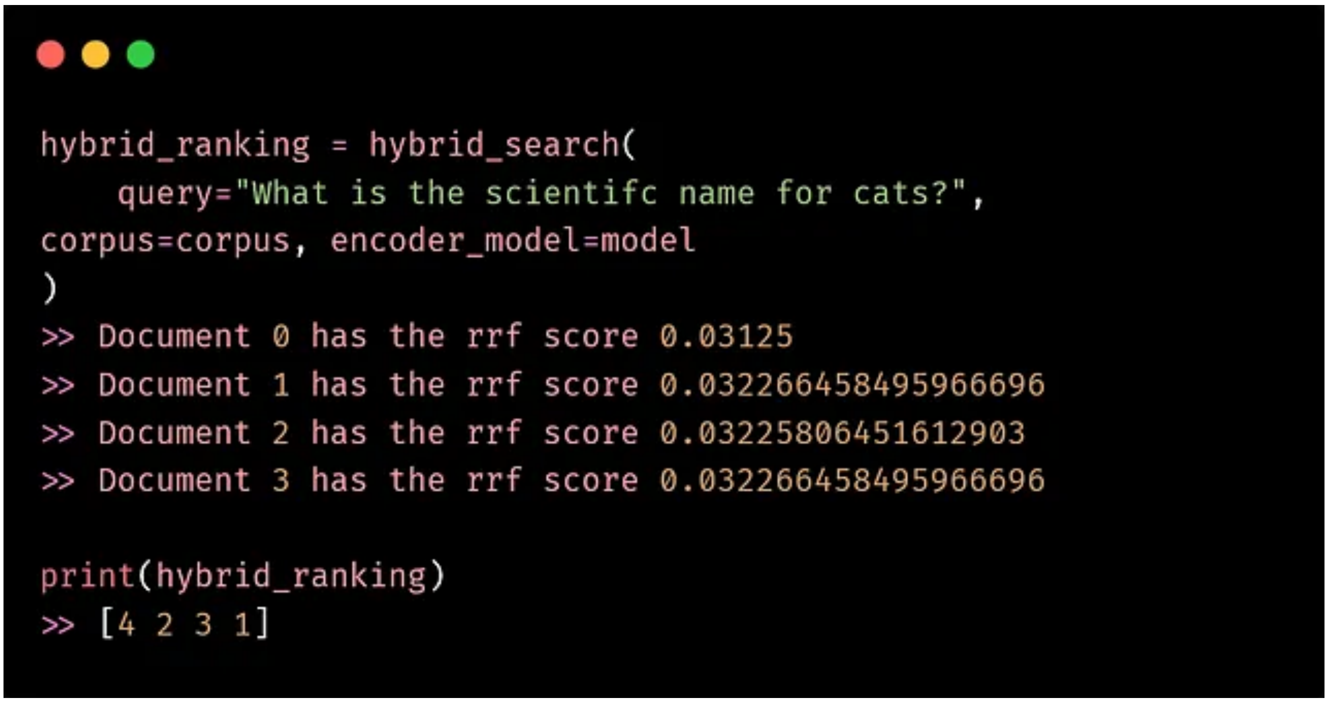
How to Use Hybrid Search for Better LLM RAG Retrieval

Building an advanced local LLM RAG pipeline by combining dense embeddings with BM25

Dr. Leon Eversberg



```
hybrid_ranking = hybrid_search(
    query="What is the scientifc name for cats?",
corpus=corpus, encoder_model=model
)
>> Document 0 has the rrf score 0.03125
>> Document 1 has the rrf score 0.03226458495966696
>> Document 2 has the rrf score 0.032258064516129O3
>> Document 3 has the rrf score 0.03226458495966696

print(hybrid_ranking)
>> [4 2 3 1]
```

Code snippet from the hybrid search we are going to implement in this article. Image by author

The basic Retrieval-Augmented Generation (RAG) pipeline uses an encoder model to search for similar documents when given a query.

This is also called **semantic search** because the encoder transforms text into a high-dimensional vector representation (called an embedding) in which semantically similar texts are close together.

Before we had Large Language Models (LLMs) to create these vector embeddings, the BM25 algorithm was a very popular search algorithm. BM25 focuses on important keywords and looks for exact matches in the available documents. This approach is called **keyword search**.

If you want to take your RAG pipeline to the next level, you might want to try **hybrid search**. Hybrid search combines the benefits of keyword search and semantic search to improve search quality.

In this article, we will cover the theory and implement all three search approaches in Python.

Table of Contents

RAG Retrieval

Hybrid search is the combination of keyword search and semantic search. We will cover both search strategies separately and then combine them later on.

Keyword Search With BM25

BM25 is the algorithm of choice for keyword search. With BM25, we get a score for our query for each document in our corpus.

BM25 is based on the **TF-IDF algorithm**, which means that the core of the formula is the product of the **term frequency (TF)** and the **inverse document frequency (IDF)**.

The TF-IDF algorithm is based on the idea that "matches on less frequent, more specific, terms are of greater value than matches on frequent terms" [1].

In other words, the TF-IDF algorithm looks for documents that contain rare keywords from our query.

The BM25 algorithm looks for documents that contain important keywords from the search query. Image by author

There are many variations of the BM25 algorithm, each aiming to improve upon the original algorithm. However, none seems to be systematically better than the others [2].

So in practice, it should be fine to pick one and stick with it.

If we look at the LangChain source code, we can see that it uses the *BM25Okapi* class from the rank_bm25 package, which is a slightly modified version of the ATIRE BM25 algorithm [3].

The formula to get a score for a document `d` and a given query `q` consisting of multiple terms `t` in the ATIRE BM25 version is as follows [2]:

The formula for the ATIRE BM25 score.

- `N` is the number of documents in the corpus
- `df_t` is the number of documents containing the term `t` (also called the **document frequency**)
- `tf_td` is the number of occurrences of the term `t` in document `d` (also called the **term frequency**)
- `L_d` is the length of our document and `L_avg` is the average document length
- There are two empirical tuning parameters: `b`, and `k_1`

Intuitively, we see that the formula sums over all terms `t`, which we can think of as words.

The left-hand factor `log(N/df_t)` in the BM25 equation is called the **inverse document frequency**. For common words such as “the”, all of our documents may contain, so the inverse document frequency will be zero (because log(1) is zero).

On the other hand, very rare words will appear in only a few documents, thus increasing the left factor. **Therefore, the inverse document frequency is a measure of how much information is contained in the term `t`.**

The right factor is influenced by the number of times the term `t` occurs in document `d`.

The document `d=["I like red cats, black cats, white cats, and brown cats"]` has a very high term frequency `tf_td` for the term `t="cats"`, which will result in a high BM25 score for a query containing the word “cats”.

Let’s use BM25 to get some intuition using the Python library `rank_bm25`.

```
pip install rank_bm25
```

First, we load the library and initialize BM25 with our tokenized corpus.

```
from rank_bm25 import BM25Okapi
```

```
corpus = [
    "The cat, commonly referred to as the domesti
    "The dog is a domesticated descendant of the 
    "Humans are the most common and widespread sp
    "The scientific name Felis catus was proposed
]
tokenized_corpus = [doc.split(" ") for doc in cor

bm25 = BM25Okapi(tokenized_corpus)
```

Next, we tokenize our query.

```
query = "The cat"
tokenized_query = query.split(" ")
```

Finally, we compute scores using the BM25 algorithm. A high score indicates a good match between the document and the query.

```
doc_scores = bm25.get_scores(tokenized_query)

print(doc_scores)
>> [0.92932018 0.21121974 0. 0.1901173]# scores f
```

Because BM25 looks for exact term matches, querying for the terms “cats”, “Cat”, or “feline” will all result in scores of `doc_scores = [0,0,0]` for our three example documents.

Semantic Search With Dense

Embeddings

When we perform semantic search via dense embeddings, we transform words into a numerical representation. The idea is that **similar words are close together** in this new mathematical representation.

Using dense embeddings to group semantically similar text. Image by author

Text embeddings are high-dimensional vectors of single words or whole sentences. They are called *dense* because each entry in the vector is a meaningful number. The opposite is called *sparse* when many vector entries are simply zero.

Before turning words into embeddings, they are first converted to **tokens**, which is a mapping from string to integer. A neural network embedding model called the **encoder** then converts the tokens into **embeddings**.

From input string to sentence embedding. Image from https://medium.com/towards-data-science/how-to-reduce-embedding-size-and-increase-rag-retrieval-speed-7f903d3cecf7

After converting all the text from our corpus of documents into embeddings, we can then perform a semantic search to see which embedded document is closest to our embedded query.

We can visualize this task by plotting the embedding dimensions and finding the closest document matches to our query.

Scatter plot of the sentence embeddings from documents and a question. Image from https://medium.com/towards-data-science/how-to-reduce-embedding-size-and-increase-rag-retrieval-speed-7f903d3cecf7

Mathematically, we find the closest match using the **cosine distance function**. For two embedding vectors `a` and `b`, we can compute the cosine similarity using the dot product as follows:

The formula for the cosine similarity.

Where the numerator is the dot product of the two embedding vectors and the denominator is the product of their magnitudes.

Geometrically, **the cosine similarity is the angle between the vectors**. The cosine similarity score ranges from -1 to +1.

A cosine similarity score of -1 means that the embeddings `a` and `b` face exactly in opposite directions, 0 means that they have an angle of 90 degrees (they are unrelated), and +1 means that they are the same. **So we look for a value close to +1 when matching a search query to documents**.

If we normalize our embeddings beforehand, the cosine similarity measure becomes equivalent to the dot product similarity measure (the denominator becomes 1).

Let's use the Python package `sentence-transformers` to perform a basic semantic search.

```
pip install sentence-transformers
```

First, we load the library and download the all-MiniLM-L6-v2 encoder model from HuggingFace. This encoder model is trained to produce 384-dimensional dense embeddings. If you have an OpenAI API key, you can also use their `text-embedding` model instead.

```
from sentence_transformers import SentenceTransfo
```

```
# 1. Load a pretrained Sentence Transformer model
model = SentenceTransformer('sentence-transformers
```

Then, we use the same corpus of documents as before.

```
# The documents to encode
corpus = [
    "The cat, commonly referred to as the domesti
    "The dog is a domesticated descendant of the 
    "Humans are the most common and widespread sp
    "The scientific name Felis catus was proposed
]

# Calculate embeddings by calling model.encode()
document_embeddings = model.encode(corpus)

# Sanity check
print(document_embeddings.shape)
>> (4, 384)
```

And we embed our query:

```
query = "The cat"
query_embedding = model.encode(query)
```

Finally, we can compute the cosine similarity scores. Instead of coding the formula ourselves, we can use the utility function `cos_sim` from `sentence_transformers`.

```
from sentence_transformers.util import cos_sim
```

```
# Compute cosine_similarity between documents and
scores = cos_sim(document_embeddings, query_embed

print(scores)
>> tensor([[0.5716],  # score for document 1
>>         [0.2904],  # score for document 2
>>         [0.0942],  # score for document 3
>>         [0.3157]]) # score for document 4
```

To see the power of semantic search with dense embeddings, I can re-run the code with the query “feline”:

```
query_embedding = model.encode("feline")

scores = cos_sim(document_embeddings, query_embed

print(scores)
>> tensor([[0.4007],
>>         [0.3837],
>>         [0.0966],
>>         [0.3804]])
```

Even though the word “feline” does not appear in the corpus of documents, the semantic search still ranks the text about cats as the highest match.

Semantic Search or Keyword Search?

Which search approach is better? That depends. Both have advantages and disadvantages. Now that we know how both work, we can see where they can be useful and where they can fail.

Keyword search with BM25 looks for exact matches of the query term. This can be very useful when we are looking for exact matches of phrases.

If I’m looking for “The Cat in the Hat”, I’m probably looking for the book/movie. And I don’t want semantically similar results that are close to hats or cats.

Another use case for keyword search is programming. If I am looking for a specific function or piece of code, I want an exact match.

Semantic search, on the other hand, looks for semantically similar content. This means that semantic search also finds documents with synonyms or different spellings, such as plurals, capitalization, etc.

Since both algorithms have their use cases, hybrid search uses both and then combines their results into one final ranking.

The disadvantage of hybrid search is that it requires more computing resources than running only one algorithm.

Hybrid Search

We can combine the results of BM25 and cosine similarity using **Reciprocal Rank Fusion (RRF)**. RRF is a simple algorithm for combining the rankings of different scoring functions [4].

First, we need to get a document ranking for each scoring algorithm. In our example, this would be:

```
corpus = [
    "The cat, commonly referred to as the domesti
    "The dog is a domesticated descendant of the 
    "Humans are the most common and widespread sp
    "The scientific name Felis catus was proposed
]
query = "The cat"

bm25_ranking = [1, 2, 4, 3] # scores = [0.9293201
cosine_ranking = [1, 3, 4, 2] # scores = [0.5716,
```

The formula for the combined RRF score for each document `d` is as follows:

The formula for the RRF score.

Where `k` is a parameter (the original paper used `k=60`) and `r(d)` are the rankings from BM25 and from cosine similarity.

Putting It All Together

Now we can implement our hybrid search by doing BM25 and cosine similarity separately and then combining the results with RRF.

First, let's define functions for RRF and a helper function to convert float scores to int rankings.

```
import numpy as np

def scores_to_ranking(scores: list[float]) -> lis
    """Convert float scores into int rankings (ra
    return np.argsort(scores)[::-1] + 1


def rrf(keyword_rank: int, semantic_rank: int) ->
    """Combine keyword rank and semantic rank int
    k = 60
    rrf_score = 1 / (k + keyword_rank) + 1 / (k +
    return rrf_score
```

Here is my simple hybrid search implementation using the concepts described above.

```
from rank_bm25 import BM25Okapi
from sentence_transformers import SentenceTransfo
from sentence_transformers.util import cos_sim

model = SentenceTransformer("sentence-transformer
def hybrid_search(
    query: str, corpus: list[str], encoder_model:
) -> list[int]:
    # bm25
    tokenized_corpus = [doc.split(" ") for doc in
    tokenized_query = query.split(" ")
    bm25 = BM25Okapi(tokenized_corpus)
    bm25_scores = bm25.get_scores(tokenized_query
    bm25_ranking = scores_to_ranking(bm25_scores)

    # embeddings
    document_embeddings = model.encode(corpus)
    query_embedding = model.encode(query)
    cos_sim_scores = cos_sim(document_embeddings,
    cos_sim_ranking = scores_to_ranking(cos_sim_s

    # combine rankings into RRF scores
    hybrid_scores = []
    for i, doc in enumerate(corpus):
        document_ranking = rrf(bm25_ranking[i], c
        print(f"Document {i} has the rrf score {d
        hybrid_scores.append(document_ranking)

    # convert RRF scores into final rankings
    hybrid_ranking = scores_to_ranking(hybrid_sco
    return hybrid_ranking
```

Now we can use `hybrid_search` with different queries.

```
hybrid_ranking = hybrid_search(
    query="What is the scientifc name for cats?",
)
print(hybrid_ranking)
```

```
>> Document 0 has the rrf score 0.03125
>> Document 1 has the rrf score 0.03226645849596
>> Document 2 has the rrf score 0.03225806451612
>> Document 3 has the rrf score 0.03226645849596
>> [4 2 3 1]
```

As a next step, we could add more knowledge to our corpus of documents. In my article How to Use Re-Ranking for Better LLM RAG Retrieval, I integrated Wikipedia into the knowledge corpus.

Adding a re-ranker on top of the hybrid search will further improve the overall RAG pipeline.

How to Use Re-Ranking for Better LLM RAG Retrieval

Building an advanced local LLM RAG pipeline with two-step retrieval using open-source bi-encoders and cross-encoders

Conclusion

Hybrid search combines semantic search and keyword search to produce a better overall search result.

To perform keyword search, we implemented the BM25 algorithm, which looks for important keyword matches.

For semantic search, we used cosine similarity with a pre-trained encoder model that produces dense embeddings.

While hybrid search can improve RAG retrieval, it also requires more computational resources than running only one search algorithm.

Hybrid search is an interesting building block for improving your RAG pipeline. Give it a try!

References

[1] K. Spärck Jones, A statistical interpretation of term specificity and its application in retrieval (1972), Journal of Documentation Vol. 28 Nr. 1

[2] A. Trotman, A. Puurula, and B. Burgess, Improvements to BM25 and Language Models Examined (2014), ADCS '14: Proceedings of the 19th Australasian Document Computing Symposium

[3] A. Trotman, X.-F. Jia, and M. Crane, Towards an Efficient and Effective Search Engine (2012), Proceedings of the SIGIR 2012 Workshop on
Open Source Information Retrieval

[4] G. V. Cormack, C. L. A. Clarke, and S. Büttcher, Reciprocal Rank Fusion outperforms Condorcet and individual Rank Learning Methods (2009), Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval