Hybrid Search a method to Optimize RAG implementation

Akash Chandrasekar Follow 11 min read · Feb 24, 2024

Hybrid Search: Combining Traditional Keyword-Based Search with Modern Vector Search

In the context of Retrieval-Augmented Generation (RAG) pipelines, developers are actively exploring the complexities of achieving production-ready performance. Analogous to various endeavors, the Pareto Principle applies here as well. While accomplishing the initial 80% is relatively straightforward, the remaining 20% required for full production readiness presents significant challenges.

Unveiling the Essence of Hybrid Search:

At its core, hybrid search is akin to an alchemist's concoction, seamlessly merging disparate search algorithms to distill a potion of unparalleled relevance. Picture traditional keyword-based search as the sturdy foundation, rooted in precise term matching but vulnerable to the whims of typos and synonyms. Contrastingly, the advent of vector or semantic search introduces a shimmering veil of context-awareness, transcending linguistic barriers and typographical errors. By fusing these elements, hybrid search births a synergy that transcends the limitations of individual methodologies, unveiling a treasure trove of insights previously concealed beneath layers of digital clutter.

Navigating the Landscape of Search Techniques:

Delving deeper, let's navigate the contrasting landscapes of keyword-based and vector searches. The former, reminiscent of a seasoned detective, diligently scours the digital realm for exact matches, adept at unearthing specific nuggets of information. Yet, its Achilles' heel lies in its susceptibility to linguistic nuances, often leaving vital clues obscured in the shadows. On the flip side, semantic search emerges as a beacon of enlightenment, traversing the semantic web with finesse, unfazed by linguistic barriers or minor deviations. However, its reliance on high-quality vector embeddings and vulnerability to out-of-domain terms present formidable challenges.

The Art of Synthesis:

Crafting Relevance through Hybrid Search: Enter hybrid search, the virtuoso conductor orchestrating a symphony of search algorithms. Like a master chef blending ingredients to perfection, hybrid search amalgamates the precision of keyword-based search with the contextual acumen of semantic search. Imagine querying “How to merge two Pandas DataFrames with .concat()?” — a conundrum that defies conventional search paradigms. Here, keyword search dutifully unveils results pertaining to the “.concat()” method, while semantic search lends its expertise in recognizing synonyms like “combine” and “join”, enriching the search experience manifold.

Unlocking the Potential:

Harnessing Hybrid Search for the Future: As we gaze into the crystal ball of technological evolution, the potential of hybrid search gleams brightly on the horizon. From e-commerce to academia, the applications are boundless — streamlining research endeavors, enhancing customer experiences, and transcending linguistic barriers with ease. Yet, with great power comes great responsibility; the quest for relevance must be tempered with a commitment to ethical data practices and user privacy.

Implementation of Hybrid Search process

Hybrid search is a sophisticated approach that seamlessly combines two distinct search methodologies: keyword-based search and vector-based search. By leveraging the strengths of both approaches, hybrid search delivers more accurate and relevant search results.

Keyword-based search,

Keyword-based search, also known as sparse vector search, relies on generating sparse embeddings where most values in the vector are zero except for a few non-zero values. These sparse embeddings are created using

algorithms such as BM25, which builds upon the TF-IDF (Term Frequency-Inverse Document Frequency) approach.

Keyword-based search, also known as sparse vector search, relies on generating sparse embeddings where most values in the vector are zero except for a few non-zero values. These sparse embeddings are created using algorithms such as BM25, which builds upon the TF-IDF (Term Frequency-Inverse Document Frequency) approach.

In a keyword-based search:

1. Preprocessing: The search query and documents are preprocessed to extract relevant keywords and remove noise.
2. Sparse Embedding Generation: For each document in the corpus, a sparse embedding is generated based on the frequency of keywords within that document compared to their frequency across all documents. This is typically done using algorithms like BM25.
3. Query Embedding: The search query is also converted into a sparse embedding using the same algorithm.
4. Similarity Calculation: The cosine similarity or another similarity metric is then used to compare the query embedding with each document's embedding, determining the relevance of each document to the query.
5. Ranking: The documents are ranked based on their similarity scores, with the most relevant documents appearing at the top of the search results.

Here's a Python code example illustrating these steps:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
```

```
# Sample documents
documents = [
    "This is a sample document containing some keywords.",
    "Keywords are important for keyword-based search.",
    "Document analysis involves extracting keywords.",
    "Keyword-based search relies on sparse embeddings."
]
# Sample search query
query = "Keyword-based search"

# Step 1: Preprocessing (Skipped for brevity)
import re

def preprocess_text(text):
    # Convert text to lowercase
    text = text.lower()
    # Remove punctuation
    text = re.sub(r'[^\w\s]', '', text)
    return text

# Sample documents
documents = [
    "This is a sample document containing some keywords.",
    "Keywords are important for keyword-based search.",
    "Document analysis involves extracting keywords.",
    "Keyword-based search relies on sparse embeddings."
]

# Sample search query
query = "Keyword-based search"

# Preprocess documents
preprocessed_documents = [preprocess_text(doc) for doc in documents]

# Preprocess query
preprocessed_query = preprocess_text(query)

print("Preprocessed Documents:")
for doc in preprocessed_documents:
    print(doc)
```

```
print("\nPreprocessed Query:")
print(preprocessed_query)

# Step 2: Sparse Embedding Generation (BM25)
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(documents)

# Step 3: Query Embedding
query_embedding = vectorizer.transform([query])

# Step 4: Similarity Calculation (Cosine Similarity)
similarities = cosine_similarity(X, query_embedding)

# Step 5: Ranking
ranked_indices = np.argsort(similarities, axis=0)[::-1].flatten()
ranked_documents = [documents[i] for i in ranked_indices]

# Output the ranked documents
for i, doc in enumerate(ranked_documents):
    print(f"Rank {i+1}: {doc}")
```

This code demonstrates the essential steps of keyword-based search, including generating sparse embeddings using TF-IDF (TfidfVectorizer), computing cosine similarity, and ranking the documents based on their similarity scores.

Vector search method

Vector search, also known as dense vector search, utilizes modern machine learning algorithms to generate dense numerical representations (vectors) of data objects such as text documents. These dense vectors contain rich semantic information and are typically comprised of non-zero values.

1. Vector Embedding: Each document in the corpus is transformed into a dense vector representation using advanced machine-learning models like Transformers. These models encode the semantic meaning of the text into dense numerical vectors, capturing relationships and nuances in the data.

2. Query Embedding: Similarly, the search query is converted into a dense vector representation using the same embedding model. This ensures that the query is represented in the same semantic space as the documents.
3. Similarity Calculation: Once the query and document embeddings are obtained, a similarity metric such as cosine similarity is applied to calculate the similarity between the query vector and each document vector. Cosine similarity measures the cosine of the angle between two vectors and provides a measure of similarity ranging from -1 to 1, with higher values indicating greater similarity.
4. Ranking: The documents are ranked based on their similarity scores, with those having higher similarity scores to the query appearing at the top of the search results.

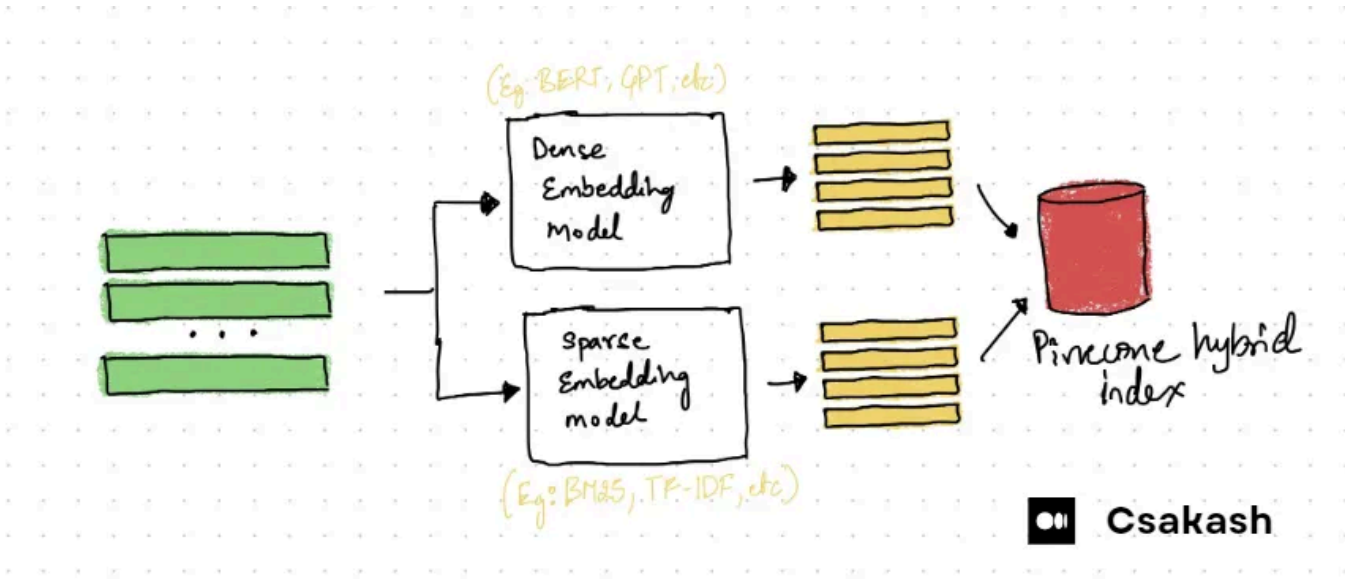
Here's a simplified Python code example demonstrating vector search using pre-trained word embeddings:

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Sample documents (represented as dense vectors)
document_embeddings = np.array([
    [0.634, 0.234, 0.867, 0.042, 0.249],
    [0.123, 0.456, 0.789, 0.321, 0.654],
    [0.987, 0.654, 0.321, 0.123, 0.456]
])

# Sample search query (represented as a dense vector)
query_embedding = np.array([[0.789, 0.321, 0.654, 0.987, 0.123]])

# Calculate cosine similarity between query and documents
similarities = cosine_similarity(document_embeddings, query_embedding)

# Rank documents based on similarity scores
ranked_indices = np.argsort(similarities, axis=0)[::-1].flatten()
```

```
# Output the ranked documents
for i, idx in enumerate(ranked_indices):
    print(f"Rank {i+1}: Document {idx+1}")
```

Mixing vector search and keyword search for Hybrid search

Combining hybrid and vector search follows a similar principle to the fusion of keyword-based and vector search results. However, in this scenario, we're integrating the outcomes of a hybrid search, which has already combined keyword-based and vector search techniques, with the results of a standalone vector search.



Once again, the integration involves scoring the results from both searches, followed by a weighted combination to produce a unified ranking. The alpha parameter, as mentioned earlier, plays a pivotal role in determining the influence of each approach.

Let's delve into the process:

1. Scoring the Results: The search results from both the hybrid and vector searches are scored based on a chosen metric, such as cosine distance or positional rank within the results list. Each result receives a numerical score reflecting its relevance to the search query.
2. Weighted Combination: The scores obtained from the hybrid search (which already incorporates both keyword-based and vector search) and the standalone vector search are combined using the alpha parameter. The hybrid_score is calculated using the formula:

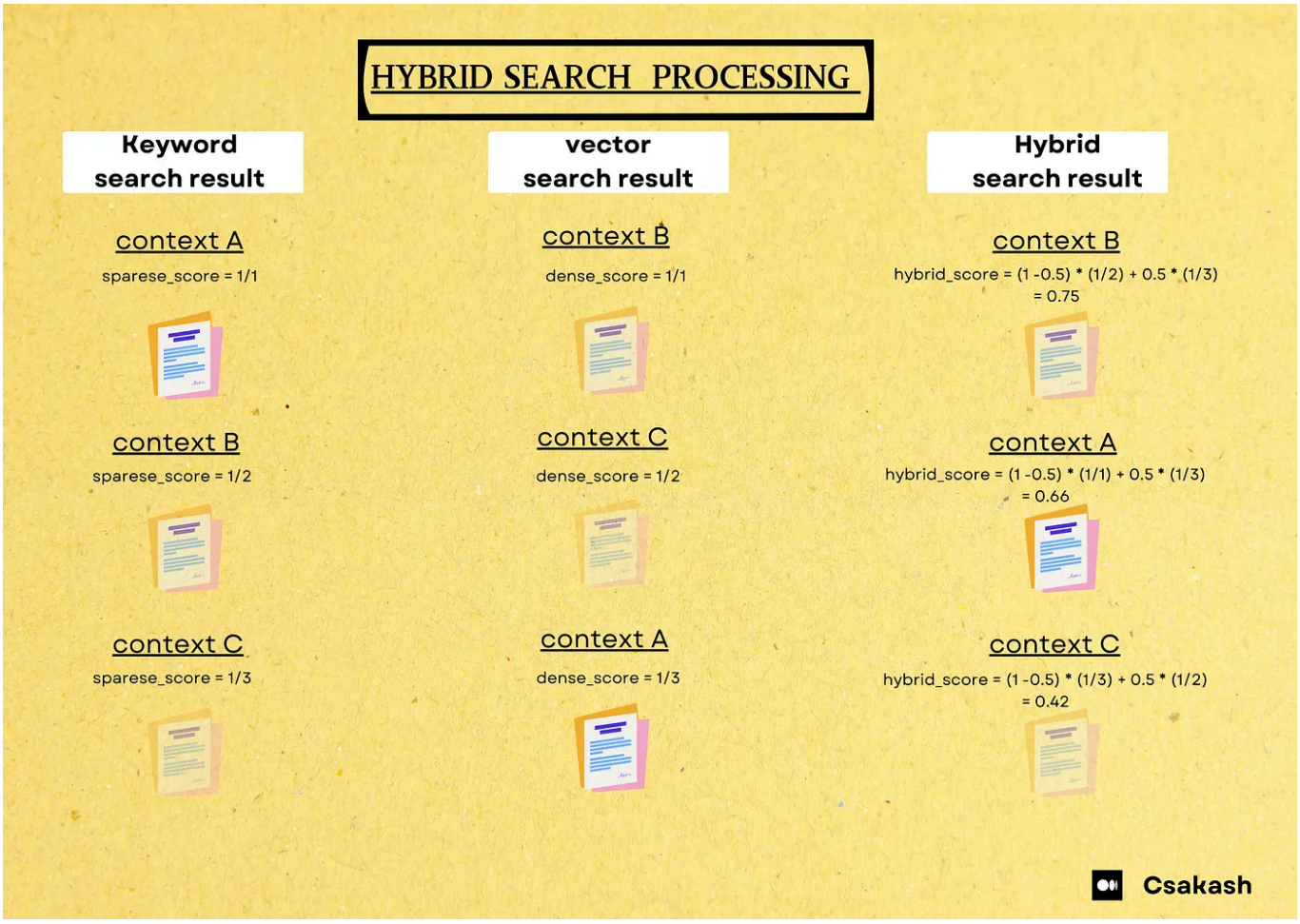
```
hybrid_score = (1 - alpha) * sparse_score + alpha * dense_score
```

Where:

- hybrid_score is the score obtained from the hybrid search.
- vector_score is the score obtained from the standalone vector search.
- alpha determines the weight assigned to each score, with values typically ranging between 0 and 1.
- `alpha = 1`: Pure vector search
- `alpha = 0`: Pure keyword search

1. Re-ranking: The combined scores are used to re-rank the search results. Results with higher hybrid scores are prioritized in the final ranking, reflecting their enhanced relevance to the search query.

This integration of hybrid and vector search leverages the strengths of both approaches, further enhancing the precision and relevance of search outcomes. By seamlessly blending the contextual awareness of hybrid search with the robustness of vector search, users are presented with a more comprehensive and accurate set of results, facilitating more effective information discovery and retrieval.



How Hybrid Search Elevates RAG Pipeline Performance

Enhancing the performance of a RAG (Retrieval-Augmented Generation) pipeline entails adjusting various parameters to fine-tune its efficacy. Among these parameters, optimizing the relevance of retrieved-context emerges as a pivotal aspect. This significance stems from the fact that the retrieved context serves as input for the Language Model (LM), and if it fails

to align with the given question, the LM's ability to generate a pertinent answer is compromised.

Tailoring the search technique to your specific context type and query is crucial for maximizing the benefits of your RAG application. Hence, the parameter alpha, dictating the balance between keyword-based and semantic search, assumes the role of a hyperparameter necessitating meticulous calibration.

In a standard RAG pipeline leveraging LangChain, configuring the retriever component involves designating the vector store component as the retriever using the .as_retriever() method:

```
# Define and populate vector store

vectorstore = #chose your own vectore store
# Set vectorstore as retriever
retriever = vectorstore.as_retriever()
```

However, this approach exclusively facilitates semantic search. To enable hybrid search functionality within LangChain, a dedicated retriever component with hybrid search capabilities must be defined.

Example:

`WeaviateHybridSearchRetriever`:

```
from langchain.retrievers.weaviate_hybrid_search import WeaviateHybridSearchRetr
retriever = WeaviateHybridSearchRetriever(
    alpha = 0.5,                # defaults to 0.5, which is equal weighting betwe
    client = client,            # keyword arguments to pass to the Weaviate clien
    index_name = "LangChain",   # The name of the index to use
    text_key = "text",          # The name of the text key to use
    attributes = [],            # The attributes to return in the results
)
```

The rest of the vanilla RAG pipeline will stay the same.

This small code change allows you to experiment with different weighting between keyword-based and vector searches. Note that setting `alpha = 1` equals a fully semantic search as is the equivalent of defining the retriever from the `vectorstore` component directly (`retriever = vectorstore.as_retriever()`).

When Hybrid search be utilized ?

1. **Information Retrieval Systems:** Hybrid search can improve the performance of information retrieval systems such as search engines, knowledge bases, and document repositories. By combining keyword-based and semantic search techniques, these systems can deliver more accurate and contextually relevant results to users.

2. **E-commerce Platforms:** E-commerce websites often rely on search functionality to help users find products quickly and efficiently. Hybrid search can enhance product search by considering both keyword matches (such as product names and descriptions) and semantic similarities (related products, synonyms) to provide more tailored recommendations.

3. **Customer Support and FAQs:** Hybrid search can be employed in customer support portals and FAQ sections to assist users in finding relevant answers to their queries. By combining keyword-based search with semantic understanding, these systems can better match user questions with appropriate answers, improving user satisfaction and reducing support overhead.

4. **Academic and Research Databases:** Academic search engines and research databases can benefit from hybrid search techniques to improve the discoverability of scholarly articles, papers, and other academic resources. By combining keyword-based indexing with semantic analysis of content, these platforms can offer more comprehensive search results to researchers and students.

5. **Healthcare Information Systems:** In healthcare, hybrid search can be applied to medical databases and electronic health records to assist healthcare professionals in finding relevant patient information, research articles, treatment protocols, and diagnostic guidelines. By combining

keyword-based search with semantic understanding of medical concepts, hybrid search can support more accurate clinical decision-making.

6. **Legal and Regulatory Compliance:** Legal professionals and compliance officers can leverage hybrid search techniques to navigate complex legal documents, case law databases, and regulatory frameworks more effectively. By combining keyword-based search with semantic analysis of legal concepts and terminology, hybrid search can facilitate more efficient legal research and compliance analysis.

Real-time example:

Consider the case of Stack Overflow, a platform renowned for its vast repository of coding solutions. In its quest to enhance search capabilities, Stack Overflow has recently embraced semantic search, thereby augmenting its existing keyword-based approach with a more context-aware methodology.

Originally, Stack Overflow relied on TF-IDF to match keywords with relevant documents. However, articulating coding problems can be inherently nuanced, leading to variations in search results based on the choice of words. For instance, tasks like combining Pandas DataFrames could be described using different terms like merging, joining, or concatenating. In such scenarios, a context-aware search method like semantic search proves invaluable.

Conversely, many users frequent Stack Overflow to troubleshoot errors by directly copying and pasting error messages. Here, precise keyword matching reigns supreme. Furthermore, for queries involving method and argument names (e.g., .read_csv() in Pandas), exact keyword matching is indispensable.

Thus, the real-world utility of Stack Overflow exemplifies the symbiotic relationship between context-aware semantic searches and exact keyword matching. This synergy underscores the importance of implementing a hybrid search retriever component, which can seamlessly integrate both approaches to cater to a diverse array of user needs.

End of Optimization

This article delves into the realm of hybrid search, which amalgamates keyword-based and vector searches to enhance the relevance of search results. By combining outcomes from separate search algorithms and recalibrating the rankings, hybrid search offers a comprehensive approach to information retrieval.

Within the framework of hybrid search, the parameter alpha plays a crucial role in determining the balance between keyword-based and semantic searches. Regarded as a hyperparameter, alpha's optimization in RAG (Retrieval-Augmented Generation) pipelines is pivotal for refining search result accuracy.

Drawing insights from the Stack Overflow case study, we illustrate the practical utility of hybrid search in scenarios where semantic search augments the search experience. Nonetheless, we emphasize the continued significance of exact keyword matching, particularly in contexts where specific terms are prevalent.

Hybrid Search | LIm Evaluation | Rag Evaluation | Vector Database | Weaviate