Documentazione Chatbot

Romano Mancini

13 settembre 2024

Indice

1 Introduzione

1.1 Panoramica generale

L'applicazione sviluppata si colloca nel settore della finanza personale, sfruttando modelli di AI per semplificare l'interazione con un database di spese personali.

Interagendo mediante comandi vocali in linguaggio naturale, l'utente sarà in grado di inserire all'interno dell'applicazione le proprie spese, specificandone l'importo e una breve descrizione. Algoritmi di NLP e RAG, affiancati da un LLM, provvederanno a generare una query SQLite di inserimento nel database. Qualora l'utente dovesse invece porre una domanda, sarà generata una query di ricerca.

Attualmente in fase embrionale, il progetto ha il potenziale per espandersi verso implementazioni su piattaforme Mobile, Web o Desktop, aggiungendo ulteriori funzionalità quali l'estrazione di insight sulle proprie spese e proiezioni rispetto al futuro.

Questa documentazione si prefigge l'obiettivo di fornire una guida completa per sviluppatori e utenti tale da consentire un'agevole comprensione, utilizzo e sviluppo dell'applicazione.

1.2 Motivazioni

Il periodo storico di forte incertezza economica acuisce sempre più l'interesse del pubblico nei confronti di strumenti che possano aiutare nelle gestione delle proprie finanze. [13] Il coinvolgimento collettivo nel settore è aumentato del 10% rispetto al 2021, con i giovani che si mostrano particolarmente attenti al loro presente e futuro finanziario. Nel 2023 l'Italia conferma il trend positivo, registrando una percentuale dell'85% (dal 76% del 2021) sul totale del campione di intervistati che si dichiarano "molto" o "abbastanza" interessati. Nonostante questo, il 2023 ha visto anche il raggiungimento del minimo storico dei risparmi degli italiani [14], che, tuttavia, storicamente si è sempre confermato essere un popolo incline al risparmio.

Appare quindi particolarmente evidente una discrepanza tra il bisogno e la volontà del pubblico nei confronti di un maggiore risparmio, e la loro incapacità di determinarlo.

L'impiego dell'intelligenza artificiale risulta significativo, in quanto mira a semplificare di molto un processo che può risultare particolarmente fastidioso, quale quello di tracciare le proprie spese. Riuscendo poi a trarre degli insight personalizzati, l'utente non avrà più la necessità di dedicarsi in modo attivo allo studio delle proprie disponibilità economiche. Essendo l'80% degli italiani riluttante nei confronti della gestione del patrimonio [11], che si ritiene essere una cosa troppo complessa, quest'applicazione ha il potenziale di ritagliarsi un ruolo importante nel contesto sociale contemporaneo.

2 Richiami teorici

2.1 Introduzione all'Intelligenza Artificiale

L'intelligenza artificiale, nel suo significato più ampio, è la capacità (o il tentativo) di un sistema informatico di simulare l'intelligenza umana attraverso l'ottimizzazione di funzioni matematiche [2]. Si tratta di un settore specifico delle scienze informatiche, che si articola in tecnologie diverse. Tra queste, emergono:

- Machine Learning;
- Computer Vision;
- Natural Language Processing;
- Document Intelligence;
- Knowledge Mining;
- IA Generativa.

2.2 Natural Language Processing

Per far sì che i sistemi informatici possano interpretare il soggetto di un testo in modo simile a come lo farebbe un essere umano, viene utilizzata l'elaborazione del linguaggio naturale (NLP), un'area dell'intelligenza artificiale che si occupa di comprendere il linguaggio scritto o parlato e rispondere di conseguenza.
In questo contesto intervengono i Large Language Model (LLM), modelli computazionali in grado di generare linguaggio o svolgere altre attività di elaborazione del linguaggio naturale. Il nome deriva dal fatto che il loro processo di addestramento prevede l'utilizzo di un'enorme quantità di testo.

2.3 RAG

Una delle tecniche che più riescono a migliorare l'output degli LLM è la Retrieval-Augmented Generation (RAG). Mediante la stessa, si riesce a far si che il modello faccia riferimento a una base di conoscenza autorevole al di fuori delle sue fonti di addestramento prima di generare una risposta.

2.4 Ricerca vettoriale

La ricerca vettoriale è un metodo comune per archiviare e cercare dati non strutturati (come il linguaggio naturale). L'idea alla base consiste nel memorizzare, al posto del testo, un vettore numerico che ne rappresenti le caratteristiche semantiche. Se c'è la necessità di effettuare una ricerca, si può tramutare la ricerca in vettore e cercare il risultato che più vi somiglia.

2.5 Conclusioni

Avendo a che fare con un'applicazione che si premette l'obiettivo di interagire sia con il linguaggio naturale dell'utente che con un database, appare evidente quanto NLP, RAG e ricerca vettoriale saranno impiegati necessariamente in collaborazione al fine di adempiere allo scopo.

3 Tecnologie utilizzate

3.1 Framework per lo sviluppo dell'app

Lo sviluppo dell'applicazione (primariamente mobile) è avvenuto mediante Flutter [4], un framework open-source creato da Google per la creazione di interfacce native per iOS, Android, Linux, macOS e Windows. Flutter presenta una bassa barriera d'ingresso, poiché non è necessario conoscere HTML, CSS o JavaScript. Questo aspetto risulta particolarmente vantaggioso in particolare per studenti o programmatori novizi. Risulta sufficiente avere delle buone basi di programmazione a oggetti per trovare la transizione particolarmente intuitiva.

3.2 Framework per sviluppo AI

Per la creazione di una pipeline coerente di modelli di intelligenza artificiale si è impiegato LangChain [9], un framework di orchestrazione open source per lo sviluppo di applicazioni che utilizzano modelli linguistici di grandi dimensioni (LLM, large language model). Gli strumenti e le API di LangChain sono disponibili in librerie basate su Python e JavaScript, le quali semplificano il processo di creazione di applicazioni basate su LLM come chatbot e agenti virtuali.

3.3 Modello di LLM scelto e soluzione Cloud impiegata

Il modello di LLM scelto è stato llama3 (Meta) [10], in particolare la sua versione da 70 miliardi di parametri. Si tratta, al momento, di uno dei modelli più avanzati. Supporta il fine-tuning, riesce a gestire contesti particolarmente estesi ed è efficiente da un punto di vista computazionale.

Come è comprensibile, nonostante l'ottimo lavoro di ottimizzazione svolto sul modello, la possibilità di farlo girare in locale è stata immediatamente scartata in quanto non realizzabile. Ci si è quindi rivolti ad una soluzione Cloud di nome Groq [8] (da non confondere con Grok, il modello di AI Generativa proposto da X, ex Twitter).

Groq, Inc. è un'azienda statunitense specializzata in intelligenza artificiale che sviluppa un "AI

accelerator application-specific integrated circuit" (ASIC) da loro denominato "Language Processing Unit" (LPU), progettato per accelerare le prestazioni di inferenza dei workload AI. Esempi di carichi di lavoro AI che possono essere eseguiti sulla LPU di Groq includono modelli linguistici di grandi dimensioni, classificazione delle immagini, rilevamento di anomalie e analisi predittiva.

La scelta di Groq è stata dettata dalla facilità di utilizzo, che di fatto si limita ad una chiamata API, nonchè ai limiti particolarmente laschi realtivi al piano gratuito, che consente 30 chiamate al minuto. Inizialmente si era pensato di utilizzare l'API di Google per accedere a Gemini, il loro modello di LLM. Essendo però fornite solo 5 chiamate al minuto gratuitamente, la fase di testing risultava significativamente lenta. Peraltro, Gemini è decisamente meno potente dell'attuale llama3.

3.4 Database

Per il database, necessario a contenere i dati inseriti dall'utente, si è deciso di procedere con un'implementazione in SQLite3 [15], una libreria open source scritta in C che implementa un DBMS SQL di tipo ACID incorporabile all'interno di applicazioni.
La scelta è stata dettata dalla volontà di avere un database che sia veloce, di piccole dimensioni e affidabile, così da non dover far caricare all'utente i suoi dati su cloud.

3.5 Login Utente

Lo sviluppo dell'app è ultimato dall'interazione con Firebase [3], una piattaforma cloud che offre una serie di servizi particolarmente utili per lo sviluppo di applicazioni web e mobile. In particolare, si è usufruito del servizio di registrazione e login degli utenti, che prevede anche l'invio di email di conferma.

3.6 Linguaggi di programmazione

Per lo sviluppo delle interfacce Flutter impiega il linguaggio Dart [1], particolarmente sfruttato anche nel settore delle web app. Si tratta di un linguaggio open source, sviluppato da Google. Il prospetto è quello di un supporto a lungo termine, che punti a sopperire a tutte le mancanze e le incongruenze del "rivale" JavaScript. Dart si caratterizza per essere molto semplice da imparare, specie se si ha esperienza con programmazione a oggetti o linguaggi come C#. Tuttavia, essendo un linguaggio recente, la community è ancora relativamente ristretta.

La parte di sviluppo orientata all'implementazione dei modelli di AI è stata svolta interamente in Python [12], linguaggio di riferimento per questo tipo di applicazioni. La scelta è stata pressocchè forzata dalla presenza di innumerevoli librerie che semplificano enormemente lo sviluppo AI. Peraltro, la community è estremamente ampia e il linguaggio è open-source. E' evidente quanto Python perda in efficienza rispetto ad altri linguaggi, ma questa mancanza è trascurabile rispetto alla già importante pesantezza dei modelli impiegati.

3.7 Controllo delle versioni

Il controllo delle versioni, noto anche come controllo del codice sorgente, è la pratica di monitorare e gestire le modifiche al codice del software. Nell'informatica moderna i software adibiti al controllo delle versioni risultano pressoché essenziali, specialmente in ambiti in cui più programmatori collaborano sullo stesso progetto. In locale, il sistema di riferimento è Git [5], software per il controllo di versione distribuito utilizzabile da interfaccia a riga di comando, creato da Linus Torvalds nel 2005. Procedendo, ci si è spostati dapprima in cloud su GitHub [6], per poi convergere verso GitLab [7]. Il vantaggio di GitLab è quello di essere installabile anche su un server locale, garantendo una maggiore indipendenza e privacy nei confronti di quello che è il codice scritto.

Ricapitolando, la toolchain utilizzata è la seguente:

- Framework per la costruzione dell'app: Flutter;
- Framework per sviluppo AI: LangChain;
- Modello LLM: llama3, 70b;
- Soluzione Cloud: Groq.

- Controllo delle versioni: Git (Github, Gitlab)
- Database: SQLite3
- Linguaggi di programmazione: Dart, Python
- Servizi ausiliari: Firebase

4 Spiegazione del codice

La prima parte del codice scritto consiste nella costruzione di un servizio che, data una qualunque domanda proveniente dall'utente, generi una query SQLite3 da inserire nel database in modo tale da estrarre i dati richiesti.

4.1 Chiamata API

La chiave API utile ad effettuare chiamate a Groq è stata salvata in un file .env, già incluso nel .gitignore. Si tratta quindi di un file che non sarà caricato all'interno delle repository dei diversi software di gestione delle versioni. Nel file Python sarà riportato:

```
import os
from dotenv import load_dotenv
# carica il contenuto del file .env
load_dotenv()
# salva la chiave API come environment variable
os.environ["GROQ_API_KEY"] = os.getenv("GROQ_API_KEY")
```

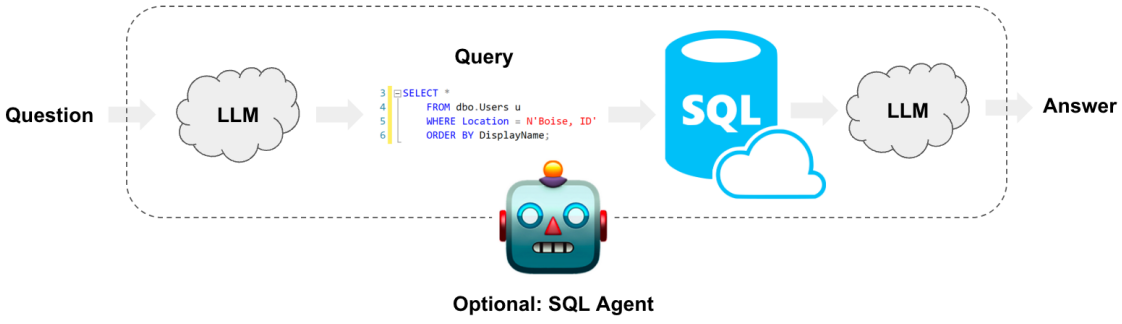
In questo modo, di fatto, si può pubblicare il codice in modo sicuro senza rischiare che la chiave API appaia in chiaro.

4.2 Connessione del database e definizione della chain

Lo step successivo è quello di connettere il database in locale al codice. Per farlo, si sfrutta la libreria LangChain in questo modo:

```
from langchain_community.utilities import SQLDatabase
db = SQLDatabase.from_uri("sqlite:///googleDb.sqlite3")
```

Successivamente, si procede con la creazione della chain. Con il termine 'chain' ci si riferisce ad una serie di chiamate, che possono riguardare un LLM, un tool, o una fase di pre-elaborazione dei dati. Ad alto livello, la chain utilizzata in questo momento è così rappresentabile:



Fonte: https://shorturl.at/6ke7V

Si procede quindi con la definizione dell'LLM da utilizzare all'interno della funzione di libreria di LangChain che permette la creazione di una prima chain.

```
from langchain.chains import create_sql_query_chain
from langchain_groq import ChatGroq

llm = ChatGroq(
    model="llama3-groq-70b-8192-tool-use-preview",
```

```
    temperature=0,
    max_retries=2,
)
chain = create_sql_query_chain(llm, db)
```

La "temperature" è un iperparametro molto importante degli LLM, il quale regola la casualità dell'output. Di solito è un numero compreso tra 0 e 1. Una "temperature" di 0 farà si di garantire risposte molto prevedibili, "più deterministiche". Essendo l'output una query, si preferisce che ci sia quanta meno variabilità possibile nelle risposte.

4.3 System prompt, output parser

Il mero utilizzo di un LLM non è sufficiente: è necessario impiegare un system prompt di supporto, che dia delle regole all'LLM in modo da renderlo più adatto a generare del codice "più corretto". Una porzione piuttosto importante del tempo di sviluppo di questo servizio è stata dedicata alla creazione di un system prompt quanto più coerente possibile, che aiutasse l'LLM a generare risultati migliori. All'interno del system prompt si è proceduto anche con l'inserimento della data e ora attuale: il modello faceva fatica a reperirla, poichè probabilmente il suo "today" si riferiva al giorno in cui è stato allenato.
Al termine della chain si riporta un output parser, utile a rendere leggibile la risposta.

```
from datetime import datetime
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
llm = ChatGroq(
    model="llama3-groq-70b-8192-tool-use-preview",
    temperature=0,
    max_retries=2,
)
db.run("DROP TABLE IF EXISTS 'EXPENSES';")
chain = create_sql_query_chain(llm, db)
today = datetime.today().strftime('%Y-%m-%d')
time = datetime.today().strftime('%H:%M:%S')
cur_year = datetime.now().year
system = """
            You are given a database where there are all the items/purchases done
            by a person. Whatever you do, you must not output the word INTERVAL.
            Whatever you do, you must not use syntax like date('2024-09-13') - 7
            days.
            Whatever you do, you must not subract to dates.
            - Use SQLITE3 syntax.
            Follow this rules:
            - only query existing tables
            - Use current date only if other time is not given.
            - Month Year: >= <=
            - The present year is {cur_year}
            - If you need to subtract time from a date, use a syntax like
            date('2024-09-12', '-30 days').
            - "What is the least expensive item bought" and similiar requests
            want the description (yes), not the price (no)
            - Don't use the INTERVAL keyword: adjust the date calculations using
            strftime or date functions supported by SQLite.
            - Don't use BETWEEN: go for direct comparisons.
            - If you need to go in the past, remember that today is {today}.
            - How many: COUNT
            - Questions SIMILIAR TO 'What is the least expensive item' MUST
            output an item.
            - If there are singular names, the probability of a single row is
            higher.
            - MUST Suppose the year {cur_year}, UNLESS not already given in the
            question.
            - If no year/month/day/time is provided, test all possible
```

```
            year/month/day/time.
            - Don't use built-ins unless comparing dates: today is {today}, time
            is {time}.
            - Use time only when specified (e.g., 6 PM, 18:00).
            - For date comparisons, prefer >= or <=.
            - If time is given without a date, use only the time.
            - Use DISTINCT for category queries.
            - evenings or mornings without date do not care about the date but only
            about the time
            - Output the queries only.
            """
prompt = ChatPromptTemplate.from_messages(
    [("system", system), ("human", "{query}")]
).partial(dialect=db.dialect,cur_year=cur_year, time=time, today=today)
validation_chain = prompt | llm | StrOutputParser()
full_chain = {"query": chain} | validation_chain
```

4.4 Funzione "get_from_database"

Per una questione di comodità di utilizzo si è preferito racchiudere in una funzione la chiamata alla chain. Durante la fase di stesura del system prompt, ci si è resi conto del fatto che lo stesso fosse divenuto troppo esteso: l'LLM iniziava a non ubbidirvi più totalmente. Per questa ragione, si è deciso di introdurre una semplice logica if/else tale da arricchire in modo più deterministico le domande dell'utente in presenza di parole chiave come "item", "category" o "percentage". Sebbene sia opinabile che una logica del genere si allontani troppo dal modus operandi tipico di un'applicazione AI, c'è da tenere in conto che così facendo tramite poche linee di codice si riesce a garantire un aumento considerevole dell'accuratezza del modello. In fase di sviluppo, questo è sembrato un ottimo compromesso, utile a raggiungere l'obiettivo finale senza focalizzarsi troppo su dettagli di natura più concettuale e astratta.
La query così ottenuta viene inserita nel database. Effettuati piccoli ritocchi di parsing e arrotondamento dei numeri a virgola mobile a due decimali, si procede con lo stampare in output il risultato.

```
def get_from_database(printQuestion, printQuery, question, db, chain):
    if "item" in question:
        question = question.replace("item", "thing")
        if "expensive" in question:
          question = question + " Need the description AND the price. Use ORDER BY"
    if "items" in question:
        question = question.replace("items", "things")
    if "category" in question:
        question = question + " Select only one category."
    if "categories" in question:
        question = question + " Select all distinct categories without limits."
    if "percentage" in question:
        question = question + " Use divisions/SUM and moltiplication*100."

    response = chain.invoke({"question": question})

    response = response.replace("SQLQuery:", "").replace("```sql", "")
               .replace("```", "").replace("\n",";").strip()

    if (printQuestion):
        print(question)

    if (printQuery):
        print(response)

    result = db.run(response[response.find("SELECT"):])
    result = result.replace("[(","").replace(",)]","").replace("(","")
             .replace(")","").replace(",,",",").replace("'","").replace("]","")
```

```
if not result:
    print("No match found.")
else:
    # rounding decimals to the second place
    for word in result.split():
        if "." in word:
            try:
                fl = float(word)
                fl = round(fl, 2)
                fl = format(fl, '.2f')
                result = result.replace(word, str(fl))
            except ValueError:
                continue
    print(result)

print("---")
```

5 Testing

5.1 Descrizione del processo di test

La parte di test è stata eseguita manualmente: si è proceduto con la stesura di 38 possibili domande sulle quali si è modellato il system prompt. La funzione get_from_database() è stata usata in "modalità debug", con i booleani printQuestion e printQuery impostati a True.
In questo modo, ad ogni chiamata al modello venivano stampati:

- La domanda in linguaggio naturale;
- La query estratta dall'LLM;
- Il risultato della query.

5.2 Risultati ottenuti

Prevedibilmente, sulle domande sfruttate per definire il system prompt, il modello si comporta in modo eccellente, raggiungendo il 100% di correttezza delle query da un punto di vista sintattico, e il 98% di correttezza da un punto di vista contenutistico.
Testando il modello su altre 20 domande create ex-novo, i risultati sono variabili: la correttezza sintattica rimane del 100%, ma a livello contenutistico solo il 70% delle query ottiene il risultato sperato. A sopperire a tali mancanze interverranno modelli di RAG, decisamente più adatti a ricerche particolarmente precise.

Riferimenti

[1] *Dart (Linguaggio di programmazione)*. URL: https://dart.dev/.

[2] *Definizione di Intelligenza Artificiale*. URL: https://it.wikipedia.org/wiki/Intelligenza_artificiale.

[3] *Firebase*. URL: https://firebase.google.com/.

[4] *Flutter*. URL: https://flutter.dev/.

[5] *Git*. URL: https://git-scm.com/.

[6] *GitHub*. URL: https://github.com/.

[7] *GitLab*. URL: https://about.gitlab.com//.

[8] *Groq*. URL: https://groq.com/.

[9] *Langchain HomePage*. URL: https://www.langchain.com/.

[10] *llama3*. URL: https://ollama.com/library/llama3.

[11] Panorama. *Gli italiani sanno poco di finanza, e ci rimettono*. URL: https://www.panorama.it/economia/italiani-finanza-investimenti-soldi-banche.

[12] *Python (Linguaggio di programmazione)*. URL: https://www.python.org/.

[13] Corriere della Sera. *Finanza, gli italiani sono sempre più interessati a risparmio e investimenti ma c'è un gap nell'educazione*. URL: https://www.corriere.it/economia/finanza/23_ottobre_25/finanza-italiani-sono-sempre-piu-interessati-risparmio-investimenti-ma-c-gap-nell-educazione-d5e9e47a-7328-11ee-b12d-3a48784b526b.shtml.

[14] Sole24Ore. *Istat, nel 2023 risparmio italiani al minimo storico. Imposte famiglie in aumento di 24,6 miliardi, Irpef al top*. URL: https://www.ilsole24ore.com/art/istat-2023-risparmio-italiani-minimo-storico-AFfcPzLD.

[15] *SQLite3*. URL: https://www.sqlite.org/.