UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

TESI DI LAUREA

Studio ed applicazione di modelli di Natural Language Processing per la ricerca di contenuti digitali su domini chiusi

Study and application of Natural Language Processing models for digital content search on closed domains

Relatore

Prof. Domenico Ursino

Candidato

Luca Agostinelli

ANNO ACCADEMICO 2021-2022

Sommario

Il fine della presente tesi è di esporre e valutare il percorso progettuale che ha portato alla creazione di un software di Machine Learning per il Question Answering. Sempre più diffusi sono i tentativi di innestare all'interno del processo demografico le innovazioni prodotte dalle tecnologie, e questo elaborato ne è un esempio. Infatti, il lavoro sviluppato propone di valutare quanto sia efficace, in termini assoluti, l'estrazione di un testo, tramite Machine Learning, come risposta ad una domanda data in input. Nel primo e nel secondo capitolo vengono esposte le caratteristiche e l'architettura della tecnologia utilizzata. L'impostazione metodologica adottata segue il classico sviluppo che comprende l'analisi dei requisiti e la progettazione, presentate, rispettivamente, nel terzo e nel quarto capitolo. Il quinto capitolo illustra, tramite codice, l'architettura e le funzionalità integrate nel software. È compreso, nel sesto capitolo, un esempio del funzionamento del sistema. Per la conclusione e la discussione del lavoro è riservato il settimo capitolo, mentre, nell'ottavo capitolo, sono presentate le conclusioni, con un occhio rivolto verso il futuro.

Keywords: Intelligenza Artificiale, Machine Learning, Deep Learning, Haystack, Natural Language Processing, Huggingface, DPR, Python, Jupyter, Retriever, Pipeline, NLP.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Introduzione

È sotto gli occhi di tutti il fatto che, da qualche decennio a questa parte, stiamo vivendo cambiamenti radicali sotto il punto di vista della tecnologia. Una della aree su cui bisogna porre particolare attenzione è quella del Machine Learning. La crescente attenzione creatasi su questa disciplina è motivata dai risultati conseguibili grazie alla maturità tecnologica raggiunta, sia nel calcolo computazionale sia nella capacità di analisi in tempi brevi di enormi quantità di dati in qualsiasi forma.

L'Intelligenza Artificiale ed il Machine Learning trovano largo impiego nella vita quotidiana della maggior parte degli individui. Infatti, i vari strumenti di riconoscimento vocale che vengono regolarmente utilizzati si basano su algoritmi tipici dell'Intelligenza Artificiale. Un significativo utilizzo di questa tecnologia si riscontra anche nel settore automobilistico, ad esempio nei veicoli in grado di muoversi nel traffico anche senza pilota, oppure nei sistemi di guida semi-autonoma. Le potenzialità e le prospettive di impiego di queste tecnologie sono in continua evoluzione.

Gli aspetti principalmente analizzati nel presente elaborato si riferiscono all'ambito del Natural Language Processing e alle possibili applicazioni nel contesto del Question Answering (QA). I modelli di risposta alle domande sono modelli di Machine Learning o Deep Learning in grado di rispondere a domande in una determinata circostanza. Possono estrarre frasi di risposta da paragrafi, parafrasare la risposta in modo generativo o creare a loro volta domande su un elaborato. Tali modelli devono capire la struttura del linguaggio, avere una comprensione semantica del contesto e delle domande ed avere la capacità di individuare la posizione di una frase di risposta. Quindi, senza alcun dubbio, è difficile addestrare modelli che svolgano tali compiti. Fortunatamente, esistono le reti neurali, adatte per svolgere operazioni così difficili. Esse sono sistemi costituiti da unità in grado di attivarsi in conseguenza della ricezione di segnali di ingresso, veicolando l'attivazione da un'unità all'altra.

Per portare a termine l'obiettivo di QA che ci siamo prefissati, utilizzeremo un framework intuitivo e ben documentato chiamato Haystack. Questa piattaforma open source è adibita alla creazione di sistemi di ricerca su grandi moli di dati. Essa si integra con altri framework open source come Huggingface, un repository di modelli di NLP, o Elasticsearch, un server di ricerca.

Tutto ciò ha stimolato molto la nostra curiosità, e le difficoltà ci hanno spronato ed invogliato ad investire tempo e risorse nel progetto alla base del presente elaborato.

La tesi in oggetto è composta da otto capitoli, strutturati come di seguito specificato:

- Nel primo capitolo verrà proposta una panoramica generale sulla storia della tecnologia, a partire all'Intelligenza Artificiale, passando per il Machine Learning ed arrivando al Deep Learning. Verrà introdotto, poi, il Natural Language Processing con le proprie applicazioni.
- Nel secondo capitolo verrà analizzato il framework di riferimento, cioè Haystack. In aggiunta a tutte le caratteristiche ed ai componenti messi a disposizione, verranno illustrati, anche, diversi esempi di funzionamento di tale piattaforma.
- Nel terzo capitolo verrà descritta l'analisi dei requisiti, sia funzionali che non, del progetto in questione. Verranno presentate, anche, una descrizione della componente dati ed una trattazione del software di sviluppo utilizzato, ovvero Project Jupyter.
- Nel quarto capitolo verrà illustrata, invece, la progettazione completa di tutti i componenti impiegati. Nella parte finale del capitolo è presente una descrizione matematica delle funzioni di similarità di cui ci siamo serviti.
- Nel quinto capitolo verrà presentata tutta l'implementazione delle componenti che costituiscono il nostro notebook. In ogni blocco di codice è affiancata una descrizione del funzionamento.
- Nel sesto capitolo verrà proposta una panoramica generale sulla messa in funzione del software, comprendente di output intermedi e possibili errori. Inoltre, verranno presentati i risultati ottenuti.
- Nel settimo capitolo verrà descritta la SWOT Analysis, con i suoi quattro step, e saranno elencate le lezioni apprese durante lo sviluppo del software.
- Nell'ottavo capitolo, infine, verranno tratte le conclusioni e verranno delineati alcuni possibili sviluppi futuri.

CAPITOLO 1

Introduzione al Machine Learning

In questo primo capitolo daremo un'occhiata da vicino allo scenario tecnologico di riferimento; in particolare, parleremo della storia del Machine Learning e lo analizzeremo ampiamente sotto diversi punti di vista, con un approfondimento sul campo della comprensione e dell'elaborazione del linguaggio naturale umano.

1.1 Storia della tecnologia

Per capire in modo approfondito il Machine Learning (ML), bisogna partire da un concetto più ampio, che lo ingloba, cioè l'Intelligenza Artificiale (AI). Il primo ad utilizzare quest'ultima fu l'assistente universitario di matematica *John McCarthy* (Figura 1.1) nel 1956, in occasione di un convegno per il quale sarebbe stato necessario trovare una terminologia specifica per differenziare questo campo di ricerca dalla già nota cibernetica.

Figura 1.1: *John McCarthy*, il padre dell'Intelligenza Artificiale

Oltre a *John McCarthy*, sono tre le figure che si sono distinte in questo decennio per aver portato novità e innovazione nel campo dell'AI: *Alan Turing*, *Arthur Samuel* e *Frank Rosenblatt*.

Alan Turing è noto per aver ideato un test che si poneva l'obiettivo di paragonare l'Intelligenza Artificiale a quella umana, conosciuto appunto come "Test di Turing" o "Imitation game".

Arthur Samuel (Figura 1.2), informatico statunitense, sull'onda del forte entusiasmo per l'evoluzione tecnologica, invece, realizzò il suo "giocatore di dama", un programma ideato affinché si auto-migliorasse fino a superare le abilità del creatore.

Figura 1.2: *Arthur Samuel* che realizza il programma di dama per il computer

Il terzo pioniere in questo campo fu *Frank Rosenblatt*, uno psicologo che aveva appena finito il dottorato alla Cornell University. Egli elaborò l'idea che cambiò per sempre il corso della storia. Da buon psicologo, sapeva che le regressioni lineari erano (e sono tuttora) lo strumento perfetto per formulare predizioni in base ad una serie di attributi predittori e, inoltre, che questa configurazione era perfetta per simulare lo stato di un neurone automatico, il quale riceve una serie di segnali in input e genera una predizione della quantità di segnale da restituire in output. Questo neurone artificiale, detto percettrone, era finalmente in grado di prendere decisioni automaticamente. L'invenzione del percettrone ebbe una grandissima risonanza nei media e accese l'entusiasmo di molte comunità di ricerca. Proprio in questo momento storico, quindi, si coniò il termine "Machine Learning", cioè apprendimento automatico.

Gli anni '50, densi di cambiamenti e novità, si conclusero con il riconoscimento dell'Intelligenza Artificiale come campo di ricerca indipendente, dando vita così ad una nuova definizione di tecnologia e ponendo le basi per le numerose evoluzioni che seguiranno nei decenni successivi.

Sul finire degli anni '70 si assistette ad un aumento esponenziale dell'utilizzo di computer che, essendo più piccoli ed economici, vennero acquistati in massa, specialmente dalle aziende. Conseguentemente, anche la quantità di documentazione prodotta iniziò a crescere, necessitando, così, di strumenti per poterla organizzare e consultare rapidamente.

Nella seconda metà degli anni '80 venne reinventato l'algoritmo di back-propagation relativo all'apprendimento per le reti neurali. Questo permise di creare un'alternativa ai modelli simbolici (utilizzati inizialmente da *McCarthy*) attraverso i modelli connessionisti, che si posero l'obiettivo di spiegare il funzionamento della mente ricorrendo all'utilizzo di reti neurali artificiali.

1.1.1 Teoria dell'apprendimento

L'obiettivo principale dell'apprendimento automatico è che una macchina sia in grado di generalizzare dalla propria esperienza, ovvero che sia in grado di svolgere ragionamenti induttivi. Con questo termine si intende l'abilità di una macchina di portare a termine in maniera accurata esempi o compiti nuovi, che non ha mai affrontato, dopo aver fatto esperienza su un insieme di dati di apprendimento. Gli esempi di addestramento si assume provengano da una distribuzione di probabilità, generalmente casuale.

L'analisi computazionale degli algoritmi di apprendimento automatico e delle loro prestazioni è una branca dell'informatica teorica chiamata, appunto, "teoria dell'apprendimento". Poiché gli esempi di addestramento sono insiemi finiti di dati e non c'è modo di sapere l'evoluzione futura di un modello, la teoria dell'apprendimento non offre alcuna garanzia

sulle prestazioni degli algoritmi, anche se questi ultimi possono essere validi in termini di probabilità.

Oltre ai limiti prestazionali, i teorici dell'apprendimento studiano la complessità temporale e la fattibilità dell'apprendimento stesso. Una computazione è considerata fattibile se può essere svolta in tempo polinomiale.

1.2 Machine Learning

Il Machine Learning, come detto in precedenza, è un sottoinsieme dell'Intelligenza Artificiale che si occupa di creare sistemi che apprendono o migliorano le performance in base ai dati che utilizzano. Osservando la Figura 1.3, si può distinguere bene che, sebbene tutto ciò che riguardi il Machine Learning rientri nell'Intelligenza Artificiale, quest'ultima non includi solo il Machine Learning.

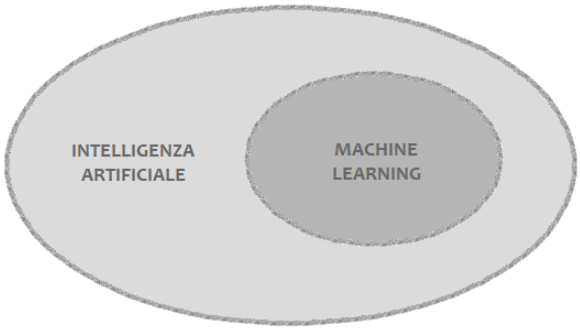

Figura 1.3: Diagramma di Venn di AI e ML

Guardando il Machine Learning da una prospettiva informatica, anziché scrivere il codice di programmazione attraverso il quale, passo dopo passo, si "dice" alla macchina cosa fare, al programma vengono forniti "solo" dei set di dati. Questi vengono elaborati attraverso algoritmi e viene sviluppata una propria logica per svolgere la funzione o il compito richiesti, come imparare a riconoscere un'immagine. Attualmente, il Machine Learning, anche detto apprendimento automatico, è largamente utilizzato. Quando interagiamo con le banche, acquistiamo online o utilizziamo i social media, vengono utilizzati gli algoritmi di ML per rendere la nostra esperienza efficiente, facile e sicura.

Il Machine Learning funziona in linea di principio sulla base di due distinti approcci, identificati dal sopracitato *Arthur Samuel*, che permettono di distinguere l'apprendimento automatico in due sottocategorie a seconda del fatto che si diano al computer esempi completi da utilizzare come indicazione per eseguire il compito richiesto, oppure che si lasci lavorare il software senza alcun aiuto. Il primo approccio è noto come apprendimento supervisionato, mentre il secondo come apprendimento non supervisionato (Figura 1.4).

1.2.1 Machine Learning supervisionato

Gli algoritmi di Machine Learning supervisionato sono i più utilizzati. Con questo modello, un data scientist agisce da guida e insegna all'algoritmo i risultati da generare. Esattamente come un bambino impara a identificare i frutti memorizzandoli in un libro illustrato, nel Machine Learning supervisionato l'algoritmo apprende da un set di dati già etichettato e con un output predefinito.

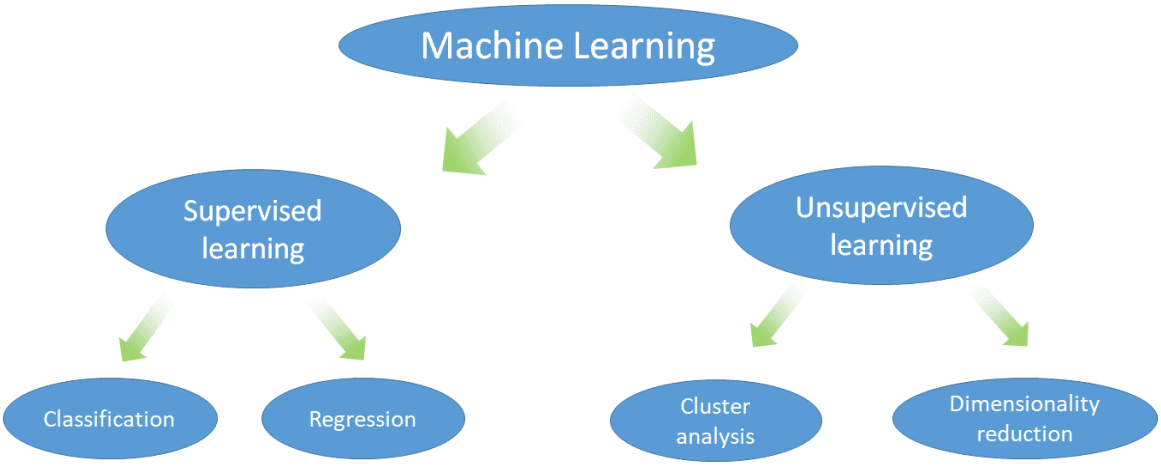

Figura 1.4: ML supervisionato e non supervisionato

Esempi di Machine Learning supervisionato sono gli algoritmi di regressione lineare, classificazione multiclasse e support vector machines.

1.2.2 Machine Learning non supervisionato

Il Machine Learning non supervisionato utilizza un approccio più indipendente, in cui un computer impara ad identificare processi e schemi complessi senza la guida attenta e costante di una persona. Il ML non supervisionato implica una formazione basata su dati privi di etichette e per i quali non è stato definito un output specifico.

Per continuare ad utilizzare l'analogia precedente, il Machine Learning non supervisionato è simile ad un bambino che impara ad identificare i frutti osservando i colori e gli schemi, anziché memorizzando i nomi con l'aiuto di un insegnante. Il bambino cercherà le somiglianze tra le immagini e le suddividerà in gruppi, assegnando a ciascun gruppo la nuova etichetta corrispondente. Gli algoritmi di clustering k-means, l'analisi di componenti principali e indipendenti e le regole associative sono esempi di ML non supervisionato.

1.2.3 Reti neurali artificiali

Precedentemente sono state citate le reti neurali artificiali; queste rappresentano un sottoinsieme del Machine Learning e sono molto utilizzate in questo campo. Il loro nome e la loro struttura sono ispirati al cervello umano, imitando il modo in cui i neuroni biologici si inviano segnali.

Una rete neurale artificiale, nella sua forma più elementare, ha tre strati di neuroni; l'informazione scorre da uno strato all'altro, proprio come avviene nel cervello umano, tali strati sono:

- il livello di input, cioè il punto di ingresso dei dati nel sistema;
- lo strato nascosto, dove l'informazione viene elaborata;
- il livello di uscita, dove il sistema decide come procedere in base ai dati.

Le reti neurali artificiali più complesse avranno più strati, alcuni dei quali saranno nascosti. In Figura 1.5 è possibile osservare un esempio di una rete composta da uno strato di input, uno di output e tre strati di neuroni nascosti.

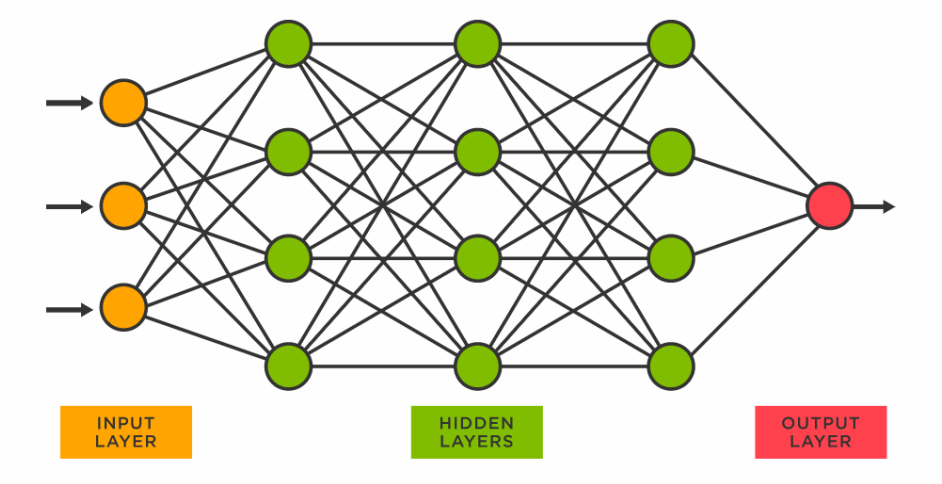

Figura 1.5: Esempio di una rete neurale artificiale

Ciascun nodo, o neurone artificiale, si connette ad un altro ed ha un peso ed una soglia associati. Se l'output di qualsiasi singolo nodo è al di sopra del valore di soglia specificato, tale nodo viene attivato, inviando i dati al successivo livello della rete. In caso contrario, non viene passato alcun dato al livello successivo della rete.

Le reti neurali sono un'alternativa alle tecniche standard, un po' limitate, che si basano sui concetti di normalità e indipendenza delle variabili. La capacità delle reti neurali di esaminare una vasta varietà di relazioni rende più facile per l'utente modellare rapidamente fenomeni che potrebbero essere abbastanza difficili, o addirittura impossibili, da comprendere altrimenti. Come grande difetto, esse non sono in grado di fornire un supporto matematico che delinei il processo. In parole povere, il processo non è spiegabile.

1.2.4 Deep Learning

È impossibile parlare di reti neurali senza menzionare il Deep Learning (DL). Tendenzialmente, si parla di Deep Learning e reti neurali in modo interscambiabile, il che può creare confusione. Di conseguenza, vale la pena notare che il termine "profondo" in Deep Learning si riferisce esclusivamente alla profondità dei livelli in una rete neurale. Una rete neurale che consiste in più di tre livelli, comprensivi degli input e dell'output, può essere considerata un modello di Deep Learning. Una rete neurale che ha solo due o tre livelli è soltanto una rete neurale di base.

Se il Machine Learning si può considerare come una sottocategoria dell'Intelligenza Artificiale, il Deep Learning, a sua volta, è una branca del Machine Learning (Figura 1.6).

1.2.5 Applicazioni del Machine Learning

È errore comune pensare al Machine Learning come a sistemi che trovino applicazione in ambiti molto particolari e specifici, compreso il settore della ricerca scientifica. Al contrario, l'apprendimento automatico vanta molte applicazioni di uso quotidiano e molto vicine a noi.

Un esempio di utilizzo del Machine Learning può essere quello che permette alle aziende di realizzare pubblicità online basata sugli interessi degli utenti, le cui necessità e i cui gusti vengano identificati per mezzo dell'analisi delle ricerche effettuate in rete. I siti Web che consigliano articoli recanti la dicitura "potrebbero interessarti" si basano proprio sulla cronologia degli acquisti fatti in precedenza e analizzati per mezzo del ML.

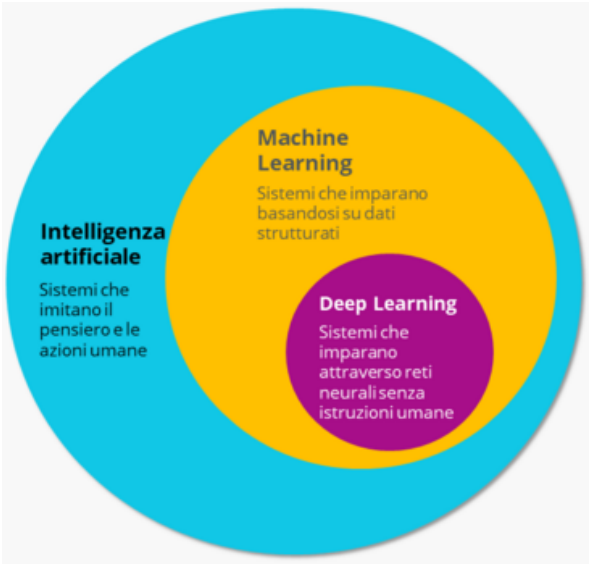

Figura 1.6: Diagramma di Venn di AI, ML e DL

E, ancora, pensiamo all'utilizzo dei motori di ricerca che, attraverso una o più parole chiave, restituiscono liste di risultati; queste ultime non sono altro che l'effetto di algoritmi di apprendimento automatico non supervisionato.

Sistemi di questo tipo vengono impiegati anche nel settore finanziario per la prevenzione delle frodi, dei furti di dati e di identità; in questi casi specifici, gli algoritmi imparano ad agire mettendo in correlazione eventi, abitudini degli utenti, preferenze di spesa e altri dati. Tutte informazioni attraverso le quali riescono, poi, ad identificare eventuali comportamenti anomali.

Gli esempi di applicazione dei sistemi di Machine Learning con apprendimento supervisionato nel settore della ricerca medica sono, allo stato attuale, innumerevoli, con previsioni sempre più accurate e tempestive, anche relativamente a diagnosi di tumori e di malattie rare.

Inoltre, una sua applicazione classica è quella del riconoscimento vocale, di cui sono dotati molti smartphone e molti dispositivi domotici (Figura 1.7). Ed è proprio nella relazione tra computer e linguaggio umano che si pone il focus di questo elaborato.

Figura 1.7: Esempio di tre diversi assistenti vocali

1.3 Machine Learning per il Natural Language Processing

Il Machine Learning per la comprensione del linguaggio umano e l'analisi del testo implica l'utilizzo di algoritmi di apprendimento automatico e Intelligenza Artificiale per capire il significato dei documenti di testo. Elaborare o comprendere i linguaggi umani, allo scopo di eseguire attività utili, non è particolarmente semplice ed immediato; per superare queste difficoltà ci viene in aiuto il Natural Language Processing (NLP).

1.3.1 Il Natural Language Processing

A differenza dei linguaggi di programmazione, che seguono regole ben precise e sono facilmente interpretabili dalle macchine, la lingua da noi utilizzata non è facilmente rappresentabile. Ma poiché interagiamo con le macchine quotidianamente (Figura 1.8), è necessario creare sistemi in grado di comprendere e rispondere all'uomo. Ed è qui che entra in gioco il Natural Language Processing. Sostanzialmente, questo filone di ricerca studia i sistemi informatici per l'analisi e l'elaborazione del linguaggio naturale; esso si concentra sui meccanismi del linguaggio, in modo da essere utilizzato su programmi eseguibili dalle macchine.

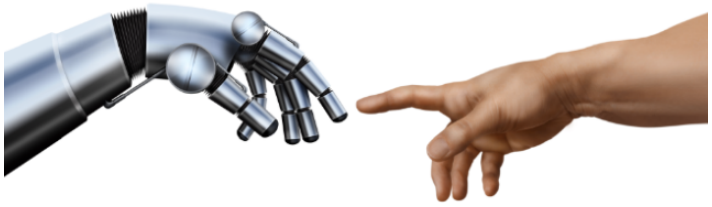

Figura 1.8: Interazione uomo-macchina

L'NLP si occupa principalmente di testi, intesi come sequenze di parole che, in una lingua, esprimono uno o più messaggi, mentre l'elaborazione del parlato, o riconoscimento vocale, è considerato un ambito a sé. Le sequenze di parole possono essere qualsiasi cosa che contenga testo: commenti sui social media, recensioni online, risposte a sondaggi, documenti finanziari, medici, legali e normativi.

Il ruolo cardine dell'elaborazione del linguaggio naturale e dell'analisi del testo è quello di migliorare, accelerare e automatizzare le funzioni di analisi del testo sottostanti e quelle che trasformano tale testo non strutturato in dati ed approfondimenti utilizzabili.

A rendere particolarmente difficoltosa la comprensione del linguaggio umano da parte di un algoritmo informatico contribuiscono le sue numerose ambiguità; infatti, per comprendere un determinato discorso, è necessario possedere anche una conoscenza della realtà e del mondo circostante. La semplice conoscenza del significato di ogni singola parola non è, infatti, sufficiente a interpretare correttamente il messaggio della frase; al contrario, può portare a comunicazioni contraddittorie e prive di significato. La ricerca in questo ambito si è focalizzata, in particolar modo, sui meccanismi che permettono alle persone di comprendere il contenuto di una comunicazione umana e sullo sviluppo di strumenti che possano fornire ai sistemi informatici la capacità di comprendere ed elaborare il linguaggio naturale.

1.3.2 Machine Learning per la comprensione testuale

Il dialogo tra uomo e macchina coinvolge diversi aspetti, quali la morfologia, la sintassi, la semantica, la pragmatica e il discorso nel suo complesso. Di conseguenza, sono numerosi i task di Natural Language Processing che automatizzano queste aree. Esempi di tali task sono:

- il riconoscimento della lingua;
- la scomposizione della frase in unità elementari;
- l'analisi semantica;
- la sentiment analysis.

Per svolgere questi task gli algoritmi di NLP utilizzano largamente il Machine Learning, in particolare l'uso di reti neurali artificiali. Come abbiamo avuto modo di vedere in precedenza, tutto ciò che riguarda l'Intelligenza Artificiale è ben legato e strettamente connesso. Nella Figura 1.9 possiamo vedere, a grandi linee, dove si posiziona il Natural Language Processing in relazione al Machine Learning e all'AI in generale.

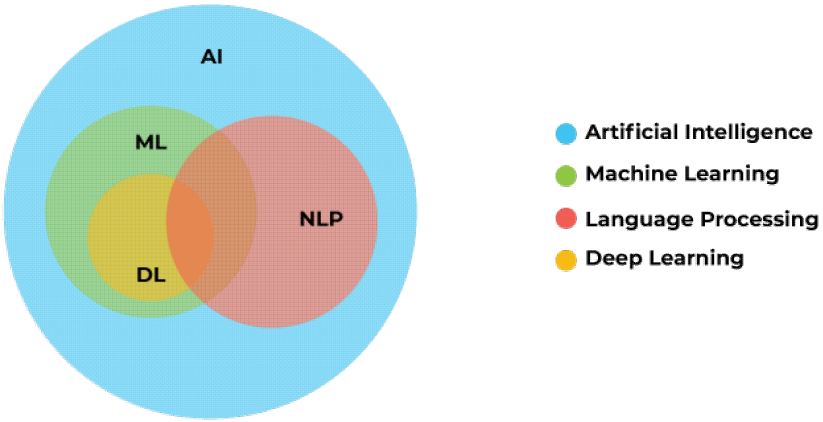

Figura 1.9: Diagramma di Venn di AI, ML, DL e NLP

1.3.3 Le applicazioni del Natural Language Processing

Le imprese sono sempre più interessate alle soluzioni di Natural Language Processing. Infatti, sono diverse le opportunità dei sistemi di elaborazione del linguaggio naturale per il business:

- l'analisi di email aziendali per riconoscere messaggi indesiderati o per classificare la posta in arrivo;
- l'estrazione di informazioni, da documenti di governance, quali report e procedure, per garantirne una rapida consultazione;
- i progetti per l'analisi di documenti amministrativi, quali fatture e contratti, e le soluzioni per l'analisi delle comunicazioni interne all'azienda;
- l'analisi di post sui Social Network per comprendere il sentiment degli utenti e consolidare la rinomanza dell'azienda;

- gli algoritmi per comprendere le query di navigazione nei siti web e reindirizzare correttamente la ricerca;
- le soluzioni per l'analisi di notizie giornalistiche, ad esempio, per riconoscere le fake news;
- l'estrapolazione, da documenti aziendali, di informazioni necessarie a rispondere ad una determinata domanda al fine di automatizzare l'assistenza ai clienti.

Questa ultima applicazione dell'NLP non è solo a scopo di esempio, ma verrà successivamente ripresa ed approfondita, essendo il fine del progetto alla base del presente elaborato. Infatti, come abbiamo visto, tramite il Machine Learning, il Natural Language Processing, e grazie all'aiuto di reti neurali artificiali, è possibile addestrare un modello a comprendere file testuali ed elaborare una risposta riguardante il più possibile il quesito che viene richiesto in input.

CAPITOLO 2

Il framework Haystack

In questo capitolo sarà introdotto il framework Haystack, che ha reso possibile il fine del progetto sviluppato nel presente elaborato. In particolare, daremo uno sguardo alla sua architettura, esamineremo i suoi componenti e, infine, analizzeremo qualche costrutto predefinito.

2.1 Introduzione ad Haystack

Haystack è un framework open source per la creazione di sistemi di ricerca, che agiscono efficacemente su raccolte di documenti di grandi dimensioni. I recenti progressi nel campo del Natural Language Processing hanno aperto la strada all'applicazione di tecniche che vedremo in seguito, come il Question Answering (QA), il retrieval e la summarization, in contesti del mondo reale. Quindi, Haystack è progettato per essere il ponte tra la ricerca e l'industria. Il logo di questo framework è visualizzato in Figura 2.1.

Figura 2.1: Il logo di Haystack

Haystack è progettato per aumentare il livello di qualità della ricerca tramite testo. Infatti, la ricerca per parole chiave è efficace ed appropriata per molte situazioni, ma il Machine Learning ha consentito ai sistemi di eseguire ricerche in base al significato delle parole, anziché sulla base della corrispondenza delle stringhe.

Tutto ciò è stato sviluppato dalla start up tedesca *Deepset*, fondata da *Malte Pietsch*, *Timo Möller* e *Milo Rusic*. Quest'ultimo, in qualità di amministratore delegato, ha affermato che Haystack può essere considerato il "pezzo mancante" tra i dati e le applicazioni software, offrendo agli utenti un modo per accedere, scrivere e persino creare dati utilizzando la lingua parlata. Inoltre, ha affermato che *Deepset* sta aiutando gli utenti a far comprendere meglio il linguaggio alle macchine, cercando di implementare il framework per ogni singolo tool software. Haystack, in altre parole, potrebbe essere, e in qualche caso sporadico già lo è, il motore che alimenta tutti i tipi di applicazioni ad attivazione vocale.

2.1.1 Le caratteristiche di Haystack

Diverse sono le caratteristiche che possono attribuirsi ad Haystack; queste, si possono riassumere in:

- *Natural Language Processing per la ricerca*: vengono scelti i componenti più adatti sulla base dell'NLP per eseguire il retrieval, il Question Answering ed altro ancora.
- *Ultimi modelli*: sono utilizzati transformer based model, come, ad esempio, *BERT* e *RoBERTa* (Figura 2.2), cambiando senza problemi quando ne vengono pubblicati di nuovi.
- *Database flessibili*: vengono caricati i dati e vengono eseguite query da una vasta gamma di database, quali Elasticsearch, FAISS, SQL ed altri.
- *Scalabilità*: il sistema si adatta per gestire milioni di documenti e per distribuire quest'ultimi tramite l'API REST.
- *Adattamento del dominio*: vengono utilizzati tutti gli strumenti necessari per annotare esempi, raccogliere feedback degli utenti, valutare componenti ed eseguire il finetuning dei modelli.

Figura 2.2: Il logo di Google Bert

Con lo sviluppo di nuovi modelli di elaborazione del linguaggio, sono possibili anche nuovi stili di ricerca. In Haystack, si possono creare sistemi che eseguano:

- *Question answering*: è possibile trovare risposte dettagliate in un certo numero di documenti partendo da semplici domande in linguaggio naturale.
- *Summarization o riassunto*: è possibile recuperare i documenti in base al significato della query e non solo sulla base delle sue parole chiave.
- *Ricerca documenti*: facendo una domanda generica, si possono ottenere i documenti più rilevanti.
- *Question generation o generazione delle domande*: tramite un file testuale come input, possono essere restituite le domande generate a cui il documento può rispondere.

Questo è solo un sottoinsieme delle caratteristiche e dei tipi di sistemi che possono essere creati nell'ampia varietà dei modelli che offre Haystack.

2.2 I tre diversi livelli

Il framework Haystack è orientato alla creazione di specifiche Pipeline di ricerca personalizzabili e pronte per l'uso. Ci sono tre diversi livelli nei quali si può interagire tramite gli svariati componenti che Haystack mette a disposizione. Questi livelli sono:

1. i Node;
2. le Pipeline;
3. le REST API.

2.2.1 I Node

I Node sono componenti principali che, tra le varie cose, elaborano il testo in input. Haystack offre molti Node che eseguono diversi tipi di analisi del testo. Essi sono spesso alimentati dagli ultimi modelli di trasformatori. Dal punto di vista del codice, sono semplici classi Python, i cui metodi possono essere direttamente richiamati. Ad esempio, se si desidera rispondere a delle domande con un modello `Reader`, tutto ciò che bisogna fare è fornire dei documenti ed una semplice domanda in input. Inoltre, i Node, possono eseguire fasi di pre-elaborazione come la pulizia o la divisione del testo.

Molto importante, però, è il fatto che tutti i Node siano progettati per essere utilizzati in una Pipeline. Infatti, un Node prende come input l'output del Node precedente, o, in molteplici casi, quello dei Node precedenti.

Lavorare a questo livello con i Node di Haystack è l'approccio "pratico" standard ed offre un modo molto diretto di manipolare gli input e di controllare gli output. Ciò può essere utile per l'esplorazione, la prototipazione ed il debug. Nel Listato 2.1 è possibile vedere un esempio di un Node.

```
reader = FARMReader(model="deepset/roberta-base-squad2")

result = reader.predict(
    query = "Che paese ha come capitale Roma?",
    documents = documents,
    top_k = 10
)
```

Listato 2.1: Esempio di un Node composto dal `Reader`

2.2.2 Le Pipeline

Haystack si basa sull'idea che i sistemi composti siano molto più della somma delle loro parti. Combinando la potenza del Natural Language Processing attraverso diversi Node, gli utenti possono creare sistemi potenti e personalizzabili. La Pipeline è la chiave per far funzionare questo approccio modulare, combinando, appunto, svariati Node.

Infatti, quando si aggiungono Node ad una Pipeline è possibile definire in che modo i dati fluiscano attraverso il sistema e quali Node eseguino la fase di elaborazione. L'idea è quella di creare un grafo aciclico diretto (DAG), in cui ogni nodo del grafo sia un elemento costruttivo dell'architettura, come il `Reader` o il `Retriever`. Grazie a questa idea, il flusso dei dati sarà semplificato e facilmente gestibile; ciò consente anche opzioni di instradamento complesse, come quelle che coinvolgono i Node decisionali. Nel Listato 2.2 è possibile vedere un esempio di una Pipeline composta da due Node.

```
p = Pipeline()
p.add_node(component=retriever, name="Retriever", inputs=["Query"])
p.add_node(component=reader, name="Reader", inputs=["Retriever"])
result = p.run(query="Come si chiama il fiume di Parigi?")
```

Listato 2.2: Esempio di una Pipeline composta da un `Reader` ed un `Retriever`

2.2.3 Le REST API

Per distribuire un sistema di ricerca, si ha bisogno di più di un semplice script Python. Infatti, è necessario un servizio che possa rimanere sempre attivo, gestire le richieste non appena arrivano ed essere anche richiamabile da molte applicazioni diverse. Per questo, Haystack viene fornito con una REST API progettata per funzionare in ambienti di produzione. L'utilizzo tramite un'API può avvantaggiare gli sviluppatori che desiderano implementare, ad esempio, la funzionalità di Question Answering nei loro progetti, sia per app Web che per app mobile.

Utilizzando delle configurazioni come quelle mostrate nel Listato 2.3 è possibile caricare Pipeline da file YAML, interagire con esse tramite richieste HTTP e connettere Haystack ad un'interfaccia grafica rivolta all'utente finale.

```
curl -X "POST" \
  "http://127.0.0.1:8000/query"\
  -H "accept: application/json" \
  -H "Content-Type: application/json" \
  -d {
  "query": "Come si chiama il padre di Arya Stark?",
  "params": {}
}
```

Listato 2.3: Esempio di una REST API

2.3 I componenti di Haystack

Haystack fornisce tutti gli strumenti necessari per creare un'applicazione basata sul Natural Language Processing personalizzata che funzioni per svariati compiti. Ciò include diverse funzionalità, partendo dalla prototipazione e arrivando al deploy. Di seguito troviamo un elenco delle componenti principali di Haystack:

- i `Documents`, le `Answers` e le `Labels`;
- il `Retriever`;
- il `Reader`;
- i Document Stores;
- le Pipelines;
- le REST API.

2.3.1 `Documents`, `Answers` e `Labels`

In Haystack ci sono diverse classi principali che vengono regolarmente utilizzate in molti ambiti distinti. Queste sono classi che trasportano i dati attraverso il sistema. Gli utenti, probabilmente, interagiranno con esse come input oppure output della loro Pipeline.

La classe `Document`

La classe `Document` è l'oggetto principale che concerne i dati in Haystack. Essa memorizza dati testuali, tabulari o di immagine, insieme al proprio ID ed ai propri metadati. Può anche contenere informazioni create nella Pipeline, incluso il punteggio di affidabilità del modello o l'embedding sviluppato per esso durante l'indicizzazione.

La classe `Answer`

La classe `Answer` contiene tutte le informazioni sulla previsione fatta da un modello `Reader` o da una Pipeline con un modello `Reader` alla fine. Essa conterrà la stringa di risposta, il punteggio di affidabilità del modello, il contesto attorno alla risposta, l'ID del documento ed i relativi metadati. Si possono trovare, anche, gli offset di inizio e di fine della stringa di risposta, rispetto al testo completo del documento.

La classe `Label`

Una classe `Label` contiene tutte le informazioni rilevanti per l'estrazione di un documento o l'annotazione per il Question Answering. Viene generalmente utilizzata per la valutazione e può essere recuperata da un archivio di documenti tramite `store.get_all_labels()`. Tuttavia, negli scenari in cui potrebbe esserci più di un'annotazione per query, usando `store.get_all_labels_aggregated()`, verrà restituito un elenco di oggetti MultiLabel che, a sua volta, contiene un elenco di Labels. Le Label, inoltre, vengono restituite quando viene chiamato un Document Store contenente dati etichettati.

2.3.2 `Retriever`

Il `Retriever` è uno dei Node principali forniti da Haystack. Esso esegue il retrieval dei documenti scorrendo un archivio e restituendo una serie di documenti candidati. Questi sono i più rilevanti per la query fornita in input. Se utilizzato in combinazione con un `Reader`, può vagliare rapidamente i documenti irrilevanti, evitando di svolgere più lavoro del necessario ed accelerando il processo di interrogazione. In poche parole, funziona da filtro per alleggerire il costo computazionale totale del processo.

Come è possibile notare nel Listato 2.4, il `Retriever` è sempre accoppiato con il Document Store. Infatti, è strettamente necessario specificare quest'ultimo durante l'inizializzazione del `Retriever`.

```
1 from haystack.nodes import BM25Retriever
2 retriever = BM25Retriever(document_store)
```

Listato 2.4: Esempio di inizializzazione di un `Retriever` chiamato `BM25Retriever`

2.3.3 `Reader`

Anche il `Reader`, come il `Retriever`, è uno dei Node principali forniti da Haystack. Esso riceve una query ed un insieme di documenti come input e restituisce una o più risposte, selezionando un intervallo di testo all'interno dei documenti. Il `Reader` è anche noto come sistema di Question Answering a dominio aperto nel Machine Learning, contenendo tutti i componenti necessari, tra cui il caricamento dei pesi del modello, la tokenizzazione e l'embedding computation.

L'unico grande problema del `Reader` è il fatto che richieda una GPU per funzionare rapidamente, dato che, per essere eseguito, richiede parecchie risorse computazionali. Nel Listato 2.5, è presente un esempio di un `Reader` che utilizza un modello basato su *RoBERTa*.

```
1 from haystack.nodes import FARMReader
2 model = "deepset/roberta-base-squad2"
3 reader = FARMReader(model, use_gpu=True)
```

Listato 2.5: Esempio di un `Reader` chiamato `FARMReader`

2.3.4 Document Store

Un Document Store è, semplificando, un database che memorizza i testi ed i metadati dell'utente e li fornisce al `Retriever` al momento della query. Il modo più comune per utilizzare un Document Store in Haystack è recuperare i documenti utilizzando, appunto, un `Retriever`. È necessario fornire un Document Store come argomento per l'inizializzazione di esso. Si noti che il `Retriever` funziona come un Node, a differenza del Document Store.

A seconda che si voglia utilizzare Elasticsearch, Opensearch, FAISS, o altri server di ricerca, l'inizializzazione di un nuovo Document Store all'interno di Haystack è semplice e si può racchiudere in poche righe di codice. Di seguito, nel Listato 2.6, è presente un esempio di Document Store implementato con l'utilizzo di Elasticsearch.

```
1 from haystack.document_stores import ElasticsearchDocumentStore
2 document_store = ElasticsearchDocumentStore()
3 dicts = [
4     {
5         "content": DOCUMENT_TEXT_HERE,
6         "meta": {"name": DOCUMENT_NAME, ...}
7     }, ...
8 ]
9 document_store.write_documents(dicts)
```

Listato 2.6: Esempio di una scrittura di un Document Store utilizzando Elasticsearch

2.3.5 Pipeline

Per costruire moderne Pipeline di ricerca, si ha bisogno di due fattori: potenti elementi costitutivi ed un modo semplice per unirli insieme. Le Pipeline sono state create proprio per questo scopo e consentono molti scenari di ricerca. Esse si possono classificare in due grandi gruppi:

- indexing Pipeline;
- querying Pipeline.

Questi due gruppi verranno esaminati in dettaglio nelle prossime sottosezioni.

Indexing Pipeline

Le Pipeline di indicizzazione preparano i file per la ricerca. Il loro obiettivo principale è convertire i file in documenti Haystack in modo che possano essere salvati in un Document Store.

Esse iniziano, spesso, con un convertitore di file o un classificatore di file seguito da convertitori di file. Come si può notare nel Listato 2.7, l'ultimo Node in una Pipeline di indicizzazione è sempre un Document Store.

Aggiungendo Node nel mezzo di una pipeline di indicizzazione è possibile anche eseguire ulteriori passaggi di elaborazione sui documenti, prima che vengano salvati nel Document Store. Ad esempio, è possibile aggiungere un preprocessore per eseguire la divisione e la pulizia del documento.

```
1 from haystack.pipelines import Pipeline
2 p = Pipeline()
3 p.add_node(component=text_converter, name="TextConverter", inputs=["File"])
```

```
p.add_node(component=preprocessor, name="PreProcessor", inputs=["TextConverter"])
p.add_node(component=doc_classifier, name="DocClassifier",inputs=["PreProcessor"])
p.add_node(component=doc_store, name="DocumentStore", inputs=["DocClassifier"])

p.run(file_paths=["filename.txt"])
```

Listato 2.7: Esempio di una indexing Pipeline

Querying Pipeline

Le Pipelines di query, invece, eseguono ricerche su una serie di documenti che sono stati indicizzati in un Document Store. Il primo Node in una Querying Pipeline accetta sempre query come input. Come output, invece, sarà presente la risposta alla query.

In Figura 2.3 possiamo notare il modo in cui le due tipologie di Pipeline si interfacciano tra loro, come detto in precedenza, attraverso il Document Store.

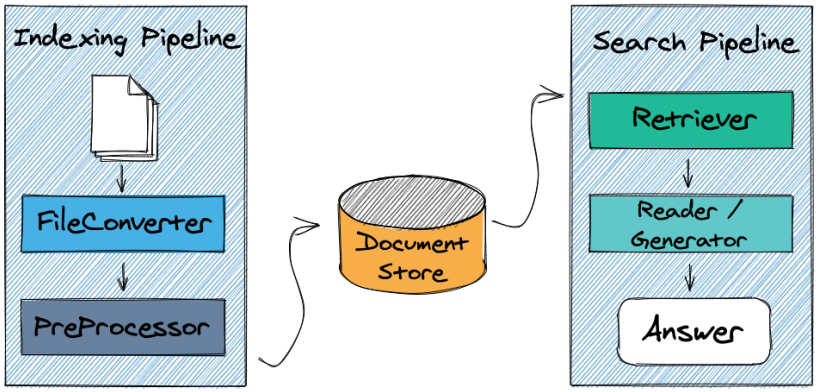

Figura 2.3: Esempio di architettura in Haystack

2.4 Hugging Face

Hugging Face (Figura 2.4) è un'azienda con la missione di democratizzare l'accesso ai sistemi di Natural Language Processing, contribuendo allo sviluppo di tecnologie che migliorino il mondo attraverso le Intelligenze Artificiali. Non si può parlare di Haystack senza citare Hugging Face.

Sostanzialmente, Hugging Face è un repository in cui possiamo trovare diverse librerie basate su Python. In particolare, il focus è incentrato sulla libreria Transformers, che offre un'API pensata per l'utilizzo di molte architetture di trasformatori, come le già citate *BERT* e *RoBERTa*. Grazie a questa libreria si possono ottenere risultati all'avanguardia su una varietà di attività di Natural Language Processing, come la classificazione del testo, l'estrazione di informazioni ed il Question Answering. Tali architetture vengono pre-addestrate con diversi set di pesi.

Hugging Face mette a disposizione svariati modelli allenati e pronti all'uso, in oltre 100 lingue differenti, compreso, ovviamente, l'italiano.

2.5 Esempi di Pipeline predefinite

Haystack viene fornito con una serie di Pipeline predefinite, che si adattano alla maggior parte dei modelli di ricerca standard, consentendo, ad esempio, di creare un sistema di Question Answering in pochissimo tempo. Di seguito verranno trattate le più importanti.

Figura 2.4: Il logo di Hugging Face

2.5.1 `ExtractiveQAPipeline`

Il Question Answering estrattivo ha il compito di cercare, in un'ampia raccolta di documenti, un intervallo di testo che risponda ad una domanda. L'`ExtractiveQAPipeline` (Listato 2.8) combina `Retriever` e `Reader` in modo tale che il `Retriever` analizzi un database e restituisca solo i documenti che ritiene più rilevanti per la query, mentre il `Reader` accetti i documenti restituiti dal `Retriever` e selezioni un intervallo di testo come risposta alla query.

L'output della Pipeline è un dizionario Python, il quale fornisce informazioni aggiuntive come il contesto da cui è stata estratta la risposta e la confidenza del modello nell'accuratezza della risposta estratta.

```
1 pipeline = ExtractiveQAPipeline(reader, retriever)
2
3 query = "Quanto tempo si impiega per andare da Roma a Milano?"
4 result = pipeline.run(query=query, params={"Retriever": {"top_k": 10},
5     "Reader": {"top_k": 1}})
```

Listato 2.8: Esempio di una `ExtractingQAPipeline`

2.5.2 `DocumentSearchPipeline`

In genere passiamo l'output del `Retriever` ad un altro componente. Tuttavia, possiamo utilizzarlo anche autonomamente per la ricerca semantica al fine di trovare i documenti più rilevanti per la nostra query.

`DocumentSearchPipeline` incorpora il `Retriever` in una Pipeline. Si noti che questo incorporamento non fornisce al `Retriever` funzionalità aggiuntive, ma consente, invece, di utilizzarlo in modo coerente con altri componenti di Haystack, tramite la stessa sintassi familiare. Nel Listato 2.9 possiamo vederne un esempio.

```
1 pipeline = DocumentSearchPipeline(retriever)
2 query = "Dammi informazioni riguardo al calcio."
3 result = pipeline.run(query, params={"Retriever": {"top_k": 5})
```

Listato 2.9: Esempio di una `DocumentSearchPipeline`

2.5.3 `SearchSummarizationPipeline`

Il `Summarizer` aiuta a dare un senso all'output del `Retriever`, creando un riepilogo dei documenti reperiti. Ciò è utile per eseguire un rapido controllo di integrità e confermare la qualità dei documenti candidati suggeriti dal `Retriever`, senza dover ispezionare ogni documento singolarmente. Nel Listato 2.10 è possibile notare un esempio di

`SearchSummarizationPipeline`.

```
pipeline = SearchSummarizationPipeline(summarizer=summarizer,
    retriever=retriever, generate_single_summary=True)
result = pipeline.run(query="Descrivi il capo dello stato.",
    params={"Retriever": {"top_k": 5}})
```

Listato 2.10: Esempio di una `SearchSummarizationPipeline`

2.5.4 `FAQPipeline`

`FAQPipeline` (Listato 2.11) racchiude il `Retriever` in una Pipeline e ne consente l'utilizzo per il Question Answering con i dati delle FAQ. Rispetto ad altri tipi di risposta alle domande, il QA in stile FAQ è significativamente più veloce. Tuttavia, è in grado di rispondere solo alle domande identiche o molto simili alle FAQ, poiché questo tipo di QA confronta le stringhe di testo con le domande esistenti nel dataset delle FAQ.

```
pipeline = FAQPipeline(retriever=retriever)
result = pipeline.run(query=query, params={"Retriever": {"top_k": 1})
```

Listato 2.11: Esempio di una `FAQPipeline`

2.5.5 `QuestionGenerationPipeline`

La versione più semplice di una `QuestionGenerationPipeline` accetta un documento come input e genera domande a cui il documento può rispondere. L'output di questa Pipeline sarà, dunque, un array composto dalle domande che ha generato. Come si può vedere nel Listato 2.12, la Pipeline è composta da un solo Node.

```
question_generation_pipeline = QuestionGenerationPipeline(question_generator)
result = question_generation_pipeline.run(documents=[document])
```

Listato 2.12: Esempio di una `QuestionGenerationPipeline`

CAPITOLO 3

Analisi dei requisiti

Questo capitolo è dedicato a tutto ciò che riguarda l'inizio di un progetto, partendo dalla raccolta di informazioni con annesso studio della fattibilità, passando per l'analisi dei requisiti funzionali e non funzionali ed arrivando a dare uno sguardo agli strumenti utilizzati per la programmazione.

3.1 Introduzione al progetto

Nel mondo attuale, caratterizzato da una estrema varietà e quantità di contenuti espressi nel linguaggio naturale, l'uso dell'Intelligenza Artificiale assume rilevanza strategica, favorendo la realizzazione di soluzioni innovative per l'elaborazione, la comprensione e la produzione in maniera automatica di dati testuali. In particolare, negli ultimi anni, si è assistito alla nascita di nuovi approcci, che integrano l'elaborazione del linguaggio naturale con gli algoritmi di Deep Learning, producendo risultati straordinari in differenti scenari applicativi.

Proprio per questo ambito nasce l'idea del progetto associato alla presente tesi, ovvero implementare, tramite Haystack, un modello di Natural Language Processing, addestrato su un dominio chiuso, che risponda a delle domande specifiche riguardo tale ambito. Successivamente, il modello sarà testato per capire se le performance siano accettabili, o si possano migliorare, constatando le criticità.

Per sviluppare script di Machine Learning è necessario possedere nozioni di programmazione in Python, visto che tale linguaggio è quello più usato quando si parla di apprendimento automatico.

In ogni caso, prima di vedere la progettazione con il relativo ambiente di sviluppo e tutti i suoi strumenti, concentriamoci sull'analisi dei requisiti per il fine in questione.

3.2 Raccolta informazioni

Quando si ha un'idea è meglio metterla in pratica al più presto per evitare che qualcuno lo faccia prima di noi. Ma ancor prima di attuarla è bene prendersi qualche momento per assicurarsi che possa essere competitiva sul mercato. Che si voglia sviluppare un progetto su più livelli o un task più contenuto, la prima cosa da fare è partire da alcune buone prassi. Una di queste è, sicuramente, l'analisi dei requisiti.

3.2.1 Studio fattibilità

Uno dei punti di partenza è capire se lo sviluppo del software è profittevole: lo studio di fattibilità aiuta a comprendere se il progetto renderà, partendo dalla definizione delle spese e dagli sforzi tecnici che gli sviluppatori o le aziende dovranno sostenere. Successivamente dovrà essere affrontata un'analisi del mercato; bisognerà capire, infatti, qual'è lo stato dell'arte e quali sono gli elementi con cui differenziarsi.

Come abbiamo visto in precedenza, un software open source che sia in grado di aiutare nella realizzazione del nostro task è presente sul mercato; si tratta di Haystack. La sua libreria così vasta e piena di risorse è in grado di soddisfare a pieno tutte le nostre esigenze. Inoltre, come si può notare nella Figura 3.1, il suo sito web è ricco di tutorial e linee guida che ci possono assistere nello sviluppare all'interno del nostro dominio chiuso.

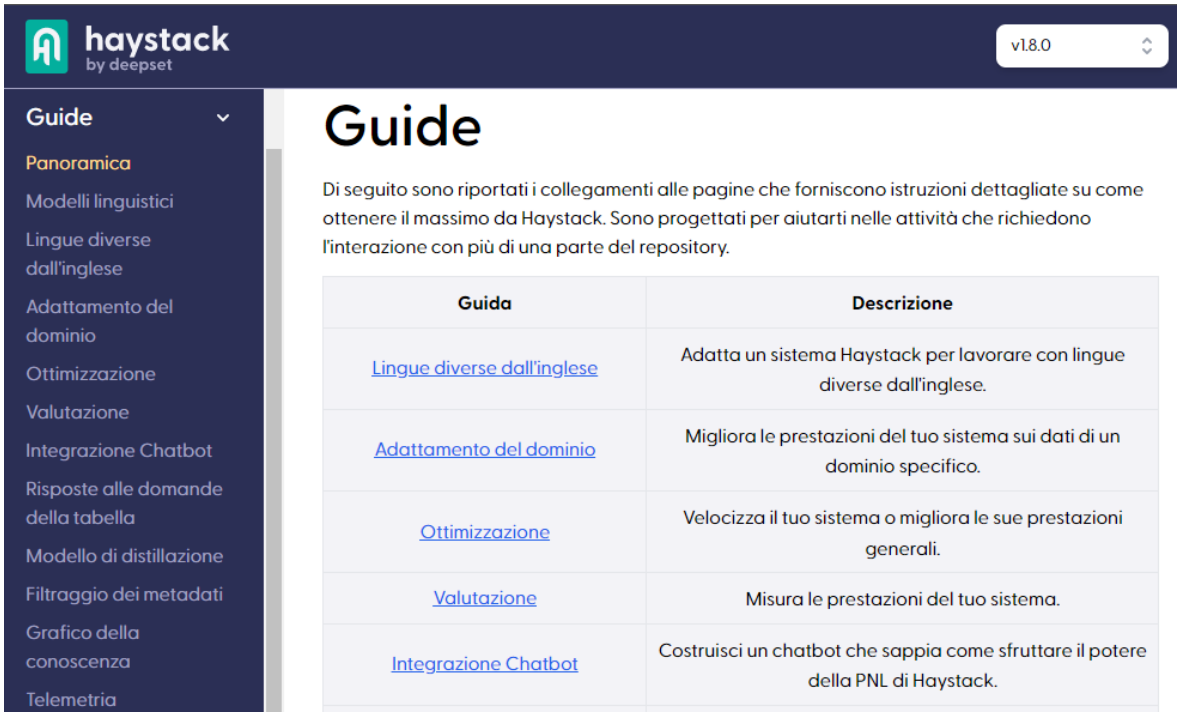

Figura 3.1: Un elenco delle guide messe a disposizione da Haystack

Com'era prevedibile, non esistono dei siti web, delle applicazioni o, più in generale, delle risorse open source che siano in grado di affrontare il problema senza modifiche o senza la programmazione nel complesso. Perciò, si è deciso di provare ad affrontare il problema, creando un codice che, presi come input dei documenti e delle domande, sappia rispondere logicamente e che fornisca anche il livello di accuratezza delle risposte.

3.3 Descrizione della componente dati

Il formato dei dati che sono stati resi disponibili per questo lavoro non è un semplice JavaScript Object Notation (JSON). Infatti, i documenti forniti dall'azienda committente sono dello stesso formato del dataset SQuAD, introdotto nella prossima sottosezione.

3.3.1 SQuAD e DPR

Lo Stanford Question Answering Dataset (SQuAD) rappresenta una forte sfida per i modelli NLP. Esso è formato da un insieme di coppie di domanda-risposta. Come suggerisce

il suo nome, SQuAD si concentra sul compito di rispondere alle domande.

Il processo di creazione del dataset è il seguente:

- A partire da documenti testuali grezzi vengono selezionati i singoli paragrafi, assicurandosi il filtraggio delle sezioni di testo più piccole.
- Successivamente, per ogni paragrafo selezionato, vengono formulate cinque domande riguardanti il contenuto del testo in questione.

Purtroppo Haystack, per il fine che ci siamo prefissati, non elabora il formato SQuAD. Esso, però, mette a disposizione un comodo script, chiamato `squad_to_dpr.py`, che effettua una semplice conversione nel formato migliore per leggere il dataset. In questo caso, tale formato è il formato Dense Passage Retrieval (DPR).

Nel Listato 3.1 possiamo vedere un esempio dei due formati appena citati.

```
# SQuAD format
{
    version: "Versione del dataset"
    data:[
        {title: "Titolo dell'articolo di Wikipedia"
            paragraphs:[
                {context: "Paragrafo dell'articolo"
                    qas:[
                        {d: "ID di una coppia di domanda-risposta"
                            question: "Question"
                            answers:[
                                {"answer_start": "Posizione della risposta"
                                    "text": "Risposta"}
                            ],
                            is_impossible: (not in v1)
                        }
                    ]
                }
            ]
        }
    ]
}

# DPR format
[
    {
        "question": "....",
        "answers": ["...", "...", "..."],
        "positive_ctxs": [{
            "title": "...",
            "text": "...."
        }],
        "negative_ctxs": ["..."],
        "hard_negative_ctxs": ["..."]
    },
    ...
]
```

Listato 3.1: Esempio dei formati SQuAD e DPR

3.4 Requisiti

La raccolta e l'analisi dei requisiti è uno dei processi di pianificazione ed è finalizzato all'acquisizione di tutte le informazioni necessarie per configurare gli obiettivi di progetto e le modalità della loro definizione. I requisiti si possono suddividere in requisiti funzionali e requisiti non funzionali.

3.4.1 Requisiti funzionali

I requisiti funzionali descrivono le interazioni tra il sistema ed il suo ambiente, indipendentemente dalla sua implementazione. L'ambiente include l'utente ed ogni altro sistema esterno. Nel caso del presente progetto, i requisiti funzionali sono i seguenti:

- *Addestramento di un modello pre-allenato*: preso il modello pre-allenato più adatto, individuato nel repository Huggingface, il sistema dovrà addestrarlo sul nostro dominio chiuso, per adattarlo alla risposta di specifiche domande riguardanti il nostro ambito. Questo processo è, anche, chiamato fine-tuning.
- *Risposta alle domande*: il sistema dovrà essere in grado, ricevuta una domanda in input, di rispondere ad essa. Saranno visualizzate le "k" risposte migliori in ordine di pertinenza.
- *Vista delle performance*: tramite diverse metriche, l'utente dovrà poter visualizzare le performance del modello per capire se sia accettabile o se possa migliorare.

3.4.2 Requisiti non funzionali

I requisiti non funzionali descrivono gli aspetti del sistema che non sono direttamente legati al comportamento, cioè alle funzionalità. Essi includono una grande varietà di richieste che si riferiscono a diversi aspetti del sistema. I requisiti non funzionali relativi al nostro progetto sono i seguenti:

- *Affidabilità*: l'affidabilità è la capacità di un sistema o di una componente di fornire la funzione richiesta sotto determinate condizioni. La persona che utilizzerà il nostro script vuole che faccia esattamente ciò che è richiesto, per cui sarà necessario che lo script stesso sia progettato in modo affidabile.
- *Scalabilità*: in informatica, la scalabilità è la caratteristica di un sistema software o hardware di essere facilmente adattabile. In questo caso, il nostro sistema deve essere in grado di cambiare agevolmente il modello pre-addestrato iniziale e, inoltre, di addestrare con diversi set di dati.
- *Leggibilità del codice*: essendo una parte di un servizio più ampio, il codice relativo al sistema deve poter essere letto e compreso anche da altri sviluppatori. Quindi, è buona regola programmare in maniera leggibile e comprensibile, ad esempio commentando spesso e associando a variabili e funzioni dei nomi che ne indichino la semantica.

3.5 Strumenti per la programmazione

3.5.1 Il linguaggio di programmazione Python

Come già accennato in precedenza, il linguaggio attualmente più utilizzato per implementare progetti basati sul Machine Learning è Python (Figura 3.2). La comunità di sviluppo

Figura 3.2: Logo di Python

che vi gravita intorno è molto ampia ed è disponibile una grande quantità di moduli e di librerie aggiuntive, soprattutto per l'elaborazione dei dati.

Per semplificare l'installazione di Python è stata utilizzata la distribuzione Anaconda, che, oltre all'interprete del linguaggio, include i principali pacchetti usati per il calcolo scientifico. Questa distribuzione offre, anche, un ambiente di sviluppo molto funzionale ed efficace, costituito dai Notebook Jupyter che permettono lo sviluppo e l'esecuzione interattiva del codice.

3.5.2 Project Jupyter

Project Jupyter (Figura 3.3) è un progetto con l'obiettivo di sviluppare software open source e servizi per l'elaborazione interattiva in diversi linguaggi di programmazione. È stato scorporato da IPython nel 2014 da *Fernando Pérez* e *Brian Granger*. Il nome del progetto Jupyter è un riferimento ai tre linguaggi di programmazione principali da esso supportati: Julia, Python ed R. Il suo nome ed il suo logo sono un omaggio alla scoperta, da parte di Galileo, delle lune di Giove.

Figura 3.3: Logo di Jupyter

Project Jupyter ha sviluppato e supportato prodotti di calcolo interattivo, quali JupyterHub, JupyterLab e Jupyter Notebook.

Jupyter Notebook

Jupyter Notebook è un'applicazione Web che permette di creare e condividere documenti testuali interattivi, contenenti oggetti quali equazioni, grafici e codice sorgente eseguibile. Jupyter è diventato uno standard per data scientist poiché offre la possibilità di realizzare, documentare e condividere analisi di dati all'interno di un framework. Quest'ultimo supporta:

- operazioni di data cleaning e di data trasformation, simulazioni numeriche, Machine Learning ed altro;
- l'esecuzione di applicazioni Scala e Python su piattaforme di big data, grazie all'integrazione con Apache Spark.

Jupyter Notebook è costituito da una serie di celle, disposte in una sequenza lineare. Come si può vedere nella Figura 3.4, ogni cella può contenere testo o codice. Il testo viene formattato in MarkDown, un linguaggio di markup con la semplice sintassi di formattazione di un testo normale. Le celle di codice, invece, contengono termini nella lingua di programmazione

associata con il notebook specifico. Nel nostro caso, per ovvie ragioni, abbiamo utilizzato Jupyter Notebook mediante il linguaggio interpretato Python.

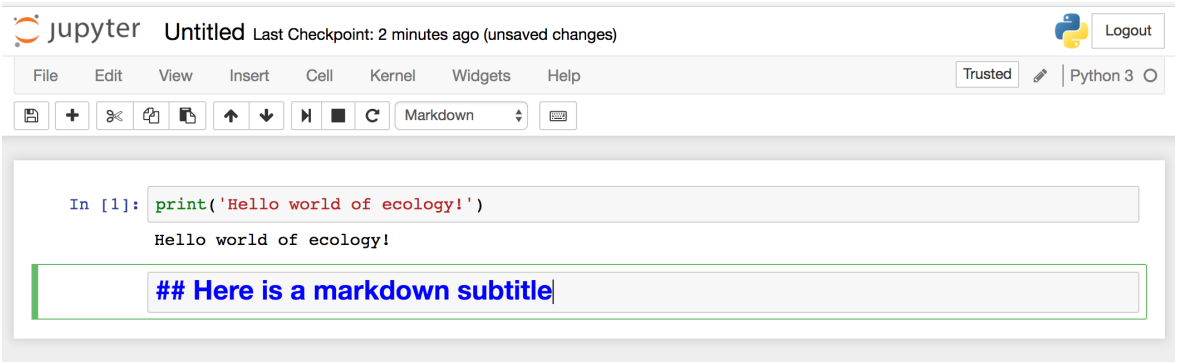

Figura 3.4: Esempio dell'interfaccia di Jupyter Notebook

CAPITOLO 4

Progettazione delle attività di re-engineering

In questo quarto capitolo vedremo la progettazione di tutti i componenti che caratterizzeranno il lavoro svolto; in particolare, parleremo della pre-elaborazione, dei modelli utilizzati, del Document Store scelto, del Retreiver che abbiamo impiegato e delle funzioni di similarità selezionate per il nostro obiettivo.

4.1 Componenti principali per la progettazione

Riguardo alla progettazione del lavoro oggetto della presente tesi, è possibile identificare diverse componenti principali che caratterizzano l'operatività e la messa in atto. Tali componenti, che verranno illustrate in dettaglio nei prossimi paragrafi, si possono riassumere nei seguenti macro-argomenti:

- il Modello di Huggingface;
- il Document Store;
- il `Retriever`;
- la pre-elaborazione;
- le Pipeline;
- le funzioni di similarità.

4.2 Modello di Huggingface

Prima di capire quale Pipeline progettare, che `Retriever` utilizzare o di che Document Store servirsi, è bene studiare e capire qual è il modello di Huggingface migliore per il nostro scopo. Infatti, esistono più di ottantacinque mila modelli pronti all'uso su questa speciale piattaforma, spaziando dalla classificazione di immagini, al riconoscimento di file audio, al riassunto di testi, e molto altro ancora.

Il nostro caso specifico, però, si occupa di NLP basandosi su *BERT* o *RoBERTa* e, filtrando sul sito la ricerca dei modelli sulla base di questo ambito, possiamo vedere una drastica, ma prevedibile, diminuzione dei modelli. Sono, infatti, poco meno di novemila. Questo numero, però, è composto per la quasi totalità da modelli preaddestrati sulla lingua inglese. Il nostro dataset, ed il progetto in generale, tuttavia, si basano sulla lingua italiana, ragion per

cui, purtroppo, siamo molto limitati nella scelta. I modelli totalmente in italiano si contano sulle dita di una mano e si vede che sono molto acerbi. Diversamente, i multilingua, che comprendono l'italiano, sono circa due dozzine; perciò, è opportuno valutare se scegliere un modello totalmente in italiano oppure uno multilingua.

La scelta è ricaduta sul modello in Figura 4.1 per i seguenti motivi; in primis, il grande numero di download mensili è sinonimo di grande affidabilità, ed acquista maggiore importanza se rapportato agli altri modelli; inoltre, esso utilizza il formato DPR, che risulta essere lo stesso del nostro dataset.

Quindi, si è deciso di utilizzare il `dpr-ctx_encoder-bert-multilingual`. Questo modello verrà dato in input al `Retriever` per migliorare la qualità dei testi che verranno selezionati come risposta alla query in input.

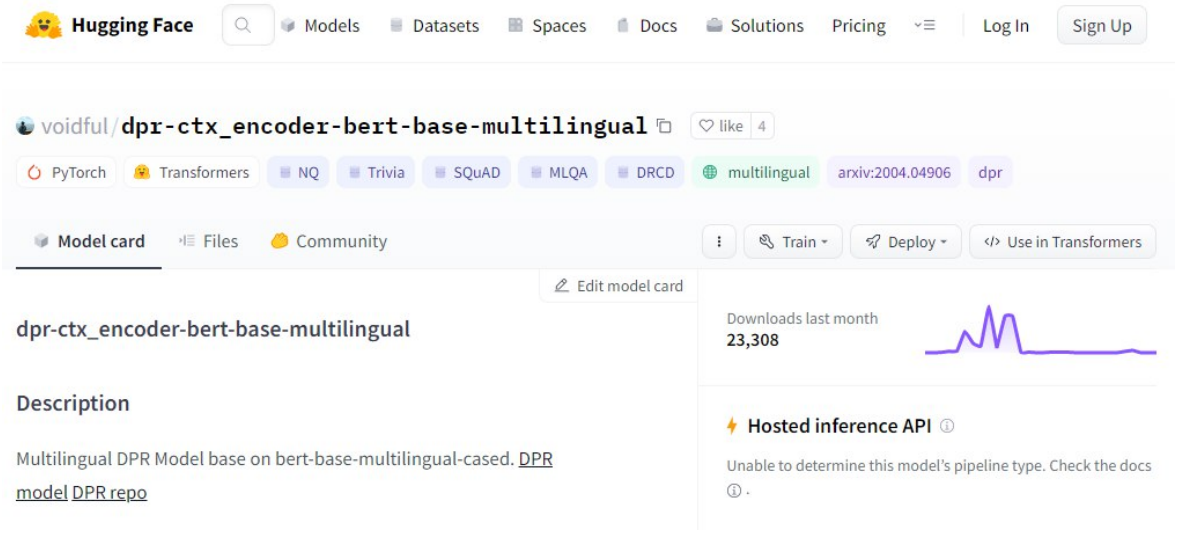

Figura 4.1: Il modello di Huggingface utilizzato

4.3 Document Store

Come abbiamo visto nella sezione 2.3.4, un Document Store è un database progettato per memorizzare e cercare dati, come, ad esempio, documenti di tipo JSON; questa tipologia di dati rappresenta l'alternativa più popolare ai database relazionali. Infatti, offre una grande varietà di vantaggi, tra cui:

- un modello di dati intuitivo con cui gli sviluppatori possono lavorare in modo facile e veloce;
- uno schema flessibile che consenta al modello di dati di evolversi in base alle esigenze dell'applicazione;
- la possibilità di ridimensionare orizzontalmente.

Grazie a questi vantaggi, i Document Store sono database generici che possono essere utilizzati in una varietà di casi d'uso e di settori.

Poichè essi sono dei database non relazionali (o NoSQL), invece di archiviare i dati in righe e colonne fisse, utilizzano documenti flessibili. Di conseguenza, appare chiaro il motivo per cui il formato più utilizzato per archiviare dati nei Document Store è JSON, essendo quest'ultimo un formato semplice e molto versatile.

Esistono diversi tipi di Document Store, come, ad esempio, Elasticsearch, OpenSearch, Milvus, FAISS e Weaviate. Riguardo al Document Store più adatto al nostro ambito, la scelta è ricaduta sull'Elasticsearch Document Store.

4.3.1 Elasticsearch

Elasticsearch è un archivio di documenti distribuito. Quando un documento viene archiviato, esso è indicizzato ed è completamente ricercabile quasi in tempo reale, entro pochi decimi di secondo. Elasticsearch utilizza una struttura di dati chiamata indice invertito che supporta ricerche full-text. Un indice invertito elenca ogni parola univoca che appare in qualsiasi documento ed identifica tutti i documenti in cui si trova tale parola.

Un indice può essere pensato come una raccolta ottimizzata di documenti; ogni documento è una raccolta di campi, che sono le coppie chiave-valore contenenti i propri dati. Come impostazione predefinita, Elasticsearch indicizza tutti i dati in ogni campo ed ogni campo indicizzato ha una struttura dati dedicata ed ottimizzata. Ad esempio, i campi di testo sono archiviati in indici invertiti ed i campi numerici e geografici sono archiviati in alberi. La capacità di utilizzare le strutture dati per campo, al fine di assemblare e restituire risultati di ricerca, è ciò che rende Elasticsearch così veloce.

Elasticsearch ha anche la capacità di essere senza schema, il che significa che i documenti possono essere indicizzati senza specificare esplicitamente come gestire ciascuno dei diversi campi che potrebbero essere presenti in essi. Quando la mappatura dinamica è abilitata, Elasticsearch rileva ed aggiunge automaticamente nuovi campi all'indice. Tale comportamento predefinito semplifica l'indicizzazione e l'esplorazione dei dati: basta avviare l'indicizzazione dei documenti ed Elasticsearch rileverà e "mapperà" valori booleani, valori interi ed in virgola mobile, date e stringhe, come tipi di dati di Elasticsearch appropriati.

In Figura 4.2 è possibile vedere una semplificazione di come vengono gestite le differenze tra Elasticsearch ed un database relazionale.

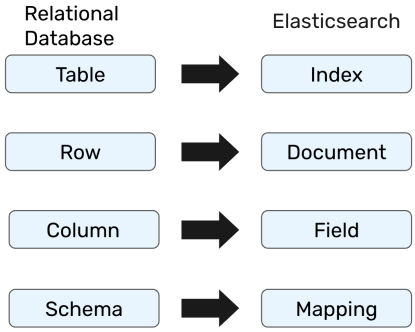

Figura 4.2: Differenze tra un database relazionale ed Elasticsearch

4.4 `Retriever`

A differenza del modello di Huggingface e del Document Store, la scelta del `Retriever` è stata più semplice ma anche più limitata. Infatti, ne esistono di svariati, ciascuno focalizzato su un aspetto in particolare. Noi, avendo un modello di dati ed un modello di Huggingface basati sul formato DPR, abbiamo dovuto scegliere il `DensePassageRetriever`.

4.4.1 `DensePassageRetriever`

Un `DensePassageRetriever` è responsabile del recupero dei passi rilevanti, rispetto alla domanda posta, in base alla somiglianza sostanziale tra le rappresentazioni dei passaggi testuali e quelle delle domande. Inoltre, poiché l'intero sistema deve essere ragionevolmente veloce nel soddisfare le richieste dell'utente, viene pre-calcolato e mantenuto un indice contenente tali rappresentazioni. Quindi, durante il tempo di inferenza, per ogni nuova domanda o richiesta che emerge, possiamo recuperare in modo efficiente tutti i "k" passaggi principali ed, in seguito, eseguire il nostro `Reader` solo su questo sottoinsieme più piccolo. La dimensione di "k" dipende da un paio di fattori, come il ritardo previsto nella Pipeline o la potenza di calcolo disponibile. In generale, qualsiasi valore di k compreso tra 3 e 40 serve allo scopo.

La similarità tra la domanda e l'embedding dei passi è rappresentata dal calcolo del prodotto scalare tra di essi; tale metodo è scelto per la sua semplicità. Più alto è lo score della similarità e più il passaggio è rilevante per la domanda. Questo ci servirà per capire quali passi sono più rilevanti tra i "k" scelti.

Come accennato in precedenza nel Capitolo 4.2, al `Retriever` viene passato un modello, preso da Huggingface, su cui quest'ultimo baserà i suoi calcoli. Per migliorare ulteriormente ed in maniera significativa la comprensione dei testi, il `DensePassageRetriever` verrà, inoltre, addestrato sul nostro dataset convertito in DPR.

4.5 Pre-elaborazione

Haystack include una suite di strumenti per, ad esempio, estrarre testo da diversi tipi di file, normalizzare lo spazio bianco o dividere il testo in parti più piccole al fine di ottimizzare il retrieval. Questi passaggi di pre-elaborazione dei dati possono avere un grande impatto sulle prestazioni dei sistemi, ed una gestione efficace dei dati è fondamentale per ottenere il massimo dal proprio progetto. In questo caso, si è scelto di utilizzare il `Logging` ed il `PreProcessor`.

4.5.1 `Logging`

Il `Logging` è un semplice script messo a disposizione da Haystack per la visualizzazione di messaggi. Infatti, basterà configurare la visualizzazione dei messaggi di `Logging` e quale livello di log deve essere utilizzato. Un esempio banale può essere rappresentato dalla distinzione tra messaggi di errore, di warning e di info. Questo componente aiuta l'output del codice ad essere più pulito e più facilmente leggibile.

4.5.2 `PreProcessor`

Invece, la classe `PreProcessor` è progettata per aiutare a pulire il testo, dividendolo anche in ripartizioni sensate. La suddivisione dei file può avere un impatto molto significativo sulle prestazioni del sistema ed è assolutamente obbligatoria per i modelli DPR. In generale, è consigliato dividere il testo dei file in piccoli documenti di circa 100 parole. Al fine del nostro progetto, inoltre, si è utilizzato lo "split overlap", parametro che aiuta a dividere le frasi seguendo la punteggiatura, senza il rischio del troncamento nel pieno di una frase.

4.6 Pipeline

Iniziando a mettere insieme i componenti citati in precedenza, si viene a creare la Pipeline utilizzata in questo progetto. Essa, quindi, non sarà una delle varie Pipeline predefinite messe a disposizione da Haystack, ma sarà una Pipeline custom.

Tale Pipeline sarà strutturata sequenzialmente come di seguito specificato:

- `Logging`;
- `PreProcessor`;
- `ElasticsearchDocumentStore`;
- `DensePassageRetriever`.

In questo modo, le prime due componenti formeranno l'indexing Pipeline, mentre il solo `DensePassageRetriever` formerà la Pipeline di ricerca.

4.7 Funzioni di similarità

Il Clustering consiste nel raggruppare determinati oggetti simili tra loro e può essere utilizzato per decidere se due elementi sono simili o dissimili nelle loro proprietà.

Nel Data Mining, la misura della similarità è una distanza calcolata utilizzando le dimensioni che descrivono le caratteristiche dell'oggetto. Ciò significa che se la distanza tra due punti dati è piccola, c'è un alto grado di somiglianza tra gli oggetti, mentre quando la distanza è grande, ci sarà un basso grado di somiglianza. Quest'ultima è soggettiva e dipende fortemente dal contesto e dall'applicazione. Infatti, la somiglianza tra due oggetti può essere determinata, ad esempio, dalla loro dimensione, dal loro colore, etc.

Alcune delle misure di somiglianza popolari sono:

- distanza euclidea;
- distanza di Manhattan;
- Jaccard similarity;
- Cosine similarity.

In questo progetto, per la valutazione dei risultati ottenuti, si è deciso di utilizzare in prima istanza la semplice Jaccard similarity e, successivamente, il TF-IDF embedding ed il semantic embedding, metodi che possono integrare la cosine similarity.

4.7.1 Jaccard similarity

Per capire bene la Jaccard similarity bisogna partire dall'indice di Jaccard. Quest'ultimo, noto anche come coefficiente di similarità di Jaccard, è un indice statistico utilizzato per confrontare la similarità e la diversità di insiemi campionari. Esso è definito come la dimensione dell'intersezione divisa per la dimensione dell'unione degli insiemi campionari:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}. \tag{4.7.1}$$

Invece, la distanza di Jaccard, che misura la dissimilarità tra insiemi campionari, è complementare al coefficiente di Jaccard e si ottiene sottraendo il coefficiente di Jaccard da 1, o, in modo equivalente, dividendo la differenza delle dimensioni dell'unione e dell'intersezione di due insiemi per la dimensione dell'unione:

$$J_\alpha(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{4.7.2}$$

Sostanzialmente, questa distanza è la similarità di Jaccard ed è propriamente una metrica.

4.7.2 TF-IDF embedding

Quando si parla di TF-IDF, si discute di comprensione del linguaggio da parte delle macchine. La sigla TF-IDF significa "Term Frequency - Inverse Document Frequency" e rappresenta una formula matematica per calcolare l'importanza di una parola inserita in un documento in rapporto ad altri documenti simili.

Questo valore è utilizzato come fattore di ponderazione in Information Retrieval e Data Mining. Come si può vedere in Figura 4.3, il valore TF-IDF aumenta proporzionalmente al numero di volte che una parola compare nel documento. Tuttavia, viene compensato dalla frequenza della stessa parola nel corpo di altri documenti simili; ciò aiuta a controllare se alcuni termini sono generalmente più comuni di altri.

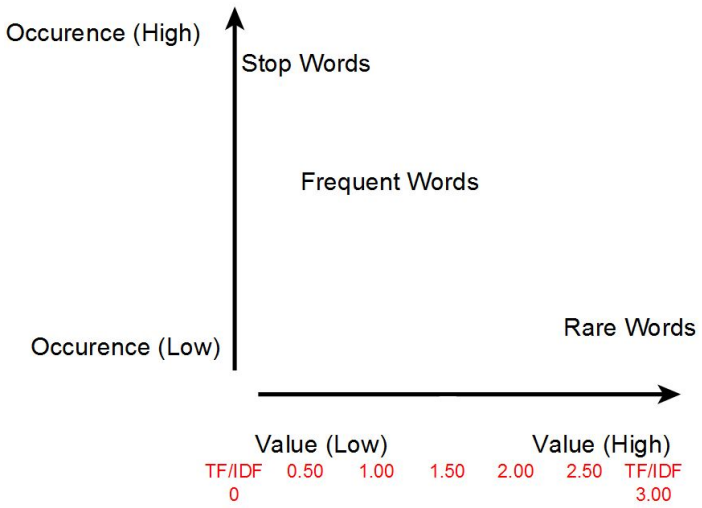

Figura 4.3: Grafico del funzionamento dell'indice TF-IDF

4.7.3 Semantic embedding

La semantica distribuzionale comprende una serie di teorie e metodi di linguistica computazionale per lo studio della distribuzione semantica delle parole nel linguaggio naturale. Questi modelli sono perlopiù empirici ed assumono che ci sia una distribuzione statistica dei termini dominante, che verrà utilizzata per delinearne il comportamento semantico.

Questa teoria propone il paradigma per cui le parole sono distribuite ad una distanza proporzionale al loro grado di similarità. Quest'ultima proprietà segue l'ipotesi fondamentale della semantica distribuzionale secondo la quale due parole sono tanto più simili semanticamente, quanto più tendono a comparire nello stesso contesto linguistico. La frase emblematica

di questa teoria è:

$$<< \textit{conoscerai una parola dalla compagnia che frequenta} >> . \tag{4.7.3}$$

4.7.4 Cosine similarity

I due metodi sopra citati sono stati utilizzati in questo progetto assieme alla cosine similarity, per normalizzare il risultato e poterlo confrontare.

La similarità del coseno, o cosine similarity, è una tecnica euristica per la misurazione della similitudine tra due vettori, effettuata calcolando il coseno tra di essi. Essa è usata generalmente per il confronto di testi nell'estrazione di dati e nell'analisi testuale.

Dati due vettori di attributi numerici, A e B, il livello di similarità tra di essi è espresso utilizzando la formula:

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} \tag{4.7.4}$$

Un altro modo di indicare la formula, del tutto equivalente, è il seguente:

$$\frac{\sum_{k=1}^{n} A(k)B(k)}{\sqrt{\sum_{k=1}^{n} A(k)^2}\sqrt{\sum_{k=1}^{n} B(k)^2}} \tag{4.7.5}$$

Il valore di similitudine così definito è compreso tra -1 e +1, dove -1 indica una corrispondenza esatta ma opposta e +1 indica due vettori uguali. Nel caso tipico del confronto fra testi il contenuto dei due vettori è la frequenza dei termini, ossia il numero di volte in cui una certa parola ricorre all'interno del testo. Il k-esimo elemento di ogni vettore conterrà, dunque, il numero di volte in cui la parola numerata con k ricorre nel testo, oppure 0 se quest'ultima non ricorre mai.

Nel caso dell'analisi dei testi, poiché le frequenze dei termini sono sempre valori positivi, si otterranno valori che vanno da 0 a +1, dove +1 indica che le parole contenute nei due testi sono le stesse (ma non necessariamente nello stesso ordine) e 0 che non c'è nessuna parola che appare in entrambi.

CAPITOLO 5

Implementazione delle attività di re-engineering

In questo capitolo vengono mostrate le implementazioni e le funzionalità di tutte le celle che compongono il Notebook utilizzato per il progetto discusso nella presente tesi.

5.1 Implementazione degli import

Le librerie sono funzioni scritte da altri programmatori per risolvere problemi specifici. Queste funzioni sono raggruppate in moduli e sono messe a disposizione della comunità di sviluppatori Python. In questo modo, non è necessario sviluppare ex novo una funzione ogni volta che serve; basta richiamare la funzione esterna con `import` o `from import` ed usarla.

Per il nostro progetto, abbiamo dovuto importare le seguenti librerie:

- `json`, per la gestione del dataset nel formato JSON;
- `pandas`, per effettuare le attività di ETL;
- `logging`, per il controllo dei messaggi in output;
- `os`, per interagire con le variabili d'ambiente.

Oltre a queste librerie, sono stati importati anche i moduli `Path` e `tqdm`, utilizzati, rispettivamente, per la gestione dei percorsi dei file e per l'utilizzo delle barre di avanzamento.

Come si può vedere nel Listato 5.1, sono stati importati anche tutti i moduli di Haystack necessari per il nostro obiettivo finale.

```
1  import json
2  import pandas as pd
3  import logging
4  import os
5
6  from pathlib import Path
7  from tqdm import tqdm
8  from haystack import Pipeline
9  from haystack.document_stores import ElasticsearchDocumentStore
10 from haystack.nodes import PreProcessor, DensePassageRetriever
11 from transformers import DPRContextEncoder, DPRQuestionEncoderTokenizer
```

Listato 5.1: Gli import necessari per il nostro progetto

5.2 Implementazione ed addestramento del `Retriever`

Haystack contiene tutti gli strumenti necessari per addestrare il proprio modello di Dense Passage Retrieval. Prima di effettuare il train vero e proprio, però, bisogna inizializzare i path e definire le variabili che conterranno il Document Store ed il `Retriever`.

5.2.1 Definizione dei path

In questa cella del nostro Notebook (Listato 5.2) ci occupiamo, semplicemente, di inizializzare le variabili contenenti i vari percorsi necessari per la lettura del dataset e per il download del nostro modello direttamente da Huggingface.

```
doc_dir = "data"
save_dir = "data"

train_filename = "dataset-dpr.json"
dev_filename = "dataset-dpr.json"

query_model = "voidful/dpr-ctx_encoder-bert-base-multilingual"
passage_model = "voidful/dpr-ctx_encoder-bert-base-multilingual"
```

Listato 5.2: Elenco delle varibili contenenti i path

5.2.2 La variabile `document_store`

Come visto nella Sezione 4.3, abbiamo deciso di utilizzare Elasticsearch come Document Store. Nel Listato 5.3 si possono notare tutti i parametri necessari per l'inizializzazione, a partire dallo username, dalla password e dall'host, compreso di porta, fino ad arrivare ai parametri più tecnici, come, ad esempio, la dimensione dell'embedding. Quest'ultima va impostata in riferimento alla dimensione del modello di Huggingface stesso. In questo caso, il modello scelto in precedenza ha la dimensione dell'embedding uguale a 768. Inoltre, si noti come viene utilizzata la cosine similarity invece del prodotto scalare.

```
document_store = ElasticsearchDocumentStore(
    host="localhost",
    port=9200,
    username="elastic",
    password="password",
    index="document",
    embedding_dim=768,
    similarity="cosine",
    scheme="https",
    verify_certs=False
)
```

Listato 5.3: Definizione della variabile contenente il Document Store

5.2.3 La variabile `retriever`

Come per il Document Store, anche per il `Retriever` è necessario istanziare una variabile ed assegnarla alla parola chiave per poterla utilizzare in seguito. In questo caso, la parola chiave è `DensePassageRetriever`. Il primo parametro da passare a tale componente è il Document Store appena inizializzato, seguito dai due path del modello di Huggingface.

Come si può vedere nel Listato 5.4, il parametro `use_gpu` è posto a `True` per velocizzare l'elaborazione tramite l'uso della scheda video. Anche in questo caso è stata utilizzata la cosine similarity.

```
retriever = DensePassageRetriever(
    document_store=document_store,
    query_embedding_model=query_model,
    passage_embedding_model=passage_model,
    max_seq_len_query=64,
    max_seq_len_passage=256,
    use_gpu=True,
    similarity_function="cosine",
    batch_size=1
)
```

Listato 5.4: Definizione della variabile contenente il `Retriever`

5.2.4 L'addestramento del `Retriever`

Arrivati a questo punto, si può iniziare ad addestrare il `Retriever`. Ovviamente, quest'ultima è un'operazione lunga ed onerosa a livello computazionale, soprattutto impostando un numero elevato di epoche. Infatti, esse definiscono il numero di volte in cui l'algoritmo di apprendimento processerà l'intero set di dati di addestramento. Inoltre, anche la grandezza del dataset ed il batch size, cioè gli elementi processati ad ogni iterazione, incidono sul tempo di train. Nel nostro caso, tale tempo si aggira intorno all'ora di esecuzione. Nel Listato 5.5 è presente, nel dettaglio, il setting di tutti i parametri utilizzati.

```
retriever.train(
    data_dir=doc_dir,
    train_filename=train_filename,
    dev_filename=dev_filename,
    test_filename=dev_filename,
    max_processes=4,
    n_epochs=5,
    batch_size=4,
    grad_acc_steps=8,
    save_dir=save_dir,
    evaluate_every=3000,
    embed_title=True,
    num_positives=1,
    num_hard_negatives=1
)
```

Listato 5.5: Train del `Retriever`

5.3 Implementazione della gestione dei dati

Nell'apprendimento automatico, la pre-elaborazione, o pre-processing, è il momento in cui si preparano e si organizzano i dati, prima di elaborarli. Avviare il processo di Machine Learning su dati grezzi implica il rischio di prolungare il procedimento su dati ridondanti e mal strutturati. Analizzare i dati prima dell'elaborazione riduce la complessità computazionale.

5.3.1 La pre-elaborazione

Come visto nella Sezione 4.5, abbiamo utilizzato due componenti per la pre-elaborazione. Il primo è il `Logging`, finalizzato alla gestione dei messaggi in output, mentre il secondo è il `PreProcessor`. Tale componente, invece, si occupa di pulire il testo e dividerlo in documenti più piccoli, formati da circa 100 parole.

Un focus particolare va posto sul parametro `split_respect_sentence_boundary` (Listato 5.6); esso permette di non troncare una frase a metà, ma cerca segni di punteggiatura con un range di 20 parole prima e dopo la centesima. Ciò permette di migliorare la semantica dello split.

```
logger = logging.getLogger("WEBingester")
logger.info("Processing data before loading...")

split_length = int(os.getenv("SPLIT_LENGTH", "100"))
split_respect = eval(os.getenv("SPLIT_RESPECT_SENTENCE_BOUNDARY", "True"))

processor = PreProcessor(
    clean_empty_lines=True,
    clean_whitespace=False,
    clean_header_footer=False,
    split_by=os.getenv("SPLIT_BY", "word"),
    split_length=split_length,
    split_overlap=int(os.getenv("SPLIT_OVERLAP", "20")),
    split_respect_sentence_boundary=split_respect,
    language=os.getenv("LANGUAGE", "it")
)
```

Listato 5.6: La pre-elaborazione del dataset

5.3.2 Lo splitting del dataset

La funzione in questione si occupa, semplificando, di eseguire la divisione del dataset in frasi di circa 100 parole, "appendendole" alla lista `dicts`. Essa utilizza i due moduli di pre-elaborazione visti nella sottosezione precedente. Come si può notare dal Listato 5.7, sono presenti due funzioni di appoggio, `handle_nl_splitting` e `find_next_idx`, non inserite nel codice per non appesantire il tutto. La prima si occupa di gestire al meglio il new-line considerando "`\n`" come una parola; la seconda, invece, viene richiamata per trovare l'identificatore successivo alla fine di una determinata frase.

```
dicts = []

for path in Path(docs_path).rglob("*.json"):
    logger.info(f"Got file: '{path}'")

    with open(path, "r") as fp:
        doc = json.load(fp)

        if doc["meta"]["source"] == "html":
            dicts += processor.process(doc)

    if eval(os.getenv("HANDLE_SPLIT_BOUNDARY", "True")) and split_respect):
        logger.info("Going to execute HANDLE_SPLIT_BOUNDARY")

        new_dicts = []
        for _dict in dicts:
```

```
            words = _dict["content"].split(" ")
            if len(words) > split_length:
                words = handle_nl_splitting(_dict["content"])

                original_len = len(words)
                splits = []

                while len(words) > 0:
                    if len(words) < split_length:
                        fixed_sentence = words
                        splits.append(fixed_sentence)
                        break
                    else:
                        next_idx = find_next_idx(words, split_length)
                        fixed_sentence = []

                        for i in range(next_idx):
                            fixed_sentence.append(words.pop(0))
                        splits.append(fixed_sentence)

                logger.info(f"Split complete. Got {len(splits)}
                            over sentence length {original_len}")

                for s in splits:
                    new_dict = _dict.copy()
                    new_dict["content"] = " ".join(s)
                    new_dicts.append(new_dict)
            else:
                new_dicts.append(_dict)
        dicts = new_dicts
```

Listato 5.7: Funzione che si occupa di "splittare" il dataset

5.3.3 Il caricamento dei dati

Questa parte del Notebook è composta da poche righe di codice, per quanto sia molto lunga temporalmente e pesante computazionalmente. In una prima fase avviene la pulizia completa del Document Store, il quale viene subito ripopolato dai documenti creati precedentemente. Tale Document Store, appena riempito, viene passato al `Retriever`, il quale si occupa di effettuare l'embedding. Questi semplici passi, nel nostro caso, possono durare ore, a seconda dell'hardware che si ha a disposizione. Nel Listato 5.8 è presente l'implementazione di questa sezione.

```
document_store.delete_documents()

document_store.write_documents(dicts)

retriever = DensePassageRetriever.load(
    load_dir = save_dir,
    document_store = document_store
)

document_store.update_embeddings(retriever)
```

Listato 5.8: La pre-elaborazione del dataset

5.4 Implementazione del componente per la comparazione

Nel Listato 5.9 è presente la funzione che ricava le domande e le risposte contenute nel dataset di riferimento e le aggiunge ad una lista che verrà utilizzata successivamente. Si può notare come venga creato un campo `"clean_answers"` per filtrare ulteriormente il testo che non sia "printable" e che contenga caratteri non presenti nello standard ASCII.

```
with open("dataset.json", "r", encoding="utf-8") as f:
    val_data = json.load(f)

qea_list = []
for idx, el in enumerate(val_data["data"]):
    el = el["paragraphs"][0]

    for qs in el["qas"]:
        qea = {}
        qea["question"] = qs["question"]
        answer = qs["answers"][0]["text"]
        words = answer.split()
        printable = [
                word for word in words
                if all(i.isprintable() and i.isascii() for i in word)
            ]
        qea["answer"] = answer
        qea["clean_answer"] = " ".join(printable)
        qea_list.append(qea)
```

Listato 5.9: Creazione della lista domande-risposte

5.5 Implementazione dei modelli di similarità

Come visto nella Sezione 4.7, abbiamo deciso di implementare tre funzioni di similarità per valutare la correttezza e le performance dei risultati ottenuti. Esse saranno illustrate nelle prossime tre sottosezioni.

5.5.1 La funzione `jaccard_similarity`

La prima funzione per il calcolo degli indici è anche la più semplice. Si tratta della Jaccard similarity ed è banalmente implementata come nel Listato 5.10. Essa ricava in input due documenti, esegue lo split per avere a disposizione le singole parole ed effettua l'intersezione dei due insiemi così ottenuti. L'indice risultante sarà l'intersezione rapportata all'unione. Non tenendo conto della semantica delle parole, questa similarità è, semplicemente, l'indice di quante parole si ripetano nei due documenti passati in input.

```
def jaccard_similarity(doc_1, doc_2):
    a = set(doc_1.split())
    b = set(doc_2.split())
    c = a.intersection(b)

    return float(len(c)) / (len(a) + len(b) - len(c))
```

Listato 5.10: La funzione di similarità di Jaccard

5.5.2 La funzione `tfidf_embedding`

La seconda funzione di similarità riguarda l'indice TF-IDF (Listato 5.11). Esso misura l'importanza di un termine rispetto ad un documento o ad una collezione di documenti. Anche questa funzione ricava in input due documenti e li processa tramite il modulo `TfidfVectorizer`. L'altro modulo utilizzato, anch'esso importato da Scikit-learn, è il `cosine_similarity`. Come si può intuire dal nome, quest'ultimo effettua il check tra i due documenti vettorizzati tramite la similarità del coseno. Pertanto, sarà restituito in output l'indice di similarità normalizzato.

```
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer

def tfidf_embedding(doc_1, doc_2):
    tfidf = TfidfVectorizer()
    X = tfidf.fit_transform([doc_1, doc_2])
    return cosine_similarity(X)[0][1]
```

Listato 5.11: La funzione di similarità TF-IDF

5.5.3 La funzione `semantic_embedding`

Questa terza ed ultima funzione di similarità è la più complessa. Infatti, a differenza delle altre due, riesce a riconoscere il contesto delle parole, utilizzando il modello di Huggingface scelto precedentemente, per comprenderne il significato.

Come si può notare nel Listato 5.12, in maniera speculare alla funzione TF-IDF, in input avremo due documenti, mentre in output la funzione ritornerà l'indice di similarità del coseno normalizzato.

```
import torch

tokenizer = DPRQuestionEncoderTokenizer.from_pretrained(query_model)
model = DPRContextEncoder.from_pretrained(passage_model)

def mean_pooling(model_output, attention_mask):
    token_embeddings = model_output[0]
    input_mask_expanded = attention_mask
        .unsqueeze(-1)
        .expand(token_embeddings.size())
        .float()
    return torch.sum(token_embeddings * input_mask_expanded, 1) /
        torch.clamp(input_mask_expanded.sum(1), min=1e-9)

def semantic_embedding(doc_1, doc_2):
    sentences = [doc_1, doc_2]
    encoded_input = tokenizer(
        sentences,
        padding=True,
        truncation=True,
        return_tensors="pt"
    )
    with torch.no_grad():
        model_output = model(**encoded_input)
    sentence_emb = mean_pooling(model_output, encoded_input["attention_mask"])
    return cosine_similarity(sentence_emb)[0][1]
```

Listato 5.12: La funzione di similarità semantica

5.6 Implementazione del calcolo degli indici

Arrivati a questo punto, non resta che confrontare le prestazioni del nostro `Retriever` addestrato con le risposte originali, tramite le varie funzioni di similarità viste in precedenza. Come si può notare alla fine del Listato 5.13, vengono salvati, in un Dataframe Pandas, due indici per ogni metrica utilizzata. Ciò perché, come visto nella Sezione 5.4, abbiamo aggiunto un campo `"clean_answers"` contenente le risposte ulteriormente filtrate.

Tutti gli indici normalizzati appena calcolati vengono salvati in un file `.csv` pronti per essere visualizzati ed analizzati.

```
processed_pipes = []
empty_pipeline = Pipeline()

for file in Path("pipelines/"):
    if len(file.parts) == 2:
        print("Processing pipeline", file.name)
        pipeline = empty_pipeline.load_from_yaml(file)

        for qea in tqdm(qea_list):
            results = pipeline.run(query=qea["question"])
            for answer in results["answers"]:
                prediction = answer.context
                words = prediction.split()
                printable = [
                    word for word in words
                    if all(i.isprintable() and i.isascii() for i in word)
                ]
                clean_pred = " ".join(printable)

                jacc = jaccard_similarity(qea["clean_answer"], clean_pred)
                tfidf = tfidf_embedding(qea["clean_answer"], clean_pred)
                sem = semantic_embedding(qea["clean_answer"], clean_pred)

                prediction_list.append(prediction)
                score_list.append(answer.score)
                url_list.append(answer.meta["url"])
                clean_prediction_list.append(clean_pred)

                jacc_score_list.append(jacc)
                tfidf_score_list.append(tfidf)
                sem_score_list.append(sem)

            qea["predicted_answers"] = prediction_list
            qea["answers_score"] = score_list
            qea["answers_url"] = url_list
            qea["clean_predicted_answers"] = clean_prediction_list
            qea["jacc_scores"] = jacc_score_list
            qea["tfidf_scores"] = tfidf_score_list
            qea["sem_scores"] = sem_score_list

        for qea in qea_list:
            counter = counter + 1
            jac = jac + qea["jacc_scores"][0]
            tfidf = tfidf + qea["tfidf_scores"][0]
            sem = sem + quea["sem_scores"][0]

        jacc_accuracy = jac / counter
        tfidf_accuracy = tfidf / counter
        sem_accuracy = sem / counter

```

```
        final_results = {}
        final_results["pipe_name"] = file.name.split(".")[0]
        final_results["accuracy_jaccard"] = jacc_accuracy
        final_results["accuracy_tfidf"] = tfidf_accuracy
        final_results["accuracy_semantic"] = sem_accuracy

        processed_pipes.append(final_results)

        out_dir = Path("domande-risposte")
        if not out_dir.exists():
            out_dir.mkdir(parents=True)

        lines_list = []
        for qea in qea_list:
            tosave = {}
            tosave["question"] = qea["question"]
            tosave["answer"] = qea["answer"]
            tosave["clean_answer"] = qea["clean_answer"]
            tosave["predicted_answer1"] = qea["predicted_answers"][0]
            tosave["clean_predicted_answer1"] = qea["clean_predicted_answers"][0]
            tosave["predicted_answer2"] = qea["predicted_answers"][1]
            tosave["clean_predicted_answer2"] = qea["clean_predicted_answers"][1]
            lines_list.append(tosave)

        qafinal = pd.DataFrame(lines_list)
        qafinal.to_csv(out_dir / (file.name.split(".")[0] + ".csv"))

results = pd.DataFrame(processed_pipes)
results.to_csv("validation-results.csv")
```

Listato 5.13: Il codice per salvare e visualizzare le performance

CAPITOLO 6

Esempi di funzionamento del nuovo sistema di Business Ingelligence

Questo sesto capitolo è dedicato alla messa in atto del progetto con le proprie complessità. Verranno analizzati, infatti, i software idonei per l'esecuzione, gli output intermedi ed i possibili errori. Infine, verrà fornito un riepilogo dei risultati finali.

6.1 Come avviare il progetto

Il codice analizzato nel precedente capitolo, è facilmente eseguibile da diversi programmi open source, come, ad esempio, Pycharm o VSCode. Diversamente, per lo sviluppo ed il testing di tale progetto, abbiamo utilizzato due programmi differenti, anch'essi open source. I due software in questione, approfonditi successivamente, sono i seguenti:

- Anaconda;
- Docker.

6.1.1 Anaconda

Anaconda è la piattaforma per la Data Science con Python più utilizzata al mondo, che vede tra i suoi clienti più importanti aziende quali BMW, Samsung e Cisco.

La grande popolarità di Anaconda (Figura 6.1) è data dal fatto che semplifica sensibilmente il processo di setup di un ambiente di sviluppo, racchiudendo assieme nella stessa distribuzione tutto ciò di cui si ha bisogno per iniziare da subito a programmare: l'installer di Python, un package ed un environment manager dedicato, chiamato Conda, e circa 300 pacchetti installati e pronti all'uso come Pandas, Numpy, Matplotlib, Jupiter, SQLAlchemy, e così via.

Figura 6.1: Il logo di Anaconda

Uno dei punti di forza di questa distribuzione è l'interfaccia grafica inclusa in essa, chiamata Anaconda Navigator, che ci permette di poter gestire tutti gli aspetti della piattaforma, semplificando ulteriormente il processo di personalizzazione e rendendo Anaconda quasi un must per tutti quei professionisti, ricercatori o appassionati del settore, a cui piace avere tutto immediatamente operativo.

Dopo aver installato tale programma open source basterà accedere ad Anaconda Navigator per comprendere tutte le potenzialità che questa piattaforma mette a disposizione. Infatti, nella homepage, ci saranno molti tool ed interfacce grafiche preinstallate, ed altre da installare, facilmente scaricabili ed integrabili nella suite di riferimento; nel nostro caso, come visto nella Sottosezione 3.5.2, avremo bisogno di Jupyter Notebook.

Inoltre, in Anaconda Navigator è presente una sezione "Environments" dove è possibile personalizzare diversi ambienti di sviluppo, a seconda delle necessità. Per avere accesso a tutte le funzionalità del codice analizzato in precedenza, bisognerà creare un nuovo Environment, inserendo, oltre al preinstallato Python, tutte le librerie di Haystack.

Una volta selezionato l'Environment personalizzato, non resta che aprire l'interfaccia grafica offerta da Jupyter per poter iniziare ad usufruire delle potenzialità messe a disposizione dal progetto in esame.

6.1.2 Docker

Docker (Figura 6.2) è una piattaforma di "containerizzazione" open source; essa consente agli sviluppatori di impacchettare le applicazioni in dei container eseguibili, che combinano il codice sorgente delle applicazioni con le librerie e le dipendenze del sistema operativo necessarie per eseguire tale codice, in qualsiasi ambiente.

Gli sviluppatori potrebbero creare dei container senza Docker, ma la piattaforma rende più facile, semplice e sicuro gestire dei container. Docker è essenzialmente un toolkit che permette agli sviluppatori di creare, implementare, eseguire, aggiornare ed arrestare i container utilizzando dei semplici comandi ed un'automazione che consente di risparmiare lavoro tramite un'unica API.

I container sono resi possibili grazie alle funzionalità di virtualizzazione ed isolamento dei processi, integrate nel kernel di Linux. Tali funzionalità consentono a più componenti applicativi di condividere le risorse di una singola istanza del sistema operativo.

Di conseguenza, la tecnologia dei container offre tutte le funzionalità ed i vantaggi delle macchine virtuali, compresi l'isolamento delle applicazioni ed una scalabilità efficiente in termini di costi. Di seguito sono elencati i vantaggi più importanti:

- *Peso più leggero*: a differenza delle Virtual Machine, i container non portano il payload di un intero hypervisor o di un'intera istanza del sistema operativo; essi, infatti, includono solo i processi e le dipendenze del sistema operativo necessari per eseguire il codice.
- *Maggiore efficienza delle risorse*: con i container, è possibile eseguire diverse volte tutte le copie di un'applicazione sullo stesso hardware. Questo può ridurre la spesa in cloud.
- *Aumento della produttività degli sviluppatori*: rispetto alle macchine virtuali, è possibile implementare, eseguire il provisioning e riavviare i container in modo più veloce e facile.
- Le aziende che utilizzano i container segnalano altri vantaggi, tra cui una maggiore qualità delle app ed una risposta più rapida ai cambiamenti del mercato.

Per questi motivi, l'adozione di Docker è esplosa rapidamente e continua a crescere. Al momento della stesura del presente documento, Docker Inc. riporta più di 13 milioni di sviluppatori e più di 13 miliardi di download di "immagini" container ogni mese.

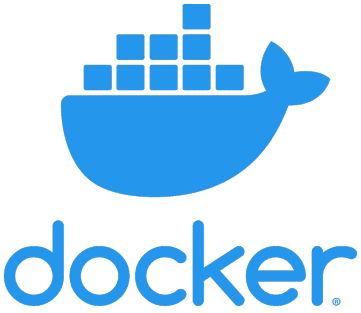

Figura 6.2: Il logo di Docker

Tuttavia, per far funzionare il nostro codice, non è necessario impacchettarlo in una "immagine" ed eseguirlo in un container. Ciò che è necessario "containerizzare", però, è l'immagine di Elasticsearch.

L'installazione di Elasticsearch tramite l'utilizzo di Docker è facile e ben documentata; bastano, infatti, i cinque comandi presenti nel Listato 6.1 per avere un'istanza di Elasticsearch attiva e pronta per l'uso.

```
1 docker pull docker.elastic.co/elasticsearch/elasticsearch:8.5.1
2 docker network create elastic
3 docker run --name es01 --net elastic -p 9200:9200 -p 9300:9300 -it \
4     docker.elastic.co/elasticsearch/elasticsearch:8.5.1
5 docker cp es01:/usr/share/elasticsearch/config/certs/http_ca.crt .
6 curl --cacert http_ca.crt -u elastic https://localhost:9200
```

Listato 6.1: Comandi per installare Elasticsearch tramite Docker

6.2 Output intermedi

Uno dei primi output che possono essere visualizzati quando si lancia il codice è quello del caricamento del modello, nel momento in cui viene inizializzato il `Retriever`. Come si può vedere nella Figura 6.3, se il modello che si vuole utilizzare non è stato precedentemente caricato in locale, esso verrà ricercato nei server di Huggingface e, nel caso esista, verrà prelevato e salvato localmente.

```
INFO - haystack.modeling.model.language_model -  LOADING MODEL
INFO - haystack.modeling.model.language_model -  =============
INFO - haystack.modeling.model.language_model -  Could not find voidful/dpr-ctx
_encoder-bert-base-multilingual locally.
INFO - haystack.modeling.model.language_model -  Looking on Transformers Model
Hub (in local cache and online)...
INFO - haystack.modeling.model.language_model -  Automatically detected languag
e from language model name: multilingual
INFO - haystack.modeling.model.language_model -  Loaded voidful/dpr-ctx_encoder
-bert-base-multilingual
```

Figura 6.3: Output del caricamento del modello di Huggingface

Un altro importante output è presente nella cella di addestramento del `Retriever`. A parte la suggestiva immagine di una locomotiva, in riferimento alla parola chiave "train" (Figura 6.4), in tale output è possibile vedere i dati di loading del dataset di train. Infatti, per caricare novantasette documenti, in questo caso specifico, esso ha impiegato circa dieci secondi, con una media di 8.87 documenti al secondo.

```
INFO - haystack.modeling.data_handler.data_silo -
Loading data into the data silo ...
                ______
                |o  |   !
                |:`_|---'-.
  __            |__|______.-/ _ \-----.|
 (o)(o)------'\ _ /     ( )

INFO - haystack.modeling.data_handler.data_silo -  LOADING TRAIN DATA
INFO - haystack.modeling.data_handler.data_silo -  ==================
INFO - haystack.modeling.data_handler.data_silo -  Loading train set from: data
\validation-dataset-dpr-html.json
INFO - haystack.modeling.data_handler.data_silo -  Got ya 4 parallel workers to
convert 97 dictionaries to pytorch datasets (chunksize = 5)...
INFO - haystack.modeling.data_handler.data_silo -   0     0     0     0
INFO - haystack.modeling.data_handler.data_silo -  /w\   /|\  /|\  /|\
INFO - haystack.modeling.data_handler.data_silo -  / \   /'\   /'\   /'\
Preprocessing Dataset data\validation-dataset-dpr-html.json: 100%|████████| 9
7/97 [00:10<00:00,  8.87 Dicts/s]
```

Figura 6.4: Output del train del `Retriever`

Successivamente viene eseguito l'effettivo l'addestramento, che può durare circa 60 minuti, avendo un setting dei parametri simile al nostro. Alla fine di ciò, il codice esegue una valutazione, calcolando e visualizzando i seguenti tre indici:

- *precision*, cioè il rapporto tra i veri positivi e tutti i positivi risultati;
- *recall*, cioè il rapporto tra i veri positivi e la somma di tutti i positivi e negativi corretti;
- *f1-score*, cioè due volte il prodotto della precision e del recall, diviso per la somma tra i due valori.

Tali indici, per ciò che riguarda il nostro lavoro, sono mostrati nella Tabella 6.1.

	precision	recall	f1-score
hard_negative	0.9034	0.8434	0.8723
positive	0.5436	0.5436	0.5436
macro_avg	0.6167	0.5964	0.6063
weighted_avg	0.8312	0.8589	0.8448

Tabella 6.1: Indici relativi al train del `Retriever`

6.3 Possibili errori

Nello sviluppo software, il programmatore informatico deve stare attento a non introdurre errori. Gli errori di programmazione possono essere distinti in due categorie, ovvero gli errori che il compilatore è in grado di riconoscere ed evidenziare e quelli che, al contrario, esso non riesce a rilevare.

Alla prima categoria appartengono gli errori formali, che possono essere distinti in errori lessicali e sintattici. Fra tutti gli errori, questi sono quelli che destano meno preoccupazione, sia perché vengono rilevati dal compilatore, sia perché, in molti casi, vengono evidenziati dall'editor in fase di scrittura.

Alla seconda categoria, invece, appartengono gli errori logici. Essi derivano dagli errori di progettazione dell'algoritmo; essi, quindi, si commettono prima ancora della scrittura del programma nel linguaggio di programmazione, e determinano degli output diversi da quelli previsti. Questi errori sono insidiosi in quanto il codice del sorgente viene compilato ed eseguito senza alcuna segnalazione di errore da parte del compilatore, eppure il risultato delle elaborazioni non è quello previsto. La ricerca di tali errori è a carico del programmatore, che può verificare la loro esistenza eseguendo più volte il programma a partire da input diversi e controllando che i risultati coincidano con quelli attesi.

Molteplici sono stati gli errori formali e logici incontrati ed affrontati nello sviluppo del progetto alla base della presente tesi. Essendo gli errori formali banali e non molto interessanti, daremo uno sguardo agli errori logici più frequenti e più interessanti in cui ci siamo imbattuti durante lo sviluppo. Essi sono i seguenti:

- errore del modello;
- errore del `Retreiver`;
- errore della dimensione dell'embedding.

6.3.1 Errore del modello

Un primo errore logico incontrato consiste nella scelta del modello di Huggingface. Infatti, utilizzando i modelli di train che non sono basati sul formato DPR, potrebbero esserci degli errori, come quello mostrato in Figura 6.5. In questo caso, il problema risiede nel fatto che la misura del tokenizer non corrisponde a quella data.

```
File ~\anaconda3\envs\Haystack\lib\site-packages\haystack\modeling\model\biadaptive_model.py:354,
ab_size(self, vocab_size1, vocab_size2)
    347 model1_vocab_len = self.language_model1.model.resize_token_embeddings(new_num_tokens=None)
    349 msg = (
    350     f"Vocab size of tokenizer {vocab_size1} doesn't match with model {model1_vocab_len}. "
    351     "If you added a custom vocabulary to the tokenizer, "
    352     "make sure to supply 'n_added_tokens' to LanguageModel.load() and BertStyleLM.load()"
    353 )
--> 354 assert vocab_size1 == model1_vocab_len, msg
    356 model2_vocab_len = self.language_model2.model.resize_token_embeddings(new_num_tokens=None)
    358 msg = (
    359     f"Vocab size of tokenizer {vocab_size1} doesn't match with model {model2_vocab_len}. "
    360     "If you added a custom vocabulary to the tokenizer, "
    361     "make sure to supply 'n_added_tokens' to LanguageModel.load() and BertStyleLM.load()"
    362 )

AssertionError: Vocab size of tokenizer 250002 doesn't match with model 250037.
If you added a custom vocabulary to the tokenizer, make sure to supply 'n_added_tokens'
to LanguageModel.load() and BertStyleLM.load()
```

Figura 6.5: Errore nella scelta del modello di Huggingface

6.3.2 Errore del `Retriever`

Un altro errore logico potrebbe essere dato dalla dichiarazione del `Retriever`. Infatti, se non si utilizza il `DensePassageRetriever`, con un modello basato sul formato DPR potrebbe verificarsi una incompatibilità come quella visualizzata nella Figura 6.6. In tale caso, è stato dichiarato un `TransformersRetriever` con il modello di Huggingface scelto per tale progetto. Il messaggio riporta, inoltre, un elenco di tutti i modelli supportati da tale `Retriever`.

```
INFO - haystack.modeling.utils - Using devices: CUDA
INFO - haystack.modeling.utils - Number of GPUs: 1
The model 'DPRQuestionEncoder' is not supported for question-answering. Supported models are ['QDQBer
tForQuestionAnswering', 'F NetForQuestionAnswering', 'GPTJForQuestionAnswering', 'LayoutLMv2ForQuesti
onAnswering', 'RemBertForQuestionAnswering', 'CanineF orQuestionAnswering', 'RoFormerForQuestionAnswe
ring', 'BigBirdPegasusForQuestionAnswering', 'BigBirdForQuestionAnswering', 'Con vBertForQuestionAnsw
ering', 'LEDForQuestionAnswering', 'DistilBertForQuestionAnswering', 'AlbertForQuestionAnswering', 'C
amembe rtForQuestionAnswering', 'BartForQuestionAnswering', 'MBartForQuestionAnswering', 'LongformerF
orQuestionAnswering', 'XLMRoberta ForQuestionAnswering', 'RobertaForQuestionAnswering', 'SqueezeBertF
orQuestionAnswering', 'BertForQuestionAnswering', 'XLNetForQ uestionAnsweringSimple', 'FlaubertForQue
stionAnsweringSimple', 'MegatronBertForQuestionAnswering', 'MobileBertForQuestionAnswer ing', 'XLMFor
QuestionAnsweringSimple', 'ElectraForQuestionAnswering', 'ReformerForQuestionAnswering', 'FunnelForQu
estionAnsweri ng', 'LxmertForQuestionAnswering', 'MPNetForQuestionAnswering', 'DebertaForQuestionAnsw
ering', 'DebertaV2ForQuestionAnswering', 'IBertForQuestionAnswering', 'SplinterForQuestionAnswerin
g'].
```

Figura 6.6: Errore nella tipologia del `Retriever`

6.3.3 Errore della dimensione dell'embedding

Un ultimo errore interessante e, soprattutto, molto frequente, può essere quello della dimensione dell'embedding. Infatti, bisogna stare attenti nella definizione del Document Store, in relazione al modello scelto. Nella Figura 6.7 è possibile vedere il classico errore di mancata corrispondenza tra la dimensione dell'embedding del modello, che risulta essere 768, e quella del Document Store, che risulta essere la metà esatta.

```
RuntimeError                              Traceback (most recent call last)
Input In [17], in <cell line: 1>()
----> 1 document_store.update_embeddings(reloaded_retriever)

File ~\anaconda3\envs\Haystack\lib\site-packages\haystack\document_stores\elasticsearch.py:1393, in
update_embeddings(self, retriever, index, filters, update_existing_embeddings, batch_size, headers)
   1390 assert len(document_batch) == len(embeddings)
   1392 if embeddings[0].shape[0] != self.embedding_dim:
-> 1393     raise RuntimeError(
   1394         f"Embedding dim. of model ({embeddings[0].shape[0]})"
   1395         f" doesn't match embedding dim. in DocumentStore ({self.embedding_dim})."
   1396         "Specify the arg `embedding_dim` when initializing ElasticsearchDocumentStore()"
   1397     )
   1398 doc_updates = []
   1399 for doc, emb in zip(document_batch, embeddings):

RuntimeError: Embedding dim. of model (768) doesn't match embedding dim. in DocumentStore (384).
Specify the arg `embedding_dim` when initializing ElasticsearchDocumentStore()
```

Figura 6.7: Errore nella scelta della dimensione dell'embedding

6.4 Risultati

Come abbiamo visto nella Sezione 3.4.2, uno dei requisiti non funzionali del progetto è la scalabilità. Infatti, il codice è facilmente riadattabile per altri dataset procurati da qualsivoglia azienda. In questo caso, però, il dataset fornito risulta abbastanza scarno, ed effettuare un addestramento con pochi dati potrebbe non essere conveniente. Questo motivo potrebbe essere una delle cause dei risultati non eccellenti che abbiamo riportato nella Tabella 6.2.

Più nello specifico, come accennato in precedenza, gli indici denominati con un "2" alla fine del nome vengono calcolati sul dataset filtrato, composto interamente dal campo `"clean_answers"` (Sezione 5.6). In tali indici si può notare un leggero aumento delle prestazioni, dovuto, appunto, alla pulizia del dataset.

	value
accuracy_jaccard	0,1137
accuracy_tfidf	0,3745
accuracy_semantic	0,5639
accuracy_jaccard_2	0,1348
accuracy_tfidf_2	0,4148
accuracy_semantic_2	0,6538

Tabella 6.2: Risultati degli indici di similarità

Mettendo a confronto i tre indici nel complesso, invece, possiamo vedere come le prestazioni più scarse siano date dall'indice di Jaccard, in cui avviene una comparazione delle parole, senza contare la loro semantica. Questo dato ci può sicuramente far capire che il testo non sia stato "copiato" in risposta; tuttavia, il significato potrebbe comunque essere il medesimo.

Il Term Frequency - Inverse Document Frequency (TF-IDF) è l'indice che ha risultati intermedi. Esso calcola l'importanza di una parola inserita in un documento in rapporto ad altri documenti simili; di conseguenza, nel 40% circa dei casi, la parola risulta essere nel contesto giusto.

L'ultimo indice è il migliore in termini assoluti; esso tiene conto della semantica delle parole, riuscendo a capire il significato della frase. Di conseguenza, il valore risultante di questo indice, che va dal 56% ad un massimo di 65% nel caso filtrato, può essere un discreto punto di partenza per sviluppi e miglioramenti futuri.

CAPITOLO 7

Discussione in merito al lavoro svolto

In questo settimo capitolo viene, inizialmente, applicata la SWOT Analysis, mentre, successivamente, viene presentata una panoramica sui parametri del codice modificabili e sulle lezioni apprese durante lo sviluppo del progetto.

7.1 SWOT Analysis

L'analisi SWOT è una tecnica utilizzata per identificare i punti di forza, di debolezza, le opportunità e le minacce della propria azienda o di un progetto nello specifico. Sebbene sia ampiamente usata da molte organizzazioni, dalle piccole imprese agli enti no-profit fino alle grandi imprese, l'analisi SWOT può essere utilizzata sia per scopi personali che professionali. Come si può vedere in Figura 7.1, l'acronimo SWOT sta per Strengths, Weaknesses, Opportunities e Threats.

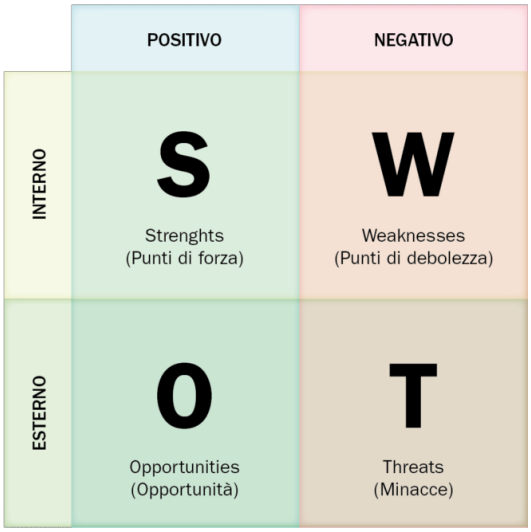

Figura 7.1: Grafico dell'analisi SWOT

7.1.1 Punti di forza

Sicuramente, uno dei punti di forza del progetto in analisi è la scalabilità. Infatti il codice può essere facilmente riadattato e rieseguito con una vasta gamma di dataset. Inoltre, è possibile cambiare facilmente il modello di Huggingface, qualora se ne presentasse uno più performante.

Un altro punto a favore è quello della compattezza del codice; non sono stati creati moduli aggiuntivi; di conseguenza, in relativamente poche righe di codice, è possibile riprodurre tutta la complessità del progetto. Ciò è dovuto, anche, a Python che riesce a racchiudere comandi multipli in qualche riga del proprio linguaggio.

L'ultimo punto di forza potrebbe essere l'affidabilità del progetto. Non essendo un progetto molto vasto, il programma esegue ciò che ci si aspetta da esso. Riducendo il codice, infatti, si riduce anche la complessità, diminuendo i fattori che potrebbero portare ad errori logici.

7.1.2 Punti di debolezza

Come si può intuire dal capitolo precedente, uno dei punti di debolezza del progetto sono i risultati, che non si rilevano eccellenti. Infatti, i valori degli indici delle performance non sono troppo alti, in relazione al fatto che, per avere una risposta corretta, gli indici di similarità debbano attestarsi sopra al 70%.

Un altro punto a sfavore potrebbe essere la dipendenza da Elasticsearch. Infatti, installare Docker e far girare il container di Elasticsearch, per poterlo utilizzare come Document Store, può risultare complicato e non immediato, soprattutto per chi lo utilizza per la prima volta.

Un ultimo punto di debolezza consiste nell'utilizzo obbligato di un dataset in formato DPR. Anche se i documenti JSON sono sempre più utilizzati, molte aziende hanno i propri dati in tabelle relazionali. Di conseguenza, la conversione potrebbe essere complessa ed, in certi casi, infattibile.

7.1.3 Opportunità

Le opportunità derivanti dal progetto in questione sono molteplici, dato che il campo di riferimento è in fase di sviluppo. Infatti, ogni anno vengono proposte tecnologie nuove riguardanti il Machine Learning e le applicazioni possono essere svariate. In questo caso specifico, si potrebbero migliorare i punti deboli con nuovi modelli Huggingface o con un nuovo dataset più ampio e completo. Inoltre, potrebbe essere opportuno condividere il codice con la community per rendere partecipi altri sviluppatori con idee differenti.

7.1.4 Minacce

Le minacce nell'analisi SWOT si riferiscono ad aree che potrebbero creare problemi, in quanto sono esterne, e generalmente fuori dal proprio controllo.

Nel nostro caso una minaccia esterna potrebbe derivare dal mancato supporto di Haystack. Infatti, Haystack è la colonna portante del progetto tanto che, senza di esso, non sarebbe nemmeno iniziato. Per questo, una chiusura inaspettata di Haystack potrebbe essere una grande minaccia. Stessa cosa, anche se in proporzioni ridotte, vale per Huggingface.

7.2 Parametri modificabili

Naturalmente, il progetto non è impeccabile sotto ogni punto di vista; infatti, sono molti i parametri o, più in generale, il codice che possono essere modificati e migliorati. Innanzitutto, è possibile modificare i parametri di train per migliorare le prestazioni. In

primis, specialmente se si dispone di un hardware performante, è opportuno aumentare il numero di epoche. Inoltre, siamo stati limitati anche dai parametri "`max_processes`" e "`batch_size`", il cui valore massimo che abbiamo potuto definire è stato pari a 4.

Un altro parametro su cui si potrebbe "giocare" è lo "`SPLIT_LENGHT`". Esso è stato settato a 100 per una questione di comodità, ma potrebbe produrre performance più elevate in presenza di un setting più accurato.

Altri miglioramenti si potrebbero effettuare sulle funzioni di similarità. Infatti, potrebbero essere aggiunte nuove funzioni, di diversa natura, per capire maggiormente sotto quali campi il sistema risponda in modo migliore o peggiore.

7.3 Lezioni apprese

Sono molte le lezioni apprese nel corso del progetto; tuttavia, esse si possono riassumere nelle seguenti:

- la necessità della documentazione;
- l'aiuto dell'ambiente;
- la pulizia del codice;
- l'opportunità di spalmare il lavoro sul tempo totale;
- la necessità di dedicare tempo per scrivere la documentazione.

Esse verranno spiegate più in dettaglio nelle prossime sottosezioni.

7.3.1 La necessità della documentazione

Una lezione appresa nelle battute iniziali potrebbe essere scontata. Essa riguarda il fatto che la documentazione sia molto rilevante. Infatti, per iniziare un progetto, è di significativa importanza dedicare un tempo consistente a studiare la documentazione a riguardo. In questo caso, sono state dedicate due settimane allo studio di Haystack, senza mettere mano al codice.

7.3.2 L'aiuto dell'ambiente

Un'altra lezione imparata presto, sta nel supporto dato dal luogo in cui si sviluppa il codice. Infatti, essere in un ambiente sereno, con persone cordiali e collaborative, giova molto alla produzione. Inoltre, chiedere aiuto quando si hanno problemi non deve essere visto come un fatto negativo, ma deve essere un segno di un ambiente cooperativo.

7.3.3 La pulizia del codice

Durante lo sviluppo software, una lezione appresa riguarda la qualità del codice. Rimettere mano ad un codice scritto in precedenza è un'operazione tediosa, che potrebbe nascondere insidie. Per questo motivo, anche a costo di perdere più tempo del previsto, è bene scrivere codice di qualità, testandolo in tutte le sue caratteristiche, prima di andare avanti nello sviluppo.

7.3.4 L'opportunità di spalmare il lavoro sul tempo totale

Un'altra lezione, imparata nella parte finale del progetto, riguarda il tempo messo a disposizione. Infatti, le scadenze sono sempre insidiose e si tende a non considerarle inizialmente, soprattutto se il tempo a disposizione è molto. L'indole umana tende ad eseguire il grosso del lavoro a ridosso della scadenza, quando, banalmente, è opportuno dividere il lavoro su tutti i giorni messi a disposizione.

7.3.5 La necessità di dedicare tempo per scrivere la documentazione

L'ultima lezione appresa riguarda la documentazione finale. Essa andrebbe scritta durante lo sviluppo, così da non perdersi dei passaggi che potrebbero essere cruciali. Il programmatore tende, spesso, ad essere sintetico, e ciò può influire negativamente sulla collaborazione, soprattutto quando altre persone dovranno capire e mettere mano al codice altrui. Di conseguenza, è opportuno redigere un'ottima documentazione leggibile e completa sotto ogni punto di vista.

CAPITOLO 8

Conclusioni

In questo capitolo verranno tratte alcune conclusioni in merito al lavoro svolto e verranno proposti alcuni possibili sviluppi futuri.

8.1 Conclusioni

Il progetto discusso nella presente tesi si è posto l'obiettivo di rispondere alla seguente domanda: è possibile realizzare un software in grado di ricercare in un dataset testuale e rispondere a delle domande? In caso di risposta affermativa, si possono visualizzare le performance?

Per rispondere a ciò, dapprima è stata proposta una panoramica sulla storia della tecnologia, introducendo il Machine Learning, il Deep Learning ed il Natural Language Processing. Successivamente è stata fatta una presentazione di Haystack, il framework di riferimento per lo sviluppo del progetto. Sono stati, inoltre, proposti diversi esempi di applicazione delle tecnologie in questione.

Dopo ciò, è stata descritta l'organizzazione iniziale del progetto con tutte le sue sfaccettature. Si è iniziato con la raccolta delle informazioni, passando per la descrizione della componente dati ed arrivando ai requisiti, funzionali e non, grazie ai quali si delinea il nostro software. Dopo di ciò, è stata illustrata la progettazione dei componenti logici, prestando una particolare attenzione alle diverse funzioni di similarità, utilizzate per comprendere il trend delle performance.

Una sezione chiave della tesi è stata quella successiva, riguardante la programmazione dei componenti. L'obiettivo è stato quello di capire come interagiscono tra loro i diversi blocchi del notebook. Successivamente, è stato affrontato l'argomento della messa in funzione del sistema, iniziando dall'avvio, passando per gli output intermedi con i possibili errori ed arrivando ai risultati ottenuti. Infine, è stata affrontata la discussione in merito al lavoro svolto, illustrando la SWOT Analysis e le relative lezioni apprese durante lo sviluppo.

8.2 Sfide future

Il software analizzato in questo elaborato è un punto di partenza per molti sviluppi futuri. Infatti, potrebbe integrarsi in diversi ambiti di risposta alle domande, come potrebbe essere un chatbot di assistenza clienti.

Tuttavia, per arrivare all'inserimento del software in un sistema automatizzato, è necessario avere un miglioramento dei risultati. Per questo fine, potrebbe essere interessante creare un proprio modello di Huggingface personalizzato, basato su un DPR preesistente. In tale modello si potrebbe applicare il freeze dei layer interni così da avere una backbone solida da cui partire. Successivamente, si potrebbero addestrare esclusivamente i layer iniziali e finali. Ciò migliorerebbe sostanzialmente le prestazioni e, nonostante ciò, il tempo di train sarebbe di gran lunga minore. Inoltre, lo sveltimento del processo di train favorisce il fatto di poter effettuare più test, apportando modifiche ai parametri. Infatti, a causa dei limiti temporali, non si sono provate varie configurazioni che avrebbero potuto aumentare le prestazioni.

Un'altra idea per uno sviluppo futuro risiede nell'aggiunta di altre funzioni di similarità per verificare diversi aspetti della risposta data in output, che non sono stati considerati durante lo sviluppo.

Un'ultima idea interessante sarebbe quella di integrare altri moduli forniti da Haystack; ad esempio, sarebbe interessante combinare nella Pipeline il `Reader`, il quale darebbe risposte più accurate. Ovviamente, aumenterebbe esponenzialmente il carico computazionale, motivo per il quale non è stato considerato in fase di progettazione.

Bibliografia

AI4BUSINESS (2022), «Cos'è il Machine learning, come funziona l'apprendimento automatico e quali sono le sue applicazioni», URL https://www.ai4business.it/intelligenza-artificiale/machine-learning/machine-learning-cosa-e-applicazioni, [Online; accessed 5-11-2022].

BRIGGS, J. (2022), «Long Form Question Answering in Haystack», URL https://www.pinecone.io/learn/haystack-lfqa, [Online; accessed 20-11-2022].

CELLI, F. (2019), «Machine Learning e Data Science: una storia di rivoluzioni», URL https://www.developersmaggioli.it/blog/machine-learning-e-data-science-una-storia-di-rivoluzioni, [Online; accessed 7-11-2022].

CHAN, B. (2020), «Question Answering at Scale With Haystack», URL https://medium.com/deepset-ai/haystack-question-answering-at-scale-c2c980e7c657, [Online; accessed 27-11-2022].

CHIUSANO, F. (2022), «Two minutes NLP — Quick Introduction to Haystack», URL https://medium.com/nlplanet/two-minutes-nlp-quick-introduction-to-haystack-da86d0402998, [Online; accessed 18-11-2022].

DOTTI, P. (2022), «Python, come funziona, esempi e librerie», URL https://www.ai4business.it/intelligenza-artificiale/machine-learning/python-come-funziona-esempi-e-libreri/, [Online; accessed 15-11-2022].

HAYSTACK (2022a), «Tutorial: Preprocessing Your Documents», URL https://haystack.deepset.ai/tutorials/08_preprocessing, [Online; accessed 14-11-2022].

HAYSTACK (2022b), «Tutorial: Training Your Own Dense Passage Retrieval Model», URL https://haystack.deepset.ai/tutorials/09_dpr_training, [Online; accessed 11-11-2022].

HAYSTACK (2022c), «What is Haystack?», URL https://haystack.deepset.ai/overview/intro, [Online; accessed 9-11-2022].

IBM (2021), «Doker», URL https://www.ibm.com/it-it/cloud/learn/docker, [Online; accessed 26-11-2022].

INTELLIGENZAARTIFICIALE.IT (2022), «MACHINE LEARNING», URL https://www.intelligenzaartificiale.it/machine-learning, [Online; accessed 2-11-2022].

LA-LA, F. (2018), «Artificially Intelligent - Uso di Jupyter Notebook», URL https://learn.microsoft.com/it-it/archive/msdn-magazine/2018/february/artificially-intelligent-using-jupyter-notebooks, [Online; accessed 16-11-2022].

LEXALYTICS (2022), «Machine Learning (ML) for Natural Language Processing (NLP)», URL https://www.lexalytics.com/blog/machine-learning-natural-language-processing, [Online; accessed 3-11-2022].

MISHRA, P. (2019), «Understanding Dense Passage Retrieval (DPR) System», URL https://towardsdatascience.com/understanding-dense-passage-retrieval-dpr-system-bce5aee4fd40, [Online; accessed 17-11-2022].

PIGRO (2021), «Storia dell'intelligenza artificiale: il machine learning e i sistemi esperti», URL https://blog.pigro.ai/it/storia-intelligenza-artificiale-machine-learning-sistemi-esperti, [Online; accessed 3-11-2022].

PROVINO, A. (2022), «Hugging Face: Free GitHub Natural Language Processing Models», URL https://andreaprovino.it/hugging-face/#:~:text=Hugging, [Online; accessed 18-11-2022].

SACHELI, G. (2014), «Come funziona TF-IDF e relazioni con la SEO», URL https://www.evemilano.com/cosa-significa-tf-idf-e-perche-e-importante, [Online; accessed 22-11-2022].

WHEATLEY, M. (2022), «Deepset, creator of the open-source NLP framework Haystack, raises 14M», URL https://siliconangle.com/2022/04/28/deepset-creator, [Online; accessed 21-11-2022].

WIKIPEDIA (2022a), «Coseno di similitudine — Wikipedia, The Free Encyclopedia», URL https://it.wikipedia.org/wiki/Coseno_di_similitudine, [Online; accessed 18-11-2022].

WIKIPEDIA (2022b), «Indice di Jaccard — Wikipedia, The Free Encyclopedia», URL https://it.wikipedia.org/wiki/Indice_di_Jaccard, [Online; accessed 18-11-2022].

WIKIPEDIA (2022c), «Machine learning — Wikipedia, The Free Encyclopedia», URL https://en.wikipedia.org/wiki/Machine_learning, [Online; accessed 2-11-2022].

WIKIPEDIA (2022d), «Natural language processing — Wikipedia, The Free Encyclopedia», URL https://en.wikipedia.org/wiki/Natural_language_processing, [Online; accessed 4-11-2022].

Ringraziamenti

Arrivato alla fine della mia carriera universitaria, sento il desiderio di ringraziare tutti coloro che mi hanno aiutato e sostenuto durante questo percorso.

Desidero ringraziare coloro che hanno dato il loro contributo nella realizzazione di questa tesi di laurea. Ringrazio, innanzitutto, il mio relatore, il Professore Domenico Ursino, che mi ha accompagnato in questo lungo percorso, facendosi trovare sempre disponibile e sciogliere i miei dubbi grazie alla sua esperienza e capacità.

Vorrei ringraziare la mia famiglia, che ha sempre creduto in me e mi ha aiutato a raggiungere tale traguardo. In particolare, ringrazio i miei genitori, che mi hanno sostenuto economicamente e moralmente in ogni mia scelta; ringrazio i miei fratelli, che mi hanno donato esperienza e spensieratezza; ringrazio i miei nonni, sempre pronti a difendermi in ogni occasione; infine, ringrazio il resto della famiglia con cui ho passato bei momenti nel corso degli anni.

Inoltre, vorrei ringraziare tutti coloro che hanno intrapreso il percorso di laurea al mio fianco e, in generale, tutti gli amici con cui mi sono relazionato e con cui ho condiviso bellissimi ricordi durante questo splendido periodo.