How to Implement a Basic Reranking System in RAG

A practical guide to easily implement a reranker capable of putting together multiple document scoring criteria in RAG systems

By **Iván Palomares Carrascosa**, KDnuggets Technical Content Specialist on November 11, 2024 in **Language Models**

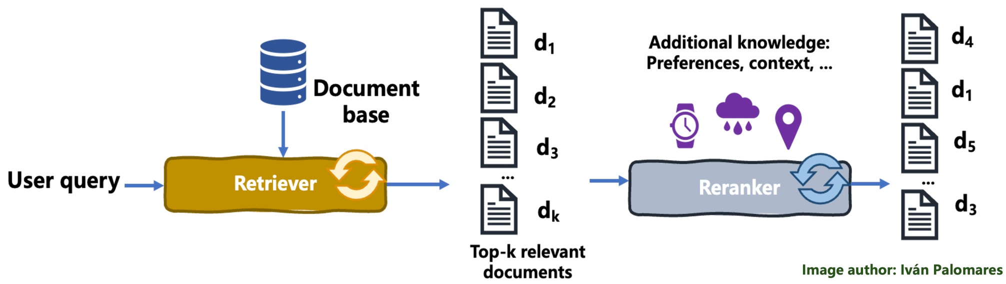
Image by Author | Ideogram

Latest Posts

- 5 LLM Prompting Techniques Every Developer Should Know
- Top 5 Freelancer Websites Better Than Fiverr and Upwork
- Creating a Useful Voice-Activated Fully Local RAG System
- 10 Little-Known Python Libraries That Will Make You Feel Like a Data Wizard
- Beginner's Guide to Subqueries in SQL
- Data Science Showdown: Which Tools Will Gain Ground in 2025

Top Posts

Reranking systems significantly enhance the relevance and accuracy of information retrieval systems, including **language models** endowed with **retrieval augmented generation (RAG)**, to prioritize the most contextually relevant and precise responses, refining the generated outputs by aligning them closer with user intent.



A basic reranking system in RAG

This tutorial walks you through building a basic reranker suitable for RAG systems in Python. The implementation shown is highly modular and focused on clearly understanding the reranking process itself.

Step-by-Step Reranking Process

We start by importing the packages that will be needed throughout the code.

```
from dataclasses import dataclass
from typing import List, Tuple
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
```

Next, we define one of the main classes for the reranking system: the `Document` class:

```
@dataclass
class Document:
    """A document with its text content, metadata, and initial relevance score'
    content: str
    embedding: np.ndarray
    initial_score: float
    metadata: dict = None
```

`@dataclass` is a decorator that simplifies the class definition by implicitly creating key methods like `__init__`, `__repr__`, and `__eq__`.

We now define the function for reranking documents as a standalone function easily integrated with a RAG system.

```
def rerank_documents(
    query_embedding: np.ndarray,
    documents: List[Document],
    semantic_weight: float = 0.5,
    initial_weight: float = 0.5
) -> List[Tuple[Document, float]]:

    # Normalizing weights
    total_weight = semantic_weight + initial_weight
    semantic_weight = semantic_weight / total_weight
    initial_weight = initial_weight / total_weight

    # Stack document embeddings into a matrix
    doc_embeddings = np.vstack([doc.embedding for doc in documents])

    # Calculate semantic similarities to user query embedding
    semantic_scores = cosine_similarity(
        query_embedding.reshape(1, -1),
        doc_embeddings
    )[0]

    # Get initial scores and normalize all scores to 0-1
    initial_scores = np.array([doc.initial_score for doc in documents])
    semantic_scores = (semantic_scores - semantic_scores.min()) / (semantic_sc
    initial_scores = (initial_scores - initial_scores.min()) / (initial_scores

    # Calculate final scores as weighted averages
    final_scores = (semantic_weight * semantic_scores) + (initial_weight * ini

    # Create sorted list of (document, score) tuples
    ranked_results = list(zip(documents, final_scores))
    ranked_results.sort(key=lambda x: x[1], reverse=True)

    return ranked_results
```

The function returns a list of document-score pairs, and takes **four inputs**:

- A vector representation (embedding) of the user query.
- A list of documents to rerank: in a RAG system, these are typically the documents obtained by the retriever component.
- The semantic weight controls the importance given to semantic similarity or document closeness to the query embedding.
- The importance weight given to the initial retrieved documents' scores.

Let's break the **function body** down **step by step**:

1. After normalizing weights to sum up to 1, the function stacks the input documents' embeddings into a matrix to ease computations.
2. The semantic similarities between documents and the user query are then calculated.
3. Access the initial document scores and normalize both initial and semantic scores. In a

full-RAG solution, the initial scores are yielded by the retriever, often using a keyword-based similarity approach like BM25.

4. Compute final scores for reranking as a weighted average of initial and semantic scores.
5. Build and return a list of document-score pairs, sorted by final scores in descending order.

Now it only remains trying our reranking function out. Let's first define some example "retrieved" documents and user query:

```
def example_reranking():
    # Simulate some "retrieved" documents with initial scores
    documents = [
        Document(
            content="The quick brown fox",
            embedding=np.array([0.1, 0.2, 0.3]),
            initial_score=0.8,
        ),
        Document(
            content="Jumps over the lazy dog",
            embedding=np.array([0.2, 0.3, 0.4]),
            initial_score=0.6,
        ),
        Document(
            content="The dog sleeps peacefully",
            embedding=np.array([0.3, 0.4, 0.5]),
            initial_score=0.9,
        ),
    ]

    # Simulate a user query embedding
    query_embedding = np.array([0.15, 0.25, 0.35])
```

Then, apply the reranking function and show the results. By specifying the two weights, you can customize the way your reranking system will work:

```
    reranked_docs = rerank_documents(
        query_embedding=query_embedding,
        documents=documents,
        semantic_weight=0.7,
        initial_weight=0.3
    )

    print("\nReranked documents:")
    for doc, score in reranked_docs:
        print(f"Score: {score:.3f} - Content: {doc.content}")

if __name__ == "__main__":
    example_reranking()
```

Output:

```

Reranked documents:
Score: 0.723 - Content: The quick brown fox
Score: 0.700 - Content: Jumps over the lazy dog
```

```
Score: 0.300 - Content: The dog sleeps peacefully
```

Now that you are familiar with reranking, the next move would be integrating your reranker with a RAG-LLM system.

Iván Palomares Carrascosa is a leader, writer, speaker, and adviser in AI, machine learning, deep learning & LLMs. He trains and guides others in harnessing AI in the real world.

More On This Topic

- How to Implement Agentic RAG Using LangChain: Part 1
- How to Implement Agentic RAG Using LangChain: Part 2
- A Simple to Implement End-to-End Project with HuggingFace
- How to Implement Named Entity Recognition with Hugging Face Transformers
- Learn how to design, measure and implement trustworthy A/B tests...
- How to Implement a Federated Learning Project with Healthcare Data

<= Previous post Next post =>