DocuGenie

Repository GitHub del progetto
Repository GitHub relativa all'estrazione

Descrizione del progetto

L'applicativo permette la consultazione di una RAG (Retrieval-Augmented Generation) che estrae informazioni pertinenti in risposta alle richieste specifiche degli utenti. Il sistema può essere completamente eseguito localmente nel browser, permettendo l'uso di modelli e LLM per la ricerca e la generazione di testo, senza la necessità di fare affidamento a server backend. Il progetto inoltre fornisce un sistema di regole per facilitare il processo di retrieval.

Sviluppo

Tecnologie utilizzate

Attualmente sono state utilizzate le seguenti tecnologie per lo sviluppo dell'applicativo:

Tecnologia	Versione	Descrizione
Vite	5.4.10	Uno strumento di compilazione e server di sviluppo realizzato per fornire una esperienza di sviluppo rapida, fluida e altamente performante.
Vitest	2.1.4	Un framework di testing per JavaScript e TypeScript basato su Vite
React	18.3.1	Una libreria JavaScript open-source che permette di creare UI interattive e dinamiche con componenti dichiarativi
PGlite	0.2.12	Una build WASM di Postgres contenuta in una client library JavaScript/TypeScript, che permette di eseguire Postgres nel browser o Node.js senza altre dipendenze
Transformers.js	3.0.3	Libreria sviluppata per essere equivalente a tranformers di python
WebLLM	0.2.74	Un motore di inferenza LLM per il browser che fornisce modelli di inferenza sui browser con l'accelerazione hardware
json-logic-engine	2.1.0	Libreria per la creazione di set di condizioni in modo sicuro, permettendo la persistenza nel database e la condivisione tra front-end e back-end

Tecnologia	Versione	Descrizione
ReactFlow	12.3.4	Componente React personalizzabile per la creazione di editor per diagrammi

Inoltre, sono state utilizzate le seguenti tecnologie per gestire l'estrazioni dei dati da PDF:

Tecnologia	Versione	Descrizione
PyMuPDF	1.24.10	Libreria python per l'estrazione di dati, l'analisi e la manipolazione di file PDF
Transformers	4.38.2	Libreria che fornisce API per scaricare ed usare modelli
scikit-learn	1.5.2	Un set di moduli per machine learning e data mining
langchain	0.3.2	Libreria che fornisce un set di strumenti per costruire applicazioni basate su LLM

Linguaggi utilizzati

Linguaggio	Versione
Typescript	5.6.3
Python	3.12.4

RAG

Le **LLM** (Large Language Models) hanno dimostrato una capacità notevole di generare testo di qualità simile a quella umana, rispondere a domande, tradurre frasi e molto altro. Tuttavia, una delle principali limitazioni di questi modelli è il loro affidamento esclusivo ai dati di addestramento. Poiché le LLM non possono accedere a informazioni aggiornate o contestuali al di fuori del loro training, possono produrre risposte errate, vecchie o poco accurate. Questo problema diventa particolarmente evidente quando si ha a che fare con domande specifiche, per cui i dati nel loro training non sono sufficienti.

Il paradigma **RAG** (Retrieval-Augmented Generation) nasce per risolvere questa problematica, integrando le capacità di generazione delle LLM con una componente di retrieval, cioè il recupero di informazioni rilevanti in tempo reale. Questo approccio consente di ottenere risposte più precise, superando i limiti di un modello statico e basato solo su conoscenze vecchie. Quando si parla di RAG, si intende:

- **Retrieval**: Quando un utente inserisce una richiesta o un prompt, il sistema RAG inizia con la fase di retrieval. In questa fase, viene eseguita una ricerca in una serie di documenti (che può essere costituita da testi, articoli, database o qualsiasi fonte di dati). Il sistema seleziona i documenti o frammenti più rilevanti, basandosi su algoritmi di ranking come **BM25** o metodi di similarità semantica. L'obiettivo è fornire alla LLM un contesto specifico e pertinente che possa guidare la generazione del testo.
- **Augmented**: Questa fase non si limita semplicemente a fornire alla LLM i dati recuperati, ma utilizza varie tecniche per ottimizzare l'apprendimento contestuale della LLM. Qui entrano in gioco approcci come il **chunking**, che suddivide documenti di grandi dimensioni in parti più piccole e gestibili, o la **fusione delle informazioni** (RAG Fusion), che combina diverse fonti di dati per aumentare la ricchezza del contesto. Questo arricchimento permette alla LLM di avere una comprensione più approfondita del contenuto e di rispondere in modo più corretto.
- **Generation**: Una volta arricchito il contesto grazie alle informazioni recuperate, la LLM utilizza queste informazioni per generare una risposta. La LLM ora non si basa più esclusivamente sui dati di addestramento pre esistenti, ma sfrutta in maniera attiva le informazioni rilevanti recuperate in tempo reale. Questo permette al modello di rispondere in modo molto più accurato e specifico, integrando dati specifici che rispondono direttamente alla richiesta dell'utente.

Vantaggi

- **Aggiornamento continuo**: Poiché le informazioni vengono recuperate in tempo reale, la RAG può rispondere con dati aggiornati e corretti anche se l'LLM è stato addestrato su un corpus precedente.
- **Flessibilità**: Il sistema può essere facilmente adattato a nuovi domini o set di dati, migliorando le risposte su argomenti specifici.
- **Miglioramento della precisione**: Recuperando e integrando informazioni pertinenti, la LLM è in grado di ridurre gli errori e fornire risposte più contestuali e precise.

Limitazioni

- **Dipendenza da informazioni esterne**: Nonostante il sistema RAG offra una grande flessibilità grazie al recupero di informazioni, esso è strettamente legato alla qualità e all'accuratezza dei dati esterni su cui si basa. Se il sistema accede a documenti o fonti di informazioni che contengono errori, dati incompleti o informazioni obsolete, anche la risposta generata dall'LLM sarà influenzata negativamente. Questo è particolarmente problematico quando la qualità delle fonti non è verificata o le informazioni recuperate sono ambigue.
- **Limitazioni delle tecnologie di ricerca attuali**: Molti sistemi RAG si affidano a tecniche di ricerca tradizionali come la ricerca vettoriale, che utilizza rappresentazioni numeriche

per trovare somiglianze tra query utente e documenti. Tuttavia, queste tecnologie, pur avanzate, possono risultare insufficienti per alcune applicazioni. BM25, ad esempio, è un potente strumento per il retrieval testuale ma ha i suoi limiti quando le query sono vaghe o complesse.

- **Ambiguità delle query umane**: Un'altra problematica da non sottovalutare riguarda il modo in cui gli utenti formulano le loro richieste. Spesso le query inserite sono poco chiare o incomplete. Gli utenti potrebbero non esprimere con precisione quello che cercano, il che può portare a problemi nella fase di retrieval, dove non vengono recuperate tutte le informazioni chiave.

BM25

BM25 è un algoritmo che considera la frequenza con cui i termini di ricerca appaiono in un documento e li normalizza in base alla lunghezza del documento stesso. Questo approccio permette di restituire risultati più rilevanti, evitando il bias verso documenti più lunghi o termini comuni. Questa tecnica è ottima per organizzare collezioni di documenti, come le librerie digitali.

Elementi chiave di BM25

- TF (Term Frequency): Conta il numero di volte in cui un termine cercato appare in ciascun documento.
- IDF (Inverse Document Frequency): Dà più importanza ai termini meno frequenti, assicurandosi che le parole più comuni non dominino.
- Document Length Normalization: Normalizza il ranking rispetto alla dimensione del documento, evitando che i documenti più lunghi siano sempre rilevanti.
- Query Term Saturation: Evita che i risultati siano distorti da termini ripetuti successivamente.

Quando è ideale BM25

- Grandi collezioni di documenti: Si adatta bene per database dove è necessario navigare diverse informazioni e dati
- Bilanciare la rilevanza: Riduce il bias associato alla lunghezza dei documenti o alla frequenza di termini comuni.
- Recupero generale di informazioni: È utile in vari scenari di ricerca, offrendo un mix di semplicità e efficacia nel fornire risultati pertinenti.

Applicazione pratica: ricerca ibrida

La ricerca ibrida può essere immaginata come una lente di ingrandimento che non guarda solo la superficie ma esplora in maniera profonda. Questo metodo sfrutta sia la ricerca basata su keyword che la ricerca vettoriale per offrire una visione più completa del contenuto:

- Keyword search: Input di una parola o una frase, e la ricerca estrae e perfeziona i termini esatti o quelli simili fra di loro nel database o la collezione dei documenti.
- Vector search: Diversamente dal keyword search, vector search non si accontenta di semplici parole. Funziona usando significato semantico, mirando a discernere il contesto sottostante della query o del suo significato. Questo assicura che anche se le parola non combacia un documento perfettamente, se il significato è rilevante, verrà preso.

RAG-Fusion

RAG-Fusion combina il retrieval con la generazione, sfruttando una LLM per creare delle variazioni di query in base alla domanda iniziale dell'utente, in modo catturare in modo più efficiente l'intento originale dell'utente. Inoltre, RAG-Fusion introduce una tecnica di re-ranking per combinare il risultato di diverse query, per organizzare i risultati di ricerca in un ranking unificato, migliorando l'accuratezza delle informazioni rilevanti.

Come funziona

1. RAG-Fusion riformula la query dell'utente traducendola in altre query simili ma distinte attraverso una LLM
2. Inizializza i vettori di ricerca per la query originale e le sue query di ricerca simili generate.
3. Combina e raffina tutti i risultati delle query attraverso RRF
4. Sceglie i migliori risultati con le nuove query, fornendo un contesto sufficiente alla LLM per creare una risposta considerando tutte le query e una lista riclassificata di risultati.

Metodologie di chunking

Sono state individuate le seguenti metodologie di chunking:

Sentence Level Chunking

Questo approccio divide il testo in frasi in base alla punteggiatura. È semplice ed efficace per task dove le ogni frase frase ha un significato completo e conciso. Tuttavia, può non preservare il contesto tra frasi, risultando quindi problematico per task che richiedono un contesto più ampio.

Paragraph-Level Chunking

Questo metodo effettua il chunking del testo in paragrafi, mantenendo più informazioni contestuali rispetto al chunking a livello di frasi. È particolarmente utile per la classificazione di documenti o task dove il contesto all'interno di un singolo blocco di discorso è rilevante. La limitazione è che non tutti i paragrafi sono uguali in lunghezza o densità di informazioni e ciò può portare a processing loads irregolari.

Character Chunking / Fixed Level Chunking

Questa strategia divide il testo in chunk in base a un numero fisso di caratteri o parole. La sua semplicità lo rende un'ottima strategia di inizio, semplificando la modellazione e la computazione. Tuttavia, chunk di dimensioni fisse tendono a perdere il contesto o informazioni critiche.

Recursive Character Chunking

Questa strategia divide il testo in chunk affinché certe condizioni siano rispettate, come la quantità minima di chunk. Questo metodo assicura che il processo di chunking si allinei con la struttura del testo, preservando maggiore significato. La sua adattabilità rende il Recursive Character Chunking ottimo per i testi con strutture variegate.

Document Specific Chunking

Document Specific Chunking è una strategia che rispetta la struttura del documento. Invece di usare un certo numero di caratteri o un processo ricorsivo, questa tecnica crea dei chunk che si allineano con la separazione logica del documento, come paragrafi o sottosezioni. Questo approccio mantiene l'organizzazione originale del contenuto redatto dall'autore e e aiuta a mantenere la coerenza del testo, rendendo le informazioni ottenute rilevanti e utili, in particolare per documenti strutturati con sezioni ben definite.

Token-based Chunking

Questa strategia suddivide il testo in chunk in base a criteri predeterminati in base ai token. Ogni token rappresenta una unità minimale di significato. I token possono essere parole, caratteri o la punteggiatura, in base alla metodologia di tokenizzazione selezionata.

Semantic Chunking

Questa strategia considera le relazioni all'interno del testo: divide il testo in chunk significativi e semanticamente completi. Questo approccio assicura che l'integrità dell'informazione durante il recupero, portando a risultati più accurati e contestualmente appropriati. Il chunking semantico consiste ne prendere gli embeddings di ciascuna frase nel documento, paragonando la similarità di tutte le frasi tra di esse, e poi raggruppando insieme le frasi con l'embedding più simile.

Concentrandosi sul significato del testo e il contesto, Semantic Chunking migliora significativamente la qualità del recupero. È una scelta ottima quando è importante mantenere l’integrità semantica.

Tuttavia, questo metodo richiede più sforzo ed è notevolmente più lento delle altre strategie.

Agent Chunking

Con la strategia di Agent Chunking si parte dall'inizio del documento, trattando la prima parte come un chunk. Successivamente si prosegue nel documento, decidendo se una nuova frase o un pezzo di informazione appartiene al primo chunk o se è necessario iniziarne uno nuovo. Infine si prosegue con il processo definito fino alla fine del documento.

Window-Based Chunking

Questa strategia consiste nella creazione di chunk di dimensione fissa con sovrapposizione tra chunk consecutivi. Questa sovrapposizione può aiutare a mitigare la perdita di contesto visto nel chunking a dimensione fissa assicurando che l'informazione ai lati dei chunk sia anche considerata in congiunzione con i chunk vicini. Questa strategia bilancia tra il mantenere contesto e processing loads gestibili ma aumenta l'overhead computazionale a causa del processing ridondante nella aree in sovrapposizione.

Dynamic Chunking

Gli algoritmi di dynamic chunking aggiustano la dimensione e i confini dei chunk in base al contenuto, come porre fine ai chunk ad interruzioni linguistiche naturali o cambiamenti di tematiche. Questo approccio è più flessibile e più preservante del contesto rispetto al chunking di dimensioni fisse, ma richiede algoritmi complessi e adattivi, che siano in grado di analizzare e comprendere la struttura del testo in tempo reale.

Quale strategia utilizzare

- Preservazione del contesto ? chunking **dinamico** e **semantico** perché mantengono la continuità a logica e tematica all'interno dei chunk, preservando contesto e riducendo la probabilità di generare risposte non sequitur.
- Efficienza computazionale ? **fixed-length** e **window based** chunking, poiché offrono dimensioni di chunk prevedibili e gestibili, facilitando il processing di dati efficiente e facilitando la scalabilità tra risorse computazionali distribuite.
- Accuratezza e qualità di recupero ? **window-based** chunking, perché aiuta ad assicurare che le informazioni importanti ai confini dei chunk non siano persi. Semantic chunking eccelle allineando i confini di chunk con le interruzioni naturali nel contenuto, catturando quindi completamente le idee.

- Adattabilità e model training ? **dynamic** chunking, perché è in grado di aggiustare le strutture di testo e contenuti variegati. Questa adattabilità lo rende ideale per allenare modelli su diversi dataset, portando a generalizzazioni migliori su tutti i tipi di testo.
- Processing in real-time ? **sentence-level** e **fixed-length** chunking perché sono semplici e veloci da implementare. Questi metodi permettono un processing veloce dei dati, il ché è essenziale in scenari che richiedono una risposta immediata.
- Gestione di grandi documenti ? **paragraph-level** chunking può essere effettivo perché aiuta a mantenere un bilancio tra la preservazione del contesto e dimensioni di chunk maneggiabili, assicurando che ciascun chunk contenga un'idea completa e indipendente.

Un altro fattore importante da tenere in considerazione è la dimensione dei chunk: con chunk troppo piccoli si rischia di perdere contesto e di aumentare l'overhead computazionale causato da un processing notevolmente elevato; chunks larghi invece potrebbero rallentare il modello, specialmente in ambienti con risorse limitate, oppure potrebbero contenere informazioni non necessarie, rendendo difficile l'estrazione delle informazioni più rilevanti.

Dimensione chunk	Tempo di risposta medio (s)	Fedeltà media	Rilevanza media
128	1.55	0.85	0.78
256	1.57	0.90	0.78
512	1.66	0.85	0.85
1024	1.68	0.93	0.90
2048	1.72	0.90	0.89

I chunk con 1024 caratteri risultano essere i migliori in termini di fedeltà, offrendo un ottimo bilancio tra rilevanza e un tempo di risposta.

Approcci ibridi

Nella pratica, la strategia di chunking ottimale richiede un approccio che combina elementi di tutte le strategie che raggiungono molteplici obiettivi. Per esempio, un sistema può usare chunking semantico per determinare interruzioni naturali e successivamente applicare chunking fixed-length tra blocchi semantici più larghi per bilanciare la preservazione del contesto con l’efficienza computazionale.

Le migliori strategie di chunking per un sistema RAG quindi dipendono dai requisiti specifici dell'applicazione, della natura del dataset, e dal bilancio desiderato tra l'accuratezza, l’efficienza, e la scalabilità.

Rimozione dell'intestazione e piè di pagina da documenti PDF

Per l'estrazione del testo da PDF e altri tipi di documenti, è stata suggerita la libreria Apache Tika. **Apache Tika** è un toolkit che permette di estrarre testo e metadati da oltre mille tipi di file, tra cui PDF, Word, HTML, attraverso un'unica interfaccia. Questa flessibilità rende Tika particolarmente utile per una varietà di applicazioni, come l'indicizzazione per motori di ricerca, l'analisi del contenuto e la traduzione automatica.

Tuttavia, è emersa una problematica significativa: Tika non offre un controllo granulare nell'elaborazione dei file PDF. In particolare, non permette di gestire in modo efficace la rimozione di elementi strutturali non desiderati come intestazioni e piè di pagina ripetitivi. Questi elementi possono influenzare negativamente la qualità del testo estratto, introducendo rumore nelle successive fasi di analisi e processamento, soprattutto quando si hanno documenti strutturati in modo complesso o con layout ricchi di grafici e tabelle.

Per risolvere questa limitazione, è stata adottata la libreria PyMuPDF esclusivamente per la gestione dei PDF. **PyMuPDF** offre un controllo più fine sui contenuti del documento, permettendo di accedere e manipolare le diverse parti del PDF, inclusa la possibilità di eliminare o modificare in modo selettivo le intestazioni, i piè di pagina e altri elementi grafici o testuali superflui.

Strategie di rimozione

Le strategie con cui si sono ottenuti i risultati migliori sono state due: DBSCAN e l'estrazione dei blocchi che rientrano nei threshold per l'header e footer.

DBSCAN

DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*) è un algoritmo che trova cluster basati sulla densità dei punti dati. È particolarmente efficace nel rilevare cluster di forme irregolari e può identificare punti dati isolati come rumore. Questa strategia è molto utile quando si vuole rimuovere header e footer senza interventi manuali a discapito della velocità e inconsistenza nell'estrazione. Infatti, non è possibile rimuovere tutte le intestazioni e piè di pagina da tutti i PDF vista l'assenza di una struttura ben definita per questo tipo di file.

```
def remove_hf(pdf_path):
 document = fitz.open(pdf_path)
 n_pages = document.page_count
 if n_pages == 1:
  document.insert_file(pdf_path)
  coordinates = {'x0': [], 'y0': [], 'x1': [], 'y1': [], 'width': [],
```

```
'height': []}
 for page in document:
  blocks = page.get_text('blocks')
 for block in blocks:
  coordinates['x0'].append(block[0])
  coordinates['y0'].append(block[1])
  coordinates['x1'].append(block[2])
  coordinates['y1'].append(block[3])
  coordinates['width'].append(block[2] - block[0])
  coordinates['height'].append(block[3] - block[1])

 df = pd.DataFrame(coordinates)

 quantile = 0.15
 upper = np.floor(df['y0'].quantile(1 - header_threshold))
 lower = np.ceil(df['y1'].quantile(footer_threshold))
 x_min = np.floor(df['x0'].min())
 x_max = np.ceil(df['x1'].max())
 y_min = np.floor(df['y0'].min())
 y_max = np.ceil(df['y1'].max())

 # Frequenza header/footer
 hff = 0.8
 min_clust = min_cluster_size = int(np.floor(n_pages * hff))
 if min_clust < 2:
  min_clust = 2
  hdbscan = HDBSCAN(min_cluster_size = min_clust)
  df['clusters'] = hdbscan.fit_predict(df)
  df_group = df.groupby('clusters').agg(
  avg_y0=('y0','mean'), avg_y1=('y1','mean'),
  std_y0=('y0','std'), std_y1=('y1','std'),
  max_y0=('y0','max'), max_y1=('y1','max'),
  min_y0=('y0','min'), min_y1=('y1','min'),
  cluster_size=('clusters','count'), avg_x0=('x0', 'mean')).reset_index()

 df_group = df_group.sort_values(['avg_y0', 'avg_y1'], ascending=[True,
True])

 std = 0
 footer = np.floor(df_group[(np.floor(df_group['std_y0']) == std) &
(np.floor(df_group['std_y1']) == std) & (df_group['min_y0'] >= upper) &
(df_group['cluster_size'] <= n_pages)]['min_y0'].min())
 header = np.ceil(df_group[(np.floor(df_group['std_y0']) == std) &
(np.floor(df_group['std_y1']) == std) & (df_group['min_y1'] <= lower) &
(df_group['cluster_size'] <= n_pages)]['min_y1'].max())
```

```
 if not pd.isnull(footer):
  y_max = footer
 if not pd.isnull(header):
  y_min = header

 # Coordinate blocco del corpo del documento
 return x_min, y_min, x_max, y_max
```

Soluzione alternativa

La seconda soluzione consiste nell'estrarre i blocchi di testo da ogni pagina, ordinarli dal più alto al più basso, individuare i blocchi che rientrano nei threshold per gli header e footer e rimuoverli dalla lista di blocchi totali. Questa strategia è notevolmente consistente e veloce ma richiede un intervento manuale per regolare i threshold qualora non fossero adatti.

```
HEADER_THRESHOLD = 0.1 # 10% of the page
FOOTER_THRESHOLD = 0.9 # 90% of the page

EXCLUDE_HEADERS = True
EXCLUDE_FOOTERS = True

# Function to detect headers and footers
# It takes a pdf path and an optional exclude range,
# extracts the blocks for each page, sorts them by y-coordinate,
# ignores the blocks within the header and footer thresholds
# and then creates a cropped pdf with the body of the pages,
# a json with the chunks and the page numbers and a text file with the
body of the pages
def remove_hf(pdf_path, exclude_range=None):
 document = fitz.open(pdf_path)
 page_rectangles = []
 cropped_document = fitz.open()
 text_with_pages = []
 exclude_pages = []

 try:
  if exclude_range:
   exclude_pages = parse_range(exclude_range)
  else:
   exclude_pages = []
 except:
  print("Invalid range format. Please use comma-separated numbers or
number ranges (e.g., 1,2,3-5).")
  return []
```

```
 for page_num in range(len(document)):
  page_to_exclude = page_num + 1

  if page_to_exclude in exclude_pages:
   continue

  page = document.load_page(page_num)
  blocks = page.get_text("blocks")
  blocks = sorted(blocks, key=lambda b: b[1])
  header_blocks = []
  footer_blocks = []
  body_blocks = []

  page_height = page.rect.height
  header_threshold = 0
  footer_threshold = page_height

  if EXCLUDE_HEADERS:
   header_threshold = page_height * HEADER_THRESHOLD

  if EXCLUDE_FOOTERS:
   footer_threshold = page_height * FOOTER_THRESHOLD

  for block in blocks:
   if block[1] < header_threshold:
    header_blocks.append(block)
   elif block[1] > footer_threshold:
    footer_blocks.append(block)
   else:
    body_blocks.append(block)

  if body_blocks:
   min_y = min(block[1] for block in body_blocks)
   max_y = max(block[3] for block in body_blocks)
  else:
   min_y = header_threshold
   max_y = footer_threshold

  rect = fitz.Rect(0, min_y, page.rect.width, max_y)
  page_rectangles.append(rect)

  cropped_page = page.set_cropbox(rect)
  cropped_document.insert_pdf(document, from_page=page_num,
to_page=page_num)

  body_text = "\n".join([block[4] for block in body_blocks])
```

```
  text_with_pages.append((body_text, page_num + 1))

 cropped_output_path = "./output/cropped_output.pdf"
 cropped_document.save(cropped_output_path)
 cropped_document.close()

 chunks, chunk_page_numbers =
recursive_chunking_with_pages(text_with_pages)

 metadata = {
  "title": document.metadata.get("title", "Unknown"),
  "author": document.metadata.get("author", "Unknown"),
  "num_pages": len(document),
  "num_chunks": len(chunks),
 }

 output_data = {
  "metadata": metadata,
  "chunks": [{ "index": i, "page": chunk_page_numbers[i], "text": chunk }
for i, chunk in enumerate(chunks)]
 }

 json_output_path = "./output/output.json"
 with open(json_output_path, 'w', encoding='utf-8') as json_file:
  json.dump(output_data, json_file, ensure_ascii=False, indent=4)

 txt_output_path = "./output/output.txt"
 with open(txt_output_path, 'w', encoding='utf-8') as txt_file:
  txt_file.write("\n".join([text for text, _ in text_with_pages]))
```

La funzione **remove_hf** applica la soluzione alternativa per creare un file json contente i metadati e un array di testi con le corrispettive pagine originali di appartenenza, insieme ad un file pdf con header e footer rimossi e un file di testo per convenienza. Si può usare una funzione che ottiene il corpo del testo insieme ad un array di pagine, effettua il chunking usando la strategia desiderata e assegna la pagina di appartenenza a ogni chunk creato.

```
def recursive_chunking_with_pages(text_with_pages, chunk_size=800,
chunk_overlap=300):
 all_text = ""
 page_boundaries = []
 current_position = 0

 for text, page_number in text_with_pages:
  all_text += text
  page_boundaries.append((current_position, current_position + len(text),
```

```
page_number))
  current_position += len(text)

 result = RecursiveCharacterTextSplitter(
  chunk_size=chunk_size,
  chunk_overlap=chunk_overlap,
  length_function=len,
  is_separator_regex=False,
  separators=["\n\n", "\n", ".", "?", "!", " ", ""],
 ).create_documents([all_text])

 chunks = [doc.page_content for doc in result]
 chunk_page_numbers = []

 for chunk in chunks:
  chunk_start = all_text.find(chunk)
  chunk_end = chunk_start + len(chunk)

 for start, end, page_number in page_boundaries:
  if start <= chunk_start < end or start < chunk_end <= end:
   chunk_page_numbers.append(page_number)
   break

 return chunks, chunk_page_numbers
```

Descrizione dell'applicativo

Al fine di sperimentare le funzionalità di PGlite, Transformers.js e WebLLM è stata creata una semplice interfaccia web con Vite, React e la libreria di componenti Shadcn.

Database

Per salvare gli embeddings si fa uso di PGlite, che non è altro che PostgreSQL in WASM.

```
const db = new PGlite("idb://rag-app", {
 extensions: {
  vector,
  pg_trgm,
  ltree,
  adminpack
 },
});

export const initSchema = async (db) => {
 await db.exec(`
```

```
  create extension if not exists vector;
 create extension if not exists pg_trgm; -- Per BM25
 create extension if not exists ltree;
 create extension if not exists adminpack;

 create table if not exists chunks (
 id bigint primary key generated always as identity,
 page_id integer,
 chunk_id integer,
 content text not null,
 embedding vector (384)
 );

 create table if not exists rules (
 id serial primary key,
 name text not null,
 conditions text not null,
 page integer not null,
 prompt text not null,
 salience integer not null,
 parent_id integer references rules(id),
 created_at timestamp default current_timestamp
 );

 create index if not exists chunks_hnsw on chunks using hnsw (embedding
vector_ip_ops);
 create index if not exists chunks_gin on chunks using gin (content
gin_trgm_ops); -- Index per BM25
  `);
};
```

PGlite ha uno strato di file system virtuale che gli consente di essere eseguito in ambienti dove normalmente non può avere accesso al filesystem. Gli sviluppatori di PGlite consigliano di utilizzare un VFS IndexedDB al momento poiché OPFS non è ancora supportato da Safari.

Una volta ottenuti gli embeddings, essi possono essere caricati con un semplice insert per poter effettuare successivamente la ricerca.

```
/**
* Searches the database for chunks matching a query.
* @param {PGliteWorker} pg - The PGliteWorker instance.
* @param {any[]} embedding - The embedding to search for.
* @param {string} query - The query to search for.
* @param {number} match_threshold - The threshold for matching (default is
0.8).
```

```
* @param {number} limit - The maximum number of results to return (default
is 3).
*/
export const search = async (
 pg: PGliteWorker,
 embedding: any[],
 query: string,
 match_threshold = 0.8,
 limit = 3,
) => {
 let vectorResults;
 /**
 * Search for chunks using vector similarity
 * <#> is a system operator for computing the inner product of two
vectors.
 * It determines whether the vectors point in the same or opposite
directions.
 * The inner product is negative, because PostgreSQL only supports ASC
order index scans on operators.
 * @see https://github.com/pgvector/pgvector?tab=readme-ov-file#querying
 */
 vectorResults = await pg.query(
  `
  select * from chunks
  where chunks.embedding <#> $1 < $2
  order by chunks.embedding <#> $1
  limit $3;
  `,
  [JSON.stringify(embedding), -Number(match_threshold), Number(limit)],
 );

 /**
 * Search for chunks using BM25
 * ts_rank_cd is a system function for computing a score showing how well
a  tsvector matches a tsquery
 * to_tsvector converts the text into a vector of words, ignoring stop
words
 * plainto_tsquery converts the query into a vector of words
 * @@ is a boolean operator that checks if the query is a subset of the
text
 * these operators are built-in operators provided by postgres
 */
 const bm25Results = await pg.query(
  `
  select *, ts_rank_cd(to_tsvector(content), plainto_tsquery($1)) as rank
  from chunks
```

```
  where to_tsvector(content) @@ plainto_tsquery($1)
  order by rank desc
  limit $2;
  `,[query, Number(limit)],
 );



 const combinedResults = [...vectorResults.rows, ...bm25Results.rows]
 .sort((a: any, b: any) => (a.distance || 0) - (b.distance || 0) + (b.rank
|| 0) - (a.rank || 0)).slice(0, limit);

 return combinedResults.map((result: any) => ({
  content: `- ${result.content}`,
  page_id: result.page_id,
  }));
};
```

La funzione effettua la ricerca nearest neighbor (vicini più prossimi), usando un criterio di similarità basato sul prodotto interno. La soluzione fornita introduce anche BM25 per realizzare un sistema di ricerca ibrida. Questa funzione è particolarmente veloce ma la qualità dei risultati dipende dagli embedding generati.

Transformers.js e WebLLM

Al fine di testare sia Transformers.js e WebLLM, per generare gli embedding è stato utilizzato **Transformers.js**, mentre per la generazione del testo è stato impiegato **WebLLM**.

Le **LLM** possono richiedere risorse computazionali significative e, se non gestite correttamente, rischiano di essere scaricate ogni volta che la pagina viene ricaricata. Per risolvere questa problematica, si è fatto uso di Web Workers e React hooks. I **Web Workers** permettono di eseguire operazioni in background, evitando che attività pesanti come la generazione di embedding o testo blocchino il thread principale. I **React hooks** invece semplificano la gestione dello stato e degli effetti collaterali all'interno dell'applicazione React.

È stato creato un file `worker.ts` che contiene due classi, una dedicata a **Transformers.js** e una a **WebLLM**. Entrambe le classi seguono il pattern singleton, il che significa che viene creata un'unica istanza della classe solo quando necessario, evitando così di crearla nuovamente ad ogni utilizzo.

```
import { pipeline } from '@huggingface/transformers'
import { CreateMLCEngine, InitProgressCallback } from '@mlc-ai/web-llm';

/**
```

```
 * Singleton class for the embedding pipeline.
 * @class
 * @property {string} task - The task to use for the pipeline.
 * @property {string} model - The model to use for the pipeline.
 * @property {any} instance - The instance of the pipeline.
 */
class EmbeddingPipelineSingleton {
  static task: string = 'feature-extraction';
  static model: string = 'Supabase/gte-small';
  static instance: any = null;

  static async **getInstance**(**progress_callback**: ((_progress_: any)
=> void) | null = null) {
    if (this.instance === null) {
      this.instance = pipeline(this.task as any, this.model, {
        progress_callback: progress_callback as Function | undefined,
        dtype: 'fp32',
        device: 'gpu' in navigator ? 'webgpu' : 'wasm',
      });
    }
    return this.instance;
  }
}

/**
 * Singleton class for the text generation pipeline.
 * @class
 * @property {string} model - The model to use for the text generation.
 * @property {any} instance - The instance of the text generation.
 */
class TextGenerationPipelineSingleton {
  static model: string = 'Llama-3.2-1B-Instruct-q4f32_1-MLC';
  static instance: any = null;

  static async **getInstance**(**progress_callback**: ((_progress_: any)
=> void) | null = null) {
    if (!('gpu' in navigator)) {
      throw new Error('GPU not supported');
    }
    if (this.instance === null) {
      this.instance = await CreateMLCEngine(this.model, {
        **initProgressCallback**: progress_callback as
InitProgressCallback | undefined,
      });
    }
    return this.instance;
```

```
  }
}
```

Diversamente da quanto mostrato, entrambe le due classi possono essere configurabili sia per generare embedding, che per produrre il testo.

https://github.com/mlc-ai/web-llm/blob/main/examples/embeddings/src/embeddings.ts

```
// RAG with Langchain.js using WebLLM for both LLM and Embedding in a
single engine
// Followed
https://js.langchain.com/v0.1/docs/expression_language/cookbook/retrieval/
// There are many possible ways to achieve RAG (e.g. degree of integration
with Langchain,
// using WebWorker, etc.). We provide a minimal example here.
async function simpleRAG() {
  // 0. Load both embedding model and LLM to a single WebLLM Engine
  const embeddingModelId = "snowflake-arctic-embed-m-q0f32-MLC-b4";
  const llmModelId = "gemma-2-2b-it-q4f32_1-MLC-1k";
  const engine: webllm.MLCEngineInterface = await webllm.CreateMLCEngine(
    [embeddingModelId, llmModelId],
    {
      initProgressCallback: initProgressCallback,
      logLevel: "INFO", // specify the log level
    },
  );

  const vectorStore = await MemoryVectorStore.fromTexts(
    ["mitochondria is the powerhouse of the cell"],
    [{ id: 1 }],
    new WebLLMEmbeddings(engine, embeddingModelId),
  );
  const retriever = vectorStore.asRetriever();

  const prompt =
    PromptTemplate.fromTemplate(`Answer the question based only on the
following context:
  {context}

  Question: {question}`);

  const chain = RunnableSequence.from([
    {
      context: retriever.pipe(formatDocumentsAsString),
      question: new RunnablePassthrough(),
```

```
    },
    prompt,
  ]);

  const formattedPrompt = (
    await chain.invoke("What is the powerhouse of the cell?")
  ).toString();
  const reply = await engine.chat.completions.create({
    messages: [{ role: "user", content: formattedPrompt }],
    model: llmModelId,
  });

  console.log(reply.choices[0].message.content);

  /*
    "The powerhouse of the cell is the mitochondria."
  */
}
```

La soluzione tradizionale per chiamare i Worker in JavaScript è la seguente:

```
const worker = new Worker(new URL('./worker.ts', import.meta.url), {
 type: 'module',
})
```

Vite tuttavia offre un modo più semplice per importare i worker, usando il suffisso ?worker nell'importazione:

```
import Worker from './worker.ts?worker'

const worker = new Worker()
```

Supporto

> **Supporto**
>
> A partire da Novembre 2024, secondo caniuse.com il supporto globale per WebGPU è di circa 72%, il che significa che potrebbe non essere disponibile per alcuni utenti. Se il progetto non funziona nel browser, potrebbe essere necessario attivare WebGPU tramite i feature flag:
>
> - Firefox: con il flag `dom.webgpu.enabled` (guarda qui).
> - Safari: con il feature flag `WebGPU` (guarda qui).

- Versioni meno recenti dei browser Chromium (su Windows, macOS, Linux): con il flag `enable-unsafe-webgpu` (guarda qui).

Per utilizzare i modelli PyTorch, TensorFlow, o JAX con Transformers.js, i modelli devono essere ricompilati per supportare ONNX, il runtime per eseguire i modelli sul browser. Il developer che sviluppa Transformers.js consiglia questo script di conversione basato su 🤗 Optimum. È importante indicare che al momento Transformers.js riporta il seguente errore `Error: Could not locate file (500 error)` cercando di interagire con LLM o modelli di text generation.

WebLLM invece supporta solamente i modelli MLC.

Note

Per ridurre i tempi di attesa durante il caricamento dei dati, ho sperimentato diverse strategie per ottimizzare l'inserimento degli embedding nel database. Ho provato infatti a caricare file JSON contenenti embedding pre-calcolati, inserendoli direttamente nel database. Questa soluzione ha mostrato un notevole miglioramento delle prestazioni, poiché viene completamente ignorata la fase di classificazione e calcolo degli embedding.

In parallelo, ho testato il caricamento di file JSON separati, dove ogni file rappresenta un singolo chunk di testo con gli embedding già inclusi. Ho riscontrato che la dimensione complessiva dei file risultava più contenuta rispetto al caricamento di un unico file di grandi dimensioni contenente tutti i chunk (da ~12,2mb a ~9,7mb).

Per un file PDF di 128 pagine, suddiviso in chunk da 800 caratteri con un overlap di 300 caratteri, la velocità media di inserimento per un file unico con embedding pre-calcolati è di circa 9 secondi. Utilizzando invece file JSON separati per ciascun chunk, il tempo di inserimento aumentava leggermente, impiegando in media uno o due secondi in più. Questa differenza è probabilmente dovuta al sovraccarico del gestire più file separati, ma la gestione di file più piccoli può comunque offrire vantaggi in termini di flessibilità e organizzazione.

Quando ho caricato invece un semplice file JSON contenente solo il corpo del testo (sia chunkato che non), e ho effettuato la generazione degli embedding e l'inserimento nel database in tempo reale, l'intera procedura non ha superato 3 minuti. Questa lentezza può essere attribuita alla natura di PGlite, che utilizza un singolo thread (il thread principale del browser) e supporta una sola connessione.

Per mitigare queste limitazioni, ho optato per l'utilizzo della funzionalità Multi-tab Worker, offerta da PGlite, che separa il carico dal thread principale, migliorando l’efficienza e riducendo i tempi di esecuzione. Questo approccio ha migliorato significativamente le performance, riducendo il

tempo necessario per l'intera procedura ad un minuto. Multi-Tab Worker inoltre permette di interagire con il database da più di una scheda o finestra.

```
// pg-worker.ts
worker({
 async init(options) {
  const pglite = new PGlite({
   ...options,
   extensions: {
    vector,
    pg_trgm,
    ltree,
    adminpack
   }
  })
  return pglite
 },
})
```

```
const pg = new PGliteWorker(
 new Worker(new URL('./pglite-worker.ts', import.meta.url), {
  type: 'module',
 }),
 {
  dataDir: 'idb://rag-app',
 }
)
```

Motore di Regole

Per ottimizzare il processo di retrieval, è stato suggerito di implementare un motore di regole basato su un sistema if-then, utilizzando Drools. Drools sfrutta l'algoritmo RETE, che consente una valutazione efficiente delle condizioni delle regole per generare il risultato desiderato. Tuttavia, per rendere l'applicazione completamente utilizzabile dal lato client, si è optato per una soluzione personalizzata, evitando l'uso di Drools. Tale scelta ha portato all'adozione di un algoritmo proprietario, progettato per risolvere le diverse problematiche emerse, affrontate in modo incrementale:

- **Gestione dei conflitti** (in caso di regole multiple soddisfatte)
- **Persistenza su database**
- **Rimozione di nodi intermedi**
- **CRUD delle regole tramite interfaccia utente**

- **Valutazione simultanea di più condizioni**

Soluzione Architetturale

Classe `RuleNode`

La classe `RuleNode` rappresenta un nodo o foglia in una struttura ad albero di regole. Ogni nodo/foglia contiene informazioni su una specifica regola, incluse le condizioni da soddisfare affinché la regola sia valida e possa essere applicata.

Classe `RulesEngine`

La classe `RulesEngine` gestisce le regole e offre un meccanismo di valutazione rispetto a un set di fatti. Questo motore funge da supervisore della logica di matching tra regole e condizioni, centralizzando quindi il processo decisionale.

Interazione tra `RuleNode` e `RulesEngine`

L'interazione tra `RuleNode` e `RulesEngine` si realizza con la valutazione delle regole. Le regole vengono aggiunte al motore, dove ciascun nodo può avere figli, formando una struttura ad albero. Quando viene avviato il processo di valutazione, il `RulesEngine` analizza ciascuna regola radice, utilizzando la classe `LogicEngine` per verificare le condizioni di ogni `RuleNode`. Se una regola radice è soddisfatta, il motore valuta ricorsivamente i figli per trovare ulteriori regole valide. In caso di regole multiple soddisfatte, il motore sceglie la regola con priorità più alta. Se esistono più regole soddisfatte con la stessa priorità (salienza), viene applicata la strategia LIFO (Last In, First Out) per selezionare la regola più recente. Una volta individuata la regola appropriata, `RulesEngine` restituisce la pagina e il prompt da utilizzare corrispondenti, per restituire la pagina specifica da cui ottenere i chunk oppure utilizzare un prompt specifico.

Gestione e Persistenza delle Regole

Inizialmente, la soluzione prevedeva l'uso di funzioni per definire condizioni e azioni. Se tutte le funzioni nell'array delle condizioni restituivano un valore boolean positivo, allora veniva eseguita la funzione di azione associata. Tuttavia, il salvataggio delle regole nel database richiedeva la conversione delle funzioni in stringhe, una soluzione che presentava gravi criticità sia in termini di sicurezza sia di efficienza. Per risolvere questo problema, si è scelto di utilizzare la libreria json-logic-engine, che permette la creazione delle regole in formato JSON, rendendo così il processo più sicuro e semplificando la gestione delle regole tramite interfaccia utente.

Le regole sono strutturate nel seguente formato JSON:

Logica:

```
{
    "cat": [
        "Hello, ",
        {
            "var": "input"
        },
        "!"
    ]
}
```

Input:

```
"World"
```

Risultato:

```
"Hello, World!"
```

Questa struttura consente di creare regole complesse utilizzando vari operatori disponibili:

```
{
    "and": [
        {"find": [{"var": "input"}, "testo da individuare"]},
        {"find": [{"var": "input"}, "altro testo da individuare"]},
        {"or": [...]}
        ...
    ]
}
```

Con questa configurazione, è possibile verificare il soddisfacimento di molteplici condizioni all'interno di una singola regola. Gli operatori sono completamente personalizzabili, permettendo di creare ed implementare nuovi operatori in base alle necessità.

```
const logicEngine = new LogicEngine();
logicEngine.addMethod("find", ([str, keyword]: [string, string]) =>
 new RegExp(`\\b${keyword}\\b`, 'i').test(str)
);
```

Interfaccia e Visualizzazione delle Regole

Per una gestione visiva efficace delle regole e delle loro relazioni, è stata realizzata una tabella riepilogativa con tutte le informazioni rilevanti, affiancata da un grafico interattivo creato con ReactFlow. Tuttavia, a causa delle limitazioni imposte alle funzionalità avanzate di ReactFlow, si è scelto di utilizzare esclusivamente le feature gratuite, come l'eliminazione dei nodi e la modifica tramite doppio clic.

La creazione delle regole è stata semplificata per restituire unicamente la pagina contenente le informazioni da visualizzare e il prompt da passare al modello LLM. Questo approccio migliora la gestione delle regole e ottimizza l'esperienza utente, riducendo complessità e potenziali punti di errore.

Valutazione affidabilità dei risultati

Esistono diverse tecniche per valutare l'affidabilità e la correttezza delle risposte generate dai modelli. Di seguito vengono descritte quelle più comuni.

Valutazione correttezza in base a ground truth

La principale tecnica individuata per valutare la qualità e affidabilità delle risposte è tramite l'utilizzo di dataset di risposte verificate e accurate ("ground truth") come benchmark per valutare le performance e la qualità delle risposte. Per le RAG, i dataset ground truth consentono di valutare sia il retrieval che le risposte generate, paragonandole alle risposte verificate del ground truth. La valutazione si basa su quattro metriche principali: Context Precision, Context Recall, Faithfulness e Response relevancy.

Context Precision

Context Precision misura la proporzione di blocchi informativi rilevanti (chunk) identificati nei contesti recuperati. Si calcola con la media del $Precision@k$ per ogni chunk presente nel contesto, dove $Precision@k$ rappresenta il rapporto tra i blocchi rilevanti tra i primi k risultati e il totale dei blocchi in tale range.

$$\text{Context Precision@K} = \frac{\sum_{k=1}^{K} (Precision@k \times v_k)}{\text{Numero totale elementi rilevanti tra i migliori K risultati}}$$

$$Precision@k = \frac{\text{veri positivi@k}}{(\text{veri positivi@k} + \text{falsi positivi@k})}$$

Dove K è il numero totale dei chunk nei context rilevati e $v_k \in \{0, 1\}$ l'indicatore di rilevanza al rank k. I veri positivi al range k sono i blocchi rilevanti identificati dal modello al range k.

Tipi di approcci dell Context Precision

1. Precisione del contesto basata su LLM (senza riferimento): utilizza un LLM per valutare la rilevanza del contesto rispetto all'input dell'utente e alla risposta.
2. Precisione del contesto basata su LLM (con riferimento): utilizza un LLM per confrontare ciascun contesto recuperato con una risposta di riferimento per determinare la pertinenza.
3. Precisione del contesto non basata su LLM (con contesti di riferimento): applica misure di distanza tradizionali e non LLM per valutare la rilevanza dei contesti recuperati rispetto ai contesti di riferimento forniti.

Context Recall

Context Recall misura quanti documenti rilevanti sono stati recuperati con successo, riducendo al minimo la perdita delle informazioni importanti. Un recall elevato significa che pochi documenti rilevanti sono stati tralasciati.

LLM Based Context Recall

Questa metrica si basa sull'input dell'utente, sul ground truth e sui contesti recuperati, utilizzando valori tra 0 e 1, dove valori più elevati indicano performance migliori. Questa metrica utilizza il riferimento come proxy per i context recuperati, il che la rende anche più facile da usare, poiché l'annotazione dei contesti di riferimento può richiedere molto tempo. Per stimare il richiamo del contesto dal riferimento, il riferimento viene suddiviso in affermazioni: ciascuna affermazione nella risposta di riferimento viene analizzata per determinare se può essere attribuita al contesto recuperato o meno. In uno scenario ideale, tutte le affermazioni nella risposta di riferimento dovrebbero essere attribuibili al contesto recuperato.

$$\text{context recall} = \frac{|\text{Affermazioni del GT che possono essere attribuite al contesto}|}{|\text{Numero delle affermazioni nel GT}|}$$

Dove GT sta per "Ground Truth".

Non LLM Based Context Recall

Questa metrica è calcolata utilizzando i context recuperati, il riferimento ai context e i valori che vanno da 0 a 1, con i valori elevati che indicano performance migliori. Questa tecnica usa metriche di confronto non basate su LLM per identificare se il context recuperato è rilevante o meno.

$$\text{context recall} = \frac{|\text{Numero dei contesi rilevanti recuperati}|}{|\text{Numero totale di contesti di riferimento}|}$$

Faithfullness

La metrica della fedeltà valuta la coerenza fattuale della risposta generata rispetto al contesto fornito. Viene calcolata dalla risposta e dal contesto recuperato. La risposta è scalata

all'intervallo (0,1). Più alto è il punteggio, meglio è. La risposta generata è considerata fedele se tutte le affermazioni fatte nella risposta possono essere dedotte dal contesto dato. Per calcolare ciò, viene prima identificato un insieme di affermazioni dalla risposta generata. Ciascuna di queste affermazioni quindi viene sottoposta a un controllo con il contesto dato per determinare se può essere dedotta dal contesto. Il punteggio di fedeltà è dato da:

$$\text{Faithfulness score} = \frac{|\text{Numero delle affermazioni nella risposta generata che possono essere dedotte dal}}{|\text{Numero totale delle affermazioni nella risposta generata}|}$$

Tipi di approcci per la fedeltà

1. Fedeltà di base: utilizza un semplice metodo di controllo incrociato in cui le affermazioni nella risposta vengono convalidate direttamente rispetto al contesto recuperato.
2. Fedeltà con HHEM-2.1-Open: utilizza il modello HHEM-2.1-Open di Vectara, un classificatore basato su T5, per rilevare le allucinazioni nel testo generato. Questo modello aiuta a identificare le affermazioni non supportate, migliorando l'affidabilità delle valutazioni di fedeltà.

Response Relevancy

Questa metrica mira ad esaminare quando le risposte generate siano pertinenti al prompt dato. Un punteggio minore è assegnato alle risposte che sono incomplete o contengono informazioni ridondanti, mentre un punteggio elevato indica una rilevanza migliore. La pertinenza della risposta viene calcolata come la similarità media del coseno tra l'embedding della domanda originale e l'embedding di più domande artificiali generate in base alla risposta del modello.

$$\text{Responce Relevancy} = (\frac{1}{N}) \times \sum_{i=1}^{N} \cos\left(E_{g_i}, E_o\right)$$

Dove:

- E_{g_i} è l'embedding dell i-esima domanda generata.
- E_o è l'embedding delle domanda originale.
- N è il numero delle domande generate.

Feedback umano

Questa tecnica prevede l'uso del feedback umano per stabilire criteri di valutazione chiari e consistenti. I valutatori vengono chiamati a esprimere un giudizio su aspetti come la rilevanza, la correttezza e la completezza delle risposte. Tuttavia, il feedback umano può spesso risultare soggetto a variabilità e influenze soggettive, specialmente se i criteri di valutazione non sono ben definiti o la natura della task è soggettiva.

Per migliorare la coerenza e la qualità dei giudizi, esistono diverse strategie: una prima opzione consiste nel calcolare la media delle valutazioni ottenute dai diversi valutatori, riducendo così le discrepanze e i bias individuali; alternativamente, si può adottare un approccio basato sull'accordo tra i valutatori, prendendo in considerazione solo i casi in cui si raggiunge un consenso. In tal modo, si ottiene un set di giudizi più coerente e affidabile, garantendo che il feedback raccolto sia di alta qualità e possa contribuire a valutazioni più oggettive.

Threshold vector similarity search

Questa strategia prevede l'uso dello score della ricerca vettoriale per determinare una soglia di approvazione o esclusione dei documenti recuperati. Un punteggio elevato indica un alto grado di similarità tra il documento e la query, suggerendo una maggiore rilevanza e permettendo di stabilire criteri di selezione più stringenti. Con un threshold alto, si garantisce una selezione più accurata, limitando il set di documenti ai più pertinenti e riducendo al minimo il rumore informativo.

Con un threshold più basso invece , si amplia il pool di documenti selezionati. Questa impostazione può risultare vantaggiosa in scenari esplorativi, dove una maggiore quantità di informazioni è preferibile per ottenere una visione più ampia. Tuttavia, soglie basse possono introdurre una quantità significativa di contenuti meno rilevanti, rendendo necessario un bilanciamento tra quantità e qualità per evitare un eccesso di dati non pertinenti.

```
// query_results
// Lo "score" è il valore della similarità del coseno
{'matches': [{'id': 'vec1', 'score': 1.0, 'values': [1.0, 1.5]},
{'id': 'vec2', 'score': 0.868243158, 'values': [2.0, 1.0]},
{'id': 'vec3', 'score': 0.850068152, 'values': [0.1, 3.0]}],
```

Usare diversi LLM per giudicare le risposte

Un ulteriore metodo per valutare l'affidabilità delle risposte consiste nel generare risposte utilizzando diversi modelli (LLM) e confrontarle tra loro. Nonostante questa tecnica non rappresenti una soluzione ideale, può essere utile per ottenere una stima della qualità del recupero dei dati e della generazione delle risposte, specialmente in assenza di un dataset che fornisca un ground truth di domande, contesti e risposte corrette.

Confrontando le risposte fornite da diversi modelli, si possono identificare le similitudini e le discrepanze, valutando così la coerenza e la qualità dei risultati. Se le risposte concordano su aspetti rilevanti, è probabile che tali contenuti siano affidabili. Viceversa, risposte divergenti possono indicare zone di incertezza o errori nel recupero delle informazioni e nella generazione delle risposte, suggerendo un'area che potrebbe beneficiare di ulteriori perfezionamenti.

Inoltre, questo approccio offre un modo per identificare errori o bias specifici di un modello. Ad esempio, se un modello fornisce ripetutamente risposte meno complete o inaccurate rispetto ad altri, si possono trarre indicazioni sui suoi punti deboli e sui miglioramenti necessari.

TODO

- https://github.com/lyogavin/airllm
- CPU inference
- Database sync https://pglite.dev/docs/sync
- Docker WASM
- Differenze lingua italiana, inglese ecc
- https://www.crunchydata.com/blog/hnsw-indexes-with-postgres-and-pgvector#using-hnsw--a-code-sample
- https://www.postgresql.org/docs/current/gin.html
- funzioni di distanza pgvector