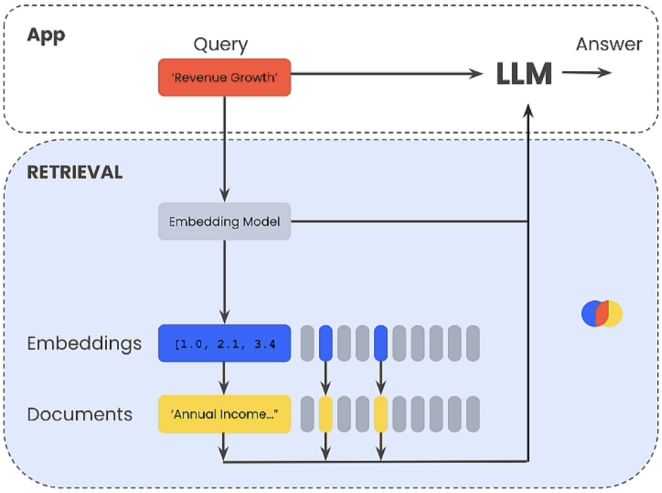
Advanced Retrieval

RAG pipeline:



take the query → run the query through the embedding model used to embed the docs, which generates an embedding → embed the query and then the retrieval system finds the most relevant docs acc to the embedding of that query by finding nearest neighbour embeddings of those docs → return both the query and relevant docs to the LLM → LLM synthesizes info from the retrieved docs to generate an answer

RecursiveCharacterTextSplitter (LangChain)

```
character_splitter = RecursiveCharacterTextSplitter(
    separators=["\n\n", "\n", ". ", " ", ""],
    chunk_size=1000,
    chunk_overlap=0
)
```

```
character_split_texts = character_splitter.split_text('\n\n'.jo

print(word_wrap(character_split_texts[10]))
print(f"\nTotal chunks: {len(character_split_texts)}")
```

so basically → first, separator will be \n\n → if the chunk size exceeds 1k, the next separator will be used on the text and so on

TokenSplitter (chunk acc to the token count, 256 is the context window length of the SentenceTransformersTokenTextSplitter)

```
token_splitter = SentenceTransformersTokenTextSplitter(chunk_ov

token_split_texts = []
for text in character_split_texts:
    token_split_texts += token_splitter.split_text(text)

print(word_wrap(token_split_texts[10]))
print(f"\nTotal chunks: {len(token_split_texts)}")
```

ChromaDB → Using the SentenceTransformmerEmbeddingFunction (basically extension of the BERT architecture → SBERT)

→ embedding of the token_split_texts is made from this

LLM to answer query → create a system prompt (add query and context)

```
def rag(query, retrieved_documents, model="gpt-3.5-turbo"):
    information = "\n\n".join(retrieved_documents)

    messages = [
        {
            "role": "system",
            "content": "You are a helpful expert financial resea
```

```
            "You will be shown the user's question, and the rele
        },
        {"role": "user", "content": f"Question: {query}. \n Info
    ]

    response = openai_client.chat.completions.create(
        model=model,
        messages=messages,
    )
    content = response.choices[0].message.content
    return content
```

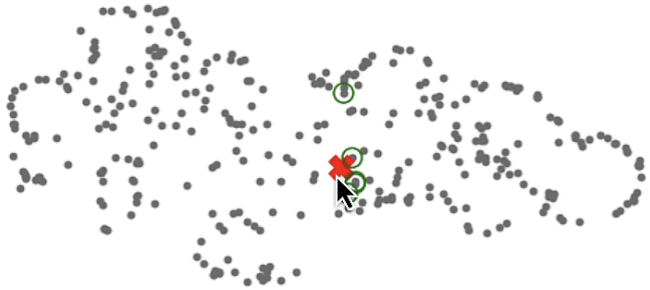
Pitfalls of Retrieval with Vectors

misc:

visualising embeddings → UMAP (projecting the vectors onto 2D plane)

UMAP (Uniform Manifold Approximation) → high dimensional data to 2D 3D (similar to PCA or t-SNE but it tries to preserve the data as much as it can (distance between vectors))

Relevancy and Distraction



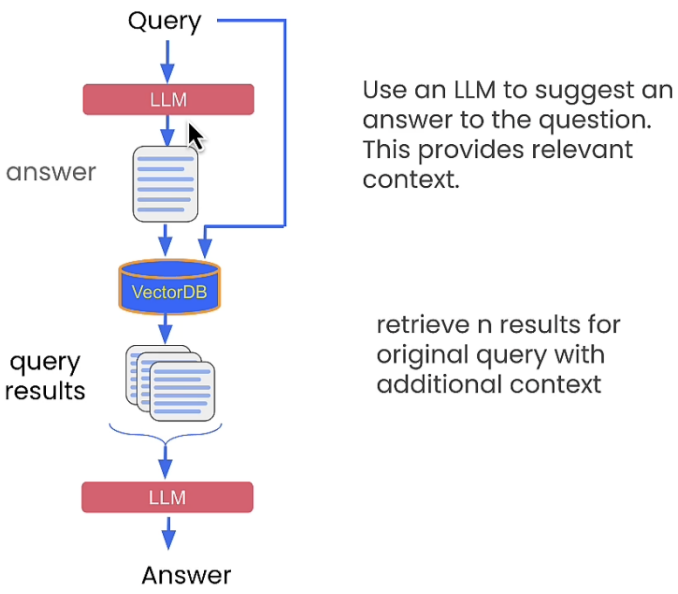
An image of the embeddings (2d and tries its best to represent distance) created→ red cross marks the query embedding, green marks the retrieved embeddings → clearly, one of the retrieved embedding is pretty far from the query embedding

There are **distractors** in the results (not relevant to the query, will distract the LLM when this will be passed to the LLM with the other relevant info)

Irrelevant queries (not in the dataset) → still gives the nearest neighbours → completely made of distractors

---

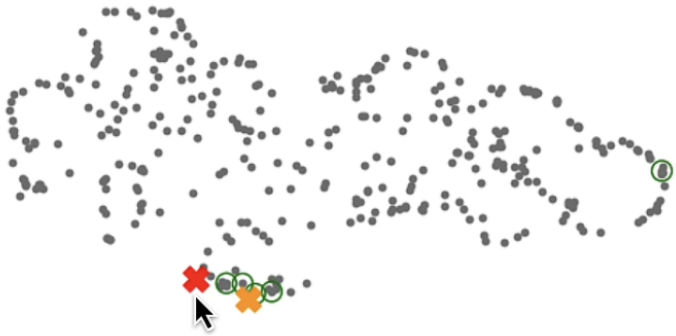
Query Expansion



Expansion with generated answers

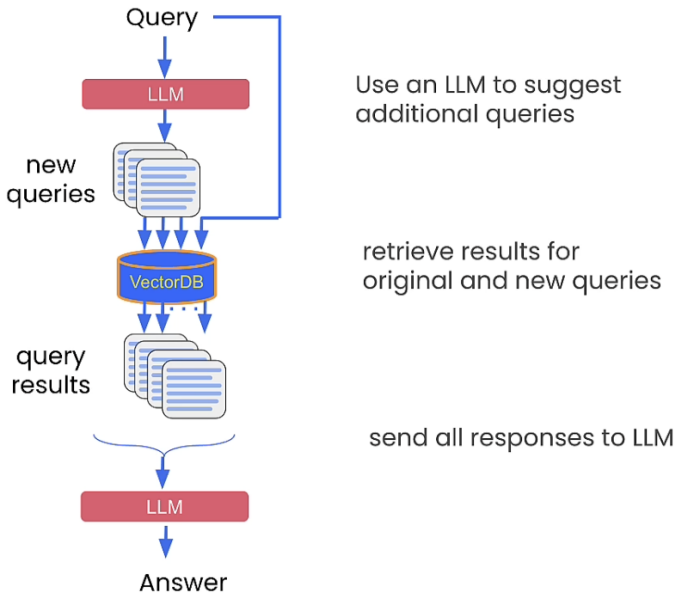
```
def augment_query_generated(query, model="gpt-3.5-turbo"):
messages = [
{
"role": "system",
"content": "You are a helpful expert financial research assistar
},
{"role": "user", "content": query}
]

response = openai_client.chat.completions.create(
    model=model,
    messages=messages,
)
content = response.choices[0].message.content
return content
```

gives a hypothetical answer (we make the LLM hallucinate) → use this as the query (query + answer) for the vector database



red cross → original query
orange cross → new query
green circles → retrieved vectors

Expansion with multiple queries

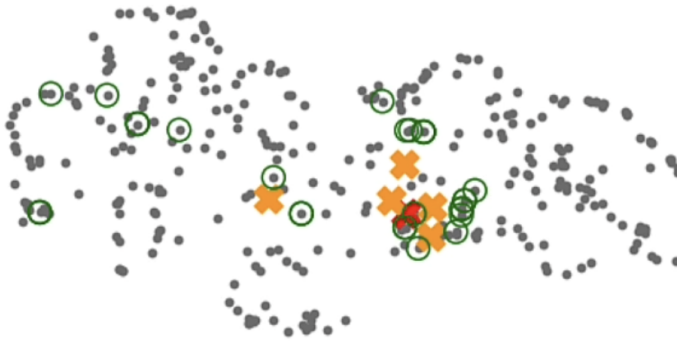


```
def augment_multiple_query(query, model="gpt-3.5-turbo"):
    messages = [
        {
            "role": "system",
            "content": "You are a helpful expert financial resea
            "Suggest up to five additional related questions to
            "Suggest only short questions without compound sente
            "Make sure they are complete questions, and that the
            "Output one question per line. Do not number the que
        },
        {"role": "user", "content": query}
    ]

    response = openai_client.chat.completions.create(
        model=model,
        messages=messages,
    )
    content = response.choices[0].message.content
    content = content.split("\n")
    return content
```

→ a more detailed message!

→ “Suggest a variety of questions that cover different aspects of the topic.” because we are asking for different but related queries (because it could simply rephrase the query)

→ add all the queries to a single array → retrieve the documents for each query



red → original query

orange → new queries

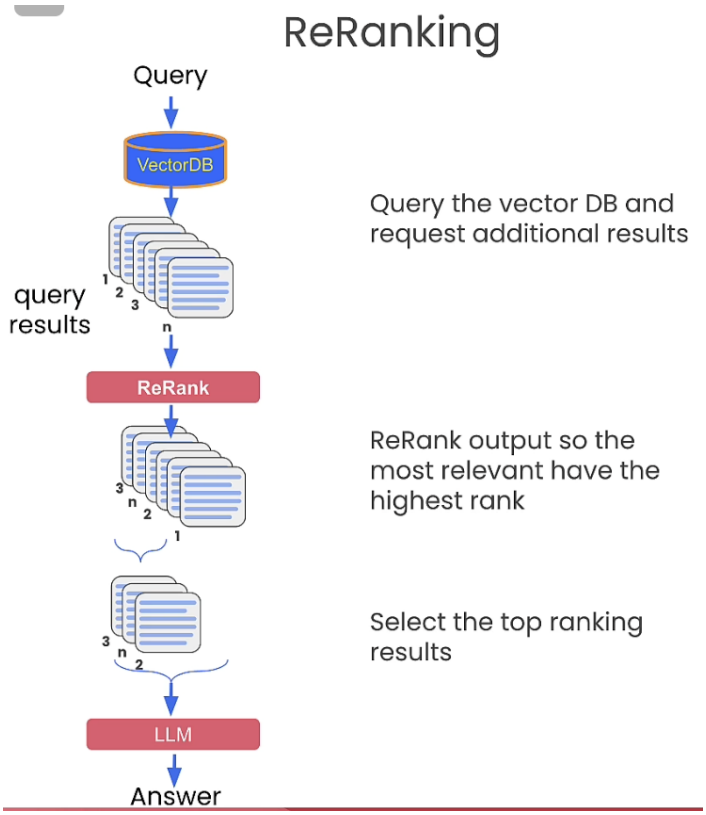
green circle → retrieved data

multiple places reached using this to give a better chance at capturing all the related info

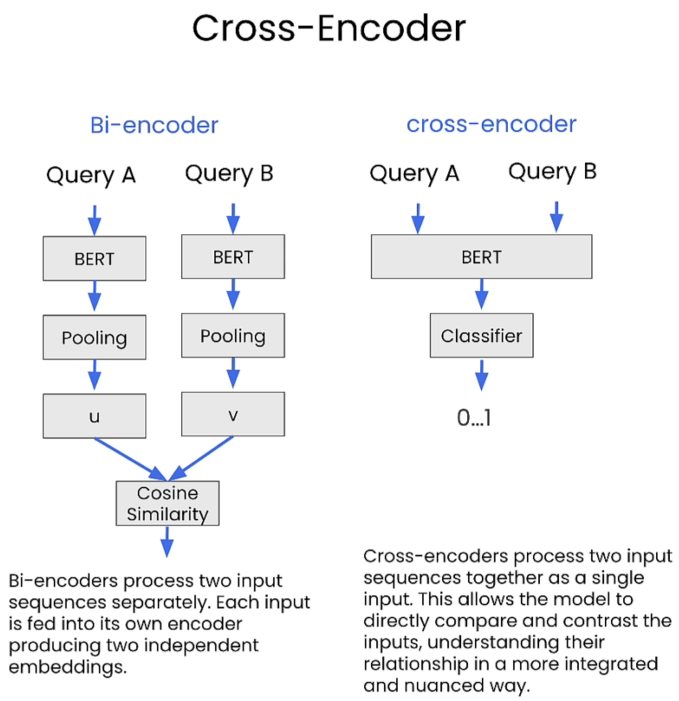
Downside : We have a lot more results than we had originally and we don't know what is relevant and what isn't.

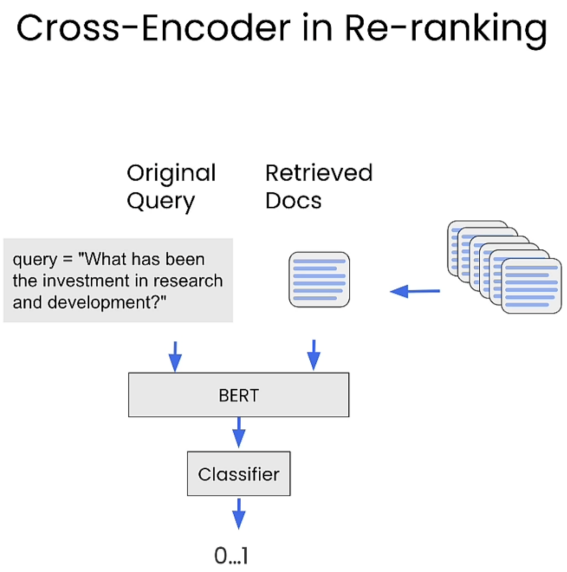
Cross-Encoder Re-Ranking

→ to score the relevancy of the retrieved results for the queries that we sent




```
#what is used?

from sentence_transformers import CrossEncoder
cross_encoder = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6
```





the resulting score is used as the relevancy score for re-ranking

Re-ranking with Query Expansion

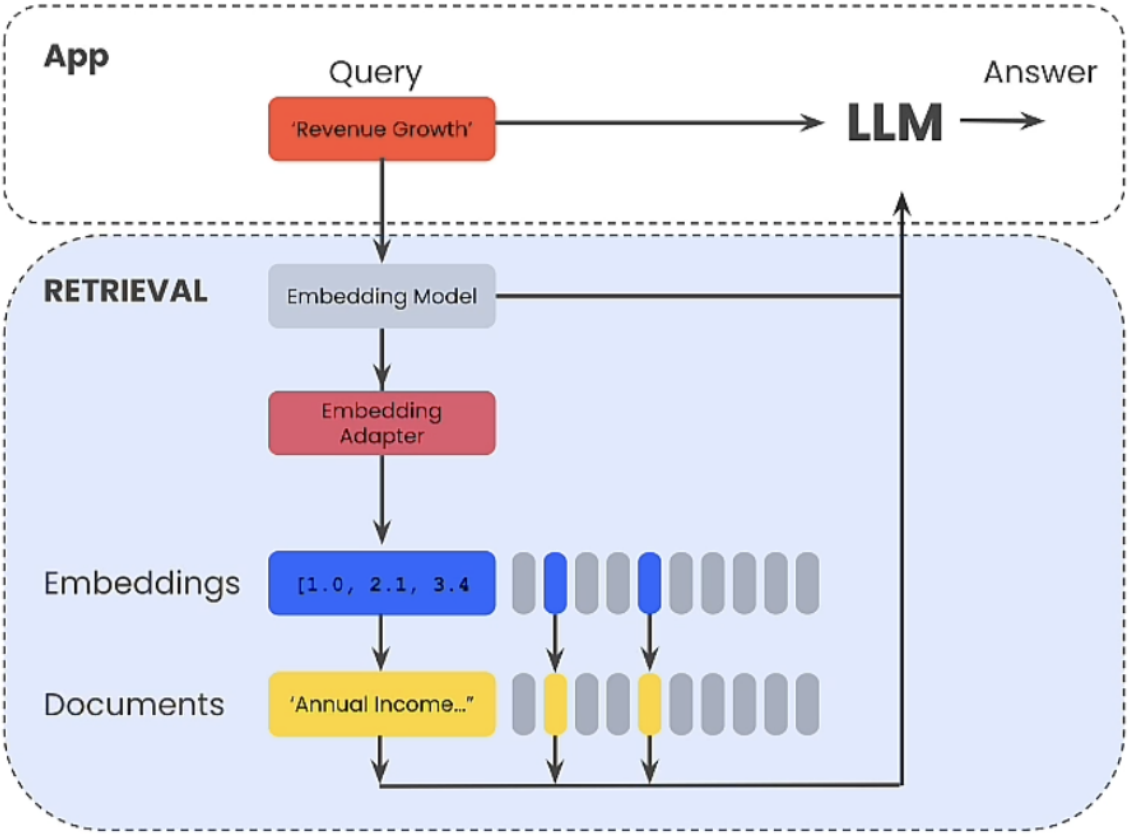
same as what we were doing to get query expansion → make pairs of the query and the documents → we make pairs of each query and each retrieved documents → we can compute the relevancy of the retrieved results for the augmented

queries and select among them the five best that we actually want to pass to the LLM

used cross-encoder as a re-ranking model

Embedding Adapters

→ embedding adaptors are a way to alter the embedding of a query to produce better results



additional stage (embedding adapter) is added → happens after the embedding model but before retrieving the most relevant results → it is trained using the user

feedback on the relevancy of the retrieved results on the set of queries

```
def generate_queries(model="gpt-3.5-turbo"):
    messages = [
        {
            "role": "system",
            "content": "You are a helpful expert financial resea
            "Suggest 10 to 15 short questions that are important
            "Do not output any compound questions (questions wit
            "Output each question on a separate line divided by
        },
    ]

    response = openai_client.chat.completions.create(
        model=model,
        messages=messages,
    )
    content = response.choices[0].message.content
    content = content.split("\n")
    return content
```

```
def evaluate_results(query, statement, model="gpt-3.5-turbo"):
    messages = [
    {
        "role": "system",
        "content": "You are a helpful expert financial research
        "For the given query, evaluate whether the following sat
        "Output only 'yes' or 'no'."
    },
    {
        "role": "user",
        "content": f"Query: {query}, Statement: {statement}"
    }
    ]
```

```
    response = openai_client.chat.completions.create(
        model=model,
        messages=messages,
        max_tokens=1
    )
    content = response.choices[0].message.content
    if content == "yes":
        return 1
    return -1
```

use the evaluated results to train the adapter → it will try to get the cosine similarity close to -1 for irrelevant stuff and close to 1 for relevant stuff (MSE loss function)