Documentazione Chatbot

Romano Mancini

8 ottobre 2024

Indice

1 Introduzione: Richiami teorici

1.1 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 [5]. 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.

1.2 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.

1.3 RAG

Tra le tecniche che più riescono a migliorare l'output degli LLM emerge 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.

1.4 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, la si può tramutare in vettore e cercare il risultato che più vi somiglia.

1.5 DBSCAN

DBSCAN è un algoritmo di clustering che cerca di raggruppare i dati basandosi su come i punti sono distribuiti nello spazio. A differenza di altri algoritmi che impongono una struttura specifica, come i cluster circolari o ellittici, DBSCAN identifica aree dense di punti, chiamate "cluster", e ignora quelli isolati o sparsi, trattandoli come rumore.
L'idea di base è che i punti in un cluster devono trovarsi a una certa distanza massima l'uno dall'altro, chiamata **eps**. Se un punto ha almeno un certo numero di altri punti (convenzionalmente definito come **min_samples**) nel suo intorno di raggio eps, viene considerato un punto "centrale". Questo punto centrale diventa il nucleo del cluster e si espande includendo i punti vicini, che a loro volta possono far parte del cluster se soddisfano lo stesso criterio di densità.
Il vantaggio principale di DBSCAN è che può identificare cluster di qualsiasi forma e che non richiede di specificare a priori il numero di cluster, come fa ad esempio il K-Means. Inoltre, è in grado di gestire bene il rumore, isolando i punti che non appartengono a nessun cluster.

1.6 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.

2 Introduzione all'applicazione sviluppata

2.1 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 del database integrato nell'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. Qualora l'applicazione dovesse discernere la volontà dell'utente di generare dei grafici riguardanti le sue spese, ne saranno generati di interattivi.

Ognuno degli obiettivi appena presentati viene raggiunto mediante specifici modelli di intelligenza artificiale, garantendo all'utente ampia libertà nonchè flessibilità di utilizzo. Attualmente in fase embrionale, il progetto ha il potenziale per espandersi verso implementazioni su piattaforme Mobile, Web o Desktop, eventualmente 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.

2.2 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. [21] 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 ha confermato 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 [22]. Si tratta di un dato che sorprende, vista la tendenza storica degli stessi alla parsimonia.

Appare quindi particolarmente evidente una discrepanza tra il bisogno (e la volontà) del pubblico di migliorare la propria gestione delle finanze, rispetto alla capacità di farlo.

L'impiego dell'intelligenza artificiale risulta significativo, in quanto mira a semplificare di molto un processo che può risultare particolarmente insidioso, quale quello di tracciare le proprie spese. Riuscendo (in aggiornamenti successivi) a trarre degli insight personalizzati, l'utente non necessiterà più di avere importanti competenze finanziarie per riuscire a gestire il proprio patrimonio. Considerando che l'80% degli italiani si dichiara riluttante nei confronti della gestione delle finanze [19], ritenendola una cosa troppo complessa, quest'applicazione ha il potenziale di ritagliarsi un ruolo importante nel contesto sociale contemporaneo.

3 Tecnologie utilizzate

3.1 Framework per lo sviluppo dell'app

Lo sviluppo dell'applicazione (primariamente mobile) è avvenuto mediante Flutter [8], un framework open-source ideato da Google per la creazione di interfacce native per iOS, Android, Linux, macOS e Windows.

3.2 Framework per sviluppo AI

Per la creazione di una pipeline coerente di modelli di intelligenza artificiale si è impiegato LangChain [16], 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

Tra i modelli di LLM scelti spicca per frequenza di utilizzo Llama3 (Meta) [17], 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 [13] (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 [23], 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.
In fase di sviluppo ci si è resi conto della necessità di avere un database secondario che desse la possibilità di effettuare ricerche vettoriali: senza quest'ultimo la ricerca mediante RAG sarebbe stata impraticabile. A tal proposito, quindi, si è impiegata la libreria Chroma [2], all'interno del framework di LangChain. Si tratta di una soluzione gratuita e open-source, ben documentata e perfettamente adatta allo scopo.

3.5 Login Utente

Lo sviluppo dell'app è ultimato dall'interazione con Firebase [7], 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 Libreria di Creazione Grafici

Per la creazione dei grafici si è utilizzata la libreria Chart.js [1]. La scelta è stata dettata dalla necessità di avere delle rappresentazioni interattive da poter facilmente impiegare in progetti successivi.
All'interno del progetto si sono anche sfruttate le librerie Python PandasAI [18] e ggplot [9] a tale scopo. Tuttavia, l'output di queste librerie è in formato SVG. Necessitando di maggiore flessibilità e interattività, la versione finale del progetto si basa su Chart.js.

3.7 Linguaggi di programmazione

Per lo sviluppo delle interfacce, Flutter impiega il linguaggio Dart [4], 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 [20], 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.

Come già spiegato, per la creazione di grafici si è reso necessario l'impiego di JavaScript [15], insieme ad HTML [14] e CSS [3]. Mediante questi linguaggi si è creata una semplice pagina web (visualizzata in localhost), aggiornata ad ogni grafico creato. Al termine del progetto, questi grafici saranno inclusi direttamente all'interno dell'applicazione mobile, sebbene il codice originale rimarrebbe di grande valore per eventuali applicazioni web sviluppate in futuro.

3.8 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 [10], 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 [11], per poi convergere verso GitLab [12]. 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.

3.9 Riassunto

Ricapitolando, la toolchain utilizzata è la seguente:

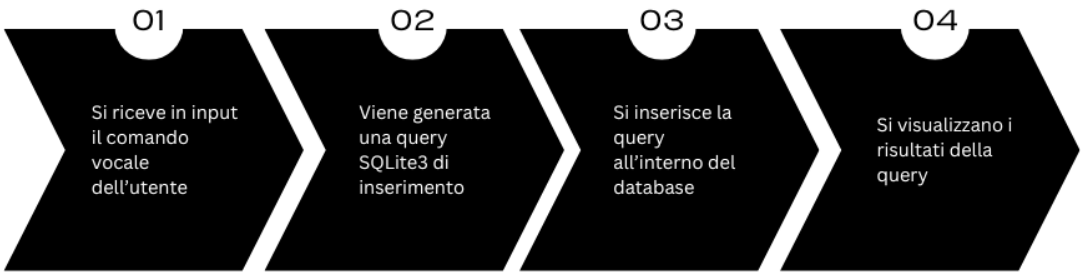
- **Framework per la costruzione dell'app**: Flutter;
- **Framework per sviluppo AI**: LangChain;
- **Modello LLM**: llama3, 70b;
- **Soluzione Cloud**: Groq;
- **Controllo delle versioni**: Git (Github, Gitlab);
- **Database**: Chroma, SQLite3;
- **Librerie di creazione grafici**: Chart.js (PandasAI, ggplot);
- **Linguaggi di programmazione**: Dart, Python, HTML, CSS, JavaScript;
- **Servizi ausiliari**: Firebase;

4 Schema riassuntivo di alto livello

Il programma è sostanzialmente diviso in due sezioni separate: la prima sezione implica l'inserimento all'interno del database di dati, la seconda prevede l'ottenimento di dati dal database stesso.

4.1 Inserimento

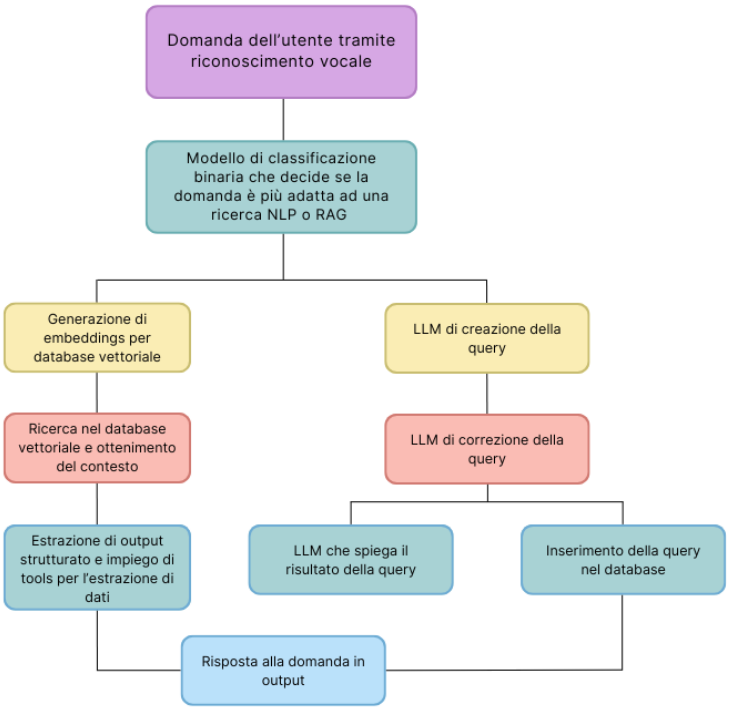
Per quanto concerne la prima fase, essa è così riassumibile:



Gli input che ci si aspetta dall'utente sono del tipo "I spent 300€ on a pair of shoes". L'applicazione provvederà ad inserire altri dati importanti quali, ad esempio, la data e la descrizione.

4.2 Ricerca di dati all'interno del database

La seconda parte dell'applicazione prevede l'ottenimento di dati dal database. L'utente deve essere in grado di formulare domande del tipo "How much did I spend in the last 7 days?" ed ottenerne una risposta strutturata. Lo schema è il seguente:

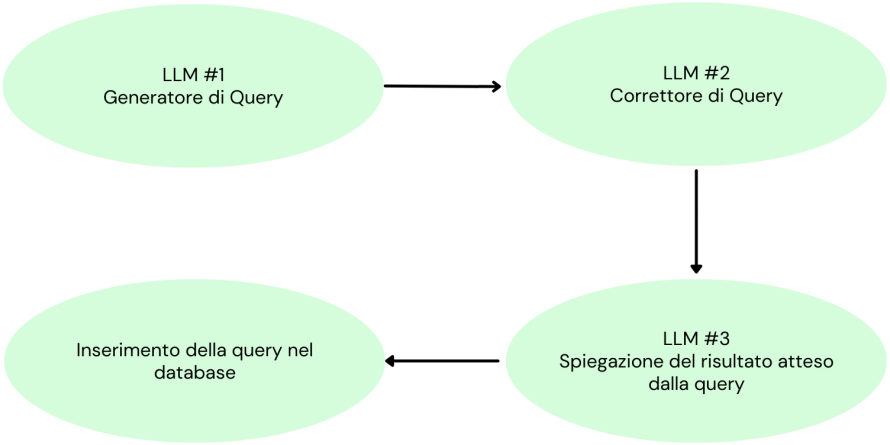


Si denota quindi una struttura ad albero, che si suddivide sul terzo livello. In fase di scrittura del codice ci si è resi conto della necessità di avere due modelli di ricerca diversi: uno di RAG e uno basato sul NLP. Domande basate su "ambiguità lessicali", magari dovute all'utilizzo di sinonimi, saranno rilegate alla ricerca all'interno di un database vettoriale, quindi mediante RAG. Al contrario, le domande più strutturate, che richiedono soprattutto una ricerca basata su informazioni di tipo temporale, saranno appannaggio dei modelli basati sul NLP. Per maggiori approfondimenti in merito si rimanda ai paragrafi in cui si spiega il codice prodotto.

5 Modello di NLP

5.1 Introduzione

Come già rappresentato all'interno del diagramma introduttivo, la parte di ricerca legata al NLP si articola in tre chain separate, di seguito definite.
Innanzitutto, è bene specificare che 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. In questo caso, ogni chain sarà composta da un prompt, un LLM e un Output Parser, utile a rendere il risultato leggibile.
Il primo modello del programma genera una query SQLite3 a partire dalla domanda in linguaggio naturale dell'utente, pervenuta all'applicazione mediante riconoscimento vocale. Il secondo modello ottiene in input la query generata dal primo e la corregge sintatticamente. Il terzo modello spiega il risultato atteso della query (ora corretta), in modo tale che l'utente sappia decidere se l'output ottenuto sia soddisfacente o meno.
Lo schema riassuntivo è il seguente:



5.2 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
load_dotenv()
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.

5.3 Connessione del database e definizione della prima 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 prima chain, contenente il primo modello di LLM.

```
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.

5.4 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
            - For questions like "How much do I spend in the evenings?" you
            should output the total spending after 18:00 from the first day.
            - If the question is asked in present tense, start from the first
            day to today.
            - Use current date only if other time is not given.
            - For questions like "How much did I spend on yoga?" you the
            description should contain 'yoga', not be entirely it.
            - The present year is {cur_year}
            - "What is the least expensive item bought" and similiar requests
            want the description (yes), not the price (no)
            - 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
```

5.5 Definizione della seconda chain

La seconda chain è definita in modo molto simile alla prima, ma con un prompt diverso. Si noti la ripetizione di alcune regole: in fase di testing si è verificato che la reiterazione di alcuni criteri comportasse un aumento dell'affidabilità del modello.

```
    llm2 = ChatGroq(
    model="llama3-groq-70b-8192-tool-use-preview",
    temperature=0,
    max_retries=2,
)
parser = StrOutputParser()
template = ChatPromptTemplate.from_messages([("system", """You are a SQLite3 query
            checker. You will receive an SQLite3 query here {query} and correct
            it syntattically.
            The query should respond to this question: {question} Respond only
            with the corrected query.
            Always remove INTERVAL keyword: adjust the date calculations using
            strftime or date functions supported by SQLite.
            Never use INTERVAL keyword: adjust the date calculations using
            strftime or date functions supported by SQLite.
            Never use INTERVAL keyword: adjust the date calculations using
            strftime or date functions supported by SQLite.
            Never use INTERVAL keyword: adjust the date calculations using
            strftime or date functions supported by SQLite.
            Correct syntax: date('2024-09-12', '-30 days')
            Correct syntax: date('2024-09-12', '+1 day')
            Correct syntax: date('2024-09-12', '-30 days')
            Correct syntax: date('2024-09-12', '+1 day')
            Correct syntax: date('2024-09-12', '-30 days')
            Correct syntax: date('2024-09-12', '+1 day')
            Correct syntax: date('2024-09-12', '-30 days')
            Correct syntax: date('2024-09-12', '+1 day')
            The table name is expensesok.
            How much: SUM()
            How much: SUM()
            How much: SUM()
            In which: ORDER BY
```

```
            present tense: NO date()
            present tense: NO date()
            You only have columns: price, description, category, timestamp
            (which include date and time)
            If the date of today is needed, remember that today is {today}.
            NEVER subract numbers to dates. """)])
correction_chain = template | llm2 | parser
```

5.6 Definizione della terza chain

Anche la terza chain ha una definizione simile alle altre, con un prompt diverso.

```
llm3 = ChatGroq(
    model="llama3-groq-70b-8192-tool-use-preview",
    temperature=0,
    max_retries=2,
)
parser2 = StrOutputParser()
template2 = ChatPromptTemplate.from_messages([("system", """You will receive
    an SQL3 query and a result. You will describe what the query gets to me,
    as if the database and the query did not exist. I only see the result.
    The query is {query}. Give a one line result. Don't talk about the result
    and the query. Template: The search found: (short description of what that
    query should find).""")])
description_chain = template2 | llm3 | parser2
```

5.7 Funzione "get_from_database"

All'interno della funzione get_from_database() si effettuano tutte le chiamate alle chain. In particolare, si aggiungono funzioni di estrazione dei risultati e di arrotondamento dei risultati, in caso di numeri a virgola mobile.

```
def get_from_database(printQuestion, printQuery, printDescription,
     printCorrectedQuery, question, db, chain, correction_chain, today,
    description_chain):
    start = timer()
    if (printQuestion):
        print(f"Question: {question}")

    response = chain.invoke({"question": question})
    response = response.replace("SQLQuery:", "").replace("```sql", "")
                .replace("```", "").replace("\n",";").strip()

    if (printQuery):
        print(f"Original Query: {response}")

    response = correction_chain.invoke({"query":response, "question":question,
                "today":today}).strip()

    if (printCorrectedQuery):
        print(f"Corrected query: {response}")

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

    if (printDescription):
        queryDescritption = description_chain.invoke({"query":response})
        print(queryDescritption)

    if not result or result=="None":
```

```
        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("---")

    end = timer()

    return (end-start)
```

Si noti la presenza delle variabili "start" ed "end": le stesse sono utilizzate in fase di testing per avere una stima del tempo richiesto per l'esecuzione delle query.

6 Testing

6.1 Metodi e Risultati

La funzione get_from_database() è chiamata in modo iterativo su una lista di 40 domande. Su 30 di queste 40 domande è stato formulato il system prompt. Su di esse, la precisione sia sintattica che contenutistica raggiunge il 100%, come prevedibile. Sulle 10 domande rimanenti, la precisione sintattica rimane del 100%, grazie soprattutto al secondo modello che pone rimedio a tutte le imprecisioni del primo. Tuttavia, la precisione contenutistica tende a diminuire, con l'80% delle query che soddisfano appieno le richieste iniziali. La descrizione dell'output previsto, che avviene a carico della terza chain, risulta quindi particolarmente importante. Senza la stessa, l'utente potrebbe considerare corretto un risultato errato, che ad egli giungerebbe senza un sufficiente contesto. In fasi successive di sviluppo dell'applicazione si potrebbe aggiungere un suggerimento da far pervenire all'utente, tale da aiutarlo a riformulare con maggiore precisione le sue richieste. Il codice di test è il seguente:

```
max_time = 0
total_time = 0
for question in TestQuestions:
    x = get_from_database(True, True, True, True, question, db, full_chain,
    correction_chain, today, description_chain)
    total_time += x
    if x > max_time:
        max_time = x
print(f"The maximum time for a result was: {x:.2f} seconds.")
print(f"The average waiting time was: {(total_time/len(questions1ok+newQuestions))
      :.2f} seconds")
```

I parametri booleani (printQuestion, printQuery, printDescription, printCorrectedQuery) sono utilizzati puramente in fase di test. Impostando ogni booleano a "True" l'output sarà il seguente:

```
Question: In which category I spend the biggest amount of money?
Original Query: SELECT category, MIN(price) FROM expensesok GROUP BY category LIMIT 1;
Corrected query: SELECT category, SUM(price) FROM expensesok GROUP BY category ORDER BY SUM(price) DESC LIMIT 1;
The search found: The highest spending category in the expenses database.
electronics, 3425.00
---
Question: How much do I spend in the mornings?
Original Query: SELECT "price" FROM expensesok WHERE "timestamp" <= date('2024-09-16') AND "timestamp" >= date('2024-09-15') AND strftime('%H:%M:%S', "timestamp") < '12:00:00';
Corrected query: SELECT SUM("price") FROM expensesok WHERE "timestamp" <= date('2024-09-16') AND "timestamp" >= date('2024-09-15') AND strftime('%H:%M:%S', "timestamp") < '12:00:00';
The search found: The total cost of expenses recorded before noon on September 15th and 16th, 2024.
No match found.
---
Question: How much do I spend in the evenings?
Original Query: SELECT "price" FROM expensesok WHERE "timestamp" >= date('2024-09-15') AND "timestamp" <= date('2024-09-16') AND "timestamp" >= '18:00:00' AND "timestamp" <= '23:59:59';
Corrected query: SELECT SUM("price") FROM expensesok WHERE "timestamp" >= date('2024-09-15') AND "timestamp" <= date('2024-09-16') AND strftime('%H:%M:%S', "timestamp") >= '18:00:00' AND
 strftime('%H:%M:%S', "timestamp") <= '23:59:59';
The search found: the total sum of expenses made between 6 PM and 11:59 PM on September 15th and 16th, 2024.
No match found.
---
Question: How much did I spend on yoga?
Original Query: SELECT "price" FROM expensesok WHERE "description" LIKE '%yoga%';
Corrected query: SELECT SUM("price") FROM expensesok WHERE "description" LIKE '%yoga%';
The search found: the total cost of all expenses related to yoga activities.
40.00
---
The maximum time for a result was: 10.46 seconds.
The average waiting time was: 9.20 seconds.
```

Il massimo tempo di attesa registrato è di 10.46 secondi, mentre il tempo di attesa medio è di 9.20 secondi. Si tratta di tempi accettabili in funzione della complessità del modello, che potrebbero anche essere resi più sopportabili mediante la tecnica dello streaming. Essendoci tuttavia la necessità di avere a che fare con dati di tipo atomico, ci si vede costretti a scartare questa ipotesi.

7 Modello di RAG

7.1 Creazione del database vettoriale

Il primo passo per lo sviluppo di un modello di RAG è stata la creazione di un database vettoriale.

```
from langchain_chroma import Chroma
from langchain_community.embeddings import OllamaEmbeddings
from langchain_core.prompts import PromptTemplate
from langchain_experimental.llms.ollama_functions import OllamaFunctions
from langchain_text_splitters import CharacterTextSplitter
from pydantic.v1 import BaseModel, Field

persist_directory = "./chroma/expenses"

embeddings = OllamaEmbeddings(model="mxbai-embed-large")

db = Chroma(persist_directory=persist_directory,
            embedding_function=embeddings)
def db_to_text():
    import sqlite3
    con = sqlite3.connect("googleDb.sqlite3")
    cur = con.cursor()

    cur.execute("SELECT * FROM expensesok;")
    rows = cur.fetchall()
    for row in rows:
        text = f"{row[0]} {row[1]} {row[2]}"
        embed(row[1], text)

def embed(description, text):
    texts = text_splitter.split_text(text)
    if texts:
        Chroma.from_texts([t for t in texts], embeddings,
                          persist_directory=persist_directory,
                          metadatas=[{"row": description}])

db_to_text()
```

Il codice appena presentato risulta di semplice comprensione: di fatto, si seleziona ogni entry dal database, e lo si inserisce all'interno del database vettoriale Chroma, scelto in funzione della sua rapidità e semplicità di utilizzo. Il modello scelto per gli embeddings è il MixedBread "mxbai-embed-large", presente all'interno della libreria di Ollama. Si tratta di un modello efficace ma

soprattutto estremamente leggero in termini di memoria occupata. E' interessante notare quanto sia stato necessario rimuovere le date dal database vettoriale: a causa delle stesse tutte le entry all'interno del database venivano salvate come "molto vicine" tra loro. Si tratta di qualcosa di non auspicabile, in quanto rende difficile la ricerca. Per le domande che riguardano informazioni di natura cronologica rimane la possibilità di adempiere alla richiesta mediante NLP.

7.2 Ricerca all'interno del database

Al fine di rendere la fase di testing più agevole, si è creata una funzione da richiamare in modo iterativo su tutte le domande appartenenti alla lista. Sussistono inoltre dei booleani, passati come argomenti, utili in fase di debug.

```
def RAG(question, db, rendered_tools, printScores, printChunks, printSmallestChunk,
      printQuestion, printContext=True, eps=35, min_samples=1, k_size=sys.maxsize,
      threshold=0.40) -> None:
    start = timer()

    context = db.similarity_search_with_score(question, k=k_size)
    scores = [score for _, score in context]
```

7.3 Suddivisione in chunks

Come è possibile notare già da subito, tra i parametri si notano eps, min_samples, k_size, threshold. Di seguito si determina una prima spiegazione:

- eps: utilizzato all'interno dell'algoritmo di clustering DBSCAN. Rappresenta il raggio massimo entro cui cercare altri punti vicini per formare un cluster. Se due punti hanno una distanza inferiore o uguale a eps, vengono considerati parte dello stesso cluster.
- min_samples: indica il numero minimo di punti (campioni) richiesti per formare un cluster. Se un punto ha meno di min_samples vicini, viene considerato rumore.
- k_size: numero massimo di documenti da estrarre dal database numerico.
- threshold: impiegato per calcolare una soglia di punteggio (score) quando si filtra il contesto basato sui risultati di ricerca.

Come anticipato, si sfrutterà l'algoritmo DBSCAN (Density-Based Spatial Clustering of Applications with Noise) per la creazione di chunks di risposte molto vicine tra loro: è molto più conveniente questo approccio che uno basato su una ulteriore rielaborazione dei documenti estratti, in quanto offre risultati "più deterministici". Il codice, quindi, rimane il seguente:

```
    chunks = divide_into_chunks(scores, eps, min_samples)
```

Con la funzione divide_into_chunks così definita:

```
    def divide_into_chunks(numbers, eps=1.0, min_samples=2):
        numbers_array = np.array(numbers).reshape(-1, 1)

        dbscan = DBSCAN(eps=eps, min_samples=min_samples).fit(numbers_array)
        labels = dbscan.labels_

        chunks = {}
        for num, label in zip(numbers, labels):
            if label not in chunks:
                chunks[label] = []
            chunks[label].append(num)

        chunks = {label: chunk for label, chunk in chunks.items() if label != -1}

        return chunks
```

7.4 Mantenimento del chunk a distanza minore

Si procede poi con la selezione del chunk che si caratterizza per la distanza minore, ovvero quello più rilevante con la domanda effettuata.

```
def keep_smallest_chunk(chunks):
    min_chunk = min(chunks.values(), key=lambda chunk: np.mean(chunk))
    return min_chunk

smallest_chunk = keep_smallest_chunk(chunks)
```

7.5 Filtro delle risposte che superano la threshold

Ulteriore scrematura avviene nel momento in cui si scartano tutte le risposte che non superano una determinata threshold.

```
max_score = max(smallest_chunk)
threshold_score = max_score * threshold - 100

context = [[(doc, score) for doc, score in context if score in smallest_chunk
         and score <= min(smallest_chunk) + 75 and score >= threshold_score]]
context_new = []
cnt = 0
for i in context:
    for j in i:
        for k in j:
            if cnt % 2 == 0:
                context_new.append(f"{k.page_content}\n")
            cnt += 1
```

7.6 Chain 1

La prima chain è utile ad estrarre le informazioni dall'output della ricerca vettoriale così da inserirle all'interno di liste. Questo tipo di parsing risulta necessario affinchè la seconda chain possa procedere in modo più sicuro ed efficace.

```
system_prompt = f"""
                    You are an expert extraction algorithm. Your goal is to
                    extract the relevant information from the context to
                    answer the user's question.
                    Only extract relevant information from the text. Do not
                    add any new information.
                    This is the context:
                    {context_new}.
                    Give back the information I should use from {context_new}
                    to answer.
                    Select the information from {context_new}. Not necessarely
                    all of it: if you think a row is not relevant, ignore it.
                    Put all of these information into a Python list, and give
                    me just that.
                    PUT INTO THE LIST ALL RELEVANT INFORMATION FROM THE CONTEXT.
                    Do a stricter selection.
                    Don't print anything that is not in the context.
                    Possible outputs: [list of row and purchases] or [empty list]
                    Return the empty list if you think there is no relevant
                    information.

                    IF YOU THINK YOU CAN'T PERFORM THE TASK, RETURN AN EMPTY LIST.
                    Do not return any information that is not in the context.
                    Just select the row. Don't select only numbers.
                    - Return list of strings
                    - Return list of strings
                    - Return list of strings
```

```
                    - Return list of strings
                    """

prompt_1 = ChatPromptTemplate.from_messages(
    [("system", system_prompt), ("user", "{input}")]
)

chain = prompt_1 | model_1 | StrOutputParser() | RunnableLambda(stripOutput)

res = chain.invoke({"input": question})
```

7.7 Chain 2

La seconda chain, sicuramente più complessa, ha la capacità di prelevare l'output di quella precedente (che, di fatto, è una lista) e di trasformarla in formato JSON. E' inoltre in grado di capire quale tool utilizzare in base alla domanda. Il tool prescelto preleva il JSON e lo rielabora, fornendo una risposta definitiva.

```
system_prompt_tools = f"""
                    You are responsible for selecting the correct tool from
                    the following list and specifying the function name and
                    arguments based on the user's question.
                    Choose the correct tool from the list below, finding the
                    one relevant to the question, and provide the function name
                    and arguments to extract the relevant information from the
                    context.
                    Here are the tools available and their descriptions:
                    {rendered_tools}
                    The input you have access to includes:
                    {res}
                    Your goal:
                    - Select the correct tool for the task.
                    - Ensure **every** part of the context is passed to the
                    tools.
                    - Provide a response in JSON format with 'name' and '
                    arguments' keys.
                    - If you believe you don't have enough context, use the
                    no_result
                    tool.
                    - If a day, month, year or time is mentioned in the input,
                    use the no_result tool.
                    - If a day, month, year or time is mentioned in the input,
                    use the no_result tool.
                    - If a day, month, year or time is mentioned in the input,
                    use the no_result tool.
                    - If the {res} is empty, use the no_result tool.
                    - If the {res} is empty, use the no_result tool.
                    - If the {res} is empty, use the no_result tool.
                    - If the {res} is empty, use the no_result tool.
                    - If "average" or "mean" in {res}, use the average tool.
                    """

model_with_tools = ChatGroq(
    model="llama3-groq-70b-8192-tool-use-preview",
    temperature=0
)
prompt_2 = ChatPromptTemplate.from_messages(
[("system", system_prompt_tools), ("user", "{input}")]
)
final_chain = prompt_2 | model_with_tools |  StrOutputParser()
| RunnableLambda(stripOutput) | JsonOutputParser() | tool_chain
```

```
    final_response = final_chain.invoke({"input": question})
    print(f"{final_response}\n\n")

    end = timer()
    sleep(1)
    return (end-start)
```

7.8 Tools

I tool impiegati sono i seguenti:

```
@tool
def total(context: list) -> int:
    """Extracts the prices and returns the total expense value.
    'context' is the list of all expenses."""
    prices = []

    for result in context:
        # Correct regex pattern to match floating-point numbers
        x = re.findall(r"\d+\.\d+", str(result))
        if x:
            for i in x:
                prices.append(float(i))

    return sum(prices)

@tool
def average(context: list) -> int:
    """Returns the average expense value. 'context' is the list of all results.
    Instead of dividing by zero, returns a test sentence."""
    prices = []

    if (len(context) != 0):
        for result in context:
            x = re.findall(r"\d+\.\d", result)
            if x:
                for i in x:
                    prices.append(float(i))
    else:
        return "The RAG was not sufficient. Try another approach."

    avg = round(float(sum(prices) / len(prices)), 2) if len(prices) != 0 else 0

    return avg


@tool
def count(context: list) -> int:
    """Returns the number of rows that are relevant to the question."""
    return len(context)

@tool
def print_all(context: list) -> str:
    """Returns all the rows that are relevant to the question."""
    return '\n'.join(map(str, context))

@tool
def no_result(context: list) -> str:
    """Returns a message if there are no results."""
    return "RAG was not sufficient. Try with NLP."
```

```python
@tool
def select_cheapest(context: list) -> str:
    """Returns the cheapest item from the context."""
    prices = []
    items = []

    for result in context:
        x = re.findall(r"\d+\.\d+", str(result))
        if x:
            for i in x:
                prices.append(float(i))
                items.append(result)

    min_price = min(prices)
    index = prices.index(min_price)
    return items[index]

@tool
def select_most_expensive(context: list) -> str:
    """Returns the most expensive item from the context."""
    prices = []
    items = []

    for result in context:
        x = re.findall(r"\d+\.\d+", str(result))
        if x:
            for i in x:
                prices.append(float(i))
                items.append(result)

    max_price = max(prices)
    index = prices.index(max_price)
    return items[index]
```

In fase di testing successivi la lista dei tool potrebbe allungarsi.

7.9 Testing e risultati

In fase di testing si è notato un ottimo comportamento del modello di RAG: il modello ha fornito risposte in modo corretto per ognuna delle domande di test. Inoltre, i tempi di elaborazione delle risposte sono estremamente rapidi, specie se comparati alla NLP. La capacità del modello di ammettere i propri limiti quando sottoposto a test i quali richiedevano informazioni non presenti nel database vettoriale è stata inoltre ottima. L'output fornito è stato il seguente:

```
Number of requested results 225 is greater than number of elements in index 45, updating n_results = 45
Question: What is the price of the 'pair of wireless earbuds'?
["65.0 pair of wireless earbuds electronics"]
{"id": 0, "name": "total", "arguments": {"context": ["65.0 pair of wireless earbuds electronics"]}}
65.0


Number of requested results 225 is greater than number of elements in index 45, updating n_results = 45
Question: What was the price of the 'book' bought in October 2023?
["15.0 book entertainment"]
RAG was not sufficient. Try with NLP.


The maximum time for a result was: 4.45 seconds.
The average waiting time was: 3.83 seconds.
```

Essendo la RAG il triplo più veloce dell'NLP, in futuro si cercherà di sfruttarla il più possibile: in caso di ambiguità, sarà comunque preferito l'utilizzo della RAG.

7.10 Refactoring

Al fine di testare il programma nella sua totalità, si è reso necessaria una fase di refactoring [6]. Sono stati creati 3 moduli, i cui file prendono il nome di module_NLP.py, module_RAG.py e module_choose_NLP_RAG.py, contenenti classi e funzioni che richiamano quanto già scritto. Si è colta l'occasione per aggiungere commenti e modificare il nome delle variabili al fine di renderle più leggibili.
In seconda sede, si è poi scritto un file di nome "main_question.py". All'interno dello stesso si è proceduto con l'importazione dei moduli sopra citati. Vengono poi definite delle costanti e aggiunta della logica tale per la quale si riceva, in un loop infinito, una domanda da terminale finchè l'utente non ha determinato la volontà di uscire. Questa domanda segue poi lo schema citato in chiusura del capitolo 4: si determina quale metodo sfruttare per l'ottenimento della risposta dal database, e lo si applica di conseguenza. Il codice è di seguito riportato:

```
import os
from dotenv import load_dotenv
from module_NLP import get_database, get_NLP_chains, NLP
from module_RAG import get_embedded_database, RAG, stripOutput
from module_choose_NLP_RAG import get_tagging_chain, get_classification

# Constants for configuration
URI_DB = "sqlite:///googleDb.sqlite3"
PERSIST_DIRECTORY = "./chroma/expenses"
PRINT_SETTINGS = {
    "print_question": False,
    "print_query": False,
    "print_description": False,
    "print_corrected_query": True,
    "print_time": False,
    "print_scores": False,
    "print_chunks": False,
    "print_smallest_chunk": False,
    "print_context": True,
    "print_method": True,
    "print_characteristics_of_the_question": False
}

# definition of databases and important variables
def load_environment_variables():
    """Load environment variables from .env file."""
    load_dotenv()
    os.environ["GROQ_API_KEY"] = os.getenv("GROQ_API_KEY")

def initialize_nlp(uri_db):
    """Initialize NLP chains and database."""
    nlp_db = get_database(uri_db)
    full_chain, correction_chain, description_chain = get_NLP_chains(nlp_db)
    k_size = int(nlp_db.run("SELECT COUNT(*) from expensesok;")
             .replace("[(","").replace(",)]",""))
    return nlp_db, full_chain, correction_chain, description_chain, k_size

def initialize_rag(persist_directory):
    """Initialize RAG database."""
    return get_embedded_database(persist_directory)

# main function: it will loop indefinitely on the questions of the user.
def main():
    """Main function to run the chatbot."""
    load_environment_variables()

    # Get the classification chain
    tagging_chain = get_tagging_chain()
```

```
    # Initialize NLP and RAG
    (nlp_db, full_chain, correction_chain,
     description_chain, k_size) = initialize_nlp(URI_DB)
    rag_db = initialize_rag(PERSIST_DIRECTORY)

    while True:
        # Get the question
        question = input("Enter your question ('X' or 'x' to exit): ")

        # Check if the user wants to exit
        if question == 'X' or question == 'x':
            exit()

        # Choose between RAG, NLP or reject the question
        method = get_classification(question, tagging_chain,
                 PRINT_SETTINGS)

        # Debug: print the chosen method
        if PRINT_SETTINGS["print_method"]:
            print(f"Chosen method: {method}.")

        # Run the chosen method
        if method == "rejected":
            response = "The question is not related to personal finance"
        elif method == "rejected (exception)":
            response = "An error occurred. Try again"
        elif method == "NLP":
            response = NLP(question, nlp_db, full_chain, correction_chain,
                           description_chain, PRINT_SETTINGS)
        elif method == "RAG":
            response = RAG(question, rag_db, stripOutput, PRINT_SETTINGS,
                           k_size)

        # Print the response
        print(f"Response: {response}.")

if __name__ == "__main__":
    main()
```

La condizione if __name__ == "__main__": viene usata per verificare se il file Python è eseguito come script principale e non importato come modulo in un altro file. Se il file è eseguito direttamente, il blocco di codice all'interno di questa condizione viene eseguito, in questo caso chiamando la funzione main().

Riferimenti

[1] *Chart.js*. URL: https://www.chartjs.org/docs/latest/samples/information.html.

[2] *Chroma*. URL: https://www.trychroma.com//.

[3] *CSS (linguaggio di programmazione)*. URL: https://it.wikipedia.org/wiki/CSS.

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

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

[6] *Definizione di refactoring*. URL: https://it.wikipedia.org/wiki/Refactoring.

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

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

[9] *ggplot*. URL: https://ggplot2.tidyverse.org/.

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

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

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

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

[14] *HTML (linguaggio di programmazione)*. URL: https://it.wikipedia.org/wiki/HTML.

[15] *JavaScript (linguaggio di programmazione)*. URL: https://it.wikipedia.org/wiki/JavaScript.

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

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

[18] *PandasAI*. URL: https://pandas-ai.com/.

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

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

[21] 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.

[22] 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.

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