato più efficace utilizzare un sistema di etichettatura gerarchico, in modo da poter identificare con maggiore precisione l'argomento della domanda, riducendo il numero di classi da distinguere.

Le classi principali, che rappresentano l'argomento generale della domanda, sono state ridotte a 7:

Classe	Descrizione	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

Avendo solo 7 classi principali di domande, considerato il numero ristretto di esempi, è stato possibile suddividerli senza ottenere un numero enorme di classi che si dividono tra di loro pochi esempi.

Le classi secondarie, che rappresentano l'argomento specifico della domanda dipendono dalla classe principale. Per questo motivo il loro numero è variabile, ma in totale si tratta di 33 classi (**count**: 29, **existence_from**: 18, **list**: 17, **description**: 16, **accepted**: 14, **representation**: 13, **existence_between**: 12, **transitions**: 12, **description_brief**: 10, **pattern**: 10, **existence_directed**: 9, **start**: 8, **final**: 8, **symbols**: 7, **off_topic**: 6, **cycles**: 4, **label**: 4, **example_input**: 4, **final_list**: 3, **states**: 3, **generic**: 3, **variation**: 2, **greet**: 2, **final_count**: 2, **validity**: 2, **simulation**: 2, **regex**: 2, **definition**: 2, **input**: 1, **existence_into**: 1, **directionality**: 1, **details**: 1, **self_loop**: 1).

Dal momento che le classi secondarie sono dipendenti dalle classi principali, ho deciso di etichettare prima le classi principali e poi le classi secondarie.

Dal momento che il numero di esempi è piuttosto ridotto (229), e inoltre le classi sono sbilanciate, ho deciso di arricchire i dati con ulteriori domande generate automaticamente e manualmente. Ho aggiunto un totale di 525 domande, con la seguente distribuzione:

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

Le domande off-topic aggiuntive (un centinaio) sono state estratte dal dataset SQUAD[1] v2 [10], [11], per avere una sufficiente varietà di domande non pertinenti.

Anche le classi secondarie hanno ricevuto alcune migliorie alla distribuzione, che tuttavia è ancora da migliorare: (**description**: 74, **accepted**: 57, **existence_from**: 42, **count**: 40, **generic**: 39, **list**: 38, **label**: 36, **transitions**: 34, **pattern**: 27, **existence_between**: 25, **existence_directed**: 21, **final**: 21, **simulation**: 20, **variation**: 19, **greet**: 19, **representation**: 19, **states**: 18, **existence_into**: 17, **definition**: 16, **description_brief**: 16, **start**: 15, **symbols**: 14, **validity**: 14, **cycles**: 12, **details**: 12, **input**: 12, **self_loop**: 11, **example_input**: 11, **regex**: 9, **final_count**: 8, **off_topic**: 6, **final_list**: 6, **optimization**: 6, **deterministic**: 5, **reachability**: 5, **start_final**: 3, **dead**: 3, **directionality**: 2, **image**: 2).

Con la nuova distribuzione, più uniforme, è possibile ottenere dei buoni risultati nel training.

L'utilizzo del dataset SQUAD ha anche introdotto un'ulteriore incremento delle performance, portando a una drastica diminuzione della classificazione di esempi OT come domande lecite.

2.1.1.4. Training

Senza addentrarmi troppo nei dettagli per il momento, ho provato due metodi per effettuare il fine-tuning, dal momento che non si tratta di una semplice classificazione multiclasse:

- **multitask training** [12], dove invece di avere un solo layer finale di classificazione posto in cima al transformer pre-trainato di BERT ne sono presenti due posizionati in parallelo
- **hierarchical training** [13], dove si utilizza una tecnica (di tante varianti possibili) in cui si procede nella classificazione delle etichette «a step».

2.1.1.5. Risultati

In seguito al training, è risultato che il modello gerarchico permette di ottenere un maggiore F1-Score, che misura la precisione e il recall del modello. Il modello multitask ha ottenuto un F1-Score dell'86.9% sul topic primario e del 64.1% sul topic secondario, mentre il modello gerarchico ha ottenuto un F1-Score del 90.6% per il topic primario e una media del 75.1% per il topic secondario.

[1] Stanford Question Answering Dataset

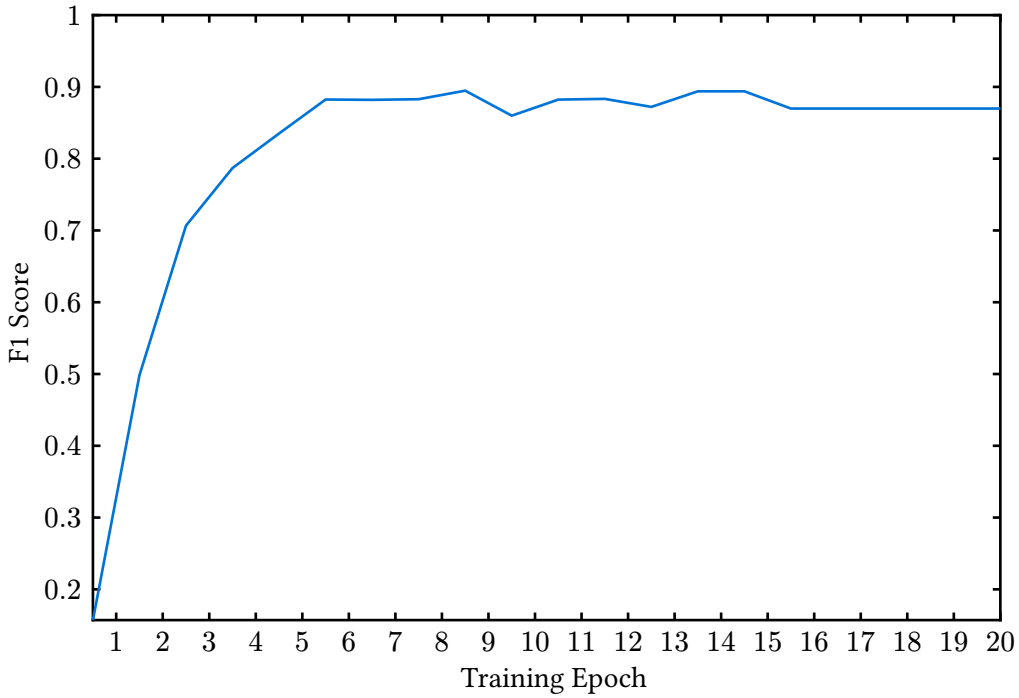

Grafico 1: F1 Score sul topic primario

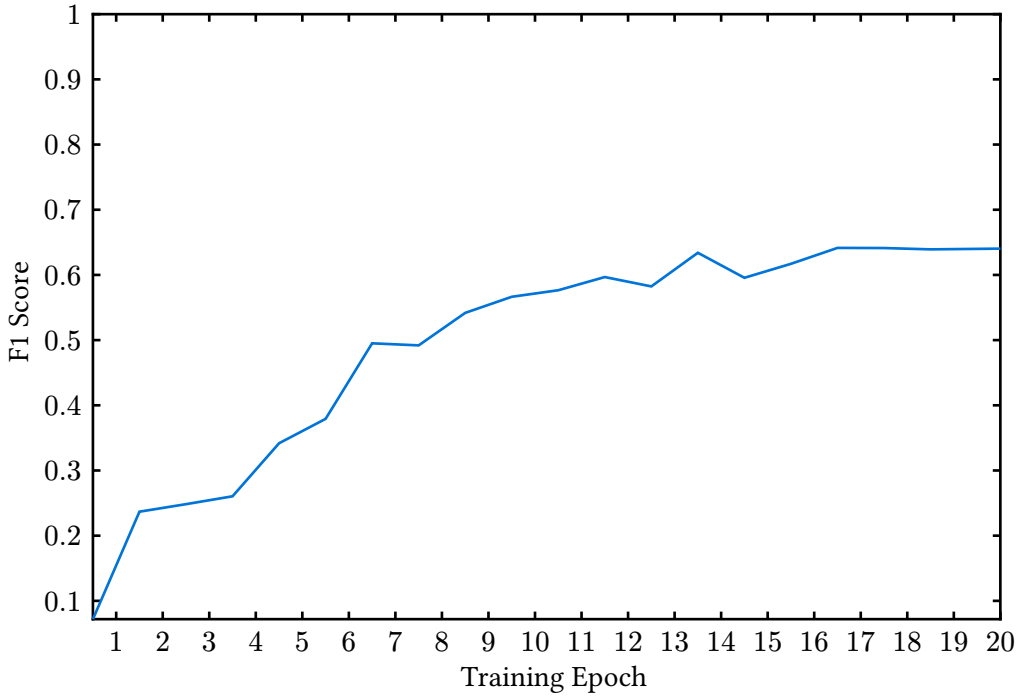

Grafico 2: F1 Score sul topic secondario

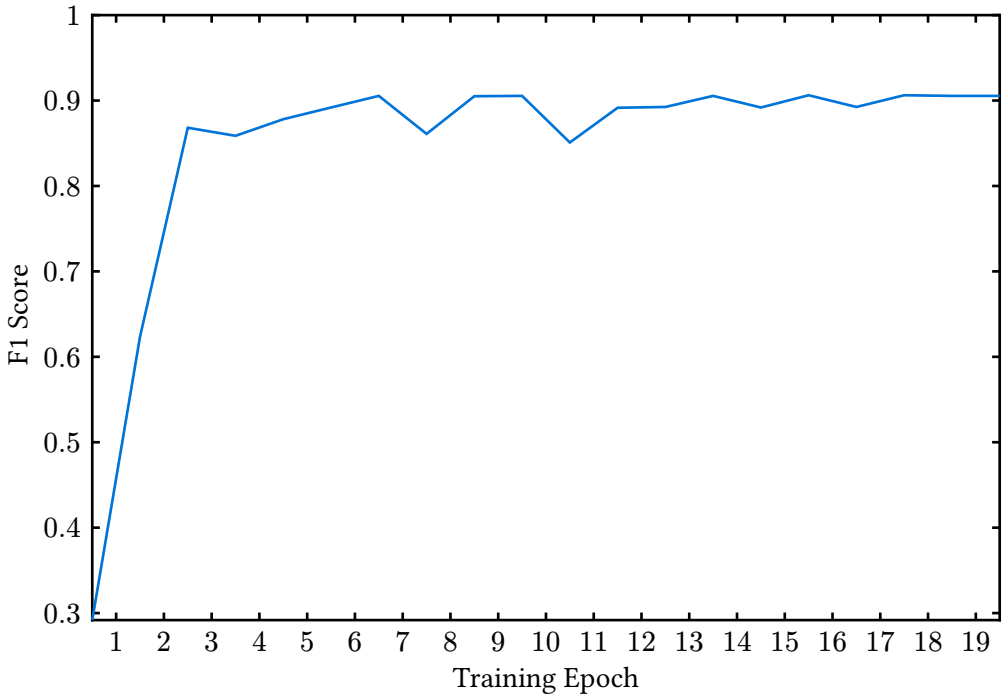

Grafico 3: F1 Score sul modello gerarchico

È possibile che il modello gerarchico sia più performante perchè isola i soli esempi che riguardano il topic secondario, permettendo di ottenere una maggiore precisione e recall.

2.1.2. Ulteriori ricerche da effettuare

Attualmente ho parlato con un mio amico del Politecnico, che mi ha suggerito di esplorare anche la via della classificazione mediante l'uso diretto degli embeddings prodotti da BERT, senza dover effettuare il fine-tuning. Questo potrebbe essere un'opzione interessante da esplorare, dal momento che permetterebbe di ottenere delle risposte più veloci e con il vantaggio di poter utilizzare anche un modello più grosso (con un numero maggiore di parametri) per ottenere una maggiore accuratezza, senza doverne effettuare il fine-tuning.

2.2. Riconoscimento dei complementi della domanda

Per alcuni dei generi di domande sono presenti anche degli elementi che dobbiamo saper estrarre per poter effettivamente rispondere ai quesiti degli utenti. Naturalmente possono essere di vario tipo, come ad esempio il riferimento a nodi o input dell'automa o degli archi.

Ho quindi effettuato ricerche per il training di modelli per la NER. Esistono modelli addestrabili basati sui transformers anche in questo caso, ma Spacy si è rivelata particolarmente efficace per questo task.

Ho etichettato manualmente le domande con Doccano [14], un tool open source per l'annotazione di testo. In seguito ho utilizzato Spacy per effettuare il training del modello NER.

Le performance sono soddisfacenti ed effettivamente il modello riesce a riconoscere con una buona precisione le entità di interesse.

3. Proposte

Il compilatore inizialmente doveva produrre soltanto un file AIML, con le interazioni e i dati già previsti e fissati nella pietra. Tuttavia, a parer mio, questo genere di approccio risulta poco flessibile e prono a produrre grosse quantità di dati ridondanti. Naturalmente è possibile anche utilizzare regole annidate e simili per mitigare queste problematiche, ma in un mondo dove oggi siamo in grado di interagire con una LLM, credo che bisogni cercare di puntare a proporre un «AIML» più adatto a quello che oggi ci si aspetta da un motore di chat e interrogazioni.

In più, utilizzare le regole in modo ricorsivo può portare a maggiore confusione e difficoltà nella comprensione di come le regole sono correlate tra di loro.

La difficoltà di ideare, mantenere e aggiornare le regole probabilmente in assoluto è il punto più problematico e importante da risolvere.

Integrare i dati nel file AIML si potrebbe anche rivelare problematico, dovendo introdurre variazioni non indifferenti alla struttura del formato.

Per questo motivo inizialmente ho dedicato tutto il tempo alla ricerca di tecniche efficaci per il riconoscimento dei quesiti dell'utente, finito con l'utilizzo di BERT dopo fine-tuning. Allo stesso modo, per identificare i soggetti di una domanda, il NER dovrebbe essere sufficiente.

Detto questo, vorrei proporre un nuovo genere di sistema, di cui il compilatore è solo una parte. Nello specifico, si tratta di una pipeline che integra diversi step, partendo da un compilatore e finendo in un motore di esecuzione per il question answering su topic ristretti e specifici.

3.1. Cosa fa il compilatore

Il compilatore invece di dover costruire un file AIML in output, lavora su più stadi, e necessita di una configurazione più approfondita.

Nello specifico, richiede i seguenti dati:

1. Per ogni genere (classe) di domanda possibile nel dominio da analizzare, devono essere forniti i dataset di domande che possano essere usati per comporre i dati con cui viene effettuato il fine-tuning di BERT.
2. Come per le classi di domande, ogni genere di Named Entity potenzialmente richiesta deve essere compresa in un dataset taggato per addestrare un modello che si occupi di NER.
3. Infine, non possono mancare anche i dati effettivi sui quali deve essere costruita una piccola knowledge base. Dato che ogni classe può richiedere informazioni diverse dalla knowledge base, possono essere utili delle regole per recuperare i dati relativi alla domanda appena posta. La regola può essere espressa in JSONPath [15] o JMESPath [16] o un altro linguaggio di query da decidere (un'opzione decisamente più complessa può essere ad esempio SPARQL [17], proveniente dal dominio del web semantico). Questo potrebbe anche lasciare spazio alla possibilità di interrogare risorse esterne per ottenere informazioni aggiornate.

Non tutto deve essere per forza implementato nel corso di questa parte della tesi (specialmente quello che riguarda la flessibilità per interrogazioni della KB che esulano dal dominio degli automi), tuttavia vorrei comunque lasciare sufficiente flessibilità e estensibilità nel sistema da permettere aggiunte indolori.

Il risultato è un insieme di file che contengono tutto il necessario per permettere al motore di interrogazione di reagire alle domande degli utenti, e di estrarre le informazioni necessarie per rispondere.

3.1.1. Componenti della configurazione

Il file di configurazione JSON deve contenere le seguenti informazioni:

1. Una sezione riguardante il training, che include:
   - I dati relativi all'automa, che verranno estratti per costruire la knowledge base. Ipoteticamente i dati possono essere forniti in vari formati; dovrebbe essere piuttosto facile supportare formati come Graphviz Dot [18], JSON o XML.
   - Il/I dataset di interazioni, che contengono le domande che l'utente può fare, già etichettati per classe di domanda. Possono essere presenti gerarchie di classi di domande, in modo da poter avere una maggiore flessibilità.
   - Il/I dataset di NER, che contengono le domande etichettate con le entità che si vogliono poter estrarre.

```
"training": {
  "domain_data": {
    "type": "automaton",
    "data": "path_to_automaton.dot"
  },
  "interactions": {
    "type": "dataset",
    "data": "path_to_interactions_dataset.csv"
  },
  "ner": {
    "type": "dataset",
    "data": "path_to_ner_dataset.csv"
  },
}
```

2. Le classi di interazione, che contengono le regole per rispondere alle domande.
   Ogni possibile interazione deve indicare:
   - Una descrizione (utile a fini di documentazione): `description`
   - Un'etichetta per la classe dei dati usati nel training che riguardano l'interazione: `label`
   - Una lista di possibili slot da riempire: `slots`. Ogni slot ha alcuni dettagli:
     - Un nome: `name`. Questo corrisponde al nome dell'entità che si vuole estrarre dalla domanda basandoci sul modello di NER addestrato
     - Una descrizione: `description`
     - Un'indicazione se è opzionale: `optional`
   - Una lista di possibili risposte, che possono contenere dei placeholder per i dati estratti: `answers`.
     Queste risposte possono essere scritte in un linguaggio di templating, come ad esempio Handlebars [19], oppure possono essere indicate anche delle configurazioni più complesse che permettono di fornire il necessario ad una LLM per generare una risposta più complessa usando i dati estratti (o anche da una memoria interna).
   - Una query per estrarre i dati dalla knowledge base che verranno usati per la risposta: `query`.

Questa query può essere espressa in JSONPath o JMESPath (o altri linguaggi che potrebbero essere identificati come più adatti).

- Una lista di sottoclassi, `subclasses`, che contengono le stesse informazioni di una classe di interazione. Questo permette di creare una gerarchia di classi di interazione, in modo da poter avere una maggiore flessibilità.
  Se una classe contiene delle sottoclassi, questa non può avere una query associata, e le risposte devono essere generate dalle sottoclassi. Ipoteticamente si potrebbe pensare anche di avere più risposte da più sottoclassi se queste matchano, oppure informare l'utente che il sistema ha compreso la domanda ma che non è in grado di rispondere (e magari suggerire delle domande più specifiche).

Ecco come potrebbe essere strutturata la sezione delle classi di interazione, per lasciare un esempio concreto:

```
"interaction_classes": {
  "InizioConversazione": {
    "label": "start",
    "description": "Gestisce i saluti iniziali dell'utente.",
    "answer": [
      "Ciao! Come posso aiutarti oggi?",
      "Salve! Sono qui per rispondere alle tue domande."
    ]
  },
  "DomandaStatoAutoma": {
    "label": "state",
    "description": "Richieste di informazioni riguardo gli stati dell'automa",
    "subclasses": {
      "StatoStart": {
        "label": "start",
        "description": "Dettagli sugli stati di start",
        "slots": [
          {
            "optional": true
            "name": "state"
          }
        ],
        "query": "$.automaton.states[?(@.type=='start')]",
        "answers": [
              "L'automa ha ${res.length} stati di start: ${res.map(s =>
s.name).join(', ')}",
         "Gli stati di start dell'automa sono: ${res.map(s => s.name).join(',
')}"
        ]
      },
      "InformazioniOrdine": {
        "descrizione": "Domande riguardanti lo stato di un ordine.",
        "sottoclassi": {},
        "slot": ["numero_ordine"],
        "query": "$.ordini[?(@.numero=='${numero_ordine}')]",
        "risposte": [
          "Lo stato del tuo ordine ${numero_ordine} è: ${risultato.stato}",
          "L'ordine ${numero_ordine} è attualmente: ${risultato.stato}"
```

```
            ]
          }
        }
      }
    }
  }
```

Naturalmente questa è solo una proposta, e il formato del file di configurazione può essere modificato in base alle esigenze. Per iniziare a costruire il sistema, si potrebbe partire con un formato più semplice e poi aggiungere funzionalità più complesse in funzione di quello che vediamo che è necessario per lavorare nel dominio degli automi.

3.2. Cosa fa il motore di interrogazione

Il motore di interrogazione è il componente che si occupa di ricevere le domande degli utenti e di rispondere in base alle regole definite nel file di configurazione.

Dovrebbe tenere traccia dello stato della conversazione, come AIML già fa utilizzando la memoria e comandi come `<think>`.

Verrebbe costruito in python, avendo bisogno di interagire con il modello di BERT per il riconoscimento delle domande e con Spacy per il riconoscimento delle entità.

L'idea sarebbe di implementarlo come una serie di moduli o una libreria, in modo da poterlo poi integrare separatamente in tanti potenziali sistemi diversi (come ad esempio un bot Telegram, un'interfaccia web, un'applicazione mobile, ecc.).

Per il nostro caso, ad esempio potrebbe essere integrato in una API scritta con FastAPI [20], che permette di costruire API RESTful in modo molto semplice e veloce.

Ritengo che spostarci da AIML a qualcosa di costruito apposta permetterebbe di ottenere un sistema più flessibile e più adatto ai tempi attuali, e darebbe anche una risposta ad alcuni dubbi che erano sorti durante le varie volte che ci siamo confrontati in riunione, come l'integrazione delle LLM, la flessibilità delle regole, la gestione delle variabili quali la conoscenza e le competenze dell'utente o la memoria del sistema.

4. Bibliografia

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

[2] S. Gururangan *et al.*, «Don't Stop Pretraining: Adapt Language Models to Domains and Tasks». [Online]. Disponibile su: https://arxiv.org/abs/2004.10964

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

[4] V. Sanh, L. Debut, J. Chaumond, e T. Wolf, «DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter». [Online]. Disponibile su: https://arxiv.org/abs/1910.01108

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

[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. [Online]. Disponibile su: https://arxiv.org/abs/2408.00118

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

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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

[15] Stefan Goessner, «JSONPath - XPath for JSON». [Online]. Disponibile su: https://goessner.net/articles/JsonPath/

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

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

[18] «Graphviz - Graph Visualization Software». [Online]. Disponibile su: https://graphviz.org/

[19] «Handlebars - Minimal Templating on Steroids». [Online]. Disponibile su: https://handlebarsjs.com/

[20] «FastAPI - Fast (high-performance) API framework for Python». [Online]. Disponibile su: https://fastapi.tiangolo.com/