POLITECNICO DI TORINO

Corso di Laurea Magistrale
In Ingegneria Matematica

Tesi di Laurea Magistrale

Sistemi di Ricerca Intelligente: tecniche di *Data Retrieval e Large Language Models* applicati a un contesto aziendale

Relatrice
Prof.ssa Tania Cerquitelli

Candidata
Cecilia Bartoletti

Anno Accademico 2023-2024

Indice

Elenco delle figure 6

1 Introduzione 9

2 Avnet 11

2.1 Storia dell'azienda . . . 11

2.2 Il modello speedboat di Avnet . . . 12

2.3 myDA . . . 14

2.3.1 X-Ref - Cross search . . . 15

3 AI and RAG 17

3.1 LLM - Large Language Model . . . 18

3.1.1 Cos'è un LLM . . . 18

3.1.2 Come sono costruiti gli LLM . . . 18

3.1.3 Limiti dei LLM . . . 19

3.2 RAG . . . 20

3.2.1 Struttura del RAG . . . 20

4 Modello 23

4.1 Assunzioni . . . 24

4.2 Attributi principali su cui fare ricerca . . . 24

4.3 Embedding . . . 26

4.3.1 Sentence embedding . . . 27

4.3.2 I predecessori di MPNet: BERT e XLNet . . . 30
4.3.3 MPNet . . . 32
4.3.4 Implementazione MPNet . . . 33
4.4 Indice di ricerca vettoriale . . . 35
4.4.1 Cos'è un indice vettoriale . . . 36
4.4.2 Indice vettoriale Atlas . . . 36
4.4.3 Costruzione dell'indice . . . 37
4.5 LLM: GPT-4o-mini . . . 41
4.6 Workflow . . . 42
4.6.1 La funzione principale: handle_user_input . . . 43
4.6.2 Definizione delle variabili di ambiente e connessione ai servizi . . . 48
4.6.3 funzione get_taxonomies . . . 52
4.6.4 funzione get_torch_model . . . 54
4.6.5 funzione _extract_category_from_input . . . 57
4.6.6 funzione _parse_response . . . 57
4.6.7 funzione _find_similar_products . . . 58
4.6.8 funzione _identify_prompt . . . 69
4.6.9 funzione _prompt_formatter_pn . . . 71
4.6.10 funzione _prompt_formatter . . . 73
4.6.11 funzione _send_chat_request . . . 74
4.7 Esempi di applicazione . . . 75
4.7.1 Ricerca di un materiale simile . . . 75
4.7.2 Descrizione di un materiale . . . 77
4.7.3 Confronto tra due materiali . . . 79
4.8 Prototipo . . . 81

5 Conclusioni 83
5.1 Obiettivi e risultati . . . 83
5.2 Sviluppi Futuri . . . 85

Elenco delle figure

4.1 Architettura Transformers 28
4.2 Architettura Sentence-Transformers 29
4.3 BERT pre-training. 30
4.4 XLNet pre-training 31
4.5 Prototipo del chatbot 81
4.6 Prototipo del chatbot durante una richiesta 81

Capitolo 1

Introduzione

Negli ultimi anni, l'intelligenza artificiale (AI) ha rivoluzionato numerosi settori, tra cui la gestione e l'analisi delle informazioni aziendali. In particolare, l'evoluzione dei modelli di linguaggio di grandi dimensioni (LLM, Large Language Models) e delle tecniche di Recupero Guidato (RAG, Retrieval-Augmented Generation) ha aperto nuove possibilità per l'elaborazione del linguaggio naturale (NLP). Queste tecnologie, grazie alla loro capacità di comprendere e generare testi in modo intelligente, trovano oggi applicazione in ambiti sempre più diversificati, dalla ricerca scientifica all'industria.

Questa tesi si propone di sviluppare un modello che integra l'approccio LLM con la tecnica RAG, al fine di permettere agli utenti di effettuare ricerche personalizzate sui materiali industriali. L'obiettivo principale è quello di superare le limitazioni dei sistemi di ricerca attualmente esistenti, che si basano esclusivamente su confronti diretti tra numeri di parte (Part Numbers, PN), introducendo una risorsa alternativa che consenta di formulare domande personalizzate basate su specifiche tecniche e requisiti. Questo modello non si limita alla ricerca di materiali simili a un Part Number esistente, ma permette di effettuare richieste più dettagliate, come la ricerca di un PN simile con determinate caratteristiche, la ricerca per tassonomia o il confronto tra due PN distinti.

Per raggiungere questo obiettivo, il progetto si articola in diverse fasi. In primo luogo, viene analizzata la struttura dei modelli di linguaggio di grandi dimensioni, evidenziandone vantaggi e limiti, con un focus sulle loro applicazioni pratiche nel contesto aziendale.

Successivamente, viene introdotta la metodologia RAG, che consente di potenziare le capacità dei LLM attraverso l'accesso a basi di conoscenza esterne e specifiche, nel nostro caso aziendali, riducendo così il rischio di informazioni inesatte o obsolete. Il cuore della ricerca consiste nella progettazione e implementazione di un modello che combina queste due tecnologie, sfruttando tecniche di embedding e ricerca vettoriale per migliorare la pertinenza e la precisione delle risposte fornite agli utenti.

L'applicazione pratica di questo modello viene testata nel contesto aziendale di Avnet, una delle principali aziende nel settore della distribuzione di componenti elettronici e soluzioni tecnologiche. Attraverso l'utilizzo di un indice di ricerca vettoriale basato su MongoDB Atlas e del modello di embedding All-MPNet-base-v2, il sistema proposto è in grado di elaborare query complesse e restituire risultati più coerenti e pertinenti rispetto ai metodi tradizionali.

I risultati ottenuti dimostrano come l'integrazione di LLM e RAG possa rappresentare un significativo passo avanti nel miglioramento dei sistemi di ricerca aziendali, permettendo un accesso più rapido e preciso alle informazioni tecniche. Infine, vengono delineate le possibili evoluzioni future del modello, con particolare attenzione all'ottimizzazione delle risorse computazionali e all'espansione del sistema verso nuovi settori applicativi.

Capitolo 2

Avnet

2.1 Storia dell'azienda

Avnet è una delle aziende più importanti al mondo nel settore della distribuzione di componenti elettronici e soluzioni tecnologiche, con una lunga e affascinante storia che risale al 1921. Fondata da Charles Avnet, un immigrato russo che si trasferì negli Stati Uniti, l'azienda ebbe inizio a New York, dove si occupava inizialmente di vendere parti radio in surplus. La sua strategia si rivelò vincente, poiché in quel periodo l'interesse per la radio si stava espandendo rapidamente nelle case americane. Durante i primi anni, Avnet si concentrò sull'acquisto e la rivendita di componenti elettronici, servendo una clientela composta principalmente da produttori di apparecchiature radio e rivenditori.
Con l'avvento della Grande Depressione, molti commercianti erano in difficoltà, ma Charles Avnet riuscì a trasformare la crisi in un'opportunità. Decise di passare dal commercio al dettaglio a quello all'ingrosso, riuscendo a mantenere l'azienda solida, saldando i debiti e persino generando un piccolo profitto. La sua abilità nel gestire l'azienda, unita a una reputazione di integrità, gli permise di superare questo periodo critico e gettò le basi per un'espansione futura.
Durante la Seconda Guerra Mondiale, Avnet continuò a crescere, producendo antenne per le forze armate statunitensi. Dopo la guerra, la società si concentrò maggiormente sull'acquisto e la rivendita di componenti elettronici e elettrici in surplus, mantenendo

il focus sull'innovazione e sul supporto alla crescita tecnologica dell'epoca. Nel 1955, l'azienda cambiò il suo nome in Avnet Electronic Supply Company e fu incorporata ufficialmente, focalizzandosi soprattutto sulla distribuzione di condensatori, interruttori e altri componenti elettronici. L'anno successivo, nel 1956, Avnet aprì un secondo impianto di assemblaggio vicino a Los Angeles, dedicato in particolare all'industria aeronautica. Negli anni '60, l'azienda ampliò le sue operazioni, diversificando la sua offerta e integrando nella sua rete anche apparecchiature audio e strumenti musicali, un settore che portò Avnet a guadagnarsi un posto alla Borsa di New York nel 1960.
Il vero salto di qualità per Avnet arrivò con l'inizio della sua collaborazione con Intel negli anni '70, diventando il primo distributore di semiconduttori Intel. Questo accordo consolidò ulteriormente la posizione dell'azienda nel settore elettronico e le permise di espandersi anche nel mercato dei computer. Nello stesso periodo, Avnet diversificò ulteriormente la sua attività, inserendo nel suo portafoglio di prodotti i sistemi completi per computer e i software. Già nel 1979, Avnet raggiunse un fatturato di un miliardo di dollari, un traguardo che simboleggiava il successo della sua strategia di espansione.
Nel corso degli anni '80 e '90, Avnet continuò a crescere, espandendo la sua presenza a livello globale. Nel 1998, l'azienda decise di trasferire la sua sede centrale a Phoenix, in Arizona, per consolidare la propria posizione come leader globale nel settore. Nel 2001, Avnet compì un'importante acquisizione, quella di Kent Electronics Corp., per 600 milioni di dollari. Allo stesso modo, acquisì Bell Microproducts nel 2010, espandendo la sua offerta nel mercato dei componenti elettronici. Queste acquisizioni e altre simili consolidarono ulteriormente la posizione di Avnet, che riuscì ad espandere la propria influenza in nuovi mercati e a entrare in nuovi segmenti tecnologici.

2.2 Il modello speedboat di Avnet

Uno degli sviluppi più significativi nella storia recente di Avnet è stato l'introduzione nel 2001 del modello di business "speedboat". Questo modello prevedeva la creazione di unità operative specializzate, ciascuna focalizzata su segmenti specifici del mercato, per rispondere più agilmente alle necessità dei clienti e per adattarsi rapidamente ai cambiamenti

del mercato. Le principali divisioni speedboat includono:

- *EBV Elektronik*: specializzata nella distribuzione di semiconduttori, che fornisce anche supporto tecnico avanzato a livello europeo

- *Avnet Abacus*: concentrata su componenti passivi, elettromeccanici e di interconnessione, con una forte attenzione all'assistenza pre e post vendita

- *Avnet Silica*: dedicata ai semiconduttori, supporta i clienti in fase di progettazione e sviluppo di soluzioni personalizzate

Questo modello ha avuto grande successo in Europa e ha permesso a Avnet di sviluppare una struttura molto più flessibile, permettendo alle divisioni di rispondere rapidamente alle esigenze dei vari segmenti di mercato e di adattarsi alle tecnologie emergenti. L'approccio speedboat ha permesso a ciascuna divisione di agire in modo indipendente, mantenendo però il vantaggio di essere parte di una struttura globale ben consolidata.
A livello globale, Avnet ha una presenza in più di 140 paesi e gestisce 14 centri di distribuzione, 13 centri di integrazione e otto centri di programmazione, il che le consente di offrire un'ampia rete logistica in grado di soddisfare le esigenze di clienti di ogni parte del mondo. L'azienda è strutturata in modo tale da garantire la massima efficienza nelle sue operazioni, con una rete globale che offre supporto tecnico, logistica e soluzioni personalizzate.
Oggi, Avnet continua a essere un leader nel settore, non solo grazie alla sua rete globale e alla struttura organizzativa snella, ma anche per l'innovazione continua. La leadership è composta da figure chiave come il CEO Phil Gallagher, che guida l'azienda verso nuove opportunità di crescita nei mercati emergenti, e dal presidente Rodney Adkins, che si concentra sulla strategia tecnologica e sull'espansione nei settori ad alta crescita. Nel 2019, Avnet ha dimostrato ancora una volta la sua attitudine all'innovazione, iniziando ad accettare criptovalute come metodo di pagamento per i suoi prodotti e servizi, un segno tangibile della sua apertura alle nuove tecnologie.

2.3 myDA

Al servizio di Avnet EMEA, il Digital Innovation Team ha sviluppato myDA, acronimo di my Digital Assistant, un'applicazione progettata per ottimizzare e supportare il processo di vendita. Questo strumento è stato creato per essere utilizzato dai dipendenti dell'azienda, offrendo loro una piattaforma centralizzata e intelligente che semplifica numerose attività legate alla gestione delle vendite. Tra le principali funzionalità di myDA vi è la possibilità di fornire agli utenti una lista di Part Numbers (PN) di materiali disponibili in tempo reale, che possono essere proposti ai clienti. La piattaforma personalizza inoltre la lista delle opportunità in base al portfolio clienti dell'utente, permettendo un approccio mirato e ottimizzato alle vendite. Un'altra funzione chiave è la possibilità di registrare design associando uno o più PN a un cliente specifico, consentendo di configurare soluzioni personalizzate in modo rapido ed efficace. Nel caso in cui un determinato PN non sia disponibile o sia necessario cercarne uno alternativo con caratteristiche simili, myDA offre una funzionalità di ricerca X-Ref (cross-reference), che permette di individuare opzioni equivalenti o con specifiche alternative. L'applicazione integra anche una sezione dedicata alle missing opportunities, identifica quote o ordini non associati a nessun progetto o design registrato. Questa funzione consente agli utenti di identificare e registrare nuovi progetti.

Infine, myDA include un'area analytics avanzata, che fornisce una panoramica dettagliata delle performance attraverso diversi KPI (Key Performance Indicators). I dati sono organizzati e visualizzati per criteri come paese, ruolo dell'utente, o categorie specifiche come le "speedboat". Questa sezione permette agli utenti di monitorare i risultati, individuare tendenze e guidare decisioni data-driven per migliorare le prestazioni complessive.

In sintesi, myDA è molto più di uno strumento di gestione: rappresenta un alleato strategico per il team vendite di Avnet EMEA, progettato per migliorare l'efficienza operativa, sfruttare al meglio le opportunità di mercato e offrire un'esperienza cliente altamente personalizzata.

2.3.1 X-Ref - Cross search

Lo strumento di ricerca di part number simili integrato in myDA si basa su un'architettura a tre livelli, progettata per identificare in modo accurato i materiali alternativi. Questo sistema sfrutta grafi di similarità, database avanzati e integrazioni con fornitori per offrire risultati precisi e personalizzati. Ecco il funzionamento nel dettaglio:

- Primo livello: al primo livello viene costruito un grafo di similarità utilizzando i materiali per cui sono disponibili informazioni tecniche aziendali archiviate in file. Nel grafo, ogni materiale è rappresentato come un nodo, mentre gli archi tra i nodi sono pesati in base al grado di similarità tra i materiali. Questa struttura consente di creare una rete relazionale tra i materiali, facilitandone il confronto.

- Secondo livello: il secondo livello si basa su un database Elastic che organizza gli attributi dei materiali secondo le tassonomie. Ogni attributo di ogni tassonomia viene suddiviso in gruppi o cluster, determinati in base alla numerosità dei dati disponibili per quella categoria. Per cercare un part number simile, viene calcolato uno score di similarità confrontando gli attributi del materiale uno per uno. Se i cluster degli attributi coincidono o risultano vicini, si applica un modello che considera il grado di vicinanza. Lo score viene calcolato anche tenendo conto del peso dei singoli parametri, poiché alcuni attributi possono essere più rilevanti di altri. Un valore soglia predefinito stabilisce infine il livello minimo di similarità necessario affinché un materiale venga considerato simile.

- Terzo livello: il terzo livello si basa su API fornite dai produttori o fornitori, che permettono di espandere ulteriormente la ricerca. Queste API, partendo da un part number specifico, restituiscono una lista di materiali alternativi presenti nei loro database, ciascuno corredato da uno score di similarità. Questo livello consente di accedere direttamente a informazioni aggiornate e dettagliate fornite dai partner aziendali.

Grazie a questa struttura a tre livelli, il sistema di ricerca di myDA combina dati interni,

analisi tassonomiche avanzate e risorse esterne, offrendo uno strumento potente per trovare materiali alternativi.

Tuttavia una limitazione di questo strumento di ricerca risiede nella sua rigidità nell'adattarsi a esigenze di ricerca specifiche. L'utente può inserire il numero di un part number esistente per avviare la ricerca e ottenere alternative simili, ma non ha la possibilità di definire ulteriori richieste personalizzate. Ad esempio, non è possibile specificare che si sta cercando un materiale con una determinata caratteristica tecnica, né filtrare i risultati per ottenere tutti i part number appartenenti a una specifica tassonomia e dotati di attributi particolari. Questa rigidità limita le possibilità di esplorazione e personalizzazione delle ricerche, soprattutto quando il semplice confronto tra part number non è sufficiente a soddisfare le esigenze dell'utente. Sebbene lo strumento X-Ref sia estremamente accurato nel fornire part number alternativi con caratteristiche simili, questo livello di precisione comporta una perdita di scalabilità e flessibilità. Il modello presentato in questa tesi si propone di superare queste limitazioni, offrendo uno spettro di ricerca molto più ampio e adattabile. L'obiettivo è quello di sviluppare uno strumento che permetta agli utenti di specificare criteri di ricerca dettagliati e personalizzati. Ad esempio, l'utente potrà definire vincoli tecnici, come l'esigenza di mantenere un attributo uguale a un determinato valore, oppure esplorare intere tassonomie filtrando per caratteristiche specifiche. In questo modo, sarà possibile estendere le capacità del sistema, rendendolo più versatile e scalabile, senza compromettere l'affidabilità dei risultati.

L'integrazione di questi miglioramenti mira a trasformare il processo di ricerca, passando da un approccio limitato e guidato esclusivamente dal confronto di part number a un sistema più dinamico, che risponde in modo preciso alle esigenze variabili degli utenti e consente una gestione più efficace delle opportunità di vendita e progettazione.

Capitolo 3

AI and RAG

Negli ultimi anni, l'intelligenza artificiale, AI, ha assunto un ruolo centrale nel panorama tecnologico, trovando applicazione in una vasta gamma di settori, dall'industria alla sanità, fino alla gestione dei dati e alla comunicazione aziendale. Tra le tecnologie più avanzate nel campo dell'AI spiccano i modelli di linguaggio di grandi dimensioni (LLM, Large Language Models) e le architetture basate sul Recupero Guidato (RAG, Retrieval-Augmented Generation). Questi strumenti stanno ridefinendo il modo in cui le aziende gestiscono le informazioni e interagiscono con i loro clienti, fornendo risposte intelligenti, personalizzate e contestualizzate in tempo reale.
In questo capitolo verranno esaminati in dettaglio i principi fondamentali che regolano queste tecnologie, con un focus specifico su:

- LLM (Large Language Models): cosa sono, come funzionano, e perché rappresentano una svolta nella comprensione e generazione del linguaggio naturale
- RAG (Retrieval-Augmented Generation): un'architettura innovativa che combina i punti di forza dei modelli di linguaggio con sistemi di recupero delle informazioni, creando un approccio ibrido capace di unire la creatività e la conoscenza storica.

Dopo una panoramica teorica, si introdurrà l'applicazione pratica di queste tecnologie nel contesto aziendale, con un approfondimento sul modello da me implementato. Quest'ultimo combina le potenzialità di un LLM con un sistema RAG per dare vita a una

chat intelligente progettata specificamente per le esigenze di Avnet. Questa chat non solo sfrutta l'intelligenza artificiale per interpretare le richieste degli utenti e generare risposte naturali e pertinenti, ma utilizza il recupero guidato RAG per accedere alle fonti informative aziendali, garantendo che le risposte siano basate su informazioni aggiornate e precise. Prima di analizzare nel dettaglio l'implementazione di questo modello, sarà utile comprendere le basi teoriche e i vantaggi di AI e RAG, che rappresentano il cuore tecnologico di questa soluzione.

3.1 LLM - Large Language Model

3.1.1 Cos'è un LLM

Un LLM, Large Language Model, è un modello di intelligenza artificiale basato sull'apprendimento automatico, progettato per comprendere e generare linguaggio naturale in modo altamente sofisticato. Questi modelli sono addestrati su grandi quantità di dati testuali per eseguire un'ampia gamma di compiti legati al linguaggio, come traduzioni, risposte a domande, generazione di testi e molto altro. Questo approccio rappresenta un'alternativa altamente efficiente rispetto allo sviluppo di modelli specifici per ogni singolo compito o dominio, un processo che sarebbe estremamente oneroso in termini di costi, infrastruttura e tempo. Inoltre, l'approccio generalista degli LLM permette di sfruttare le sinergie tra i diversi compiti linguistici, migliorandone le prestazioni e garantendo una flessibilità superiore rispetto ai modelli dedicati. Gli LLM rappresentano una svolta significativa nel mondo dell'AI e sono facilmente accessibili al pubblico tramite interfacce i modelli GPT di Open AI, Gemini di Google, Llama di Meta ed altri. In poche parole, gli LLM sono progettati per comprendere e generare testo, oltre ad altre forme di contenuto, come un essere umano, sulla base della grande quantità di dati utilizzati per addestrarli.

3.1.2 Come sono costruiti gli LLM

I modelli LLM si basano su reti neurali profonde, in particolare sull'architettura dei trasformatori, una tecnologia introdotta nel 2017 con l'articolo scientifico intitolato *"Attention*

Is All You Need". Il nucleo di questa architettura è il meccanismo di *self-attention*, che permette al modello di analizzare una sequenza di testo e identificare le parti più rilevanti in relazione al contesto. I modelli come GPT, Generative Pre-trained Transformer, utilizzano il decoder dei trasformatori per generare testo, mentre altri, come BERT, sfruttano sia encoder che decoder per quanto riguarda la comprensione del linguaggio. La peculiarità di un LLM è la sua grandezza: sono presenti miliardi, a volta trilioni, di parametri, ovvero i pesi della rete neurale che determinano il comportamento del modello. Durante la fase di addestramento, questi parametri vengono ottimizzati su grandi dataset, spesso composti da una varietà di fonti, come libri, articoli, siti web ed anche codice sorgente. Lo scopo dell'addestramento è far sì che il modello impari a predire con precisione la parola successiva in una sequenza di parole, un processo chiamato *causal language modeling*. In altri casi, l'obiettivo può essere quello di prevedere parole mancanti in un testo, come avviene nel masked language modeling. Le capacità di un LLM come si può evincere, sono incredibilmente versatili: può infatti analizzare testi complessi, riassumerli, tradurli o rispondere a domande, può generare contenuti come articoli, poesie, storie o persino codice, e supportare attività come la risoluzione di problemi, l'elaborazione di dati linguistici per le aziende o la generazione automatica di risposte nei servizi di assistenza clienti. È uno strumento che trova applicazione in diversi campi come l'elaborazione del linguaggio naturale, l'intelligenza artificiale conversazionale, l'analisi dei dati, la programmazione e la ricerca scientifica.

3.1.3 Limiti dei LLM

Nonostante le capacità e potenzialità di un modello LLM, si presentano diversi limiti. Infatti il modello apprende dai dati su cui è addestrato e di conseguenza può riflettere bias o pregiudizi presenti in tali dati, rischiando di produrre risposte non equilibrate o culturalmente inappropriate. Un altro problema risiede nei costi: la creazione e l'uso dei LLM richiedono enormi risorse computazionali ed energetiche, rendendoli costosi da sviluppare e mantenere. Gli LLM inoltre non comprendono realmente il linguaggio o il mondo come farebbe un essere umano, ma generano risposte basate su schemi statistici appresi, il che può portarli a produrre informazioni errate o non verificabili, fenomeno

noto come "allucinazioni". Tecniche come il *fine-tuning*, ovvero l'adattamento del modello su un compito specifico, o l'ingegneria dei prompt, che consiste nel progettare input particolarmente efficaci, o la tecniche RAG, Retrieval-Augmented Generation, che ora verrà introdotta, stanno migliorando ulteriormente le loro prestazioni. In conclusione, i modelli LLM rappresentano una pietra miliare nel campo dell'intelligenza artificiale e dell'elaborazione del linguaggio naturale, NLP. Nonostante i loro limiti, costituiscono comunque uno strumento potente e versatile, capace di rivoluzionare il modo in cui interagiamo con la tecnologia e il linguaggio.

3.2 RAG

In aiuto alle limitazioni dei modelli LLM arriva la tecnica di RAG, acronimo di Retrieval-Augmented Generation, ovvero "Generazione potenziata dal recupero". Questo approccio innovativo risolve i punti deboli dei modelli di linguaggio LLM, come l'incapacità di accedere a informazioni aggiornate, specifiche o riservate, e il rischio di produrre risposte inesatte o basate su dati incompleti. La RAG ottimizza l'output del modello linguistico di grandi dimensioni, in modo che faccia riferimento a una base di conoscenza al di fuori delle sue fonti di dati di addestramento prima di generare una risposta. La RAG estende le capacità già avanzate degli LLM a domini specifici o alla conoscenza base interna di un'organizzazione, il tutto senza la necessità di riaddestrare il modello. Questo approccio permette di migliorare l'output fornito dal LLM in modo che rimanga pertinente, accurato e utile in vari contesti. Le informazioni mirate ottenute con la RAG possono essere più aggiornate rispetto al LLM, ma anche più precise nel caso di organizzazioni e settori specifici. In questo modo il modello di linguaggio può fornire risposte più appropriate ai prompt, e basare tali risposte su dati attuali e veritieri.

3.2.1 Struttura del RAG

La RAG combina due fasi fondamentali per migliorare la precisione e l'efficacia delle risposte generate. La prima fase è quella del recupero delle informazioni (Retrieval), in cui il sistema ricerca contenuti rilevanti da una base di conoscenza esterna, come database,

archivi documentali o persino siti web. Questa ricerca avviene in tempo reale o quasi, permettendo di accedere a informazioni aggiornate o altamente specifiche che non sono state memorizzate durante l'addestramento del modello LLM. Successivamente, queste informazioni recuperate vengono utilizzate nella seconda fase, quella della generazione (Generation). Qui, il modello linguistico sfrutta il materiale recuperato per produrre una risposta coerente, contestualizzata e formulata in linguaggio naturale. Questo approccio è particolarmente utile perché non limita il modello linguistico alla conoscenza acquisita durante l'addestramento, ma lo rende un sistema dinamico, capace di "consultare" fonti esterne per integrare informazioni mancanti o aggiornate. Inoltre, riduce significativamente il fenomeno delle "allucinazioni", ossia quelle risposte erronee o inventate che i modelli generativi possono produrre quando si affidano esclusivamente ai loro parametri interni. L'utilità di RAG è evidente in una vasta gamma di applicazioni pratiche. In ambito medico, ad esempio, un sistema RAG può accedere a database scientifici o linee guida cliniche, offrendo risposte affidabili basate sulle ultime ricerche. Nel settore legale, può cercare normative o sentenze aggiornate per fornire analisi giuridiche dettagliate. Nel supporto tecnico, invece, consente di individuare rapidamente informazioni rilevanti all'interno di manuali o documenti interni per aiutare i clienti in modo efficace. La vera forza di RAG risiede nella sua capacità di rendere i modelli linguistici flessibili e personalizzabili. Le basi di conoscenza utilizzate per il recupero possono essere specifiche per un determinato settore o continuamente aggiornate per rispondere alle esigenze di contesti in rapida evoluzione. Ad esempio, un'azienda può caricare la propria documentazione interna o una raccolta di dati specializzati, garantendo che le risposte fornite siano non solo linguisticamente accurate, ma anche contestualmente pertinenti. In definitiva, RAG rappresenta una svolta cruciale nel superamento dei limiti degli LLM. Introducendo una componente di recupero delle informazioni, non solo potenzia l'accuratezza delle risposte, ma rende i modelli linguistici più dinamici, affidabili e adatti ad applicazioni reali. È un esempio di come la tecnologia possa evolversi per integrare capacità diverse, superando i confini di ciò che è possibile con i modelli tradizionali.

Capitolo 4

Modello

Il modello sviluppato combina tecniche di embedding, ricerca vettoriale e l'utilizzo di un Large Language Model (LLM) con l'obiettivo di poter interrogare una chatbot su specifici materiali industriali. L'utente può quindi porre domande riguardanti, ad esempio, la similarità tra materiali, le differenze tra due materiali, oppure richiedere un materiale simile ad un altro specificato o con determinate caratteristiche tecniche. La peculiarità di questo approccio risiede nella necessità di fornire al LLM informazioni contestuali specifiche per consentire risposte precise e pertinenti. Infatti, nessun modello linguistico, come GPT o Gemini, riesce a rispondere in maniera corretta ed accurata a domande che riguardano informazioni aziendali senza avere un contesto appropriato.
Per risolvere questa limitazione, il modello implementato integra la logica della Retrieval-Augmented Generation (RAG) con la generazione di linguaggio del LLM: una volta ricevuto l'input dell'utente si procede con il recupero di informazioni rilevanti tramite un processo di embedding e ricerca vettoriale (RAG), prima di passare il contesto estratto al modello linguistico insieme alla richiesta dell'utente. In questo modo, il LLM è in grado di generare risposte più accurate e contestualizzate, garantendo maggiore coerenza e precisione, soprattutto quando le domande in questione sono tecniche o riguardanti informazioni aziendali.

4.1 Assunzioni

I dati su cui è costruito il modello si trovano in una tabella che contiene le informazioni specifiche sui materiali. Ogni riga rappresenta un materiale univoco, identificato tramite un part number. La tabella è costruita secondo la seguente struttura:

- manufacturer : nome del manufacturer
- manufacturercode: codice relativo al manufacturer del materiale che possiamo ritrovare nel partnumber
- manufacturerpartnumber: identificativo univoco del materiale.
- taxonomyname: nome della tassonomia a cui appartiene il materiale.
- attributes : parametri tecnici specifici (es. resistenza alla trazione, densità, temperatura di esercizio, voltaggio ecc.).

Sia noto che la precedente tabella è solamente il punto di partenza, i dati presentati, prima di entrare nel flusso del modello, verranno manipolati come verrà poi illustrato nel prossimo paragrafo.
Prima di procedere con la spiegazione dettagliata del modello siano note le seguente assunzioni:

- I dati vengono ospitati in una collezione all'interno di un cluster a pagamento in MongoDB Atlas
- L'embedding usato è 'All-MPNet-base-v2', con dimensione 768
- L'indice con cui è svolta la ricerca vettoriale è quello offerto da MongoDB Atlas
- Il LLM utilizzato è il modello 'gpt-4o-mini' di GPT

4.2 Attributi principali su cui fare ricerca

Il processo inizia con la manipolazione dei dati, su cui poi viene effettuato l'embedding, necessario per la costruzione dell'indice vettoriale. Quest'ultimo è utilizzato nella ricerca

dei materiali simili, un processo basato sulle caratteristiche tecniche degli stessi. Tuttavia, le caratteristiche dei materiali, rappresentate nel campo 'attributes' come una lista di attributi, non hanno tutte la stessa importanza.
Se l'embedding è generato utilizzando indiscriminatamente tutti gli attributi e successivamente viene eseguita una ricerca di similitudini, possono verificarsi situazioni in cui il risultato non rispecchia la rilevanza effettiva. Ad esempio, un materiale e il suo vicino più simile potrebbero coincidere perfettamente in attributi meno rilevanti, differenziandosi però nell'attributo principale. Questo porterebbe a risultati di ricerca inaccurati.
Per ovviare a questo problema, è stata introdotta una tabella di ordinamento che assegna un livello di priorità agli attributi. Ogni riga della tabella corrisponde a un attributo, associato alla sua tassonomia di appartenenza e a un ordine di importanza. All'interno di ciascuna tassonomia, i dieci attributi principali vengono ordinati in base alla loro rilevanza, con valori che vanno da 1, corrispondente all'attributo di maggiore importanza, a 10, corrispondente all'attributo di minore importanza. Questo approccio consente di ponderare correttamente gli attributi durante il calcolo degli embedding, migliorando l'accuratezza dei risultati nella ricerca dei materiali simili.
Per prima cosa, per ogni tassonomia presente nella tabella di ordinamento degli attributi, vengono selezionati i dieci attributi principali e collocati in una nuova colonna della tabella dei materiali utilizzata poi nel processo, denominata 'Main attributes'. Questo metodo permette di isolare, per ciascun materiale, gli attributi più rilevanti in base all'ordine di importanza specificato per la tassonomia di appartenenza.
Tuttavia, non tutte le tassonomie sono presenti nella tabella di cui si fa uso per isolare gli attributi principali. Per queste tassonomie prive di ordine, non si è in grado quindi di selezionare i dieci attributi principali. Di conseguenza, nella colonna 'Main attributes' di questi materiali, saranno inclusi tutti gli attributi originali presenti nella colonna 'attributes'. Questo comportamento riflette la mancanza di una gerarchia esplicita per quelle specifiche tassonomie e si fa quindi riferimento all'elenco completo di attributi disponibili.

4.3 Embedding

Per consentire la ricerca di materiali simili basata sull'input dell'utente, il modello utilizza la tecnica degli embedding. Un embedding è un vettore di dimensione fissa, generato da un modello di machine learning, che cattura le relazioni e le caratteristiche intrinseche dei dati originali. Nel contesto del modello, ogni materiale, descritto dai suoi attributi tecnici, viene trasformato in un embedding che ne rappresenta la posizione all'interno di uno spazio vettoriale di dimensione 768, generato utilizzando il modello di embedding SBERT (Sentence-BERT).

Una volta costruiti, gli embedding consentono di effettuare la ricerca vettoriale, una tecnica che sfrutta metriche di distanza, come la distanza coseno o la distanza euclidea, per individuare vettori vicini nello spazio. La somiglianza tra materiali è quindi determinata dalla vicinanza della loro rappresentazione vettoriale nello spazio multidimensionale. Questo approccio permette di superare i limiti delle semplici corrispondenze testuali o delle regole predefinite, identificando correlazioni più profonde e semantiche tra i materiali, basate sui loro attributi tecnici.

Quando l'utente fornisce un input, che può essere la descrizione di un materiale, il nome di un materiale specifico o una lista di caratteristiche tecniche ecc., il modello trasforma anche questa richiesta in un embedding utilizzando, ovviamente, lo stesso modello con cui si sono costruiti gli embedding dei materiali. Questo embedding rappresenta il significato semantico dell'input e permette di confrontarlo con gli embedding dei materiali pre-elaborati.

La ricerca di similarità avviene quindi confrontando il vettore dell'input utente con i vettori già presenti nel database. Utilizzando metriche di distanza, che approfondiremo in seguito parlando nello specifico della ricerca vettoriale, il sistema identifica i materiali i cui vettori sono più vicini a quello dell'input. Questo approccio consente di trovare materiali simili, rispondendo a domande come: "Quali materiali sono più simili a questo?" o "Esiste un materiale con caratteristiche simili?". L'adozione della ricerca vettoriale è fondamentale per il modello, poiché consente di fornire risposte precise e contestualizzate nel chatbot, anche in presenza di un numero elevato di materiali e caratteristiche tecniche.

Grazie a questa fase preliminare di embedding dei dati, la ricerca risulta precisa ed efficiente. L'indice vettoriale offerto da MongoDB Atlas garantisce che il confronto tra vettori sia rapido anche in presenza di dataset di grandi dimensioni, abilitando un'esperienza utente reattiva e accurata.
Il modello di embedding utilizzato è il Masked and Permuted Pre-training for Language Understanding, meglio noto come MPNet. In questa sezione si analizza la sua origine e la sua costruzione.

4.3.1 Sentence embedding

Gli approcci tradizionali agli embedding si basavano spesso su metodi come bag-of-words (BoW) o n-gram, come Word2Vec e GloVe, metodi di riferimento agli albori dell'NLP e l'IA. Tuttavia, nonostante questi modelli riescano a catturare in maniera efficace il significato semantico delle singole parole, spesso non riescono a cogliere il significato semantico complessivo del testo e le sfumature contestuali.L'evoluzione nel campo del deep learning ha permesso di superare questa limitazione tramite il 'sentence embedding'.
Consideriamo ad esempio la parola "scala" in due frasi diverse : "Ho salito la scala per raggiungere il piano superiore." e "La scala musicale di questa canzone è in Do maggiore." Nel primo caso, "scala" si riferisce a uno strumento fisico per salire, mentre nel secondo caso si riferisce a una sequenza di note musicali. Al contrario, usando metodi come Word2Vec o GloVe, la parola "scala" avrebbe lo stesso embedding in entrambe le frasi, anche se utilizzata in due contesti diversi. Inoltre, questi modelli producono embedding a livello di singola parola,che può creare problemi quando si lavora con testi a livello di frase. Questo evidenzia come ci sia l'urgenza di un modello più accurato che permetta di evitare il verificarsi di queste imprecisioni.
Con i Sentence Transformers si è superato questo problema.I Sentence Transformers, come si evince dal nome, utilizzano l'architettura dei Transformers come base. L'architettura di un modello Transformer consiste in diversi blocchi encoder-decoder, come illustrato di seguito:

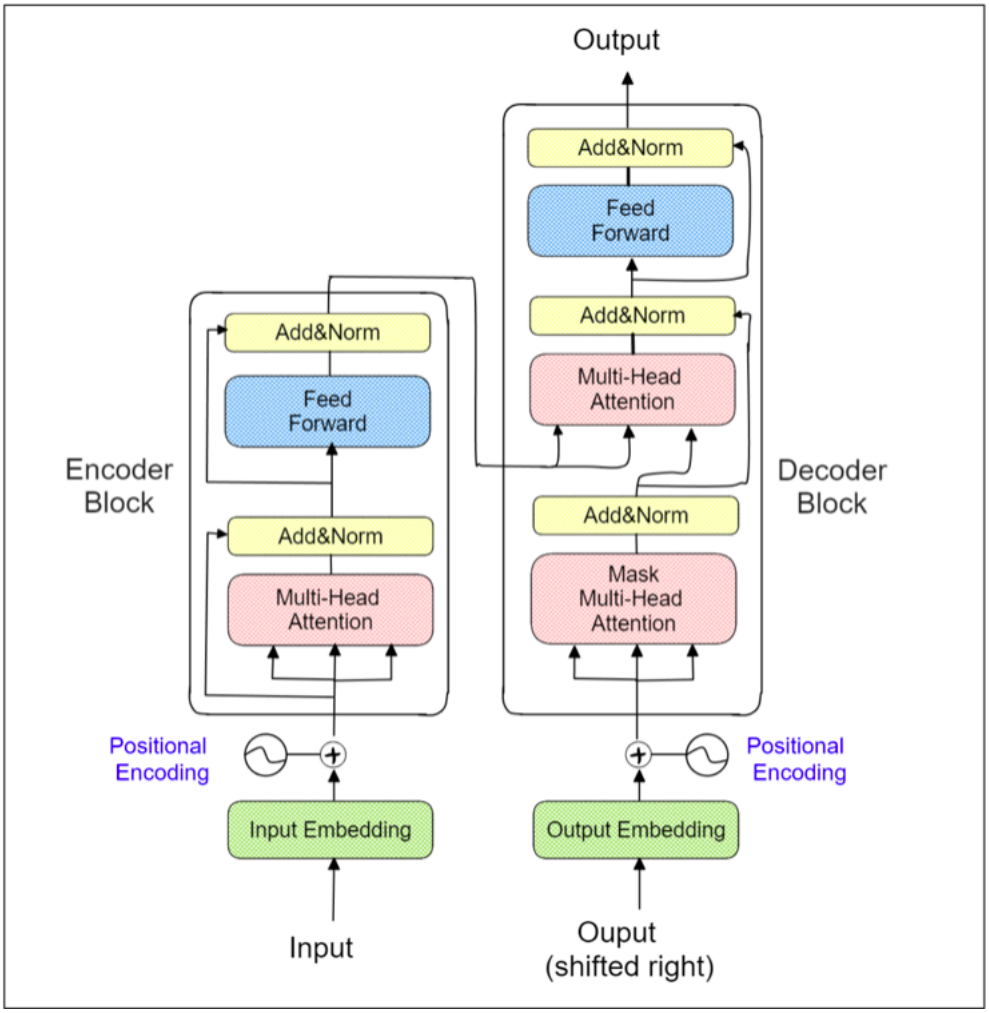

Figura 4.1. Architettura Transformers

Il livello di attenzione multi-head presente in ciascun blocco encoder del Transformer permette al modello di comprendere il contesto di ogni parola (o token) in relazione all'intera frase. Per questa ragione, i Sentence Transformers si basano esclusivamente sulla parte encoder dell'architettura dei Transformers. Grazie a questa struttura, il modello è in grado di generare embedding diversi per la parola "scala" nelle frasi "Ho salito la scala per raggiungere il piano superiore" e "La scala musicale di questa canzone è in Do maggiore" catturando il significato specifico che assume in ciascun contesto.
Tuttavia, l'output dei Transformers è ancora l'embedding di ciascuna parola (o token).

Per questo motivo, i Sentence Transformers aggiungono un ulteriore livello di pooling sopra l'ultimo blocco encoder del Transformer per ricavare l'embedding aggregato finale che rappresenta l'intero testo di input.

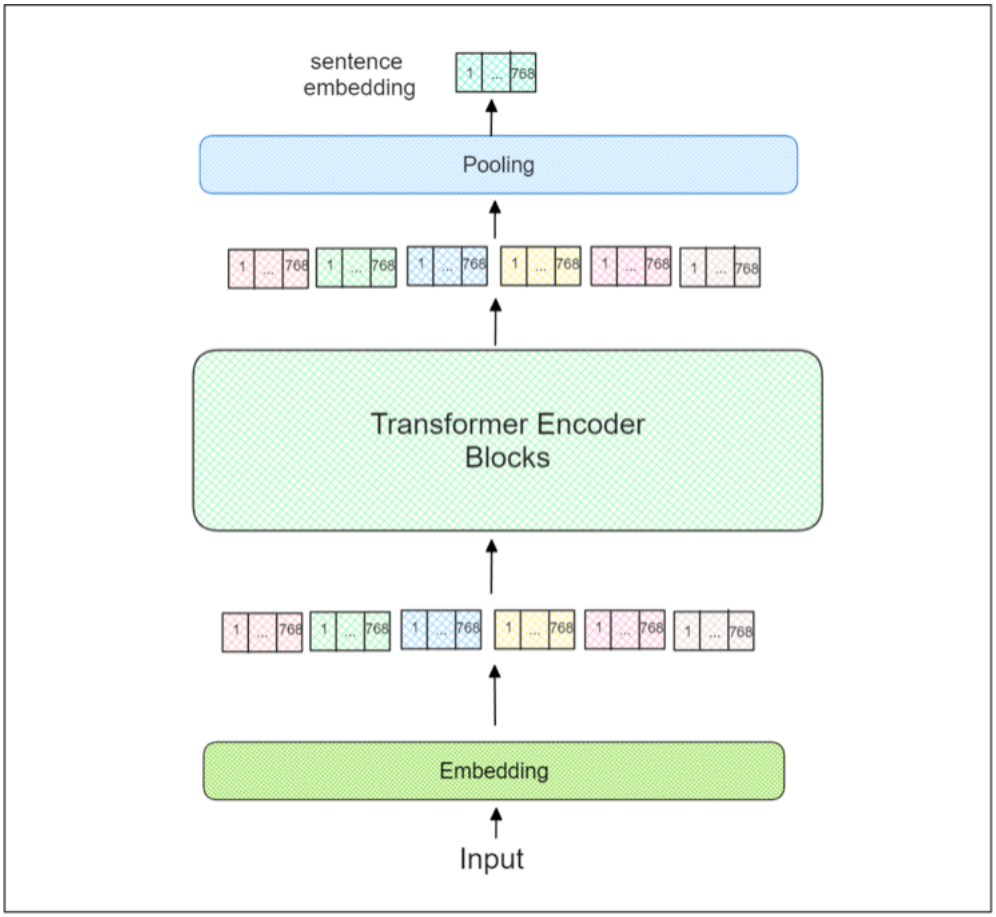

Figura 4.2. Architettura Sentence-Transformers

Tra le diverse varianti dei Sentence Transformers, MPNet è una delle più influenti. MPNet, infatti, sfrutta i punti di forza dei modelli BERT e XLNet, superando allo stesso tempo i loro principali svantaggi. Prima di procedere con il modello usato in questo progetto, analizziamo i modelli da cui deriva.

4.3.2 I predecessori di MPNet: BERT e XLNet

BERT è un modello Transfermer-based che performa allo stato dell'arte in molti task, come l'analisi del sentiment, il riconoscimento di entità, il question-answering e altro. Una delle tecniche chiave applicate durante il pre-training di BERT è il Masked Language Modeling (MLM). In questo approccio, una certa percentuale di token di input viene sostituita casualmente con un token [MASK]. L'obiettivo è quindi prevedere questo token [MASK] con il token più probabile dato il contesto dell'intera sequenza di input.

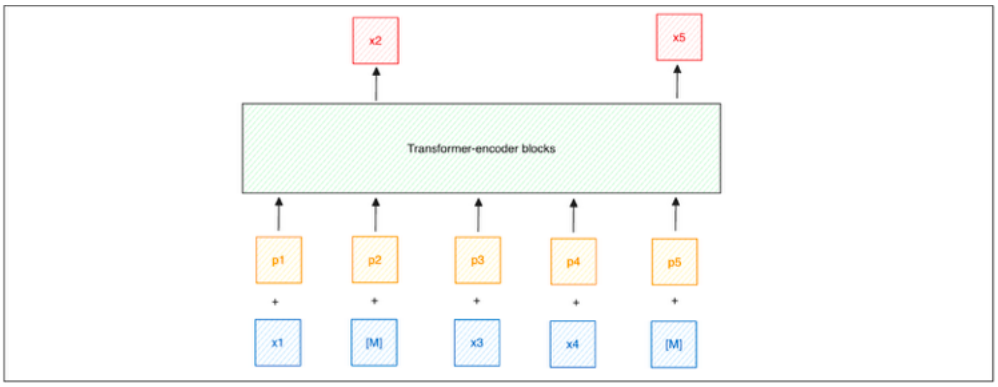

Figura 4.3. BERT pre-training.

Lo svantaggio principale di BERT è che i token [MASK] previsti sono indipendenti. Ad esempio, se abbiamo due token [MASK] in una sequenza, la previsione del primo token [MASK] non influenzerà la previsione del secondo token [MASK], anche se la previsione del primo token potrebbe essere utile per prevedere il secondo token. Questo può portare a una previsione della sequenza poco accurata.
XLNet,invece, è un modello Auto-Regressivo (AR) progettato specificamente per superare le limitazioni di BERT. Con l'approccio AR, il token da prevedere dipende sempre dai due token precedenti previsti. Tuttavia, anche l'approccio AR ha i suoi svantaggi. Infatti, quando prevede un token in una particolare posizione, considera solo i token precedenti. Questo non costituisce un problema se il token da prevedere si trova alla fine della sequenza. Tuttavia, se è posizionato all'inizio della sequenza, questo approccio può risultare limitante.

Per affrontare questa problematica, XLNet ha introdotto il Permuted Language Modeling (PLM), un metodo che permuta casualmente l'intera sequenza di input e quindi prevede il token alla fine della sequenza permutata.

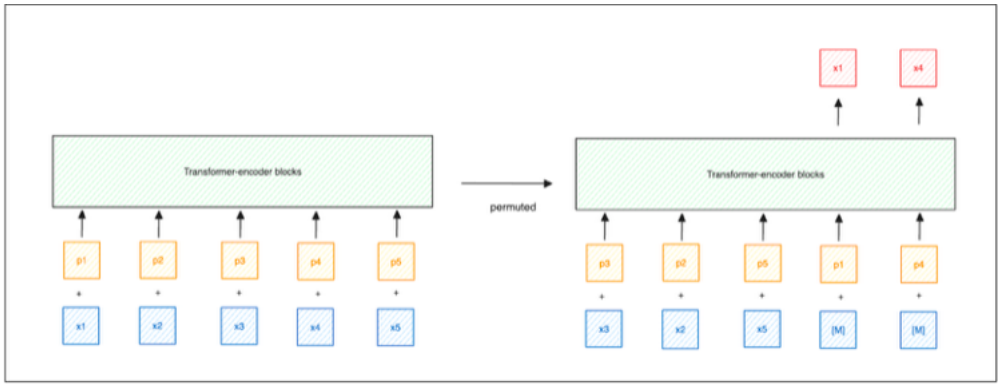

Figura 4.4. XLNet pre-training

Per esempio, data una sequenza di input di (x1, x2, x3, x4, x5), potrebbe essere permutato in (x3, x2, x5, x1, x4).Il compito di XLNet in questo caso è quello di prevedere il token x1 dato (x3, x2, x5) e x4 dato (x3, x2, x5, x1).

Tuttavia, l'introduzione del PLM non significa che XLNet sia un modello perfetto, in quanto è ancora un modello basato su encoder di trasformatore. L'essenza del blocco Transformer-encoder risiede nella sua natura bidirezionale, il che significa che il modello dovrebbe conoscere le informazioni di posizione di tutti i token nella sequenza di ingresso. Questo concetto non è pienamente realizzato nell'approccio AR utilizzato da XLNet, che considera solo le informazioni di posizione dei token precedenti.

Inoltre, il concetto di AR applicato in XLNet introduce una discordanza tra i processi di pre-formazione e di fine-tuning. Quando si utilizza XLNet per attività a valle come la classificazione del testo, le informazioni di posizione dell'intera sequenza di input sono disponibili fin dall'inizio. Pertanto, per sfruttare i vantaggi e la migliore qualità di BERT e XLNet evitando le loro limitazioni, è stato introdotto un modello unificato denominato MPNet.

4.3.3 MPNet

MPNet combina i punti di forza di MLM e PLM. Utilizza il modello di linguaggio permutato per modellare la dipendenza tra i token mascherati e, contemporaneamente, sfrutta le informazioni di posizione dell'intera sequenza. Questo approccio consente a MPNet di ottenere prestazioni superiori rispetto a modelli come BERT, XLNet e persino RoBERTa in numerosi task di inferenza linguistica naturale.
MPNet è diventato rapidamente un modello di riferimento per ottenere embedding a livello di frase, in particolare nella sua variante All-MPNet-Base-V2, che è stata addestrata su circa 1 miliardo di coppie di testi. La sua capacità di generare embedding contestuali e precisi lo rende ideale per task come il recupero di informazioni, la classificazione di testi e molto altro.
Per questi motivi, la variante All-MPNet-Base-V2 è stata scelta in questo progetto per ottenere l'embedding degli attributi dei materiali.
Il vantaggio del MLM è la sua capacità di considerare le informazioni posizionali dell'intera sequenza di input, mentre il vantaggio del PLM è la sua capacità di modellare la dipendenza tra i token mascherati attraverso il suo approccio autoregressive. MPNet combina i punti di forza del MLM e PLM nella sua architettura. In primo luogo, i token mascherati di entrambi MLM e PLM sono combinati; consideriamo una sequenza di input (x1, x2, x3, x4, x5), la sequenza viene quindi permutata in modo casuale, ottenendo ad esempio (x3, x2, x5, x1, x4). MPNet seleziona i token all'estremità destra della sequenza (ad esempio x1, x4) come token mascherati. I token non mascherati sono poi disposti come (x3, x2, x5, [M], [M]) con le loro corrispondenti informazioni di posizione (p3, p2, p5, p1, p4).
MPNet utilizza un meccanismo a due flussi, caratteristica tipica del PLM. Per esempio, quando si prevede il token x4, il modello può considerare (x3+p3, x2+p2, x5+p5) e il token precedentemente previsto (x1+p1). Questo approccio supera le limitazioni derivanti dal MLM.
MPNet incorpora anche il simbolo della maschera e le informazioni di posizione dei token mascherati ([M]+p1, [M]+p4) per garantire che ogni token possa vedere il contesto e le informazioni di posizione dell'intera sequenza di input. Per esempio, quando si prevede

token x1, il modello può considerare (x3+p3, x2+p2, x5+p5) e anche il token mascherato che lo segue via ([M]+p4). Questo approccio a sua volta, supera le limitazioni associate al PLM.

Per la sua struttura, MPNet ha ottenuto prestazioni migliori in varie attività di Natural Language Inference (NLI) rispetto ai modelli precedenti all'avanguardia come BERT, XLNet e RoBERTa.

4.3.4 Implementazione MPNet

In questo paragrafo descriviamo come è stato implementato il codice per la creazione degli embedding nel nostro caso. La colonna su cui è stato applicato il modello 'All-MPNet-base-v2' è 'concatened_attributes' dove sono stati appunto combinati i dati provenienti dalla colonna 'main attributes', la rispettiva 'taxonomyname' e 'manufacturerpartnumber': la ricerca infatti deve tenere in considerazione sia gli attributi tecnici ma anche la tassonomia di appartenenza e il part number identificativo.

I risultati ottenuti, ovvero i vettori di embedding, uno per ogni part number, sono stati inseriti in un nuovo campo della tabella 'embedding'.

L'implementazione dettagliata di seguito:

```
import pandas as pd
from sentence_transformers import SentenceTransformer
from concurrent.futures import ThreadPoolExecutor, as_completed
import logging
pim = (spark.sql('select taxonomyname, manufacturer,
    manufacturercode, manufacturerpartnumber, combined,
    combined_search from pim_embeddings_w2vec'))

df = pim.toPandas()
df['Main_attributes'] = df['Main_attributes'].fillna('None')
df['attr_main'] = df['attr_main'].fillna('None')

df['concatenated_attributes'] = df['combined_search'].apply(lambda
    x: "".join(x))
```

```
12
13 logging.basicConfig(level=logging.INFO)
14 logger = logging.getLogger()
15
16 table_name = 'pim_embeddings'
17
18 def process_batch(model, batch, batch_id):
19     try:
20         embeddings = model.encode(batch['concatenated_attributes'
               ].tolist(), batch_size=64, show_progress_bar=False)
21
22         batch.loc[:, 'embedding'] = embeddings.tolist()
23         return batch
24     except Exception as e:
25         logger.error(f"Errore nell'elaborazione del batch {
               batch_id}: {e}")
26         return None
27
28 model = SentenceTransformer('all-mpnet-base-v2')
29
30 batch_size = 1000
31 batches = [df[i:i + batch_size] for i in range(0, len(df),
       batch_size)]
32
33 table_exists = spark.catalog.tableExists(table_name)
34
35 with ThreadPoolExecutor(max_workers=10) as executor:
36
37     future_to_batch = {executor.submit(process_batch, model, batch
           , i): (batch, i) for i, batch in enumerate(batches)}
38
39     for future in as_completed(future_to_batch):
40         batch, batch_id = future_to_batch[future]
```

```
41          try:
42              processed_batch = future.result()
43              if processed_batch is not None:
44
45                  sdf_batch = spark.createDataFrame(processed_batch)
46
47
48                  if not table_exists:
49                      sdf_batch.write.mode('overwrite').saveAsTable(
                            table_name)
50                      table_exists = True
51                      logger.info(f"Tabella '{table_name}' creata e
                            batch {batch_id + 1} salvato")
52                  else:
53                     sdf_batch.write.mode('append').saveAsTable(
                          table_name)
54                      logger.info(f"Batch {batch_id + 1}/{len(
                            batches)} completato e salvato")
55              else:
56                  logger.error(f"Il batch {batch\_id} non    stato
                        processato correttamente.")
57          except Exception as exc:
58              logger.error(f"Errore nel batch {batch_id}: {exc}")
```

4.4 Indice di ricerca vettoriale

Una volta calcolati gli embedding, si procede alla creazione di un indice vettoriale basato sul database, il quale verrà successivamente utilizzato per effettuare ricerche di similarità tra l'input dell'utente e il database. L'output di questa ricerca sarà il contesto fornito, insieme alla richiesta dell'utente, al modello LLM.

4.4.1 Cos'è un indice vettoriale

Un indice vettoriale è una struttura che consente di eseguire ricerche basate sulla semantica dei dati, ossia sul loro significato sottostante. A differenza della ricerca tradizionale full-text, che si limita a trovare corrispondenze letterali tra testi, la ricerca vettoriale identifica i vettori che sono vicini alla query dell'utente in uno spazio multidimensionale. Quanto più i vettori sono vicini, tanto più i dati rappresentati da essi sono simili nel significato.
Questo approccio consente di interpretare il significato della query e dei dati, considerando l'intento del ricercatore e il contesto della ricerca per ottenere risultati più pertinenti. Ad esempio, se si cerca il termine "frutto rosso", una ricerca full-text restituirà solo dati che contengono esattamente queste parole. Al contrario, una ricerca semantica potrebbe restituire informazioni legate a frutti di colore rosso, come mele o fragole, che sono simili per significato anche se non menzionano esplicitamente il termine "frutto rosso".
L'indice vettoriale è quindi fondamentale per organizzare e interrogare i dati rappresentati in forma di vettori, rendendo possibile questo tipo di ricerca avanzata.

4.4.2 Indice vettoriale Atlas

Per abilitare una ricerca vettoriale efficiente, i dati sono stati collocati in un cluster dedicato su MongoDB Atlas. In questo cluster è stato creato un database denominato $'vector\_search'$, all'interno del quale si trova una collection chiamata $'pim\_embeddings'$, che funge da tabella per i nostri dati. MongoDB Atlas permette di costruire un indice vettoriale sulle proprie collection, all'interno del database: gli indici per la ricerca vettoriale sono separati dagli altri indici del database e servono per recuperare in modo efficiente i documenti che contengono embedding vettoriali al momento della query. Nel processo di creazione dell'indice per Atlas Vector Search, è necessario indicizzare i campi della collezione che contengono gli embedding, abilitando così la ricerca vettoriale su quei campi. Atlas Vector Search supporta embedding con una dimensione massima di 4096, nel nostro caso la dimensione degli embedding è di 768 come precedentemente indicato nelle assunzioni.
Un altro motivo per cui si è scelto l'indice di Atlas, è la possibilità di applicare filtri

preliminari ai dati indicizzando campi booleani, date, numeri, objectId, stringhe o UUID presenti nella collezione. Questo consente di restringere l'ambito della ricerca, assicurandosi che alcuni embedding vettoriali non vengano presi in considerazione per il confronto, riducendo i tempi di ricerca e risposta.

4.4.3 Costruzione dell'indice

La sintassi per la costruzione dell'indice è la seguente:

```
{
  "fields": [
    {
      "type": "vector",
      "path": "<field-to-index>",
      "numDimensions": <number-of-dimensions>,
      "similarity": "euclidean | cosine | dotProduct",
      "quantization": "none | scalar | binary"
    },
    {
      "type": "filter",
      "path": "<field-to-index>"
     },
    ...
  ]
}
```

Si analizzino ora nel dettaglio i campi richiesti durante la definizione di un indice vettoriale Atlas:

- fields: è un array di documenti, campi vettoriali e di filtro da indicizzare, uno per documento. Almeno un documento deve contenere la definizione del campo vettoriale. Si può anche opzionalmente indicizzare i campi boolean, date, number, objectId, string e UUID, uno per documento, per pre-filtrare i dati.

- fields.type: Tipo di campo da utilizzare per indicizzare i campi per la vectorSearch. È possibile specificare uno dei seguenti valori:
  - vector: per i campi che contengono incorporazioni vettoriali
  - filter: per i campi che contengono valori booleani, date, objectId, numerici, string o UUID.
- fields.path: Nome del campo da indicizzare. Per i campi nidificati, è necessario utilizzare la notazione dot per specificare il percorso dei campi incorporati. Non è possibile indicizzare i nomi dei campi con due punti consecutivi o i nomi dei campi che terminano con punti.
- fields.numDimensions: Numero di dimensioni vettoriali che la ricerca vettoriale Atlas applica a livello di indice e di query. È necessario specificare un valore inferiore o uguale a 4096. Per l'indicizzazione di BinData o vettori quantizzati, il valore deve essere uno dei seguenti:
  - 1 a 4096 per i vettori int8 per l'ingestione.
  - Multiplo di 8 per i vettori int1 per l'ingestione.
  - 1 a 4096 per i vettori binData (float32) e array (float32) per la quantizzazione scalare automatica.
  - Multiplo di 8 per i vettori binData(float32) e array(float32) per la quantizzazione binaria automatica.

    E' possibile impostare questo campo solo per i campi di tipo vettoriale
- fields.similarity: funzione di similarità vettoriale da utilizzare per cercare i vicini K-più vicini. E' possibile impostare questo campo solo per campi di tipo vettoriale. Il valore può essere uno dei seguenti:
  - euclidea: misura la distanza tra le estremità dei vettori. Questo valore consente di misurare la somiglianza in base a dimensioni diverse.
  - coseno: misura la somiglianza in base all'angolo tra i vettori. Questo valore permette di misurare la somiglianza che non è scalata dalla grandezza. Non

puoi usare vettori di magnitudine zero con coseno. Per misurare la somiglianza dei coseno, si consiglia di normalizzare i vettori e utilizzare dotProduct.

- dotProduct: misura la somiglianza come coseno, ma tiene conto della grandezza del vettore. Se si normalizza la magnitudine, cosine e dotProduct sono quasi identici nella misura della somiglianza.

- fields.quanitization: Tipo di quantizzazione vettoriale automatica per i vettori. È possibile specificare il tipo di quantizzazione da applicare ai vettori. Si consiglia di usare questa impostazione solo se gli embedding sono float o doppi vettori. Il valore può essere uno dei seguenti:

  - none: Indica che non esiste una quantizzazione automatica per l'incorporazione di vettori. Utilizzare questa impostazione se si dispone di vettori pre-quantizzati per l'ingestione. Se omesso, questo è il valore predefinito.
  - scalare: Indica quantizzazione scalare, che trasforma i valori in interi di 1 byte.
  - binario: Indica la quantizzazione binaria, che trasforma i valori in un singolo bit. Per utilizzare questa quantizzazione, numDimensions deve essere un multiplo di 8.

  Se la precisione è critica, è consigliato selezionare none o scalare invece di binario.

Si possono ancheindicizzare i campi booleani, data, numero, objectId, string e UUID per pre-filtrare i dati. Filtrare i dati è utile per restringere l'ambito della ricerca semantica e garantire che non tutti i vettori siano considerati per il confronto. Con questa opzione si riduce infatti il numero di documenti per i quali eseguire confronti di similarità, che possono diminuire la latenza delle query e aumentare l'accuratezza dei risultati di ricerca. È necessario indicizzare i campi che si desidera filtrare utilizzando il tipo di filtro all'interno della matrice dei campi.

La sintassi seguente definisce il tipo di campo *filtro*:

```
{
  "fields":[
    {
      "type": "vector",
      ...
    },
    {
      "type": "filter",
      "path": "<field-to-index>"
    },
    ...
  ]
}
```

Nel campo *path* corrispondente a *"type":"filter"* si inserisce il campo della collection che si vuole indicizzare.

L'indice usato nella nostra collection è stato così costruito:

```
{
  "fields": [
    {
      "type": "vector",
      "path": "embedding",
      "numDimensions": 768,
      "similarity": "euclidean"
    },
    {
      "type": "filter",
      "path": "taxonomyname"
```

```
      },
     {
       "type": "filter",
       "path": "manufacturerpartnumber"
      }
  ]
}
```

Come illustrato nel codice, l'indice vettoriale è stato costruito sul campo *embedding*, che ospita i vettori di embedding calcolati in precedenza utilizzando il modello *All-MPNet-base-v2* con una dimensione di 768. Su questo campo si basa la ricerca di similarità, che nel nostro caso utilizza la distanza euclidea. Inoltre, è stato implementato un pre-filtraggio dei dati sui campi *taxonomyname* e *manufacturerpartnumber*, corrispondenti rispettivamente alla tassonomia di appartenenza e al part number identificativo, per ottimizzare e focalizzare i risultati della ricerca. Questo approccio consente di restringere la ricerca di vettori simili esclusivamente al sottoinsieme di dati che corrisponde alle tassonomie e/o ai part number specificati nella query dell'utente, migliorando l'efficienza e la pertinenza dei risultati.

4.5 LLM: GPT-4o-mini

Il Large Language Model scelto nel nostro progetto è *GPT-4o-mini* di openAI.Il modello in questione, è stato lanciato da openai il 18 Luglio 2024 ed è stato progettato per bilanciare costi e prestazioni; infatti *Gpt-4o-mini* è il risultato di un processo di distillazione del più grande, GPT-4o. Questo processo consiste nell'addestrare un modello più compatto per replicare il comportamento e le prestazioni del modello originale, mantenendo un alto livello di capacità in una versione più leggera ed economica. Analizziamo le sue caratteristiche principali:

- Ampia finestra di contesto: GPT-4o Mini conserva la finestra di contesto di 128.000

token del GPT-4o, che consente di elaborare testi lunghi in modo efficiente. Questa caratteristica è particolarmente utile per applicazioni che richiedono la gestione di grandi quantità di informazioni, come l'analisi di documenti complessi o la conservazione della cronologia di lunghe conversazioni.

- Capacità multimodali: il modello è in grado di elaborare sia input testuali sia input visivi, e in futuro si prevede il supporto anche per input e output video e audio. Questa versatilità lo rende ideale per applicazioni che spaziano dall'analisi del testo al riconoscimento delle immagini e oltre.

- Costo accessibile: uno dei maggiori punti di forza del GPT-4o Mini è il suo prezzo competitivo. Elaborare un milione di token in input costa $0,15, mentre per l'output il costo è di $0,60 per milione di token. Questo lo rende significativamente più economico rispetto al GPT-4o, che ha un costo di $5,00 per milione di token in input e $15,00 per output. Rispetto al GPT-3.5 Turbo, il GPT-4o Mini è oltre il 60% più conveniente, offrendo così un eccellente rapporto qualità-prezzo.

- Sicurezza avanzata:GPT-4o Mini integra le stesse funzionalità di sicurezza del modello GPT-4o. È il primo modello nella sua API a implementare un sistema gerarchico di istruzioni, il che lo rende più resistente a tentativi di manipolazione come jailbreak, injection di prompt o estrazione del prompt di sistema. Questa robustezza ne migliora la sicurezza, rendendolo adatto a un'ampia gamma di utilizzi, anche in contesti sensibili.

GPT-4o Mini rappresenta un'importante evoluzione nell'ambito dei modelli linguistici, combinando alte prestazioni, flessibilità e convenienza in un unico modello.

4.6 Workflow

In questa sezione, viene illustrato il workflow completo del progetto. Per prima cosa, introduciamo la funzione principale, che rappresenta il cuore del flusso operativo e coordina

l'intero processo. Successivamente, verranno approfondite nel dettaglio tutte le funzioni richiamate dalla funzione principale, spiegando il loro ruolo specifico, il loro funzionamento, e come contribuiscono al workflow complessivo.

4.6.1 La funzione principale: handle_user_input

La funzione principale, *handle_user_input*, è la seguente:

```
1  def handle_user_input(user_input):
2      _DEBUG and print("user_input:", user_input)
3
4      taxonomies = get_taxonomies()
5      found_taxonomy = _extract_category_from_input(user_input,
           taxonomies)
6      if found_taxonomy:
7          identified_taxonomy = [found_taxonomy] if isinstance(
               found_taxonomy, str) else list(found_taxonomy)
8      else:
9          identified_taxonomy = None
10     response_str = _identify_prompt(user_input)
11     choice, part_numbers = _parse_response(response_str)
12     _DEBUG and print("choice+part_numbers:", choice, part_numbers)
13
14     model = get_torch_model()
15     query_vector = model.encode(user_input)
16     _DEBUG and print("vector:", query_vector[0:10], "...", f"({len
           ()} items)")
17     query_vector = [float(x) for x in query_vector]
18
19     similar_part_numbers, similar_combined =
           _find_similar_products(
20         query_vector,
21         identified_taxonomy,
```

```
22          part_numbers,
23      )
24      _DEBUG and print("similar_part_numbers+similar_combined:",
            similar_part_numbers, similar_combined)
25
26      if choice == 0:
27          prompt = _prompt_formatter(
28              user_query=user_input,
29              similar_part_numbers=similar_part_numbers,
30              similar_combined=similar_combined,
31          )
32      else:
33          prompt = _prompt_formatter_pn(
34              pn_queries=part_numbers,
35              user_query=user_input,
36              similar_part_numbers=similar_part_numbers,
37              similar_combined=similar_combined,
38          )
39      _DEBUG and print("prompt:", prompt)
40
41      response = _send_chat_request(prompt)
42      _DEBUG and print("response:", prompt)
43      print(response)
44      return response
```

La funzione *handle_user_input* è progettata per gestire l'input fornito dall'utente al chatbot. Questa funzione accetta come unico parametro la richiesta dell'utente. Una volta chiamata, la prima operazione eseguita dalla funzione consiste nell'ottenere la lista completa di tutte le tassonomie disponibili tramite la funzione *get_taxonomies*, il cui risultato viene memorizzato nella variabile *taxonomies*.

Successivamente, questa lista viene passata come parametro, insieme all'input dell'utente,

alla funzione *_extract_category_from_input*. Il compito di questa funzione, come suggerisce il nome stesso, è quello di analizzare la richiesta dell'utente e verificare se contiene una o più tassonomie all'interno della lista. Nel caso in cui la funzione identifichi delle tassonomie nella richiesta, queste vengono estratte e memorizzate nella variabile *found_taxonomy*. Questa variabile rappresenta quindi un sottoinsieme di tassonomie rilevanti, individuate nell'input fornito dall'utente. Questa estrazione costituisce un passaggio fondamentale, poiché consente al sistema di comprendere meglio il campo di interesse dell'utente e di indirizzare il flusso operativo verso la risposta più appropriata.

Per agevolare il flusso successivo della funzione, la variabile contenente le tassonomie estratte necessita essere una lista. Per questo motivo, nelle righe 6-8 viene verificato se la variabile in questione abbia un valore considerato "truthy" in Python, ad esempio, una stringa non vuota, una lista non vuota o qualsiasi oggetto diverso da None, False o un valore vuoto. Se la variabile *found_taxonomy* non contiene alcun valore, il codice passa direttamente al blocco *else*, dove la variabile *identified_taxonomy* viene impostata su *None*. Al contrario, se *found_taxonomy* contiene una o più tassonomie, viene eseguito il blocco principale. In particolare, se *found_taxonomy* è una stringa, significa che è stata identificata una sola tassonomia. In questo caso, il codice la trasforma in una lista con un unico elemento, così da garantire un formato uniforme. Se invece *found_taxonomy* non è una stringa, ma ad esempio una lista o un insieme, il codice la converte direttamente in una lista utilizzando la funzione *list(found_taxonomy)*. Alla fine di questo blocco, la variabile *identified_taxonomy* sarà una lista delle tassonomie estratte, se presenti, dall'input dell'utente; altrimenti sarà uguale a *None*.

Dopo questo blocco ipotetico, viene chiamata la funzione *_idenitify_prompt*, che prende come parametro l'input dello user (*user_input*, e tramite un API di OpenAI chiede a Chat-GPT di analizzare la richiesta dell'utente e inviare come risposta '0' se l'utente sta chiedendo di fare un confronto tra due o più materiali, altrimenti '1'; inoltre, se presenti, invia come risposta la lista dei part numbers trovati nello *user_input*. Questo passaggio è fondamentale, perché permette di classificare la natura della richiesta dell'utente e di adattare il prompt che verrà successivamente inviato a Chat-GPT.In sintesi, in base alla classificazione effettuata da *_identify_prompt*, il processo costruisce un prompt

personalizzato che risponde in modo più preciso e coerente alle esigenze dell'utente. Ad esempio, se l'utente ha richiesto un confronto tra materiali, il prompt sarà progettato per fornire una risposta specifica e dettagliata sui materiali menzionati. Al contrario, se la richiesta riguarda un altro tipo di analisi o domanda, il prompt sarà strutturato in maniera diversa per rispondere in modo appropriato al contesto. Questo concetto sarà approfondito successivamente nella descrizione dettagliata di ogni funzione. La risposta risultante dalla funzione *identify_prompt* è memorizzata nella variabile *response_str*. Successivamente, quest'ultima viene trasformata in un formato adeguato dalla funzione *_parse_response*, che restituisce nella variabile *choice* il numero identifiato, 0 o 1, e nella variabile *part_numbers* la lista dei part numbers, se presenti.

Passiamo ora alla funzione *get_torch_model*, che si occupa di gestire il caricamento e l'eventuale aggiornamento del modello di embedding *All-MPNet-base-v2*, memorizzato in remoto. Questa funzione verifica se il modello è già aggiornato e disponibile localmente. In tal caso, il modello viene caricato e assegnato alla variabile *model*. Se invece il modello non è presente o non è aggiornato, la funzione provvede a scaricarlo o aggiornarlo, e successivamente lo assegna alla stessa variabile. Si noti che il modello utilizzato per calcolare l'embedding della richiesta dell'utente (*user_input*)deve essere lo stesso con cui sono stati calcolati in precedenza gli embedding dei dati memorizzati in MongoDB Atlas, per garantire una ricerca di similarità coerente ed efficace.

Nella riga successiva, viene quindi calcolato l'embedding della richiesta dell'utente tramite la funzione *model.encode(user_input)* e viene poi memorizzato nella variabile *query_vector*. Ogni elemento di quest'ultima è poi trasformato in formato *float* per garantire che il formato dei dati sia consistente e adatto ai calcoli numerici successivi, come il calcolo di similitudini. Può infatti verificarsi che il risultato della funzione *model.encode()* sia un vettore contente numeri in formato diverso, come *Decimal* o *int*.

Una volta ottenuta la rappresentazione vettoriale della richiesta dell'utente, questa viene passata come parametro alla funzione *_find_similar_products* insieme alla lista di tassonomie identificate, *identified_taxonomy*, e la lista di part numbers trovati, *part_numbers*. Questa funzione si occupa di confrontare la query dell'utente con i dati a nostra disposizione e, tramite l'indice costruito in MongoDB Atlas, effettua una ricerca vettoriale di

similarità, basandosi anche sulle tassonomie e i part numbers inseriti dall'utente nella richiesta. Per ogni risultato ottenuto da questa ricerca di similarità, la funzione estrae due elementi principali:

- Il part number corrispondente al risultato trovato.
- Gli attributi associati al part number, che rappresentano le caratteristiche chiave del prodotto o dei dati rilevanti.

Questi due elementi vengono memorizzati rispettivamente nelle variabili *similar_part_numbers* e *similar_combined*. La variabile *similar_part_numbers* contiene quindi l'elenco dei part numbers simili trovati, mentre la variabile *similar_combined* raccoglie gli attributi corrispondenti a ciascun part number. Questa fase è cruciale, poiché i risultati trovati rappresentano i dati più simili alla richiesta dell'utente e vengono quindi utilizzati come contesto aggiuntivo nella successiva interazione con Chat-GPT. Infatti, queste informazioni servono a fornire a Chat-GPT una base di conoscenza specifica e pertinente alla query, migliorando così la qualità e la precisione della risposta che verrà generata.
Ora che abbiamo a disposizione i part numbers trovati dalla ricerca vettoriale, possiamo passare alla costruzione del prompt da inviare a Chat-GPT. Se la variabile *choice*, definita in precedenza, ha valore '0', la funzione utilizzata per costruire il prompt sarà *_prompt_formatter*. In caso contrario, se *choice* ha valore '1', verrà utilizzata la funzione *_prompt_formatter_pn*.
Queste due funzioni sono necessarie per creare il prompt finale da inviare a Chat-GPT, ma con una differenza: in entrambe le funzioni viene combinata una base di prompt personalizzata, che dipende dal valore di *choice*, concatenata a due elementi fondamentali:

- La richiesta dell'utente (*user_input*), che è il testo originale con cui l'utente ha interagito con il chatbot.
- Il contesto trovato, che include i part numbers simili e gli attributi associati, estratti dalla ricerca vettoriale precedente.

Queste funzioni creano quindi un prompt completo e mirato, che unisce una struttura di base personalizzata, basata sulla richiesta dell'utente, la query specifica dell'utente e le

informazioni rilevanti sui prodotti simili. Questo approccio permette a Chat-GPT di avere tutte le informazioni necessarie per generare una risposta accurata e contestualizzata.
A prescindere dalla funzione chiamata per la generazione del prompt, quest'ultimo viene in ogni caso memorizzato nella variabile *prompt*. Quest'ultima è l'unico parametro della funzione *send_chat_request*, che si occupa di mandare il prompt finale a Chat-GPT e restituire la risposta nella variabile *response*. Il contenuto di *response* è la risposta che riceverà l'utente nel chatbot. Nelle successive sezioni analizziamo nel dettaglio ogni funzione coinvolta nel processo e l'inizializzazione delle variabili d'ambiente necessarie.

4.6.2 Definizione delle variabili di ambiente e connessione ai servizi

Il codice che segue precede la definizione delle funzioni coinvolte nel workflow ed è un'implementazione che gestisce vari aspetti fondamentali per l'interazione con diverse risorse e servizi esterni, come il database in MongoDB, modello di embedding e servizi di archiviazione.

```
1  from pickle import loads as picke_loads
2  from zlib import decompress as zlib_decompress
3  from myda_common import Database, Auth, SPEEDBOAT
4  from decouple import config
5  from pymongo import MongoClient
6  from json import loads
7  import openai
8  from sentence_transformers import SentenceTransformer
9  from os.path import isfile, isdir
10 from azure.storage.blob import BlobServiceClient
11 from zipfile import ZipFile
12 from datetime import datetime
13 import certifi
14 openai.Client._DEFAULT_CA_BUNDLE_PATH = certifi.where()
15
16
```

```
17 CHAT_MONGO_CONN_STR = config("CHAT_MONGO_CONN_STR")
18 CHAT_OPENAI_KEY = config("CHAT_OPENAI_KEY")
19 CHAT_BLOB_STORAGE_CONN_STR = config("CHAT_BLOB_STORAGE_CONN_STR")
20
21
22 client = MongoClient(CHAT_MONGO_CONN_STR)
23 openai.api_key = CHAT_OPENAI_KEY
24
25 _LOCAL_TAXONOMIES_VERSION_FILENAME,
      _REMOTE_TAXONOMIES_VERSION_FILENAME = "taxonomies-version.bin"
      , "nlp::taxonomies.version"
26 _LOCAL_TAXONOMIES_DATA_FILENAME, _REMOTE_TAXONOMIES_DATA_FILENAME
      = "taxonomies-model.bin", "nlp::taxonomies.model"
27 _LOCAL_MODEL_VERSION_FILENAME, _REMOTE_MODEL_VERSION_FILENAME = "
      model-version.bin", "nlp::language-torch-model.version"
28 _LOCAL_MODEL_DATA_FILENAME, _LOCAL_MODEL_DATA_FOLDER,
      _REMOTE_MODEL_DATA_FILENAME = "model-torch.bin", "lp-model-
      torch-%s", "language-torch-model.data"
29 _DEBUG = True
30
31
32 class Health:
33     class taxonomies:
34         queried = 0
35         version = None
36         downloaded = False
37     class torch_model:
38         queried = 0
39         version = None
40         downloaded = False
```

Di seguito troviamo una descrizione dettagliata di ogni sezione del codice, evidenziando la logica e le funzionalità.

Le librerie necessarie per l'implementazione del processo sono:

- pickle e zlib: sono utilizzate rispettivamente per la serializzazione e la decompressione dei dati. La funzione *pickle.loads*, rinominata *picke_loads* permette di deserializzare oggetti precedentemente serializzati,al contrario *zlib.decompress*, rinominata *zlib_decompress* è utilizzata per decomprimere i dati compressi con il formato zlib.

- myda_common: questa libreria è di pertinenza aziendale e viene importata per utilizzare alcune classi comuni come 'Database', 'Auth' e 'SPEEDBOAT'. Questi componenti sono utilizzati per la gestione del database, dell'autenticazione e della configurazione della speedboat, una divisione interna dell'azienda.

- decouple: è impiegata per gestire in modo sicuro le variabili di configurazione, separando i dati sensibili dal codice. La funzione *config* viene utilizzata per caricare variabili di ambiente da un file di configurazione esterno, un file '.env', evitando di includere valori sensibili all'interno del codice.

- pymongo: libreria fondamentale per la connessione al database MongoDB; in particolare viene utilizzato *MongoClient* per stabilire la connessione al database.

- openai: questo modulo è importato per poter utilizzare l'API di OpenAI, consentendo l'accesso ai modelli linguistici come GPT. Questo modulo permette quindi di inviare richieste all'API di OpenAI e di ottenere risposte generate dai modelli di intelligenza artificiale, nel nostro caso *gpt-4o-mini*.

- sentence_transformers: utilizzata per creare vettori di embedding dall'input dell'utente, utilizzando il modello scelto *All-MPNet-base-v2*.

- os.path: le funzioni *isfile* e *isdir* di questa libreria vengono utilizzate per verificare se un determinato percorso rappresenta un file o una directory. Nel nostro caso sarà utile per controllare la presenza di determinati file o cartelle nel sistema locale.

- azure.storage.blob: impiegata per interagire con il servizio di archiviazione Blob di Azure, usato nel nostro caso per ospitare la lista doi tassonomie e il modello di embedding, consentendo di caricare e scaricare file dal cloud.

- zipfile: offre strumenti per lavorare con file compressi in formato ZIP. Viene quindi utilizzata per estrarre o comprimere file ZIP.
- datetime: consente di lavorare con date e orari, utile per la registrazione dei timestamp o per la gestione di scadenze.(RIVEDI)
- certifi: questa libreria fornisce un set di certificati CA radice aggiornati per garantire la sicurezza nelle comunicazioni HTTPS. Questa libreria è utilizzata per configurare OpenAI in modo che utilizzi i certificati sicuri forniti da 'certifi'.

Una volta importate le librerie necessarie, il codice procede con la configurazione delle variabili di connessione:

- CHAT_MONGO_CONN_STR: viene configurata la stringa di connessione per il database MongoDB, utilizzando una variabile di ambiente letta tramite la funzione *config*. Questa variabile permette di connettersi al database MongoDB dove sono memorizzati i dati del sistema.
- CHAT_OPENAI_KEY: la chiave API di OpenAI viene caricata tramite la funzione *config* e impostata nella libreria 'openai' per autorizzare l'accesso all'API di OpenAI.
- CHAT_BLOB_STORAGE_CONN_STR: viene configurata la stringa di connessione per il servizio di archiviazione blob di Azure, consentendo l'interazione con il cloud per le operazioni di caricamento e download di file necessarie al processo.

Si procede poi stabilendo le connessioni con i vari servizi esterni: MongoDB, OpenAI e Blob di Azure.
Nel codice sono inoltre definiti i nomi dei vari file locali e remoti relativi a risorse come modelli di embedding e tassonomie; in particolare le variabili _LOCAL_TAXONOMIES_VERSION_FILENAME e _REMOTE_TAXONOMIES_VERSION_FILENAME si riferiscono ai file che contengono la versione delle tassonomie, rispettivamente quella presente localmente e quella remota, mentre _LOCAL_TAXONOMIES_DATA_FILENAME e _REMOTE_TAXONOMIES_DATA_FILENAME contengono i dati stessi.
Analogamente, vengono definiti i file per il modello di embedding utilizzato, con variabili

per la versione del modello e il modello stesso, locale e remoto.
Per tenere traccia dello stato delle risorse, è stata definita una classe *Health*, che include informazioni sulle risorse chiave: la calsse *taxonomies* tiene traccia delle tassonomie, monitorando quante volte sono state interrogate, la versione delle tassonomie e se i dati sono stati scaricati correttamente, mentre *torch_model* monitora lo stato del modello di embedding, incluse informazioni sulla versione e se il modello è stato scaricato.

4.6.3 funzione get_taxonomies

La funzione *get_taxonomies* si occupa di gestire il caricamento del file contenente la lista delle tassonomie da una fonte esterna, un database Redis, dove questo file è stato precedentemte caricato, e di controllare se i dati relativi alle tassonomie sono già presenti localmente. Se i dati non sono disponibili o non sono aggiornati, la funzione si occupa anche di scaricarli e decomprimerli. Nello specifico, la funzione inizia incrementando un contatore, *Health.taxonomies.queried*, che tiene traccia di quante volte è stata effettuata la richiesta per ottenere la lista di tassonomie, utile per monitorare l'uso delle tassonomie nell'applicazione. Dopodichè la funzione accede al database Redis per ottenere informazioni relative alla versione e ai dati delle tassonomie, utilizzando una connessione autenticata tramite *Auth.tokenDataAs(speedboat=SPEEDBOAT.SILICA)* per recuperare il token di accesso al database. Successivamente cerca la versione delle tassonomie memorizzata nel database Redis e se la versione non viene trovata, la funzione solleva un'eccezione per segnalare un errore. Si procede poi verificando se esiste i file locali contenenti la lista di tassonomi e la versione delle tassonomie. Se i file non esistono o se la versione locale è diversa da quella remota, significa che i dati locali non sono aggiornati e quindi è necessario scaricarli di nuovo. In questo caso, introduciamo la variabile *Health.taxonomies.downloaded* che tiene traccia dello stato di aggiornamento delle tassonomie. Se il file contenente le tassonomie non è presente o non è aggiornati, la funzione recupera il file delle tassonomie dal database Redis, se il file non è presente, viene sollevata un'eccezione. Il file viene poi decompresso utilizzando la funzione *zlib_decompress*. Una volta decompresso il file, vengono salvati nei file locali sia i dati delle tassonomie che la versione corrispondente. Nel caso in cui i dati delle tassonomie siano già scaricati e aggiornati, la funzione li carica direttamente dal file

locale, senza la necessità di scaricarli nuovamente. Infine, la funzione restituisce i dati delle tassonomie, che sono deserializzati tramite la funzione *picke_loads(file)*, trasformando il contenuto del file in un formato utilizzabile dal codice.

```
1    def get_taxonomies():
2    Health.taxonomies.queried += 1
3    r = Database.REDIS.getDb(token=Auth.tokenDataAs(speedboat=
        SPEEDBOAT.SILICA))
4
5    remote_version: bytes = r.get(
        _REMOTE_TAXONOMIES_VERSION_FILENAME)
6    if not remote_version:
7        raise Exception(f"Version file <{
            _REMOTE_TAXONOMIES_VERSION_FILENAME}> not found in
            redis")
8    Health.taxonomies.version = remote_version.decode("ascii")
9
10   if not isfile(_LOCAL_TAXONOMIES_DATA_FILENAME) or not isfile(
        _LOCAL_TAXONOMIES_VERSION_FILENAME):
11       Health.taxonomies.downloaded = False
12   else:
13       with open(_LOCAL_TAXONOMIES_VERSION_FILENAME, "rb") as f:
14           local_version = f.read()
15       if local_version != remote_version:
16           Health.taxonomies.downloaded = False
17
18   if not Health.taxonomies.downloaded:
19       file = r.get(_REMOTE_TAXONOMIES_DATA_FILENAME)
20       if not file:
21           raise Exception(f"File <{
                _REMOTE_TAXONOMIES_VERSION_FILENAME}> not found in
                redis")
22       file: bytes = zlib_decompress(file)
```

```
23          with open(_LOCAL_TAXONOMIES_DATA_FILENAME, "wb") as f:
24              f.write(file)
25          with open(_LOCAL_TAXONOMIES_VERSION_FILENAME, "wb") as f:
26              f.write(remote_version)
27          Health.taxonomies.downloaded = True
28
29      else:
30          with open(_LOCAL_TAXONOMIES_DATA_FILENAME, "rb") as f:
31              file: bytes = f.read()
32
33      return picke_loads(file)
```

4.6.4 funzione get_torch_model

La funzione *get_torch_model* è analoga a *get_taxonomies*, con la differenza che ha come scopo il caricamento del file contenente il modello usato per calcolare l'embedding dello *user_input* e la sua versione. La struttura è identica a quella della funzione precedente, differenziandosi solamente nel tipo di documento caricato: nel primo caso la lista di tassonomie, nel secondo il modello di embedding *All-MPNet-base-v2*.La funzione *get_torch_model* si occupa di gestire il caricamento del file contenente il modello di embedding da una fonte esterna, un database Redis, dove questo file è stato precedentemte caricato in formato blob, e di controllare se il modello è già presente localmente. Se i dati non sono disponibili o non sono aggiornati, la funzione si occupa anche di scaricarli e decomprimerli. Nello specifico, la funzione inizia incrementando un contatore, *Health.torch_model.queried*, che tiene traccia di quante volte è stata effettuata la richiesta per ottenere il modello di embedding. Dopodichè la funzione accede al database Redis per ottenere informazioni relative alla versione e ai dati del modello, utilizzando una connessione autenticata tramite *Auth.tokenDataAs(speedboat=SPEEDBOAT.SILICA)* per recuperare il token di accesso al database. Successivamente cerca la versione del modello memorizzata nel database Redis e se la versione non viene trovata, la funzione solleva un'eccezione per segnalare un errore. Si procede poi verificando se esistono i file locali contenenti il modello di embedding e

la sua versione. Se i file non esistono o se la versione locale è diversa da quella remota, significa che i dati locali non sono aggiornati e quindi è necessario scaricarli di nuovo. In questo caso, introduciamo la variabile *Health.torch_model.downloaded* che tiene traccia dello stato di aggiornamento del modello. Se il file contenente il modello non è presente o non è aggiornatio, la funzione recupera il file dal database Redis; se il file non è presente, viene sollevata un'eccezione. Il file viene poi decompresso utilizzando la funzione *zlib_decompress*. Una volta decompresso, vengono salvati nei file locali sia il modello di embedding che la versione corrispondente. Nel caso in cui i dati del modello siano già scaricati e aggiornati, la funzione li carica direttamente dal file locale, senza la necessità di scaricarli nuovamente. Infine, la funzione restituisce il modello, che è deserializzato tramite la funzione *picke_loads(file)*, trasformando il contenuto del file in un formato utilizzabile dal codice.

```
1     def get_torch_model() -> SentenceTransformer:
2     Health.torch_model.queried += 1
3     r = Database.REDIS.getDb(token=Auth.tokenDataAs(SPEEDBOAT.
          SILICA))
4
5     remote_version: bytes = r.get(_REMOTE_MODEL_VERSION_FILENAME)
6     if not remote_version:
7         raise Exception(f"Version file <{
              _REMOTE_MODEL_VERSION_FILENAME}> not found in blob")
8     Health.torch_model.version = remote_version.decode("ascii")
9
10    if not isdir(_LOCAL_MODEL_DATA_FOLDER % remote_version.decode(
          'ascii')):
11        _DEBUG and print("Torch model missing folder files: must
              download")
12        Health.torch_model.downloaded = False
13    elif not isfile(_LOCAL_MODEL_VERSION_FILENAME):
14        _DEBUG and print("Torch model missing version file: must
              download")
```

```
15          Health.torch_model.downloaded = False
16      else:
17          with open(_LOCAL_MODEL_VERSION_FILENAME, "rb") as f:
18              local_version = f.read()
19          if local_version != remote_version:
20              _DEBUG and print("Torch model mismatch version: must
                    download")
21              Health.torch_model.downloaded = False
22
23      if not Health.torch_model.downloaded:
24          print("Starting download")
25          file = BlobServiceClient.from_connection_string(
                CHAT_BLOB_STORAGE_CONN_STR).get_blob_client(container=
                "pcn-email", blob=_REMOTE_MODEL_DATA_FILENAME).
                download_blob().content_as_bytes()
26          if not file:
27              raise Exception(f"File <{_REMOTE_MODEL_DATA_FILENAME}>
                    not found in blob")
28          with open(_LOCAL_MODEL_DATA_FILENAME, "wb") as f:
29              f.write(file)
30          with ZipFile(_LOCAL_MODEL_DATA_FILENAME, "r") as f:
31              f.extractall(_LOCAL_MODEL_DATA_FOLDER % remote_version
                    .decode('ascii'))
32          with open(_LOCAL_MODEL_VERSION_FILENAME, "wb") as f:
33              f.write(remote_version)
34          Health.torch_model.downloaded = True
35          _DEBUG and print("Torch model: file and version downloaded
                ")
36
37      return SentenceTransformer(_LOCAL_MODEL_DATA_FOLDER %
            remote_version.decode('ascii'))
```

4.6.5 funzione __extract__category__from__input

Questa funzione si occupa di estrarre, se presenti, le tassonomie dalla richiesta dell'utente. I suoi parametri sono infatti, *user_input*, stringa contenente la richiesta mandata al chatbot, e *categories*, una lista di stringhe contenente tutte le tassonomie presenti nei dati di riferimento.

Con un ciclo *for* la funzione itera su tutti gli elementi di *categories* e per ciascuno di essi, verifica se è contenuto in *user_input*. Ritorna tutte le tassonomie trovate.

```
1 def _extract_category_from_input(user_input: str, categories:
    list[str]) -> str:
2     for category in categories:
3         if category.lower() in user_input.lower():
4             return category
5     return None
```

4.6.6 funzione __parse__response

La funzione *_parse_response* si occupa di trasformare la risposta ottenuta, alla fine del processo, tramite API da Chat-GPT in un formato utilizzabile dal codice. In particolare, la risposta inviata da OpenAI è una stringa in formato JSON; la funzione riceve quindi in input *response_str* e da stringa JSON la trasforma in un dizionario Python utilizzando la funzione *loads* del libreria json. Dal dizionario *parsed_response* vengono poi estratti i valori associati alle chiavi "choice", valore che indica il tipo di richiesta dell'utente e "part_numbers", una lista contenente i part numbers estratti dalla richiesta dell'utente. Questi valori sono salvati rispettivamente nelle variabili *choice* e *part_numbers*.

La funzione restituisce quindi una tupla composta da choice e part_numbers, utilizzati successivamente da altre parti del programma per eseguire ulteriori operazioni.

Listing 4.1. funzione _parse_response

```
1     def _parse_response(response_str):
2     _DEBUG and print(response_str)
3     # response_str_with_quotes = f'"{str(response_str)}"'
```

```
    parsed_response = loads(response_str)
    _DEBUG and print(parsed_response)
    choice = parsed_response["choice"]
    part_numbers = parsed_response["part_numbers"]

    return choice, part_numbers
```

4.6.7 funzione __find__similar__products

La funzione *__finid__similar__products* rappresenta il nucleo del processo RAG, si occupa infatti di individuare, all'interno del nostro database, gli elementi più simili all'input fornito dall'utente, utilizzando la ricerca vettoriale offerta da MongoDB Atlas.
Questa funzionalità sfrutta inoltre la capacità di eseguire pipeline, una sequenza di passaggi operativi in cui ciascun passaggio rappresenta una specifica operazione, come filtrare, ordinare o selezionare determinati campi. Nello specifico, queste pipeline vengono eseguite sulla nostra collezione di dati *pim__embeddings*, che ospita embeddings, attributi e altre informazioni sui materiali. Gli elementi restituiti dalle diverse pipeline, che analizzeremo nel dettaglio, costituiscono il contesto che verrà passato a ChatGPT insieme alla richiesta dell'utente.
In sintesi, la funzione raccoglie i risultati della ricerca vettoriale eseguita su MongoDB Atlas, che includono informazioni specifiche come i part_numbers), le tassonomie, gli attributi, e altri dati associati agli elementi trovati nel database. Il contesto costruito serve a fornire a ChatGPT tutte le informazioni necessarie per rispondere in modo accurato e personalizzato alla richiesta dell'utente. Ad esempio, se l'utente chiede dettagli su un materiale o suggerimenti per prodotti simili, il contesto contenente i dati filtrati e rilevanti dalla pipeline permetterà a ChatGPT di generare una risposta più mirata, sfruttando sia la comprensione linguistica che i dati strutturati raccolti nel database.
Le pipeline sono composte generalmente da questi operatori:

- *$vector__search*: questa operazione è il cuore della ricerca vettoriale. In MongoDB Atlas Search, $vectorSearch cerca documenti i cui embeddings sono più vicini,in

termini di distanza nello spazio vettoriale, come la coseno-similarità o la distanza euclidea, al vettore fornito come input. Analizziamo la sua struttura nel dettaglio:

  - *index*: specifica l'indice usato per la ricerca
  - *path*: il campo su cui basare la ricerca vettoriale
  - *filter*: se presente, questo operatore filtra i documenti in base ai parametri specificati
  - *queryVector*: il vettore di query fornito come input, che rappresenta l'embedding dello *user_input* da confrontare con i documenti nel database
  - *numCandidates*: numero massimo di documenti inizialmente selezionati per il calcolo della similarità
  - *limit*: numero massimo di documenti da restituire come risultato finale

- *$project*: l'operatore *project*viene utilizzato per selezionare quali campi del documento devono essere restituiti come risultato. Questo aiuta a ridurre il volume di dati, evitando di includere informazioni superflue.

Analizziamo ora la funzione nel dettaglio; i parametri che prende in input sono *query_vector*, il vettore di embedding dell'input dell'utente, *identified_taxonomy*, le tassonomie trovate all'interno della richiesta dell'utente, *extracted_part_numbers*, i part_number identificati nella richiesta dell'utente.
La prima pipeline viene eseguita nel caso in cui siano state trovate delle tassonomie nell'input dell'utente, ovvero se il parametro *identified_taxonomy* non è vuoto, la funzione costruisce quindi una pipeline per limitare la ricerca ai documenti pertinenti a quelle tassonomie. La pipeline inizia con *$vector_search*, l'operatore di Atlas che esegue la ricerca vettoriale, tramite l'indice vettoriale precedentemente descritto, costruito sul campo *embedding* e chiamato *vector_index*; in *path* viene specificato il campo della collezione contenente gli embedding, ovvero *embedding*. Successivamente, tramite l'opzione *filter*, i documenti, su cui fare ricerca, vengono filtrati in base alle tassonomie trovate nell'input dell'utente. Ad esempio, se l'utente chiede "Puoi darmi un MOSFET con queste caratteristiche?", il filtro si assicura che vengano considerati solo i documenti che appartengono

alla tassonomia "MOSFET". In questo modo, la ricerca viene circoscritta ai risultati più rilevanti, evitando di includere documenti non pertinenti, come quelli relativi ad altri componenti elettronici. Con l'operatore *$project* vengono selezionati solo i campi utili per il contesto. Tra questi campi ci sono il part number del prodotto, *manufacturerpartnumber*, la tassonomia *taxonomyname*, gli attributi tecnici *attributes* e il punteggio di similarità *score*, calcolato sulla base della distanza vettoriale. I documenti trovati vengono restituiti ordinati in base al punteggio, così da fornire una lista di documenti classificati per rilevanza rispetto alla richiesta dell'utente. Una volta definita la pipeline, quest'ultima viene eseguita sulla collection di interesse *pim_embeddings* e tramite un ciclo for da ogni documento resituito come risultato dalla pipeline, vengono estratti tre campi specifici: il part number, la tassonomia e l'array di attributi. Una volta ottenuti la tassonomia e gli attributi dal documento, viene creato un dizionario *combined_info* che raccoglie questi due valori; questo dizionario rappresenta la combinazione di informazioni relative al prodotto, utile per il contesto da passare poi a Chat-GPT. Alla fine del processo, i part_number identificati vengono aggiunti alla lista *similar_part_numbers*, che quindi conterrà tutti i part_number dei prodotti risultanti simili alla richiesta dell'utente. Allo stesso modo, il dizionario *combined_info*, che include la tassonomia e gli attributi di ciascun prodotto, viene inserito nella lista *similar_combined*. Quest'ultima lista conterrà quindi tutte le combinazioni di tassonomie e attributi dei prodotti che hanno soddisfatto i criteri della ricerca. È importante sottolineare che ogni elemento di *similar_part_numbers* è direttamente accoppiato con il corrispondente elemento in *similar_combined*, cioè ogni part_number nella lista *similar_part_numbers* corrisponde alla combinazione di tassonomia e attributi presente nella lista *similar_combined* nella stessa posizione.

Procediamo con l'analisi della funzione nel caso in cui nella richiesta dell'utente venga identificato più di un part number. Questo scenario suggerisce con buona probabilità che l'utente voglia confrontare i prodotti corrispondenti ai part number forniti. Di conseguenza, la pipeline costruita per questo caso si limita a recuperare le informazioni relative ai part number specificati, senza effettuare ulteriori ricerche basate sugli embedding. L'obiettivo principale di questa pipeline è quindi quello di restituire, in modo diretto e preciso, i dati disponibili nella collezione per ciascuno dei part number indicati. L'operatore

$vector_search è costruito come nella pipeline *pipeline_taxonomy*, con la sola differenza nell'opzione *filter*: in questo caso infatti i documenti saranno filtrati in base ai part number trovati nello user input. L'operatore *$project* seleziona il part number, la tassonomia, gli attributi e il punteggio di similarità (non rilevante in questo caso), che successivamente sarannno inseriti nelle variabili *simialr_part_numbers* e *similar_combined*.
Quando nella richiesta dell'utente viene invece identificato un solo part number, la funzione esegue due pipeline consecutive. La prima pipeline recupera il documento associato a quel part number specifico, inclusi il suo embedding e i relativi dettagli. La seconda pipeline utilizza invece l'embedding del part number trovato come base, *queryVector* per effettuare una ricerca di prodotti simili. Questo approccio è stato scelto per garantire una maggiore precisione nella ricerca in casi in cui l'input dell'utente contenga un unico part number. Si è ipotizzato, infatti, che in questa situazione l'utente stia probabilmente cercando materiali simili a quel part number specifico. Questo metodo risponde anche all'esigenza di migliorare un sistema di ricerca preesistente, che permetteva di inserire un part number e ottenere i part number simili. Tuttavia, il vecchio sistema risultava troppo rigido e focalizzato, al punto da rendere la ricerca poco versatile e meno utile in contesti più complessi. La nuova soluzione si propone quindi di ampliare e migliorare la qualità dei risultati forniti. La prima delle due pipeline, *pipeline_1_pn*, viene eseguita se nella richiesta dell'utente viene identificato solamente un part number. Le impostazioni dell'operatore *$vectorSearch* sono analoghe alla pipeline precedente, ad eccezione dell'operatore *filter* che in questo caso filtra i documenti in base ai part number trovati nello *user_input*, ovvero l'unico contenuto in *extracted_part_numbers*. La prima pipeline restituirà esclusivamente il documento associato al part number specificato. Come definito dall'operatore *$project*, verranno estratti da questo documento solo il part number e l'embedding. L'embedding così ottenuto sarà poi memorizzato nella variabile *combined_embedding*, che servirà come base per la seconda fase del processo. Passiamo ora alla seconda pipeline, denominata *pipeline_combined*. Questa pipeline, in maniera analoga alle precedenti, utilizza l'indice *vector_index* per effettuare la ricerca vettoriale e come *path* di ricerca il campo *embedding*. Tuttavia, a differenza delle altre, il parametro *queryVector*, che rappresenta il vettore per cui trovare corrispondenze simili, sarà proprio il

valore contenuto nella variabile *combined_embedding*, ottenuto dalla prima pipeline. In questo modo, la ricerca sarà mirata a identificare i vettori più simili all'embedding del part number fornito, garantendo una corrispondenza accurata con i materiali correlati. L'operatore *$project* seleziona in questo caso il part number, la tassonomia, gli attributi e il punteggio di similarità dei materiali simili risultatnti dalla ricerca. Come si evince dal codice 3.3, in questa pipeline la collection di documenti non viene filtrata prima di eseguire la ricerca. Analogamente al caso precedente, le informazioni dei risultati della ricerca vengono inserite nelle liste *similar_part_numbers* e *similar_combined*.

Infine, nel caso in cui dalla richiesta dell'utente non vengano estratte né tassonomie né part number, la funzione *_find_similar_products* utilizza una pipeline generica, denominata *pipeline_generic*. Questa pipeline opera senza alcun tipo di pre-filtraggio sui dati, permettendo una ricerca più ampia e meno mirata. L'operatore *$vectorSearch* viene configurato in maniera analoga ai casi precedenti: utilizza l'indice vettoriale *vector_index* e come campo di ricerca *embedding*, mentre il vettore di partenza su cui effettuare la ricerca è *query_vector*, ovvero l'embedding generato dall'input dell'utente. Tuttavia, a differenza degli altri casi, in questa pipeline non viene applicata alcun filtro ai dati, consentendo di esplorare tutti i documenti presenti nella collezione senza restrizioni preliminari.

Di seguito il codice della funzione:

```
1  def _find_similar_products(query_vector, identified_taxonomy,
       extracted_part_numbers):
2      similar_part_numbers = []
3      similar_combined = []
4
5      if identified_taxonomy:
6          _DEBUG and print('pipeline tax')
7          pipeline_taxonomy = [
8              {
9                  '$vectorSearch': {
10                     'index': 'vector_index',
11                     'path': 'embedding',
```

```
                    'filter': {
                        '$and': [
                            {'taxonomyname': {'$in':
                                identified_taxonomy}}
                        ]
                    },
                    'queryVector': query_vector,
                    'numCandidates': 150,
                    'limit': 30
                }
            },
            {
                '$project': {
                    'manufacturerpartnumber': 1,
                    'taxonomyname': 1,
                    'attributes': 1,
                    'score': {'$meta': 'vectorSearchScore'}
                }
            }
        ]
        result = client["vector_db"]["pim_embeddings"].aggregate(
            pipeline_taxonomy)
        for doc in result:
            part_number = doc.get("manufacturerpartnumber")
            taxonomy = doc.get("taxonomyname")
            attributes = doc.get("attributes")

            if part_number:
                similar_part_numbers.append(part_number)


            combined_info = {
                'taxonomy': taxonomy,
```

```
43                 'attributes': attributes
44             }
45
46             similar_combined.append(combined_info)
47
48
49     elif extracted_part_numbers and len(extracted_part_numbers) >
           1:
50         _DEBUG and print(f'pipeline multiple pn')
51
52         pipeline_multiple_pn = [
53             {
54                 '$vectorSearch': {
55                     'index': 'vector_index',
56                     'path': 'embedding',
57                     'filter': {
58                         '$and': [
59                             {'manufacturerpartnumber': {'$in':
                                   extracted_part_numbers}}
60                         ]
61                     },
62                     'queryVector': query_vector,
63                     'numCandidates': 150,
64                     'limit': 30
65                 }
66             },
67             {
68                 '$project': {
69                     'manufacturerpartnumber': 1,
70                     'taxonomyname': 1,
71                     'attributes': 1,
72                     'score': {'$meta': 'vectorSearchScore'}
73                 }
```

```
74              }
75          ]
76          result = client["vector_db"]["pim_embeddings"].aggregate(
                pipeline_multiple_pn)
77
78          for doc in result:
79              part_number = doc.get("manufacturerpartnumber")
80              taxonomy = doc.get("taxonomyname")
81              attributes = doc.get("attributes")
82
83              if part_number:
84                  similar_part_numbers.append(part_number)
85
86
87              combined_info = {
88                  'taxonomy': taxonomy,
89                  'attributes': attributes
90              }
91
92              similar_combined.append(combined_info)
93
94      elif extracted_part_numbers and len(extracted_part_numbers) ==
            1:
95          _DEBUG and print(f'pipeline 1 pn')
96
97          pipeline_1_pn = [
98              {
99                  '$vectorSearch': {
100                     'index': 'vector_index',
101                     'path': 'embedding',
102                     'filter': {
103                         '$and': [
```

```
                            {'manufacturerpartnumber': {'$in':
                                extracted_part_numbers}}
                        ]
                    },
                    'queryVector': query_vector,
                    'numCandidates': 150,
                    'limit': 30
                }
            },
            {
                '$project': {
                    'manufacturerpartnumber': 1,
                    'embedding': 1,
                    'score': {'$meta': 'vectorSearchScore'}
                }
            }
        ]
        result = client["vector_db"]["pim_embeddings"].aggregate(
            pipeline_1_pn)

        embeddings = []
        for doc in result:
            part_number = doc.get("manufacturerpartnumber")
            embedding = doc.get("embedding")

            if embedding is not None:
                embeddings.append(embedding)

            if part_number:
                similar_part_numbers.append(part_number)

                combined_info = {
                    'taxonomy': None,
```

```
                    'attributes': None
                }
                similar_combined.append(combined_info)

        if len(embeddings) == 1:
            combined_embedding = embeddings[0]
        else:
            combined_embedding = None

        if combined_embedding and len(combined_embedding) > 0:
            pipeline_combined = [
                {
                    '$vectorSearch': {
                        'index': 'vector_index',
                        'path': 'embedding',
                        'queryVector': combined_embedding,
                        'numCandidates': 150,
                        'limit': 30
                    }
                },
                {
                    '$project': {
                        'manufacturerpartnumber': 1,
                        'taxonomyname': 1,
                        'attributes': 1,
                        'score': {'$meta': 'vectorSearchScore'}
                    }
                }
            ]
            result = client["vector_db"]["pim_embeddings"].
                aggregate(pipeline_combined)

            for doc in result:
```

```
                part_number = doc.get("manufacturerpartnumber")
                taxonomy = doc.get("taxonomyname")
                attributes = doc.get("attributes")

                if part_number:
                    similar_part_numbers.append(part_number)

                    combined_info = {
                        'taxonomy': taxonomy,
                        'attributes': attributes
                    }
                    similar_combined.append(combined_info)

    else:
        _DEBUG and print(f'generic pipeline')
        pipeline_generic = [
            {
                '$vectorSearch': {
                    'index': 'vector_index',
                    'path': 'embedding',
                    'queryVector': query_vector,
                    'numCandidates': 150,
                    'limit': 30
                }
            },
            {
                '$project': {
                    'manufacturerpartnumber': 1,
                    'taxonomyname': 1,
                    'attributes': 1,
                    'score': {'$meta': 'vectorSearchScore'}
                }
            }
```

```
200         ]
201         result = client["vector_db"]["pim_embeddings"].aggregate(
                pipeline_generic)
202
203         for doc in result:
204             part_number = doc.get("manufacturerpartnumber")
205             taxonomy = doc.get("taxonomyname")
206             attributes = doc.get("attributes")
207
208             if part_number:
209                 similar_part_numbers.append(part_number)
210
211                 combined_info = {
212                     'taxonomy': taxonomy,
213                     'attributes': attributes
214                 }
215                 similar_combined.append(combined_info)
216
217     _DEBUG and print(similar_part_numbers)
218     _DEBUG and print(similar_combined)
219
220     return similar_part_numbers, similar_combined
```

4.6.8 funzione __identify__prompt

La funzione *__identify__prompt* ha il compito di analizzare l'input dell'utente, *user__input* per estrarre informazioni che permettano di indirizzare correttamente la costruzione del prompt successivo. Una volta che l'utente fornisce un input, la funzione invia questa stringa al modello di intelligenza artificiale *gpt-4o-mini* attraverso una richiesta API, chiedendo al modello di restituire un oggetto JSON che contenga due chiavi principali: "choice", che indica se l'utente sta chiedendo un confronto tra prodotti o meno, e "part_numbers" , che contiene uno o più numeri di parte identificati nel testo dell'utente, se presenti.

Se la chiave choice è pari a 0, significa che l'utente sta richiedendo un confronto tra due oggetti, quindi il flusso sarà indirizzato verso la funzione *_prompt_formatter_pn*, che si occupa di elaborare una richiesta di confronto coerente con la richiesta dell'utente. Se invece la risposta indica che non c'è richiesta di confronto, ovvero choice è diverso da 0, ma potrebbero comunque esserci part number estratti, il flusso verrà indirizzato alla funzione *_prompt_formatter*, che gestisce altre situazioni, come richieste che non implicano un confronto ma necessitano comunque di una costruzione accurata del prompt da inviare a Chat-GPT. In conclusione, la funzione *_identify_prompt* restituisce gli oggetti choice e part_numbers e, in base alla risposta del modello, il flusso di lavoro sarà indirizzato alla funzione più appropriata per rispondere alla richiesta dell'utente.

```
1  def _identify_prompt(user_input: str) -> str:
2      prompt = """Send me as response a json object in this format (
           example values) :
3           {"choice": 0, "part_numbers": ["PN1", "PN2", "PN3"]}.
4          The key 'choice' is 0 if the user input is asking you to
              do a comparison, otherwise is 1.
5          The key 'part_numbers' is a list of part numbers if the
              user input contains one or more part numbers.
6          The part numbers can be in different formats, here are
              some examples of format : 1-86557-9 or 1.5KE100A or 93
              LC46AT-I/SN.
7          You just have to return the json object without any
              introduction or any comment before the object, don't
              specify it's json.
8          The user input from which you have to extract the info to
              create the JSON object is: """ + user_input
9
10     completion = openai.chat.completions.create(
11         model = 'gpt-4o-mini',
12         messages=[
13             {
14                 "role": "user",
```

```
                "content": prompt,
            },
        ],
    )

    # Extracting the content of the response
    content = completion.choices[0].message.content

    # Print the completion result
    _DEBUG and print(f"Response from OpenAI: {content}")

    return content
```

4.6.9 funzione _prompt_formatter_pn

La funzione *_prompt_formatter* ha il compito di costruire un prompt dettagliato da inviare al modello ChatGPT, costruito combinando la richiesta dell'utente, contenuta nel parametro *user_query* e un elenco di part numbers con i corrispondenti attributi, ottenuti dalla precedente ricerca vettoriale. Questa funzione viene utilizzata nel caso in cui l'input dell'utente sia una richiesta di comparazione e di conseguenza, come spiegato nella sezione precedente, la chiave *choice* del dizionario sia uguale a 0. Analizziamo ora nel dettaglio la funzione *_prompt_formatter*, che necessita di diversi parametri:

- *user_query* La domanda dell'utente, cioè il testo in cui l'utente esprime la sua richiesta
- *similar_part_numbers*: la lista di part numbers simili, estratti dalla precedente ricerca vettoriale
- *similar_combined*: la lista di liste contenenti gli attributi associati ai part numbers come informazioni specifiche sui prodotti (ad esempio, la descrizione del materiale o altre caratteristiche tecniche).

Una volta ricevute in input queste informazioni, la funzione definisce una parte base del prompt, contenuta nella variabile *base_prompt*. Questo testo include la parte iniziale della richiesta da inviare ad OpenAI in cui si specifica al modello di rispondere dettagliatamente alla richiesta dell'utente, di formulare una risposta completa, non limitandosi a semplici risultati, e di focalizzarsi sulle informazioni dei part numbers e i loro attributi. Successivamente, la funzione costruisce una lista di dizionari *context_items*, che abbina ogni part number con i suoi attributi, tramite la funzione *zip*. La lista ottenuta, *context_items* viene trasformata in una stringa *context*, ovvero per ogni elemento, viene creato un testo nel formato "Part Number: part_number, Attributes: attributes". Ogni elemento viene posizionato in una nuova riga per costruire un formato leggibile e ben strutturato. La stringa *context* con tutte le informazioni sui part numbers e attributi viene aggiunta alla variabile *prompt_part*, che rappresenta il contesto che ChatGPT userà per rispondere. Infine, il *base_prompt*, contenente la richiesta dell'utente, e *prompt_part*, contenente il contesto con i numeri di parte e i loro attributi, vengono uniti per formare il prompt finale, pronto per essere inviato al modello. La funzione restituisce quindi il prompt finale.

```
1 def _prompt_formatter_pn (user_query: str, similar_part_numbers,
    similar_combined) -> str:
2     prompt_parts = []
3     _DEBUG and print(similar_part_numbers)
4     base_prompt = f" Answer the user query: {user_query} given the
          following context items. Give me also a detailed
        description of the attributes, the differences and
        similarities of the PNs. The context is: >\n"
5     context_items = [{"part_number": pn, "attributes": comb} for
        pn, comb in zip(similar_part_numbers, similar_combined)]
6     context = "- " + "\n- ".join([f"Part Number: {item['
        part_number']}, Attributes: {item['attributes']}" for item
          in context_items])
7     prompt_part = f"{context}"
8     prompt_parts.append(prompt_part)
```

```
9
10      prompt = base_prompt + '\n'.join(prompt_parts)
11      _DEBUG and print(f"prompt:{prompt}")
12      return prompt
```

4.6.10 funzione _prompt_formatter

La funzione *_prompt_formatter* viene utilizzata quando la richiesta dell'utente non riguarda un confronto tra elementi. In questo caso la chiave *choice*, restituita dalla funzione *_identify_prompt* assume il valore 1, questo implica che l'utente sta facendo una domanda generica o richiede informazioni dettagliate, senza focalizzarsi su una comparazione esplicita tra più part numbers. La struttura della funzione è analoga a quella di *_prompt_formatter_pn*, con una differenza fondamentale: il contenuto del prompt di partenza, *base_prompt*, è formulato per rispondere a richieste generiche e non include alcuna istruzione relativa al confronto tra part numbers.

```
1   def _prompt_formatter(user_query: str, similar_part_numbers:
        list[list[str]], similar_combined: list[list[str]]) -> str
        :
2       prompt_parts = []
3       base_prompt = f"Given this context, answer the following
        question{user_query}. Formulate a sentence and give
        details about the answer, don't just give me the results,
        focus on the exact request the user is asking you.If
        possible, describe the part number  and its attributes.
        The context is:"
4
5       context_items = [{"part_number": pn, "attributes": comb} for
        pn, comb in zip(similar_part_numbers, similar_combined)]
```

```
6       context = "- " + "\n- ".join([f"Part Number: {item['
            part_number']}, Attributes: {item['attributes']}" for item
             in context_items])
7       prompt_part = f"{context}"
8       prompt_parts.append(prompt_part)
9
10      prompt = base_prompt + '\n'.join(prompt_parts)
11      _DEBUG and print(f"prompt:{prompt}")
12      return prompt
```

4.6.11 funzione _send_chat_request

L'ultima funzione, fondamentale per finalizzare il processo, è *_send_chat_request*. Il suo scopo è inviare il prompt, costruito nelle fasi precedenti, al modello di intelligenza artificiale, *gpt-4o-mini*, tramite una chiamata API, e restituire la risposta generata che sarà inviata all'utente nel chatbot. La funzione accetta un unico parametro, *prompt*, che rappresenta la richiesta formulata dalle funzioni di formattazione del prompt appena introdotte. Utilizzando la libreria OpenAI, la funzione invia una richiesta al modello specificato, *gpt-4o-mini*, tramite il metodo *openai.chat.completions.create*.
La richiesta contiene:

- *model*: specifica quale modello utilizzare, in questo caso, *gpt-4o-mini*.
- *messages*: una lista di messaggi che rappresentano la conversazione. In questo caso, viene inviato un unico messaggio, il prompt, con il ruolo *user* e il contenuto, *content* rappresentato dal parametro *prompt*.

Una volta ricevuta la risposta dal modello, la funzione accede all'attributo *completion.choices[0].message.content*, che contiene il testo generato dal modello, ovvero la risposta finale fornita dall'intelligenza artificiale che la funzione restituisce come output.

```
def _send_chat_request(prompt: str) -> str:
    completion = openai.chat.completions.create(
        model="gpt-4o-mini",
        messages=[
            {
                "role": "user",
                "content": prompt,
            },
        ],
    )
    content = completion.choices[0].message.content
    return content
```

4.7 Esempi di applicazione

In questa sezione vengono illustrati diversi esempi pratici di interazione tra l'utente e il chatbot, evidenziando la varietà di domande che è possibile porre e le relative risposte generate dal processo. Gli esempi selezionati mostrano come il chatbot sia in grado di comprendere richieste complesse, fornire risposte pertinenti e contestualizzate, e adattarsi alle esigenze specifiche dell'utente, grazie all'integrazione della Retrieval-Augmented Generation (RAG) con il modello LLM e al sistema di ricerca vettoriale ottimizzato. La lingua utilizzata nell'interazione tra utente e sistema è l'inglese per coerenza con i sistemi aziendali e per efficienza nelle risposte fornite dall LLM.

4.7.1 Ricerca di un materiale simile

In questo esempio la richiesta dell'utente è: *Can you give me a material similar to STM32F102R4T6A but with a higher pin count?*

La risposta del sistema è la seguente:

If you are looking for a microcontroller similar to the STM32F102R4T6A but with a higher pin count, several options from the STM32F1 series could be good alternatives. One strong candidate is the STM32F103VET6, which belongs to the same STM32F1 family but offers a significantly higher number of pins and additional features.

STM32F103VET6 Key Specifications

- **Pin Count**: 100 (compared to 64 on STM32F102R4T6A)
- **Core**: ARM Cortex-M3
- **Operating Frequency**: 72 MHz
- **Flash Memory**: 512 KB
- **RAM**: 64 KB
- **Interfaces**: CAN, I2C, SPI, USART, USB
- **On-Chip ADC**: 3 (16 channels, 12-bit resolution)
- **On-Chip DAC**: 2 (12-bit, 2 channels)
- **Timers**: 8
- **Operating Voltage**: 2.5V – 3.3V
- **Temperature Range**: -40°C to 85°C
- **Package**: LQFP-100

This microcontroller provides more GPIO pins (80 programmable I/Os) compared to the STM32F102R4T6A, making it an excellent choice for applications requiring more peripherals or connectivity.

Other Alternative Options

- **STM32F103VCT6**

  - **Pin Count**: 100
  - Similar features to the VET6 but with a smaller flash memory size.

- **STM32F103VCH6**
  - **Pin Count**: 100
  - Uses an LFBGA package instead of LQFP.
  - Offers the same Cortex-M3 core and peripherals.

- **STM32F107VBT6**
  - **Pin Count**: 100
  - Similar to STM32F103 but includes Ethernet support.

Why Choose These Over STM32F102R4T6A?

- More GPIO pins for expansion.
- Additional interfaces such as CAN and DAC support.
- Higher processing power (72 MHz vs. 48 MHz in STM32F102).
- More Flash and RAM for complex applications.

If you need further recommendations based on power consumption, real-time performance, or specific peripherals, let me know!

4.7.2 Descrizione di un materiale

In questo esempio la richiesta dell'utente è:
Can you tell me the characteristics of BR24T32FVT-WE2?
La risposta del sistema è la seguente:

> The BR24T32FVT-WE2 is a 32 Kb EEPROM that operates with a Serial I2C interface, designed for surface mount applications in a TSSOP-B (Thin Shrink Small Outline Package - B) form factor. It features hardware data protection

for improved reliability and supports a typical operating supply voltage range of 1.8V, 2.5V, 3.3V, and 5V, making it versatile for different circuit designs.

With a maximum random access time of 900 ns and an operating frequency of 0.4 MHz, it provides efficient data read/write operations. The EEPROM has a maximum operating current of 2 mA, ensuring low power consumption. The MSL (Moisture Sensitivity Level) is rated at MSL 1 - Unlimited, meaning it can withstand standard environmental conditions without degradation.

The maximum processing temperature is 260°C, and it is designed to function within a temperature range of -40°C to 85°C, making it suitable for industrial and commercial applications. The data retention is guaranteed for a minimum of 40 years, and it supports a programming voltage range of 1.6V to 5.5V.

Physically, the BR24T32FVT-WE2 measures 3 x 4.4 x 1 mm and has an 8-pin count configuration. Its memory organization is 4K x 8, ensuring efficient memory management. The lead finish is tin-copper, and it is not radiation-hardened (Rad Hard: No).

Alternatives

- **BR24T32FV-WE2**
  - Similar 32 Kb EEPROM, but comes in an 8SSOP-B package instead of 8TSSOP-B.
- **BR24T32F-WE2**
  - Another 32 Kb EEPROM, but in a SOP package (5 x 4.4 x 1.5 mm).
- **BR24T64FVT-WE2**
  - A higher-density 64 Kb version, in the same TSSOP-B package.
- **BR24T256FVT-WE2**
  - A larger 256 Kb variant, also in a TSSOP-B package.

The BR24T32FVT-WE2 is a compact, reliable, and low-power EEPROM ideal for embedded systems, industrial controls, and consumer electronics requiring non-volatile memory storage with I2C communication.

4.7.3 Confronto tra due materiali

In questo esempio la richiesta dell'utente è: *Can you tell me the similarities and the differences between PIC12F617-I/MF and PIC12LF1501-E/P?*
La risposta del sistema è la seguente:

> Here's a detailed comparison of the **PIC12F617-I/MF** and **PIC12LF1501-E/P** microcontrollers, highlighting their similarities, differences, and attributes.

Similarities Between PIC12F617-I/MF and PIC12LF1501-E/P:

- Both are **8-bit PIC microcontrollers** from Microchip Technology.
- They use a **RISC** (Reduced Instruction Set Computing) architecture and have a **PIC** core.
- Both use **Flash** memory for program storage.
- Both have **5 programmable I/O pins** and **3 timers**.
- Both feature a **4-channel, 10-bit ADC** (Analog-to-Digital Converter).
- Neither is **radiation-hardened**.
- Both include **1 watchdog timer**.
- Both operate at a **maximum frequency of 20 MHz**.
- Both use a **Matte Tin** lead finish and have **8 pins**.

Differences Between PIC12F617-I/MF and PIC12LF1501-E/P:
Conclusion:

Attribute	PIC12F617-I/MF	PIC12LF1501-E/P
Operating Supply Voltage	2.5V, 3.3V, 5V	3.3V
Operating Temperature Range	-40°C to 85°C (Industrial)	-40°C to 125°C (Extended)
Mounting Type	Surface Mount	Through Hole
Package Type	8DFN EP (3 x 3 x 0.88 mm)	8PDIP (9.27 x 6.35 x 3.3 mm)
Screening Level	Industrial	Extended
Analog Comparators	1	Not listed
Maximum Expanded Memory Size	14 KB	Not listed
Moisture Sensitivity Level (MSL)	MSL 1 - Unlimited	Not listed

Tabella 4.1. Comparison between PIC12F617-I/MF and PIC12LF1501-E/P

- **PIC12F617-I/MF** is more versatile in **operating voltage (2.5V–5V)**, has a **smaller form factor (3mm DFN)**, and includes an **analog comparator**. It is ideal for **compact, surface-mount applications**.

- **PIC12LF1501-E/P** supports **higher temperatures (up to 125°C)**, is in a **larger through-hole PDIP package**, and is designed for **extended temperature range applications**.

Choosing between the two depends on **voltage requirements, mounting type, and environmental conditions**. If you need a **smaller, surface-mount** microcontroller with **wider voltage flexibility**, go for **PIC12F617-I/MF**. If you need **higher temperature tolerance** and **through-hole mounting**, **PIC12LF1501-E/P** is the better choice.

4.8 Prototipo

Di seguito viene mostrato un prototipo del chatbot che verrà implementato nell'applicazione.

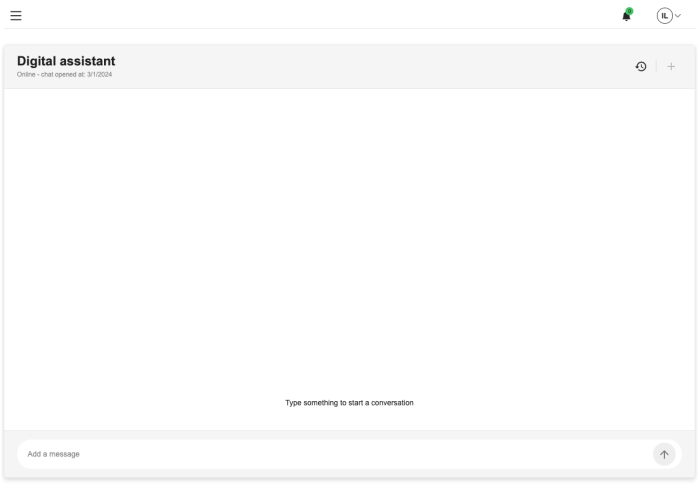

Figura 4.5. Prototipo del chatbot

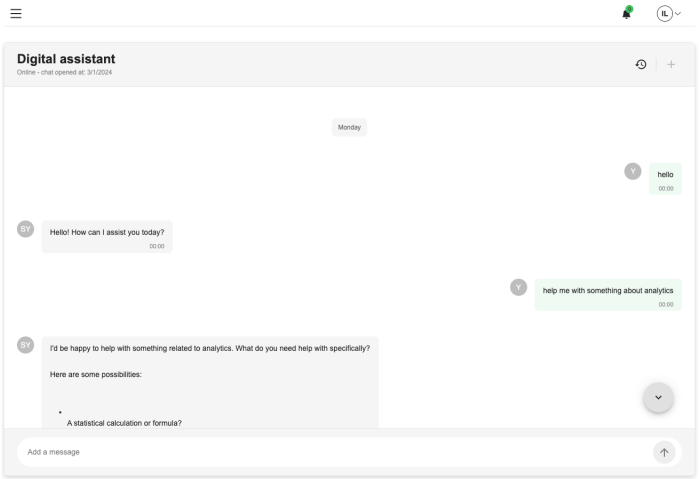

Figura 4.6. Prototipo del chatbot durante una richiesta

Capitolo 5

Conclusioni

Nel presente capitolo sono discussi i risultati della trattazione, le sue limitazioni e spunti per ulteriori sviluppi del lavoro stesso.

5.1 Obiettivi e risultati

Il lavoro svolto in questa tesi rappresenta un importante punto di partenza per il miglioramento e l'ampliamento della ricerca di materiali simili già esistente nell'applicazione myDA. L'integrazione del modello di embedding MPNet con l'indice vettoriale su MongoDB Atlas ha permesso di rendere il processo di ricerca estremamente rapido ed efficiente. Grazie alla struttura ottimizzata del processo implementato, il sistema è in grado di eseguire query senza dover caricare l'intera collezione di dati, ma prelevando direttamente solo i risultati della ricerca vettoriale. Questo approccio ha ridotto drasticamente i tempi di elaborazione consentendo un accesso più diretto alle informazioni richieste.
Uno dei principali vantaggi di questa architettura è che l'intero processo di ricerca si basa su un indice vettoriale efficiente, capace di individuare rapidamente le corrispondenze tra le richieste dell'utente e i materiali disponibili. Questo significa che il sistema non deve analizzare l'intero dataset ogni volta che riceve una nuova query, ma può limitarsi a un sottoinsieme altamente pertinente, migliorando la scalabilità e riducendo significativamente il carico computazionale.

Inoltre, la segmentazione delle informazioni e l'uso di filtri predefiniti permettono di affinare ulteriormente la ricerca, rendendo i risultati non solo più veloci, ma anche più pertinenti alle esigenze dell'utente. La possibilità di pre-filtrare i dati in fase di interrogazione, anziché dopo il recupero delle informazioni, elimina passaggi computazionalmente onerosi e garantisce una maggiore efficienza del sistema. Questo aspetto è particolarmente utile in un contesto aziendale, dove rapidità e precisione sono essenziali per migliorare i flussi di lavoro e supportare decisioni tempestive.

La combinazione di queste tecnologie ha quindi permesso di ottenere un sistema di ricerca altamente reattivo e adattabile, capace di rispondere in tempo reale a richieste specifiche senza compromettere la qualità dei risultati. Inoltre, il fatto che il sistema possa funzionare direttamente collegato al database, senza la necessità di caricare la collezione completa dei dati in memoria, rappresenta un ulteriore vantaggio in termini di gestione delle risorse, contribuendo a rendere il chatbot scalabile e adatto a un utilizzo aziendale su larga scala.

L'adozione di GPT-4o-mini ha reso più fluida l'interazione con il sistema, permettendo una comprensione più accurata delle richieste e una generazione di risposte più coerenti e contestualizzate.Grazie all'integrazione tra RAG (Retriever-Augmented Generation) e LLM (Large Language Models), gli utenti non sono più limitati a ricerche basate su parole chiave esatte, ma possono porre domande più articolate e sofisticate, che coinvolgono simultaneamente diversi attributi o richiedono un confronto dettagliato tra materiali. Questo approccio rende il sistema notevolmente più potente rispetto alle tradizionali soluzioni basate esclusivamente su LLM.

Un aspetto particolarmente rilevante di questa combinazione è che gli utenti non devono limitarsi a chiedere solamente di materiali simili o a cercare codici di prodotto (PN). Possono, infatti, richiedere informazioni specifiche su un materiale, come caratteristiche dettagliate, descrizioni tecniche o addirittura un'analisi comparativa tra diverse opzioni. Questo approccio consente di ottenere risposte molto più complete e contestualizzate, offrendo una panoramica a 360 gradi su un materiale, piuttosto che una semplice descrizione legata a un numero di parte.

Inoltre, i test effettuati hanno confermato l'affidabilità del sistema in vari scenari applicativi, dimostrando come il chatbot sia in grado di rispondere con precisione a domande

tecniche e specifiche, grazie alla capacità di elaborare richieste complesse. L'uso di RAG ha ridotto notevolmente le allucinazioni del modello, migliorando la qualità delle risposte e consentendo di rispondere in modo mirato alle necessità degli utenti.

5.2 Sviluppi Futuri

Nonostante i risultati ottenuti, vi sono diverse direzioni in cui il lavoro può essere esteso per migliorare ulteriormente le funzionalità del sistema:

- Training di un modello di embedding Ad Hoc: attualmente, il sistema utilizza un modello pre-addestrato (MPNet) per la generazione degli embedding. Un possibile sviluppo futuro consiste nell'addestrare un modello di embedding specifico sui dati aziendali, ottimizzandolo per il dominio di applicazione. Questo potrebbe migliorare ulteriormente la qualità della ricerca e la pertinenza delle risposte generate.

- Espansione delle applicazioni della chat: il chatbot basato su RAG potrebbe essere esteso a nuovi ambiti aziendali, migliorandone le capacità di supporto. Ad esempio, potrebbe essere utilizzato per:

  - fornire assistenza nella gestione dei processi aziendali, guidando gli utenti nelle procedure operative.
  - rispondere a domande relative all'uso di applicazioni aziendali, fungendo da guida interattiva per software interni.
  - supportare la formazione dei dipendenti, fornendo accesso immediato alla documentazione e alle best practice aziendali.

- Miglioramento della ersonalizzazione e dell'adattabilità: un ulteriore sviluppo potrebbe riguardare la personalizzazione delle risposte in base al profilo dell'utente. Integrare un sistema di autenticazione e profilazione consentirebbe di fornire suggerimenti più mirati e ottimizzati per le esigenze specifiche di ogni dipartimento o funzione aziendale.

L'implementazione di queste possibili migliorie potrebbe trasformare il chatbot in un vero e proprio assistente intelligente per l'azienda, in grado di supportare diversi processi e migliorare l'accesso alle informazioni aziendali in modo dinamico ed efficiente.

Sitografia

- https://my.avnet.com/emea/about-us/avnet-emea/overview
- https://www.elettronica-av.it/avnet-ventanni-di-successi-con-le-speedboat-in-europa
- https://www.avnet.com/wps/portal/us/solutions/logistics/global-facilities/
- https://en.wikipedia.org/wiki/Avnet
- https://www.ibm.com/it-it/topics/large-language-models
- https://www.oracle.com/it/artificial-intelligence/generative-ai/retrieval-augmented-generation-rag/