Documentazione Chatbot

Romano Mancini

10 ottobre 2024

Indice

1 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;

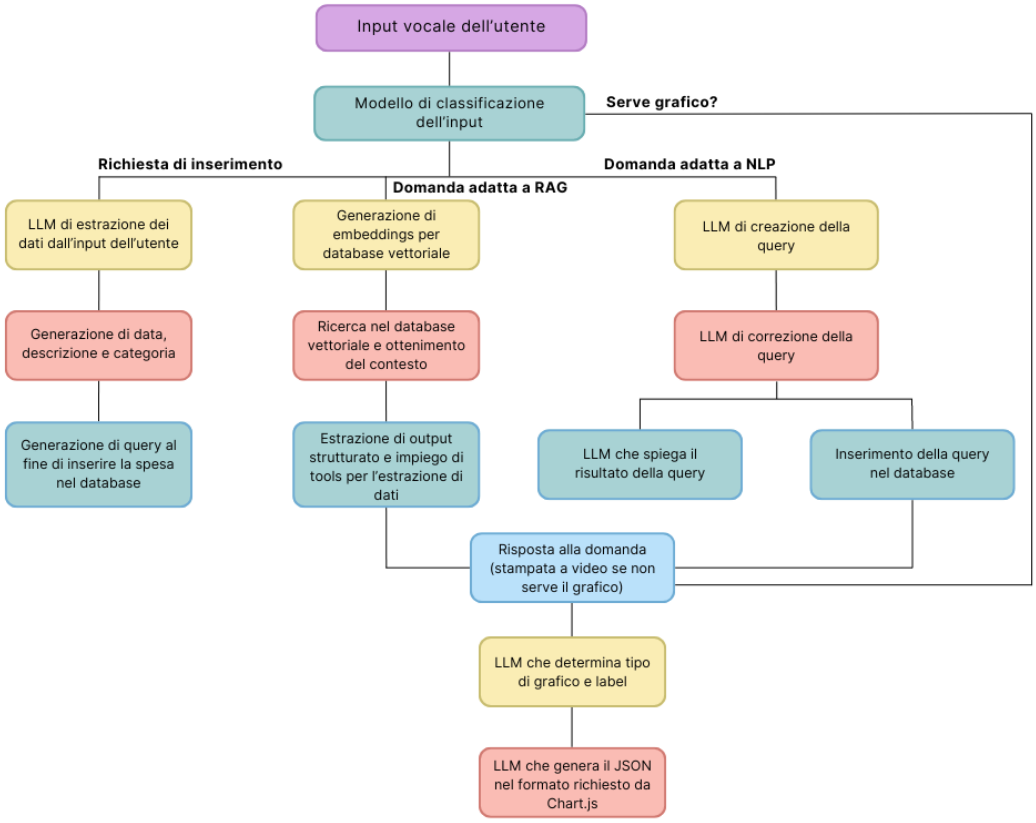
4 Descrizione del codice

4.1 Schema riassuntivo di alto livello

Il programma, come già specificato, adempie a tre scopi separati:

- Inserimento di dati all'interno del database;
- Ottenimento di dati dal database;
- Creazione di grafici interattivi a partire dalle domande dell'utente.

Per facilitare la comprensione di alto livello, si riporta di seguito uno schema riassuntivo, che sarà meglio spiegato nei paragrafi successivi.



4.2 Classificazione dell'input

Data la volontà di dare all'utente la massima flessibilità di utilizzo possibile, si è reso necessaria la creazione di un modello di classificazione tramite il quale discernere il tipo di input fornito all'applicazione. Difatti, l'input dato dall'utente potrà essere delle seguenti tipologie:

- Richiesta di inserimento di una nuova spesa nel database;
- Estrazione di dati (più adatta ad una ricerca basata su RAG);
- Estrazione di dati (più adatta ad una ricerca basata su NLP);
- Creazione di grafici basati su dati ottenuti mediante RAG;
- Creazione di grafici basati su dati ottenuti mediante NLP;
- Domande non pertinenti o non chiare, da rifiutare.

La logica di classificazione è piuttosto semplice: si fanno estrarre dei dati numerici relativi alle caratteristiche dell'input, e mediante una logica if-else si ritorna il metodo da utilizzare per adempiere alla volontà dell'utente. I dati numerici che si fanno estrarre sono salvati nei seguenti campi:

- **is_there_time**: 0 se non ci sono riferimenti al tempo nella domanda, 1 se ci sono.
- **is_specific**: Da 0 a 10, misura quanto è specifica la domanda.
- **is_broad**: Da 0 a 10, misura quanto è generica la domanda.
- **has_keyword**: Da 0 a 10, misura la presenza di parole chiave nella domanda. Se non ci sono, il valore è 0.
- **is_related_to_personal_finance**: Da 0 a 10, misura quanto la domanda è correlata alle spese o acquisti dell'utente. Se non riguarda denaro, spese o acquisti, il valore è 0.

- **NLP_or_RAG**: 0 se la domanda è più adatta per tecniche RAG, 1 se è più adatta per NLP (con valori intermedi da 0 a 10).
- **is_there_number**: 0 se non c'è un numero esplicito nella domanda, 1 se c'è.
- **is_there_category**: 0 se la parola "categoria" non è presente nella domanda, 1 se è presente.
- **is_it_question**: 1 se la frase è una domanda, 0 se è solo una frase affermativa.
- **is_asking_for_plot**: 1 se la domanda richiede un grafico, 0 se non lo richiede.

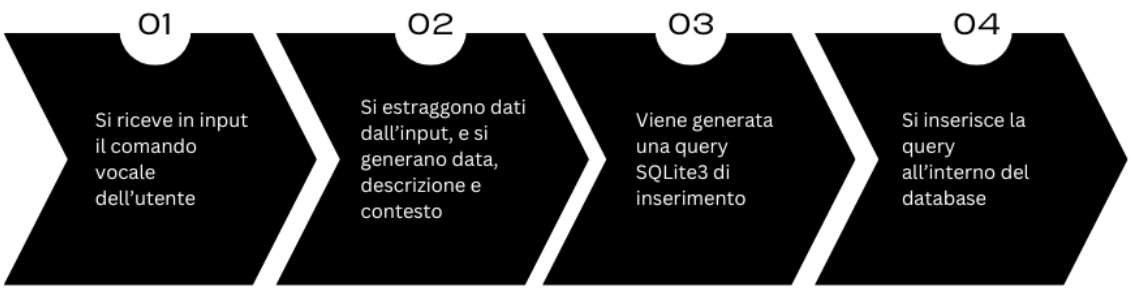
Si riporta di seguito il codice scritto.

```
from langchain_core.prompts import ChatPromptTemplate
from pydantic import BaseModel, Field
from langchain_groq import ChatGroq

class Classification(BaseModel):
    is_there_time: int = Field(description="""0 if there is nothing related to time
    in the question, 1 if there is.""")
    is_specific: int = Field(description="""From 0 to 10, how specific is the
    question?""")
    is_broad: int = Field(description="""From 0 to 10, how broad is the question?""")
    has_keyword: int = Field(description="""From 0 to 10, are there keywords in the
    question? If there are not, put 0.""")
    is_related_to_personal_finance: int = Field(description="""From 0 to 10, how is
    the question related to the expenses of the user? It is to be considered related
    if it asks anything about prices or expenses or items purchased. Put 0 if the
    question is not related to prices, expenses, money or items.""")
    NLP_or_RAG: int = Field(description="""0 if the question is better suited for RAG
    queries, 1 if the question is better suited for NLP queries. You can have a value
    between 0 and 10.""")
    is_there_number: int = Field(description="""0 if there is no explicit number in the
    question, 1 if there is and explicit number in the question.""")
    is_there_category: int = Field(description="""0 if the word 'category' is not in the
    question, 1 if there is the word 'category' in the question.""")
    is_it_question: int = Field(description="""1 if the question asks for something in
    return, 0 if it is just a sentence.""")
    is_asking_for_plot: int = Field(description="""1 if the question is asking in any way
    for a plot or a graph, 0 if it is not.""")

def get_tagging_chain():
    tagging_prompt = ChatPromptTemplate.from_template("""
            Extract the desired information from the following passage.
            You will classify the passage based on the following criteria:
            You must put a value in each field.

            1. **is_there_time**: Set to 1 if the passage mentions any time-related
            information (e.g., days, weeks, dates, months, time ranges), otherwise
            set to 0.
            2. **is_specific**: Rate the question's specificity on a scale from 0
            to 10, where 0 means the question is very vague, and 10 means it is highly
            specific.
            3. **is_broad**: Rate the question's breadth on a scale from 0 to 10, where
            0 means it is very narrow in scope, and 10 means it is very broad.
            4. **NLP_or_RAG**: Rate from 0 to 10, where 0 means the question is better
            suited for a RAG query (unstructured text), and 10 means it is better suited
            for NLP queries (structured data).
            5. **has_keyword**: Rate from 0 to 10, where 0 means there are no keywords
            in the question, and 10 means there are many keywords.
            6. **is_related_to_personal_finance**: Rate from 0 to 10, where 0 means the
            question is not related to personal finance, and 10 means it is highly related
            to personal finance.
```

```
            It has to be considered related if it asks anything about prices or expenses
            or items purchased.Put 0 if the question is not related to prices, expenses,
            money or items.
            7. **is_there_number**: Set to 1 only if there an explicit number in the
            question, otherwise set to 0.
            8. **is_there_category**: Set to 1 only if there is the word 'category'
            in the question, otherwise set to 0.
            9. **is_it_question**: Set to 1 if the question asks for something in
            return, otherwise set to 0.
            10. **is_asking_for_plot**: Set to 1 if the question is asking in any way
            for a plot or a graph, otherwise set to 0.
            Passage:
            {input}
            """)

    llm = ChatGroq(temperature=0, model="llama3-groq-70b-8192-tool-use-preview")
    .with_structured_output(Classification)

    tagging_chain = tagging_prompt | llm

    return tagging_chain

def get_classification(question, tagging_chain, PRINT_SETTINGS):
    try:
        class_calculed = tagging_chain.invoke({"input": question})
    except Exception:
        return "rejected (exception)"

    res_dict = class_calculed.dict()

    if PRINT_SETTINGS["print_characteristics_of_the_question"]:
        print(res_dict)

    if res_dict["is_related_to_personal_finance"] <= 2:
        return "REJECTED"
    elif res_dict["is_it_question"] == 0:
        return "INPUT"
    elif res_dict["is_there_time"] == 1 or res_dict["is_there_number"] == 1
    or res_dict["is_there_category"] == 1:
        if res_dict["is_asking_for_plot"] == 1:
            return "NLP|PLT"
        else:
            return "NLP"
    elif (res_dict["is_specific"] == res_dict["is_broad"])
    or res_dict["is_broad"] > 5 or res_dict["NLP_or_RAG"] > 5:
        if res_dict["is_asking_for_plot"] == 1:
            return "RAG|PLT"
        return "RAG"
    else:
        return "NLP"
```

4.3 Inserimento

Per quanto concerne il codice scritto al fine di garantire all'utente la possibilità di inserire le sue spese all'interno del database, è possibile determinarne un riassunto nello schema seguente:



Gli input che l'utente fornirà saranno del tipo: "I spent 300€ on a pair of shoes". L'applicazione integrerà automaticamente altri dettagli rilevanti, come la data dell'acquisto e una descrizione più approfondita.
Di seguito si riporta il codice Python:

```python
from pydantic import BaseModel, Field
from langchain_groq import ChatGroq
from langchain_core.prompts import PromptTemplate
from datetime import datetime
from pydantic import ValidationError

class Answer(BaseModel):
    """Outputs the structured version of the user input."""

    price: float = Field(description="The sum of money spent by the user")
    description: str = Field(description="A short description of the user's expense")
    category: str = Field(description="The category of the user's expense")


def get_input_chain():
    llm = ChatGroq(model="llama3-groq-70b-8192-tool-use-preview", temperature=0)
    prompt = PromptTemplate.from_template("""You are a finance manager who will receive,
    from the user, a textual input. You must organize the UserInput into the following
    schema: float price, string description, string category. The price and the
    description must be taken directly from the input: remember to remove the articles.
    The category must be chosen among: Clothing, Food, Health, Personal Care,
    Entertainment, Transport, Bills, Subscription, Other. UserInput: {question}.""")
    structured_llm = llm.with_structured_output(Answer)
    chain = prompt | structured_llm

    return chain


def input_into_database(question, chain, cur, MAX_DESCRIPTION_LENGTH):
    try:
        date = datetime.now().strftime("%Y-%m-%dT%H:%M:%S")

        # Generate structured response
        resp = chain.invoke({"question": question})

        # Input validation
        if resp.price <= 0:
            raise ValueError("Price must be provided as a positive value.")
        if not resp.description.strip():
            raise ValueError("A description of the expense must be provided.")
        valid_categories = {"Clothing", "Food", "Health", "Personal Care",
        "Entertainment", "Transport", "Bills", "Subscription", "Other"}
        if resp.category not in valid_categories:
            raise ValueError("Invalid category.")
```

```
        # Insert the new record into the database
        cur.execute("""INSERT INTO expensesok (price, description, category,
        timestamp) VALUES (?, ?, ?, ?);""",
                    (resp.price, resp.description, resp.category, date))

        # Commit the transaction to save the changes
        cur.connection.commit()

        # Fetch the newly inserted record to confirm
        cur.execute("""SELECT * FROM expensesok WHERE timestamp = ?;""", (date,))
        result = cur.fetchone()

        return f"""The following input has been added to the database:\nPrice:
        {result[0]}\nDescription: {result[1]}\nCategory:
        {result[2]}\nTimestamp: {result[3]}"""

    except ValidationError as ve:
        return f"Input validation error: {ve}"
    except ValueError as e:
        return f"Value error: {e}"
    except Exception as e:
        return f"An error occurred: {e}"
```

Come si può notare, si lascia la possibilità al modello di determinare il prezzo, la descrizione e la categoria della spesa inserita. La data è generata in modo deterministico, ed inserita seguendo lo standard ISO 8601.
Le categorie ritenute "valide" sono determinate manualmente, ma nulla impedisce in implementazioni future di generarle in modo graduale.

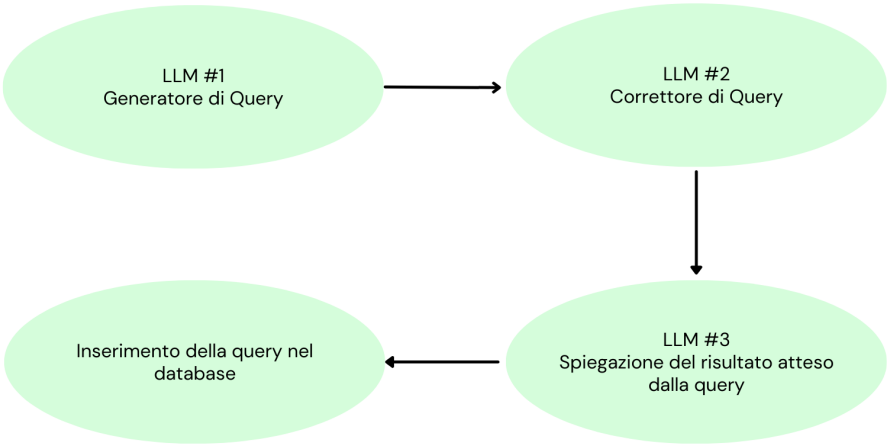
4.4 Ricerca di dati all'interno del database

La seconda parte dell'applicazione riguarda il recupero dei dati dal database partendo da una domanda dell'utente del tipo "How much did I spend in the last 7 days?"
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.

4.4.1 Modello di NLP

4.4.1.1 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:



4.4.1.2 Creazione delle chain

Il primo passo per l'ottenimento dei dati tramite NLP è la creazione delle tre chain introdotte nel precedente paragrafo. Il codice seguente è piuttosto autoesplicativo. Si ritiene tuttavia importante specificare il fatto che le ripetizioni all'interno dei diversi system prompt non sono degli errori di battitura: in fase di testing si è ritenuto utile reiterare determinati concetti al fine di aumentare le probabilità che il modello vi ubbidisse.

```
def get_NLP_chains(db):
    # definizione della chain di generazione delle query
    llm = ChatGroq(model="llama3-groq-70b-8192-tool-use-preview", temperature=0)
    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.
                You only have the columns price, description, category, and
                timestamp (which contains both day and time in isoformat).
                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 subtract from dates.
                If the question regards anything about plots, you must only
                select 2 columns.
                - Use SQLITE3 syntax.
                Follow these rules:
                - TOP: ORDER BY DESC
                - Never use LIMIT 5 if not actually required.
                - Use LIMIT 1 only if you must find the min or max value.
                - 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 no other time is given.
                - For questions like "How much did I spend on yoga?" the description
                should contain 'yoga', not be entirely it.
                - The present year is {cur_year}, today is {today}
                - "What is the least expensive item bought" and similar 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 SIMILAR 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.
            - 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

# Definizione della chain di correzione

llm2 = ChatGroq(model="llama3-groq-70b-8192-tool-use-preview", temperature=0)
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.
If the question regards plots, the query must only select 2 columns.
Correct syntax: date('2024-09-12', '-30 days')
Correct syntax: date('2024-09-12', '+1 day')
Never use LIMIT 5 if not actually required.
TOP: ORDER BY DESC
Lowest: group by ORDER BY ASC
Least: group by ORDER BY ASC
If there are words ending in "-est", LIMIT 1.
The table name is expensesok.
"What's the category I spent the lowest?" SELECT "category", SUM("price")
AS "total_spent" FROM expensesok GROUP BY "category" ORDER BY "total_spent" DESC
LIMIT 1;
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

    # Definizione della chain di descrizione

    llm3 = ChatGroq(model="llama3-groq-70b-8192-tool-use-preview", temperature=0)
    parser2 = StrOutputParser()
    template2 = ChatPromptTemplate.from_messages([("system", """You will receive an
    SQL3 query and a result. You will describe what the query gets to a user that does
    not know anything about databases, 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

    return full_chain, correction_chain, description_chain
```

4.4.1.3 Ottenimento del database

La funzione di ottenimento di un database al quale fornire le query è piuttosto semplice: si passa uno URI che identifichi univocamente il file sqlite3, e si sfrutta una funzione di libreria di LangChain per generarvi una variabile da ritornare. Si fa inoltre si di effettuare il DROP di una tabella per evitare che vi siano duplicati della stessa.

```
def get_database(uri):
    db = SQLDatabase.from_uri(uri)
    db.run("DROP TABLE IF EXISTS 'EXPENSES';")
    return db
```

4.4.1.4 Funzione di ricerca NLP

Una volta ottenuto il database e le diverse chain, si determina la presenza di una funzione che funga da wrapper. Alla stessa si passa la domanda, il database, le chain e delle opzioni di stampa. Compito principale della funzione è quello di passare le query generate al database, stampandone i risultati a video. Si noti anche la presenza di funzioni di parsing (replace, split), utili al fine di ottenere un output correttamente formattato.

```
def NLP(question, db, chain, correction_chain, description_chain, PRINT_SETTINGS):
    start = timer()
    if PRINT_SETTINGS["print_question"]:
        print(f"Question: {question}")

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

    if PRINT_SETTINGS["print_query"]:
        print(f"Original Query: {response}")

    today = datetime.today().strftime('%Y-%m-%d')
    response = correction_chain.invoke({"query":response, "question":question,
    "today":today}).strip()

    if PRINT_SETTINGS["print_corrected_query"]:
        print(f"Corrected query: {response}")

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

```
    if PRINT_SETTINGS["print_description"]:
        queryDescription = description_chain.invoke({"query":response})
        print(queryDescription)

    if not result or result=="None":
        return "No match found"
    else:
        # arrotondamento dei decimali alla seconda cifra
        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

    end = timer()

    if PRINT_SETTINGS["print_time"]:
        print(f"Time: {end-start}")

    return result
```

4.4.1.5 Testing

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.

4.4.2 Modello di RAG

4.4.2.1 Creazione del database vettoriale

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

```python
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.

4.4.2.2 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]
```

4.4.2.3 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
```

4.4.2.4 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)
```

4.4.2.5 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
```

4.4.2.6 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})
```

4.4.2.7 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)
```

4.4.2.8 Tools

I tool impiegati sono i seguenti:

```python
@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."

@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.

4.4.2.9 Testing

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.
```

4.5 Creazione di grafici

La fase di creazione dei grafici è stata articolata in due fasi distinte. Nella prima fase del progetto, è stata impiegata la libreria PandasAI insieme a ggplot, una soluzione completa e potente. Tuttavia, questa combinazione presentava un limite: i grafici venivano generati solo in formato PNG o SVG, poco adatti per applicazioni responsive o ottimizzate per il web.
Nella seconda fase è stata adottata una soluzione più avanzata, basata su diversi modelli di linguaggio (LLM), utilizzati per generare un file JSON formattato in conformità con le specifiche della libreria chart.js, garantendo così una maggiore flessibilità e integrazione web.
Una volta ultimata la fase di programmazione, si è deciso di impiegare la soluzione basata su chart.js. Tuttavia, la soluzione iniziale legata a PandasAI è ancora disponibile tra i file del progetto, poichè potrebbe essere utile per sviluppi futuri.

4.5.1 Formato SVG

La soluzione basata su PandasAI si basa sulla creazione di cosiddetti SmartDataFrame: si tratta dei comuni DataFrame già presenti all'interno della libreria di Pandas, ma con l'aggiunta di un nome, una descrizione, e un modello di LLM ad esso associato. In questo modo, in collaborazione con degli agenti atti alla creazione dei grafici, si è in grado di generare automaticamente grafici anche piuttosto complessi, senza neanche dover specificare la tipologia di visualizzazione richiesta. I DataFrame possono essere formati caricando direttamente tutto il database, oppure effettuando un input formattato dal contesto fornito da una ricerca di tipo RAG. La scelta tra i due metodi è appannaggio del modello di classificazione introdotto in apertura.
Di seguito si riporta il codice.

```
import re
import pandas as pd
from pandasai import SmartDataframe
from pandasai import Agent
from langchain_groq import ChatGroq
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

def get_plot_model():
    return ChatGroq(model="llama3-groq-70b-8192-tool-use-preview", temperature=0)

# It should receive the queries only, not the actual data
def get_plot_from_all(llm_plot, db, question, PRINT_SETTINGS):
    all_data = pd.read_sql_query("SELECT * from expensesok;", db)
    get_plot(all_data, question, llm_plot, PRINT_SETTINGS, is_there_time=True)

def get_plot_from_RAG(llm_plot, search_output, question, PRINT_SETTINGS):
    print(search_output)
    data = []
    pattern = r'(\d+\.\d+)\s+(.+)\s+(\w+)$'
    if (type(search_output) == list):
        for i in search_output:
            print(f"i: {i}")
            lines = i.split('\n')
            for line in lines:
                match = re.match(pattern, line)
                if match:
                    price = float(match.group(1))
                    description = match.group(2)
                    category = match.group(3)
                    data.append({
                        'price': price,
                        'description': description,
                        'category': category
                    })
                    print(data)
    else:
        lines = search_output.split('\n')
        for line in lines:
            print(line)
            match = re.match(pattern, line)
            if match:
                price = float(match.group(1))
                description = match.group(2)
                category = match.group(3)
                data.append({
                    'price': price,
                    'description': description,
                    'category': category
                })
```

```
    print(data)
    df = pd.DataFrame(data)
    get_plot(df, question, llm_plot, PRINT_SETTINGS, is_there_time=False)

def get_plot(dataframe, question, llm, PRINT_SETTINGS, is_there_time):
    agent = Agent(dataframe, config={"llm": llm}, description="""
    You are a data analysis agent.
    Your main goal is to help non-technical users to analyze data on their
    financial expenses.
    The charts must have numbers to ease the reading.
    Give an SVG file and not a PNG file.
    Make sure everything is visible.
    Make it very good looking.
    If you need to display categories, display everything.
    Don't print anything to terminal and don't open newly generated SVG files.
    Don't use floats for time.
    Only one plot per question.
    Always get a complete legend.
    """)
    sdf = SmartDataframe(dataframe, config={"llm": llm})
    if (is_there_time):
        sdf = SmartDataframe(sdf, name="Financial Expenses",
                                  description="""The database contains the
                                  columns price, description, category, timestamp.
                                  Each row is an expense of the user.
                                  You will be called to plot the correct data,
                                  given the question.""",
                                  config={"llm": llm})
    else:
        sdf = SmartDataframe(sdf, name="Financial Expenses",
                                  description="""The database contains the columns
                                  price, description, category.
                                  Each row is an expense of the user.
                                  You will be called to plot the correct data, given
                                  the question.""",
                                  config={"llm": llm})

    agent.chat(question + """Create an appropriate very good looking plotly chart
    using only the relevant data from the database.
    Give an SVG file and not a PNG file.
    Make sure everything is visible.
    Organize the spaces in a way that everything is readable. You don't have limits.
    If you need to display categories, display everything.
    Don't use floats for time.
    Make the chart very good looking.
    Only one plot per question.
    Always get a complete legend.""")

    if (PRINT_SETTINGS["print_explaination_plot"]):
        res = agent.explain()
        parser = StrOutputParser()
        system_template = """You will receive a description of the procedure used to
        extract a plot from the data.
        You will transform this description as if you were talking to a non-technical
        user, who only cares about which data got used and about the kind of chart
        created.
        Make the chart very good looking.
        The only thing you will talk about is what data you chose and what kind of
        chart you created.
        Don't talk about databases, SQL, SVG, titles, labels or any technical stuff.
        Don't talk about what the title and the labels are.
```

```
        Be very short and concise.

        The description is: {description}"""

        prompt_template = ChatPromptTemplate.from_messages(
                          [("system", system_template), ("user", "{description}")]
                          )

        chain = prompt_template | llm | parser
        res = chain.invoke({"description": res})
        print(f"Explaination: {res}")
```

4.5.2 Formato JSON

Volendo sfruttare la libreria Chart.js si rende necessaria la creazione di diversi modelli di LLM al fine di ottenere tutti i dati necessari. Nello specifico, si necessita di 4 modelli distinti:

1. Modello atto all'ottenimento delle label da inserire sull'asse x
2. Modello atto all'ottenimento e l'inserimento dei dati in formato JSON
3. Modello atto all'ottenimento del tipo di grafico da utilizzare
4. Modello atto alla determinazione del nome della grandezza rappresentata sull'asse y.

Come prevedibile, i dati sono ottenuti in combinazione con i modelli di NLP e RAG già precedentemente introdotti. Il modulo corrente si concentra perlopiù su funzioni di parsing più o meno complesse, utili all'ottenimento del file HTML/CSS/JavaScript richiesto per la visualizzazione del grafico. Il codice scritto è il seguente:

```
import json
import re
import ast
from langchain_groq import ChatGroq
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import JsonOutputParser
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.tools import tool
from operator import itemgetter
from langchain.tools.render import render_text_description

@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:
        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 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]

def tool_chain(model_output):
    tools = get_tools()
    tool_map = {tool.name: tool for tool in tools}
    chosen_tool = tool_map[model_output["name"]]
    return itemgetter("arguments") | chosen_tool

def get_tools():
    return [total, average, count, print_all, select_cheapest, select_most_expensive]
```

```
def get_rendered_tools():
    tools = get_tools()
    rendered_tools = render_text_description(tools)
    return rendered_tools

def stripOutput(received_input):
    if "I'm sorry" in received_input:
        return """{"id": 0, "name": "no_result", "arguments": {"context": [""]}}"""
    out = received_input.replace("<tool_call>","").replace("</tool_call>","")
    .replace("\\n","").replace("'","\"").replace('["','').replace('"]','')
    .replace('"', "").replace(", ", "\n").replace(",","\n").strip()
    return out

def get_type_chain():
    llm = ChatGroq(model="llama3-groq-70b-8192-tool-use-preview", temperature=0)
    prompt = PromptTemplate.from_template("""You are an expert at data visualization.
    You will receive a question from the user.
    You will have, as data, a list of expenses with the columns price, description,
    category, and timestamp.
    You will output the best type of chart to answer the question.
    The chart can be one of the following types: bar, pie, doughnut, line, polarArea,
    radar.
    Example UserInput: "What is the distribution of expenses by category?
    Give me the chart."
    Example Output: "pie"
    UserInput: {question}.""")

    get_type_chain = prompt | llm | StrOutputParser()
    return get_type_chain

def get_labels_chain():
    llm = ChatGroq(model="llama3-groq-70b-8192-tool-use-preview", temperature=0)
    prompt = PromptTemplate.from_template("""You are an expert at data visualization.
    You will receive a question from the user, the type of chart to realize and
    the data to use.
    You will have, as data, a list of expenses with the columns price, description,
    category, and timestamp.
    You will output only the labels for the chart, not the corresponding values.
    If you have labels to be time, order them in chronological order.
    Do not translate dates in English.
    Example UserInput: "What is the distribution of expenses by category?
    Give me the chart."
    Example Output: ["Food", "Transport", "Entertainment"]
    Give out the output, not the UserInput.
    Don't create new output, get it from {result_search}.
    Always take the label from the given data.
    UserInput: {question}.
    Type of chart: {chart_type}.
    Data: {result_search}.""")

    get_labels_chain = prompt | llm | StrOutputParser()
    return get_labels_chain

def get_label_chain():
    llm = ChatGroq(model="llama3-groq-70b-8192-tool-use-preview", temperature=0)
    prompt = PromptTemplate.from_template("""You are an expert at data visualization.
    You will receive a question from the user.
    You will understand what the user wants to know and output the label of the y axis.

    Example 1:
    UserInput: "What is the distribution of expenses by category? Give me the chart."
```

```
    Output: "Total expense"

    Example 2:
    UserInput: "Give me a chart that shows how many expenses I did in 2023."
    Output: "Number of expenses"

    You must not write anything like "Output: ", just write the title.
    UserInput: {question}.""")

    label_chain = prompt | llm | StrOutputParser()
    return label_chain

def get_data_chain():
    llm = ChatGroq(model="llama3-groq-70b-8192-tool-use-preview", temperature=0)
    prompt_template = """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.
    The number of elements in the lists must be equal to the number of labels.
    Here are the tools available and their descriptions:
    {rendered_tools}
    Your goal:
    - The data should be all of the same kind. No strings with floats, for example.
    - You must not provide all 0.0 as data.
    - Select the correct tool for the task.
    - If no specific time frame is provided, please start from today and extend
    back to the earliest year you can imagine
    - Ensure **every** part of the context is passed to the tools.
    - Provide a response in JSON format with 'name' and 'arguments' keys.
    - If "average" or "mean" in context, use the average tool.
    - You must get one value for each label.
    - Don't get more than one value for each label.
    Always output a readable JSON format for the JsonOutputParser.
    Example output: {{
    "id": 0,
    "name": "toolname",
    "arguments": {{
    "context": ["row1, row2, row3"]
    }}
    }}
    The argument "context" should always be a list. The rows are divided by a comma.
    UserInput: {question}.
    Type of chart: {chart_type}.
    Labels: {labels}.
    Context: {context}.
    Get a value for each label."""

    prompt = ChatPromptTemplate.from_template(
        prompt_template
    )

    prompt = prompt.partial(
        rendered_tools="{rendered_tools}",
        question="{question}",
        chart_type="{chart_type}",
        labels="{labels}",
        context="{context}"
    )
```

```
    data_chain = prompt | llm | StrOutputParser()  | JsonOutputParser() | tool_chain
    return data_chain


def get_graph_type(type_chain, question):
    return type_chain.invoke({"question": question})

def get_labels(lables_chain, question, chart_type, result_search, PRINT_SETTINGS):
    labels = lables_chain.invoke({"question": question, "chart_type": chart_type, "result_search":

    if labels.startswith("[") and labels.endswith("]"):
        try:
            labels = ast.literal_eval(labels)
        except (ValueError, SyntaxError):
            labels = []

        if (PRINT_SETTINGS["print_plot_labels"]):
            print(f"Labels: {labels}")
        return labels
    else:
        labels = [label.strip() for label in labels.split(',')]
        if (PRINT_SETTINGS["print_plot_labels"]):
            print(f"Labels: {labels}")
        return labels

def get_data_RAG(data_chain, question, chart_type, labels, context, PRINT_SETTINGS):
    data = data_chain.invoke({"question": question, "chart_type": chart_type,
    "labels": labels, "context": context, "rendered_tools": get_rendered_tools()})
    if (PRINT_SETTINGS["print_plot_data"]):
        print(f"Data: {data}")

    # Check if the input is a list-like string
    if isinstance(data, str) and data.startswith("[") and data.endswith("]"):
        try:
            data = ast.literal_eval(data)
        except (ValueError, SyntaxError):
            data = []
    else:
        # Handle a comma-separated string
        data = str(data)
        data = [info.strip() for info in data.split(',')]

    if (len(data) > 1):
        data_dict = {data[i]: data[i+1] for i in range(0, len(data), 2)}

        aligned_data = [data_dict.get(label, 0.0) for label in labels]
        return aligned_data

    else:
        return data

def get_data_NLP(labels, context, PRINT_SETTINGS):
    if context.startswith("[") and context.endswith("]"):
        try:
            context = ast.literal_eval(context)
        except (ValueError, SyntaxError):
            context = []
    else:
        context = [data.strip() for data in context.split(',')]

    if (PRINT_SETTINGS["print_context"]):
```

```
        print(f"Context: {context}")

    if isinstance(context[0], str) and 'T' in context[0]:
        data_dict = {entry[:10].strip('"'): context[i+1] for i, entry in enumerate(context) if i %
    else:
        # Data contains simple labels and values
        data_dict = {context[i].strip('"'): context[i+1] for i in range(0, len(context), 2)}

    aligned_data = [data_dict.get(label.strip('"').lower(), 0.0) for label in labels]

    if (PRINT_SETTINGS["print_plot_data"]):
        print(f"Aligned Data: {aligned_data}")
    return aligned_data



def get_label_title(question, model):
    return model.invoke({"question": question})

def write_chart_html(chart_type, labels, data, label, filename="chart.html"):
    data = {
        'labels': labels,
        'datasets': [{
            'label': label,
            'data': data,
            'backgroundColor': 'rgba(75, 192, 192, 0.2)',
            'borderColor': 'rgba(75, 192, 192, 1)',
            'borderWidth': 1
        }]
    }

    config = {
        'type': chart_type,  # You can change this to 'bar', 'pie', 'doughnut', 'line', 'polarAre
        'data': data,
        'options': {
            'responsive': True,
            'maintainAspectRatio': False,
            'scales': {
                'y': {
                    'beginAtZero': True
                }
            }
        }
    }

    html_content = f"""
        <!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Chart</title>
        <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
        <style>
            * {{
                margin: 0;
                padding: 0;
                box-sizing: border-box;
            }}
            body, html {{
                height: 100%;
```

```
            font-family: Arial, sans-serif;
            background-color: #f4f4f9;
        }}
        body {{
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            position: relative;
        }}
        .chart-container {{
            width: 80%;
            max-width: 900px;
            height: 70vh;
            background: #fff;
            padding: 20px;
            box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
            border-radius: 8px;
            z-index: 10;
            position: relative;
        }}
        canvas {{
            display: block;
        }}
    </style>
</head>
<body>
    <div class="chart-container">
        <canvas id="myChart"></canvas>
    </div>
 <script>
        // Data and config from Python
        const data = {data};
     const config = {json.dumps(config)};
     // Render the chart
        const myChart = new Chart(
            document.getElementById('myChart'),
            config
        );
    </script>
</body>
</html>"""

with open(filename, 'w') as f:
    f.write(html_content)
```

4.6 File main.py

Il file main si occupa di importare tutti i moduli sopra descritti, inizializzare il database, caricare le variabili di ambiente (in particoalre, la chiave API per l'accesso a Groq) e le chain ed attendere le domande dell'utente. Si definiscono inoltre le pseudo-costanti (in Python non esistono costanti, ci si affida alla convenzione di scriverne il nome in maiuscolo) utilizzate per orchestrare la stampa dei risultati. Il codice è piuttosto auto-esplicativo: lo si riporta di seguito.

```
import os
import sqlite3
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_input_plot import get_tagging_chain, get_classification
from module_input import get_input_chain, input_into_database
from module_plot_SVG import get_plot_model, get_plot_from_all, get_plot_from_RAG
from module_chartJS import (
    get_data_chain, get_labels_chain, get_type_chain,
    get_graph_type, get_labels, get_data_RAG, get_data_NLP,
    write_chart_html, get_label_title, get_label_chain
)

# Pseudo-costanti per configurazione
URI_DB = "sqlite:///googleDb.sqlite3"
PERSIST_DIRECTORY = "./chroma/expenses"
INPUT_DB = "googleDb.sqlite3"
MAX_DESCRIPTION_LENGTH = 255

PRINT_SETTINGS = {
    "print_question": False,
    "print_query": False,
    "print_description": False,
    "print_corrected_query": False,
    "print_time": False,
    "print_scores": False,
    "print_chunks": False,
    "print_smallest_chunk": False,
    "print_context": False,
    "print_method": False,
    "print_characteristics_of_the_question": False,
    "print_explaination_plot": False,
    "call_SVG_plot": False,
    "call_JSON_plot": True,
    "print_plot_type": False,
    "print_plot_labels": False,
    "print_plot_data": False
}

# Carica le variabili di ambiente dal file .env
def load_environment_variables():
    load_dotenv()
    os.environ["GROQ_API_KEY"] = os.getenv("GROQ_API_KEY")

# Inizializza le componenti per l'NLP
def initialize_nlp(uri_db):
    nlp_db = get_database(uri_db)
    full_chain, correction_chain, description_chain = get_NLP_chains(nlp_db)
    return nlp_db, full_chain, correction_chain, description_chain

# Inizializza il database RAG
def initialize_rag(persist_directory):
    return get_embedded_database(persist_directory)

# Gestisce i grafici SVG
def handle_svg_plot(method, question, connection, rag_db, PRINT_SETTINGS):
    if method[0] == "NLP":
        get_plot_from_all(get_plot_model(), connection, question, PRINT_SETTINGS)
    elif method[0] == "RAG":
        response = RAG(question, rag_db, stripOutput, PRINT_SETTINGS, is_for_plot=True)
        get_plot_from_RAG(get_plot_model(), response, question, PRINT_SETTINGS)

# Gestisce i grafici JSON
def handle_json_plot(method, question, labels_chain, label_chain, type_chain,
data_chain, rag_db, nlp_db, PRINT_SETTINGS):
```

```
    label = get_label_title(question, label_chain)

    if method[0] == "NLP":
        response = NLP(question, nlp_db, *initialize_nlp(URI_DB)[1:], PRINT_SETTINGS)
        chart_type = get_graph_type(type_chain, question)
        labels = get_labels(labels_chain, question, chart_type, response, PRINT_SETTINGS)
        try:
            data = get_data_NLP(labels, response, PRINT_SETTINGS)
        except Exception:
            print("""There was an error in the creation of the chart.
            Try to be more specific.""")
            return
    elif method[0] == "RAG":
        response = RAG(question, rag_db, stripOutput, PRINT_SETTINGS, is_for_plot=True)
        chart_type = get_graph_type(type_chain, question)
        labels = get_labels(labels_chain, question, chart_type, response, PRINT_SETTINGS)
        try:
            data = get_data_RAG(data_chain, question, chart_type, labels, response, PRINT_SETTINGS
        except Exception:
            print("""There was an error in the creation of the chart.
            Try to be more specific.""")
            return

    if not any(item != 0.0 for item in data):
        print("There was an error in the creation of the chart. Try to be more specific.")
    else:
        if len(labels) == 1:
            chart_type = "bar"
        if (PRINT_SETTINGS["print_plot_type"]):
            print(f"Chart type: {chart_type}")

        write_chart_html(chart_type, labels, data, label)

# Funzione main
def main():
    load_environment_variables()

    # Get the method chain
    tagging_chain = get_tagging_chain()

    # Initialize input database
    connection = sqlite3.connect(INPUT_DB)
    cursor = connection.cursor()
    input_chain = get_input_chain()

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

    # Initialize plot chains
    label_chain = get_label_chain()
    type_chain = get_type_chain()
    data_chain = get_data_chain()
    labels_chain = get_labels_chain()

    while True:
        question = input("Enter your question ('X' or 'x' to exit): ")
        if question.lower() == 'x':
            exit()
        method = get_classification(question, tagging_chain, PRINT_SETTINGS)
        if PRINT_SETTINGS["print_method"]:
```

```
                print(f"Chosen method: {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 == "INPUT":
                response = input_into_database(question, input_chain, cursor,
                MAX_DESCRIPTION_LENGTH)
            elif "|" not in method:
                if method == "NLP":
                    response = NLP(question, nlp_db, full_chain, correction_chain,
                    description_chain, PRINT_SETTINGS)
                else:
                    response = RAG(question, rag_db, stripOutput, PRINT_SETTINGS)
            elif PRINT_SETTINGS["call_JSON_plot"]:
                handle_json_plot(method.split("|"), question, labels_chain, label_chain,
                type_chain, data_chain, rag_db, nlp_db, PRINT_SETTINGS)
            elif PRINT_SETTINGS["call_SVG_plot"]:
                handle_svg_plot(method.split("|"), question, connection, rag_db,
                PRINT_SETTINGS)
            else:
                response = """An error occurred while trying to classify the question.
                Try again."""
            if "PLT" not in method:
                    print(f"Response: {response}.")

if __name__ == "__main__":
    try:
        main()
    except Exception as error:
        print(f"An error occurred: {error}.")
        exit(1)
```

Il costrutto if __name__ == "__main__": in Python viene utilizzato per verificare se un modulo è stato eseguito direttamente come script principale oppure importato in un altro modulo. Quando un file Python viene eseguito direttamente, la variabile speciale __name__ assume il valore "__main__". Di conseguenza, il codice all'interno di questo blocco viene eseguito solo quando il file viene eseguito direttamente e non quando è importato. In questo caso, la funzione main() verrebbe chiamata solo se il file è eseguito come script principale, permettendo di organizzare il codice in modo modulare ed evitare l'esecuzione indesiderata di alcune parti del programma quando il file è importato altrove.
Risulta interessante notare anche le modalità mediante le quali si gestiscono gli errori. Sebbene spesso si consideri scorretto trattare ogni tipo di eccezione all'interno di un unico costrutto try/except, in questo caso risulta congeniale farlo per evitare che una qualunque eccezione possa raggiungere il "livello Python", causando il crash dell'applicazione. Così facendo, si riesce a far si che l'applicazione ritorni un valore di "1" alla shell informandola del fallimento.

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/.