4.2

HOW TO SELECT AN EMBEDDING MODEL

In the previous chapter, we examined various chunking techniques to break documents down into smaller, manageable pieces called "chunks." When the user queries, the retriever can look through these smaller chunks to find relevant information quickly and accurately. Now, these chunks can be transformed into an "embedding."

Embeddings refer to dense, continuous vectors representing text in a high-dimensional space. These vectors serve as coordinates in a semantic space, capturing the relationships and meanings between words. You can have embeddings by mapping words, phrases, or even entire documents to points in this space.

Do you remember traditional text representations like one-hot encoding? These are sparse and high-dimensional. Embeddings reduce this dimensionality while preserving the semantic relationships between words. Look at Fig 4.2.1 to understand the difference between the two.

Fig 4.2.1: Difference between traditional text representations vs. embeddings

You'll see how each word in the index is assigned a unique vocabulary and is represented as a vector where all elements are zero, except for the position corresponding to the word's index, which is set to one. So, the size of the vector would be equal to the vocabulary size. This makes it excessively large in most cases.

In contrast, embeddings usually have a dimension between 50 and 300. Add to this the fact that words that are semantically similar are closer in the embedding space, which makes them much more efficient in terms of storage and computation.

THE IMPORTANCE OF EMBEDDINGS

Embeddings form the foundation for achieving precise and contextually relevant LLM outputs across different tasks. Let's explore the diverse applications where embeddings play an indispensable role.

Question Answering

Embeddings play a crucial role in enhancing the performance of Question Answering (QA) systems within RAG applications. By encoding questions and potential answers into high-dimensional vectors, embeddings allow the efficient retrieval of relevant information. The semantic understanding captured by embeddings facilitates accurate matching between queries and context, enabling the QA system to provide more precise and contextually relevant answers.

Conversational Search

Conversations involve dynamic and evolving contexts, and embeddings help represent the nuances and relationships within the dialogue. By encoding user queries and system responses, embeddings enable the RAG system to retrieve relevant information and generate context-aware responses.

InContext Learning (ICL)

The model's effectiveness in InContext Learning is highly dependent on the choice of few shot demonstrations. Traditionally, a fixed set of demonstrations was employed, limiting the adaptability of the model. Rather than relying on a predetermined set of examples, this novel approach involves retrieving demonstrations relevant to the context of each input query.

The implementation of this demonstration retrieval is relatively straightforward, utilizing existing databases and retrieval systems. This dynamic approach enhances the learning process's efficiency and scalability and addresses biases inherent in manual example selection.

Tool Fetching

Tool fetching involves retrieving relevant tools or resources based on user queries or needs. Embeddings encode the semantics of the user's request and the available tools, enabling the RAG system to perform effective retrieval and present contextually relevant tools. The use of embeddings enhances the accuracy of tool recommendations, contributing to a more efficient and user-friendly experience.

IMPACT OF EMBEDDINGS ON RAG PERFORMANCE

Which encoder you select to generate embeddings is a critical decision that will hugely impact the overall success of the RAG system. Low-quality embeddings lead to poor retrieval. Let's review some selection criteria to consider before making your decision.

Vector Dimension and Performance Evaluation

When selecting an embedding model, consider the vector dimension, average retrieval performance, and model size. The Massive Text Embedding Benchmark (MTEB) provides insights into popular embedding models from OpenAI, Cohere, and Voyager, among others. However, custom evaluation on your dataset is essential for accurate performance assessment.

Private vs. Public Embedding Model

Although the embedding model provides ease of use, it entails certain trade-offs. The private embedding API, in particular, offers high availability without the need for intricate model hosting engineering. However, this convenience is counterbalanced by scaling limitations. It's crucial to verify the rate limits and explore options for increasing them. A notable advantage is that model improvements come at no extra cost.

Companies such as OpenAI, Cohere, and Voyage consistently release enhanced embedding models. Simply run your benchmark for the new model and implement a minor change in the API, making the process exceptionally convenient.

Cost Considerations

Embedding Price Comprasion

Provider	Model	Input Price Per 1k Token	Input Price Per 1m Token	Compared To Open AI Ada	Compared To Cohere
Amazon Bedrock	Titan Embeddigs	$0.00010	$0.10000	0.00%	0.00%
(Azure) OpenAI	Ada Embeddings Embedding 3 small Embedding 3 Large	$0.00010 $0.00002 $0.00013	$0.10000 $0.02000 $0.13000	0.00% -80.00% 30.00%	0.00% -80.00% 30.00%
Cohere	Embed	$0.00010	$0.10000	0.00%	0.00%
Google Vertex AI (1 token = 4 chars)	Text Embeddings	$0.00040	$0.40000	300.00%	300.00%
Together	BGE-Base	$0.00003	$0.02800	-72.00%	-72.00%
Anyscale	theniper-gte-large	$0.00005	$0.05000	-50.00%	-50.00%
MosaicML	Instructor-Large Instructor-XL	$0.00010 $0.00020	$0.10000 $0.20000	0.00% 100.00%	0.00% 100.00%

Fig 4.2.2: Comparative analysis of the pricing for embedding services from various providers

Querying Cost

Ensure high availability of the embedding API service, considering factors like model size and latency needs. OpenAI and similar providers offer reliable APIs, while open-source models may require additional engineering efforts.

Indexing Cost

The cost of indexing documents is influenced by the chosen encoder service. Separate storage of embeddings is advisable for flexibility in service resets or reindexing.

Storage Cost

Storage cost scales linearly with dimension, and the choice of embeddings, such as OpenAI's in 1526 dimensions, impacts the overall cost. To estimate storage cost, calculate the average units per document.

Search Latency

The latency of semantic search grows with the dimension of embeddings. To minimize latency, you'll need to opt for low-dimensional embeddings.

Look at Fig 4.2.2 to get a clear idea of the pricing for embedding services from different providers. You'll notice that OpenAI's Embedding 3 Small model is **80% cheaper** than both OpenAI Ada and Cohere Embed models for 1M tokens.

Language Support

To support non-English languages, you'll need to choose a multilingual encoder or use a translation system alongside an English encoder.

Privacy Concerns

Stringent data privacy requirements, especially in sensitive domains like finance and healthcare, may influence your choice of embedding services. Evaluate privacy considerations before selecting a provider.

Granularity of Text

Various levels of granularity, including word-level, sentence-level, and document-level representations, influence the depth of semantic information embedded. Segmenting large text into smaller chunks can optimize relevance and minimize noise in the embedding process. However, due to the constrained vector size available for storing textual information, embeddings become noisy with longer text.

Types of Embeddings

Model Type	Example Model	Compute	Retrieval Performance	Granularity /Input
Sparse	SPLADE	**Low**	Good	Sentence, paragraph
Dense	Sentence transformers	**Medium**	Good	Sentence, paragraph
Multivector	COLBERT	**High**	Best	Sentence, paragraph
Long context dense	text-embedding-3-small	**Medium**	Good	Paragraphs, chapters
Variable dimension	text-embedding-3-small	**Medium**	Good	Sentence, paragraph
Code (dense)	text-embedding-3-small	**Medium**	Good	Functions, classes

Fig 4.2.3: Types of embedding models

Different types of embeddings are designed to address unique challenges and requirements in different domains. From dense embeddings capturing overall semantic meaning to sparse embeddings emphasizing specific information and from multi-vector embeddings with late interaction to innovative variable dimension embeddings, knowing your use case will help decide which embedding type to employ. Additionally, we'll explore how recent advancements, such as code embeddings, are transforming how developers interact with codebases.

Dense Embeddings

Dense embeddings are continuous, real-valued vectors representing information in a high-dimensional space. In the context of RAG applications, dense embeddings, such as those generated by models like OpenAI's Ada or sentence transformers, contain non-zero values for every element. These embeddings focus on capturing the overall semantic meaning of words or phrases, making them suitable for tasks like dense retrieval, which involve mapping text into a single embedding. This helps effectively match and rank documents based on content similarity.

Dense retrieval models utilize approximate nearest neighbor search to efficiently retrieve relevant documents for various applications. These are the embeddings usually referred to for semantic search and vector databases.

Let's say you're in Italy, and you search for "best Italian restaurants" in a semantic search engine. The search engine converts your query into a dense embedding. It then compares this vector with the embeddings of various restaurant reviews stored in its database. The engine retrieves and ranks the reviews based on how similar their embeddings are to your query and provides you with the most relevant results.

You'll need to remember that dense embeddings often have a high dimensionality in order 300, 768, or even 1024! This typically results in significant computational and memory requirements. Consequently, indexing large datasets of dense embeddings for efficient retrieval can also be a challenge.

Sparse Embeddings

Sparse embeddings, on the other hand, are representations where most values are zero, emphasizing only relevant information. In RAG applications, sparse vectors are essential for scenarios with rare keywords or specialized terms. Unlike dense vectors that contain non-zero values for every element, sparse vectors focus on relative word weights per document, resulting in a more efficient and interpretable system.

Sparse vectors like SPLADE are especially beneficial in domains with specific terminologies, such as the medical field, where many rare terms may not be present in the general vocabulary. Using sparse embeddings helps overcome the limitations of Bag-of-Words (BOW) models, addressing the vocabulary mismatch problem.

Let's extend the previous example of a search system for medical research papers. In this case, sparse embeddings can help ensure that rare medical terms are accurately represented and efficiently processed. When a user searches for a term like "hypercholesterolemia," the system can quickly retrieve relevant documents because the sparse embedding highlights this specific term, even if it appears rarely in the overall dataset.

But you'll also need to pay attention to the fact that sparse embeddings may not capture the full context of words as effectively as dense embeddings. This leads to a less accurate understanding of semantic relationships.

Multi-Vector Embeddings

Multi-vector embedding models like ColBERT feature late interaction, where the interaction between query and document representations occurs late in the process after both have been independently encoded. This approach contrasts with early interaction models, where query and document embeddings interact at earlier stages, potentially leading to increased computational complexity.

The late interaction design allows for the pre-computation of document representations, contributing to faster retrieval times and reduced computational demands, making it more suitable for processing large document collections. ColBERT's multi-vector embedding strategy involves encoding queries and documents independently, followed by a lightweight interaction step, ensuring efficiency and scalability.

Let's take an example to understand this. Take, for instance, an academic search engine that indexes millions of research papers. Using a model like ColBERT:

Pre-encoding docs
All research papers are encoded into dense vectors and stored in a database.

Query processing

- When you search for "deep learning optimization," the query is encoded into vectors.
- The query vectors interact with the pre-stored document vectors in a lightweight and efficient manner.

Retrieval
Relevant research papers are quickly retrieved based on the interaction of query and document embeddings.

Long Context Embeddings

Long documents have always posed a particular challenge for embedding models. The limitation on maximum sequence lengths, often rooted in architectures like BERT, leads to practitioners segmenting documents into smaller chunks. Unfortunately, this segmentation can result in fragmented semantic meanings and misrepresentation of entire paragraphs. It also increases memory usage, computational demands during vector searches, and latencies.

Models like BGE-M3 allow the encoder to encode sequences as long as 8,192 tokens, which helps reduce vector storage and latency without sacrificing retrieval performance.

Variable Dimension Embeddings

Variable dimension embeddings are a unique concept built on Matryoshka Representation Learning (MRL). MRL learns lower-dimensional embeddings nested into the original embedding, akin to a series of Matryoshka Dolls. Each representation sits inside a larger one, from the smallest to the largest "doll." This hierarchy of nested subspaces is learned by MRL, and it efficiently packs information at logarithmic granularities. (See Fig 4.2.4.)

Fig 4.2.4: Matryoshka Representation Learning

The hypothesis is that MRL enforces a vector subspace structure, where each learned representation vector lies in a lower-dimensional vector space that is a subspace of a larger vector space. Models like OpenAI's text-embedding-3-small and Nomic's Embed v1.5 are trained using MRL and deliver great performance at even small embedding dimensions = 256. Look at different comparisons in Fig 4.2.5.

	Ada V2	Text-Embedding-3-Small		Text-Embedding-3-Large		
Embedding size	1536	512	1536	256	1024	3072
Average MTEB score	61.0	61.6	62.3	62.0	64.1	64.6

Fig 4.2.5: Comparison of different embedding models based on embedding size and average MTEB score

This is how it would typically work. For example, a search engine uses MRL to create embeddings for web pages. At the highest level, the embeddings capture broad topics (e.g., "Technology," "Health"). Nested within these are finer details (e.g., "AI," "Blockchain," etc. for Technology). So, when you query "latest advances in AI," the search engine will first identify the broad topic using higher-level embeddings, then dig deeper into the finer details using the nested embeddings structure.

Code Embeddings

Fig 4.2.6: GitHub CoPilot Chat

Code embeddings are a recent development integrating AI-powered capabilities into Integrated Development Environments (IDEs), fundamentally transforming how developers interact with codebases. Unlike traditional text search, code embedding offers semantic understanding, allowing it to interpret the intent behind queries related to code snippets or functionalities. Code embedding models are built by training models on paired text data, treating the top-level docstring in a function along with its implementation as a (text, code) pair. (See Fig 4.2.6)

Code embedding like OpenAI's text-embedding-3-small and jina-embeddings-v2-base-code makes it easy to search through code, build automated documentation, and create chat-based code assistance.

HOW TO MEASURE EMBEDDING PERFORMANCE

Retrieval metrics, led by the widely recognized MTEB benchmark, help us measure the performance of embeddings. Each dataset in the retrieval evaluation comprises a corpus, queries, and a mapping associating each query with relevant documents from the corpus. The objective is to identify these pertinent documents. The provided model is employed to embed all queries and corpus documents, and then similarity scores are calculated using cosine similarity. Subsequently, the corpus documents are ranked for each query based on these scores and metrics, such as NDCG@10.

Although MTEB provides insights into some of the best embedding models, it fails to determine the optimal choice for specific domains or tasks. As a result, it's vital to conduct an evaluation on your own dataset. Often, we possess raw text and aim to assess the RAG performance on user queries. In such scenarios, metrics such as chunk attribution (Refer to the end of Chapter 4.1) can be quite useful.

CHOOSING THE RIGHT EMBEDDING MODEL

Let's explore how we can utilize chunk attribution to choose the optimal embedding model for our RAG system. By attributing retrieved chunks to generated outputs, we can identify which embedding model is most suitable for our use case.

Let's use Galileo's GenAI Studio to test a use case using NVIDIA 10-K annual financial reports with a simple RAG system for demonstration.

Data preparation

First, we retrieve the 10-K reports for Nvidia from the past four years. We perform straightforward parsing using the PyPDF library, yielding large chunks without applying any advanced chunking we talked about earlier. This process results in approximately 700 sizable text chunks. (See Fig 4.2.7).

```python
import glob
from langchain_community.document_loaders import PyPDFLoader

documents = []
for file_path in glob.glob("../data/nvidia_10k_*.pdf"):
    print(file_path)
    loader = PyPDFLoader(file_path)
    documents.extend(loader.load_and_split())

len(documents)

Output:
../data/nvidia_10k_2023.pdf
../data/nvidia_10k_2022.pdf
../data/nvidia_10k_2021.pdf
../data/nvidia_10k_2024.pdf

701
```

Fig 4.2.7: Code snippet for parsing through the 10-k reports to get text chunks

To test our RAG system, we need a set of questions. Leveraging GPT-turbo with a zero-shot instruction prompt, we generate a question for each text chunk. (See Fig 4.2.8).

```python
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage


def get_questions(text):
    questions = chat_model.invoke(
    [
        HumanMessage(
            content=f"""Your job is to generate only 1 short question from the given text such that it
can be answered using the provided text. Use the exact info in the questions as mentioned in the text.
Return questions starting with - instead of numbers.

Text: {text}
Questions: """
        )
    ]
)
    questions = questions.content.replace("- ", "").split("\n")
    questions = list(filter(None, questions))
    return questions

text = documents[1].page_content
print(text)
chat_model = ChatOpenAI(model="gpt-3.5-turbo-0125", temperature=1.0)
get_questions(text)

Output:
The aggregate market value of the voting stock held by non-affiliates of the registrant as of July 29,
2022 was approximately $434.37 billion (based on the closing sales price of the registrant's common stock
as reported by the Nasdaq Global Select Market on July 29, 2022). This calculation excludes 98 million
shares held by directors and executive officers of the registrant. This calculation does not exclude
shares held by such organizations whose ownership exceeds 5% of the registrant's outstanding common stock
that have
represented to the registrant that they are registered investment advisers or investment companies
registered under section 8 of the Investment Company Act of 1940.

['What was the aggregate market value of the voting stock held by non-affiliates of the registrant as of
July 29, 2022?']
```

Fig 4.2.8: Code snippet representing the generation of a question for each chunk created

We randomly select 100 chunks from the pool of 700 and create questions accordingly to have a few questions from every annual report. (See Fig 4.2.9)

```python
import pandas as pd
from tqdm import tqdm
tqdm.pandas()

df = pd.DataFrame({"text": [doc.page_content for doc in documents]})
df = df.sample(n=100, random_state=0)
df["questions"] = df.text.progress_apply(get_questions)
```

Fig 4.2.9: Code snippet for randomly selecting 100 chunks and creating questions

QA Chain

With the data prepared, we define our RAG chain using Langchain, incorporating Pinecone serverless vector index and GPT as the generator. (See Fig 4.2.10).

```python
import os

from langchain_openai import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.schema.runnable import RunnablePassthrough
from langchain.schema import StrOutputParser
from langchain_community.vectorstores import Pinecone as langchain_pinecone
from pinecone import Pinecone

def get_qa_chain(embeddings, index_name, k, llm_model_name, temperature):
    # setup retriever
    pc = Pinecone(api_key=os.getenv("PINECONE_API_KEY"))
    index = pc.Index(index_name)
    vectorstore = langchain_pinecone(index, embeddings.embed_query, "text")
    retriever = vectorstore.as_retriever(search_kwargs={"k": k})  # https://github.com/langchain-
ai/langchain/blob/master/libs/core/langchain_core/vectorstores.py#L553

    # setup prompt
    rag_prompt = ChatPromptTemplate.from_messages(
        [
            (
                "system",
                "Answer the question based only on the provided context."
            ),
            ("human", "Context: '{context}' \n\n Question: '{question}'"),
        ]
    )

    # setup llm
    llm = ChatOpenAI(model_name=llm_model_name, temperature=temperature,
tiktoken_model_name="cl100k_base")

    # helper function to format docs
    def format_docs(docs):
        return "\n\n".join([d.page_content for d in docs])

    # setup chain
    rag_chain = (
        {"context": retriever | format_docs, "question": RunnablePassthrough()}
        | rag_prompt
        | llm
        | StrOutputParser()
    )

    return rag_chain
```

Fig 4.2.10: Code snippet to define RAG chain using Langchain, incorporating Pinecone serverless vector index and GPT as the generator

RAG Evaluation Metrics

Following this, we outline the metrics for Galileo to calculate for every run. This will guide us in making the right tradeoffs later.

RAG metrics

Chunk Attribution

A chunk-level boolean metric that measures whether a 'chunk' was used to compose the response.

Chunk Utilization

A chunk-level float metric that measures how much of the chunk text that was used to compose the response.

Completeness

A response-level metric measuring how much of the context provided was used to generate a response

---

Context Adherence

A response-level metric that measures whether the output of the LLM adheres to (or is grounded in) the provided context.

Safety metrics

Private Identifiable Information (PII)

Identify instances of PII within a model's responses, specifically flagging credit card numbers, phone numbers, social security numbers, street addresses, and email addresses.

Toxicity

Flags whether a response contains hateful or toxic information. Output is a binary classification of whether a response is toxic or not.

Tone

Classifies the tone of the response into 9 different emotion categories: neutral, joy, love, fear, surprise, sadness, anger, annoyance, and confusion.

You'll read more about monitoring metrics in Chapter 6.

System metrics

Track the latency of LLM calls. (See Fig 4.2.11).

```
from typing import Optional

import promptquality as pq
from promptquality import Scorers

all_metrics = [
    Scorers.latency,
    Scorers.pii,
    Scorers.toxicity,
    Scorers.tone,
    #rag metrics below
    Scorers.context_adherence,
    Scorers.completeness_gpt,
    Scorers.chunk_attribution_utilization_gpt,
    # Uncertainty, BLEU and ROUGE are automatically included
]

#Custom scorer for response length
def executor(row) -> Optional[float]:
    if row.response:
        return len(row.response)
    else:
        return 0

def aggregator(scores, indices) -> dict:
    return {'Response Length': sum(scores)/len(scores)}

length_scorer = pq.CustomScorer(name='Response Length', executor=executor, aggregator=aggregator)
all_metrics.append(length_scorer)
```

Fig 4.2.11: Code snippet for tracking the latency of LLM calls

Workflow

Finally, we create a function with various sweep parameters, allowing us to experiment with different embedding models to test our use case and identify the optimal one.

Steps in the function

- Load the embedding model
- Delete if a vector index with the same name exists
- Create a new vector index
- Vectorize chunks and add to the index
- Load the chain
- Define the tags
- Prepare Galileo callback with metrics and tags
- Run the chain with questions to generate the answer
- Call `pq.finish()` to sync data to the Galileo console

```python
from langchain_openai import OpenAIEmbeddings
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.vectorstores import Pinecone as langchain_pinecone
from pinecone import Pinecone, ServerlessSpec
import promptquality as pq
from tqdm import tqdm

from metrics import all_metrics
from qa_chain import get_qa_chain


def rag_chain_executor(emb_model_name: str, dimensions: int, llm_model_name: str, k: int) -> None:
    # initialise embedding model
    if "text-embedding-3" in emb_model_name:
        embeddings = OpenAIEmbeddings(model=emb_model_name, dimensions=dimensions)
    else:
        embeddings = HuggingFaceEmbeddings(model_name=emb_model_name, encode_kwargs =
{'normalize_embeddings': True})

    index_name = f"{emb_model_name}-{dimensions}".lower()

    # First, check if our index already exists and delete stale index
    if index_name in [index_info['name'] for index_info in pc.list_indexes()]:
        pc.delete_index(index_name)

    # create a new index
    pc.create_index(name=index_name, metric="cosine", dimension=dimensions,
                    spec=ServerlessSpec(
                    cloud="aws",
                    region="us-west-2"
                ) )
    time.sleep(10)

    # index the documents
    _ = langchain_pinecone.from_documents(documents, embeddings, index_name=index_name)
    time.sleep(10)

    # load qa chain
    qa = get_qa_chain(embeddings, index_name, k, llm_model_name, temperature)

    # tags to be kept in galileo run
    run_name = f"{index_name}"
    index_name_tag = pq.RunTag(key="Index config", value=index_name, tag_type=pq.TagType.RAG)
    emb_model_name_tag = pq.RunTag(key="Emb", value=emb_model_name, tag_type=pq.TagType.RAG)
    llm_model_name_tag = pq.RunTag(key="LLM", value=llm_model_name, tag_type=pq.TagType.RAG)
    dimension_tag = pq.RunTag(key="Dimension", value=str(dimensions), tag_type=pq.TagType.RAG)
    topk_tag = pq.RunTag(key="Top k", value=str(k), tag_type=pq.TagType.RAG)

    evaluate_handler = pq.GalileoPromptCallback(project_name=project_name, run_name=run_name,
scorers=all_metrics, run_tags=[emb_model_name_tag, llm_model_name_tag, index_name_tag, dimension_tag,
topk_tag])

    # run chain with questions to generate the answers
    print("Ready to ask!")
    for i, q in enumerate(tqdm(questions)):
        print(f"Question {i}: ", q)
        print(qa.invoke(q, config=dict(callbacks=[evaluate_handler])))
        print("\n\n")

    evaluate_handler.finish()
```

Fig 4.2.12: Code snippet for the entire workflow

Now, let's log in to our console with one simple line! (See Fig 4.2.13).

```python
pq.login("console.demo.rungalileo.io")
```

Fig 4.2.13: Logging into the console

Sweep

We now utilize the Sweep feature to execute all configurations. (See Fig 4.2.14).

With a Chain Sweep, you can perform bulk execution of multiple chains or workflows, iterating over various versions or parameters of your system.

We have to wrap your workflow in a function that should take any experimental parameters ***(e.g., chunk size, embedding model, top_k)*** as arguments.

The previously defined function, rag_chain_executor, provides us with a wrapped workflow ready for utilization. We experiment with three models of similar dimensions to ensure comparable expressivity power. One of these models is ***all-MiniLM-L6-v2,*** a well-known embedding model with 384 dimensions. Additionally, we utilize the recently released OpenAI embedding APIs, namely ***text-embedding-3-small*** and ***text-embedding-3-large,*** which enable us to obtain text embeddings with varying dimensions. Consequently, we choose 384 dimensions for both of these models.

```
pq.sweep(
    rag_chain_executor,
    {
        "emb_model_name": ["all-MiniLM-L6-v2", "text-embedding-3-small", "text-embedding-3-large"],
        "dimensions": [384],
        "llm_model_name": ["gpt-3.5-turbo-0125"],
        "k": [3]
    },
)
```

Fig 4.2.14: Code snippet for using the Sweep feature to execute all configurations

Here are the results! Switching from the ***all-MiniLM-L6-v2*** encoder to the ***text-embedding-3-small*** encoder yields a 7% increase in attribution. This suggests that the ***text-embedding-3-small*** encoders enable us to retrieve more valuable chunks. The performance of both small and large is nearly identical, indicating that proceeding with the small would help save money while maintaining performance. See Fig 4.2.15.

Run Name	Average Attribution	Average Chunk Utilization	Average Completeness	Average Context Adherence
text-embedding-3-large-	0.427	0.21	0.938	0.827
text-embedding-3-small-...	0.437	0.161	0.921	0.827
all-minilm-16-v2-384	0.363	0.21	0.915	0.863

Fig 4.2.15: Comparison between different embedding models based on metrics like average attribution, chunk utilization, etc.

Let's navigate to the run view and effortlessly locate samples with an attribution score of 0, indicating that no useful chunk was retrieved. These represent instances where retrieval has failed. (See Fig 4.2.16 and Fig 4.2.17)

Fig 4.2.16: Displaying metrics relevant to an embedding model

			Custom Metrics	RAG Quality			
Node Type	Node Input	Node Output	Response Length	Attribution	Utilization	Completeness	Context Adherence
RunnableSequence	{"input": "What was the cu...	{"output": "The cumulative...	270	1 of 3	high	low	high
RunnableSequence	{"input": "How could exten...	{"output": "Granting exten...	405	1 of 3	low	high	high
RunnableSequence	{"input": "How does the co...	{"output": "The company ...	481	2 of 3	medium	high	high
RunnableSequence	{"input": "What licensing r...	{"output": "The licensing r...	357	1 of 3	low	high	high
RunnableSequence	{"input": "What was the to...	{"output": "Based on the p...	186	0 of 3	—	medium	high
RunnableSequence	{"input": "What factors co...	{"output": "Factors that co...	479	2 of 3	low	high	high
RunnableSequence	{"input": "How may excess...	{"output": "Excessive or s...	452	1 of 3	low	high	high
RunnableSequence	{"input": "How can a Partic...	{"output": "A Participant in...	258	1 of 3	high	high	high

Fig 4.2.17: Filtering local samples with a threshold

Later, we can probe deeper and check where the retrieval is failing. In this particular case, the chunks mention income tax, but none of them talk about income tax of the year mentioned in the question. See Fig 4.2.18.

Fig 4.2.18: Deep root analysis using Galileo GenAI Studio

This workflow enables us to rapidly conduct embedding evaluations for RAG and pinpoint the one best suited for production while simultaneously analyzing retrieval failures. And that's how you choose the right embedding model for you.

In this chapter, we looked at the importance of high-quality embeddings for effective RAG systems, considering vector dimension, performance, and model size. We went through different types of embeddings with their examples and some shortcomings for each type. We also looked at the workflow for evaluating different embedding models using chunk attribution and other metrics to identify the best model for production use.

In the next chapter, we'll examine how these embeddings integrate with vector databases to enhance information retrieval and effectively manage large datasets.

EXERCISE 4.2.1

Before you flip to the next chapter, here's a fun exercise to complete.

Try to build a rudimentary semantic search engine that uses embeddings to find the most relevant documents based on user queries. This exercise will help you understand how embedding models transform text data into vectors and how these vectors are used for similarity searches in a vector database. The system should:

- Convert documents and queries into embeddings.
- Use similarity search to find the most similar document embeddings.

Note

You can use a pre-trained model from Hugging Face to generate the embeddings.