# ~~Beyond~~ Explaining the Basics
# Of
# Retrieval (Augmented Generation)
# …
# MVPs with a twist

Ben Clavié
June {10, 11}th, 2024

# About Me

I do R&D at Answer.AI under Jeremy Howard, with other awesome people.

Prior to joining Answer.AI, I worked in a variety of roles in NLP/Information Retrieval, eventually moving to consulting.

I made the RAGatouille library, which makes ColBERT friendlier to use, and also maintain the rerankers lib (more on that in a few slides!)

If you know me, it's most likely via twitter, at @bclavie.

# Topics

**Overall theme:** Loose presentation of the core Retrieval Basics, as they should exist in all RAG pipelines:

- **Rant**: Retrieval was not invented in December 2022
- **The “compact MVP”**: Bi-encoder single vector embeddings and cosine similarity are all you need
- What’s a **cross-encoder** and why do I need it?
- **Tf-idf and full text search** is so ~~2000s~~ ~~1990s~~ ~~1980s~~ 1970s, there’s no way it’s still relevant, right?
- **Metadata Filtering**: when not all content is potentially useful, don’t make it harder than it needs to be!
- **“Compact MVP++”** : All of the above in 30 lines or less.
- Bonus: Yes, one vector is good, but how about many of them?

# ❌ Topics

What I won't be talking about today:

- ❌ How to systematically monitor and improve **RAG systems** (See Jason & Dan's upcoming course for that!)
- ❌ Evaluations: These are far too important to be covered quickly, and Jo Bergum will be covering how to efficiently do them in his upcoming talk.
- ❌ Benchmarks/Paper references: in the interest of time & space, we'll avoid big scary *Table 3.* and *Figure 2.* in those slides (except once).
- ❌ An overview of all the best performing models
- ❌ Synthetic data and training
- ❌ All the approaches you could actually use (sparse models, ColBERT...), which go beyond the very basics!

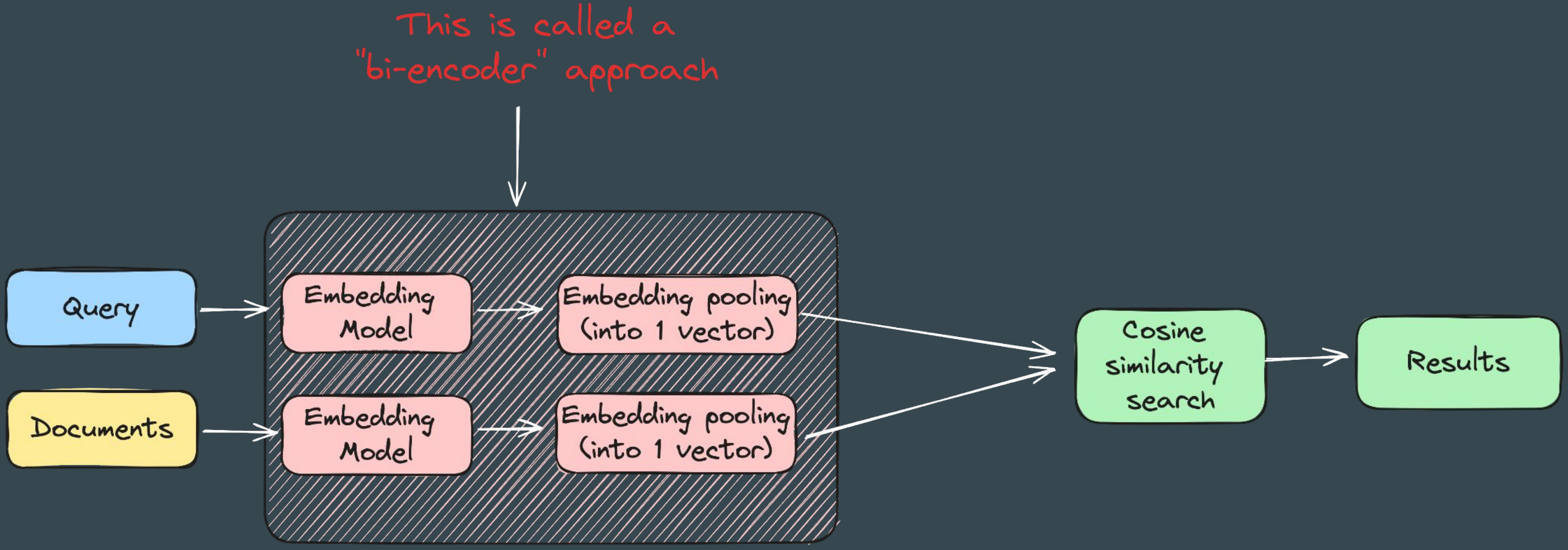
# First, a quick rant

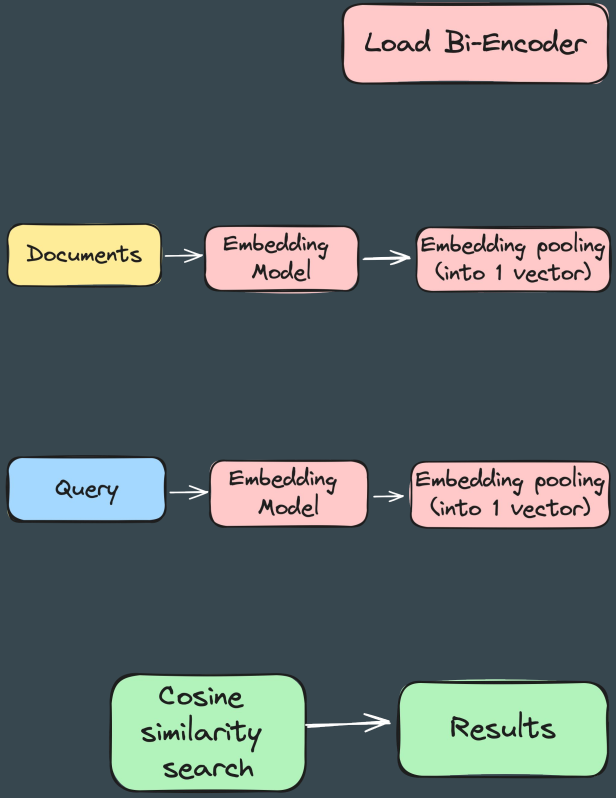
- RAG is not:
  - A new paradigm
  - A framework
  - An end-to-end system
  - Something created by Jason Liu in his endless quest for a Porsche


- RAG is the act of stitching together **R**etrieval and **G**eneration to **ground the latter**
- The **R**etrieval part comes from **Information Retrieval**, a very active field of research
- The **G**eneration part is what's handled by LLMs
- **"Good RAG"** is made up of good components:
  - Good retrieval pipeline
  - Good generative model
  - Good way of linking them up

# The compact MVP

The most compact (& most common) deep retrieval pipeline boils down to a very simple process:

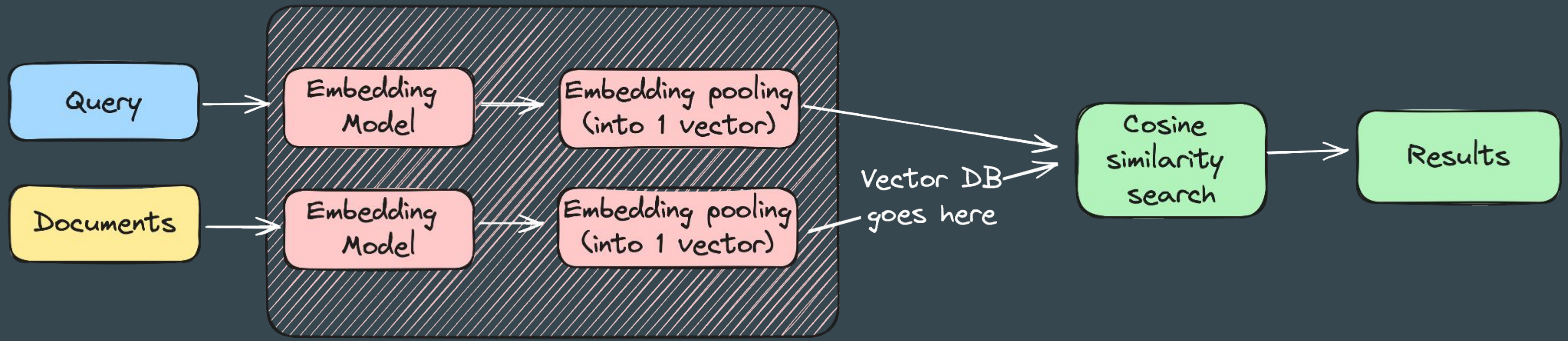





```python
# Load the embedding model
from sentence_transformers import SentenceTransformer
model = SentenceTransformer("Alibaba-NLP/gte-base-en-v1.5")

# Fetch some text content...
from wikipediaapi import Wikipedia
wiki = Wikipedia('RAGBot/0.0', 'en')
doc = wiki.page('Hayao_Miyazaki').text
paragraphs = doc.split('\n\n')
# ...And embed it.
docs_embed = model.encode(paragraphs, normalize_embeddings=True)

# Embed the query
query = "What was Studio Ghibli's first film?"
query_embed = model.encode(query, normalize_embeddings=True)

# Find the 3 closest paragraphs to the query
import numpy as np
similarities = np.dot(docs_embed, query_embed.T)
top_3_idx = similarities.topk(3).indices.tolist()
most_similar_documents = [paragraphs[idx] for idx in top_3_idx]
```

# Wait, where's the vector DB?

- **The vector db in this example is `np.array`!**
- A key point of using a vector DB (or an index) is to allow **Approximate search**, so you don't have to compute too many cosine similarities.
- **You don't actually need one to search through vectors at small scales:** any modern CPU can search through 100s of vectors in milliseconds.



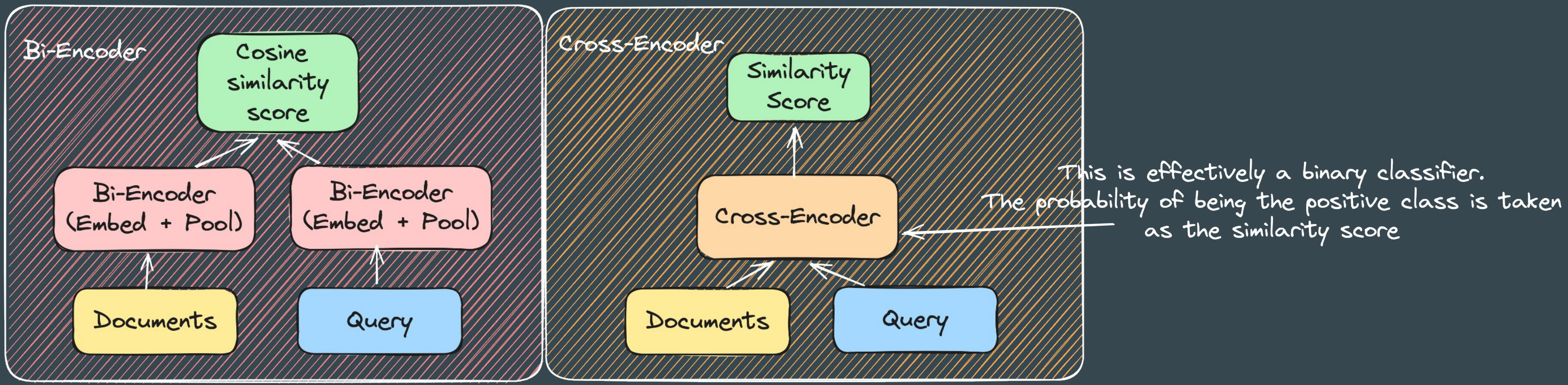
# Why are you calling embeddings “bi-encoders”?

- The representation method from the previous slides is commonly referred to as using “bi-encoders”
- Bi-encoders are (generally) used to create **single-vector representations**. They **pre-compute** document representations.
- Documents and query representations are computed **entirely separately**, they aren’t **aware of each other**.
- Thus, all you need to do at inference is to **encode your query** and search for similar document vectors
- This is **very computationally efficient**, but comes with retrieval performance tradeoffs.

# Reranking: The power of Cross-Encoders (& more!)

- So if documents & query being unaware of each other is bad, how do we fix it?

- The most common approach is using **Cross-Encoders:**




- However, It's **not computationally realistic** to compute query-aware document representations for every single query-document pair, everytime a new query comes up (imagine doing that against every Wikipedia paragraph!)

# The World of Rerankers

- You might have also heard of other re-ranking approaches: RankGPT/RankLLM, T5-based rerankers, etc...

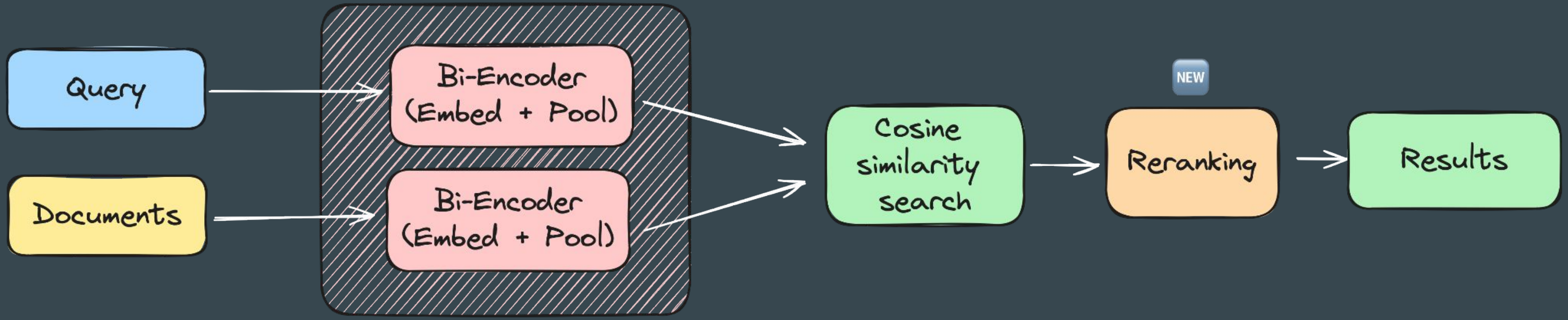
- Their method differs but the core idea is the same: **leverage a powerful but computationally expensive model to score only a subset your documents, previously retrieved by a more efficient model.**

Shameless Self-Promotion

- There are many models for you to try out, some of them API-based (Cohere, Jina...), some of them you can run locally (such as mixedbread). [Luckily, I have a library to make that easy]().

# Compact Pipeline + Reranking

- With the addition of a re-ranking step, this is what your Retrieval pipeline now looks like:



# Keyword Search: The Old Legend Lives On (½)

- Semantic search via embeddings is powerful, but **compressing information from hundreds of tokens to a single vector is bound to lose information.**

- Embeddings learn to represent information **that is useful to their training queries**.
- This training data **will never be fully representative**, especially when you use the model **on your own data**, on which it hasn't been trained.

- Additionally, **humans love to use keywords**. We have very strong tendencies to notice and use certain acronyms, domain-specific words, etc...

- To capture all this signal, **you should ensure your pipeline uses Keyword search**

# Keyword Search: The Old Legend Lives On (2/2)

- **Keyword search**, also called **"full-text search"**, is built on old technology: BM25, powered by tf-idf (a way of representing text and weighing down words that are common)

- An ongoing joke is that **information retrieval has progressed slowly because BM25 is too strong a baseline.**

- BM25 is especially powerful on longer documents and documents containing **a lot of domain-specific jargon**.

- Its inference-time compute overhead is **virtually unnoticeable,** and it's therefore a near free-lunch addition to any pipeline.

# An arXiv-style Results Table to Praise BM25

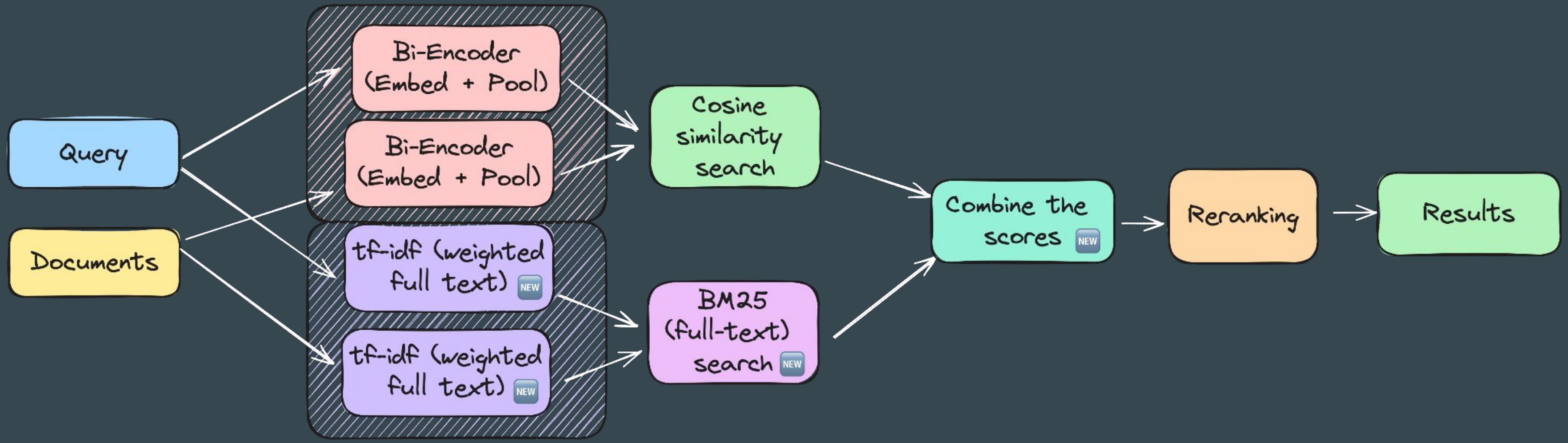
| **Model** (→) | Lexical | Sparse | | | Dense | | | |
|---|---|---|---|---|---|---|---|---|
| **Dataset** (↓) | **BM25** | **DeepCT** | **SPARTA** | **docT5query** | **DPR** | **ANCE** | **TAS-B** | **GenQ** |
| MS MARCO | 0.228 | 0.296$^{\ddagger}$ | 0.351$^{\ddagger}$ | 0.338$^{\ddagger}$ | 0.177 | 0.388$^{\ddagger}$ | 0.408$^{\ddagger}$ | 0.408$^{\ddagger}$ |
| TREC-COVID | 0.656 | 0.406 | 0.538 | 0.713 | 0.332 | 0.654 | 0.481 | 0.619 |
| BioASQ | 0.465 | 0.407 | 0.351 | 0.431 | 0.127 | 0.306 | 0.383 | 0.398 |
| NFCorpus | 0.325 | 0.283 | 0.301 | 0.328 | 0.189 | 0.237 | 0.319 | 0.319 |
| NQ | 0.329 | 0.188 | 0.398 | 0.399 | 0.474$^{\ddagger}$ | 0.446 | 0.463 | 0.358 |
| HotpotQA | 0.603 | 0.503 | 0.492 | 0.580 | 0.391 | 0.456 | 0.584 | 0.534 |
| FiQA-2018 | 0.236 | 0.191 | 0.198 | 0.291 | 0.112 | 0.295 | 0.300 | 0.308 |
| Signal-1M (RT) | 0.330 | 0.269 | 0.252 | 0.307 | 0.155 | 0.249 | 0.289 | 0.281 |
| TREC-NEWS | 0.398 | 0.220 | 0.258 | 0.420 | 0.161 | 0.382 | 0.377 | 0.396 |
| Robust04 | 0.408 | 0.287 | 0.276 | 0.437 | 0.252 | 0.392 | 0.427 | 0.362 |
| ArguAna | 0.315 | 0.309 | 0.279 | 0.349 | 0.175 | 0.415 | 0.429 | **0.493** |
| Touché-2020 | **0.367** | 0.156 | 0.175 | 0.347 | 0.131 | 0.240 | 0.162 | 0.182 |
| CQADupStack | 0.299 | 0.268 | 0.257 | 0.325 | 0.153 | 0.296 | 0.314 | 0.347 |
| Quora | 0.789 | 0.691 | 0.630 | 0.802 | 0.248 | 0.852 | 0.835 | 0.830 |
| DBPedia | 0.313 | 0.177 | 0.314 | 0.331 | 0.263 | 0.281 | 0.384 | 0.328 |
| SCIDOCS | 0.158 | 0.124 | 0.126 | 0.162 | 0.077 | 0.122 | 0.149 | 0.143 |
| FEVER | 0.753 | 0.353 | 0.596 | 0.714 | 0.562 | 0.669 | 0.700 | 0.669 |
| Climate-FEVER | 0.213 | 0.066 | 0.082 | 0.201 | 0.148 | 0.198 | 0.228 | 0.175 |
| SciFact | 0.665 | 0.630 | 0.582 | 0.675 | 0.318 | 0.507 | 0.643 | 0.644 |
| Avg. Performance vs. BM25 | | **- 27.9%** | **- 20.3%** | **+ 1.6%** | **- 47.7%** | **- 7.4%** | **- 2.8%** | **- 3.6%** |

Results table from *BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models (2021), Thakur et al.*

This paper introduces **BEIR**, aka the retrieval part of MTEB.

# The TF-IDF MVP++

With text search and reranking, this is what your pipeline now looks like:



# Metadata Filtering

- An extremely important component of **production** Retrieval is **metadata filtering.**
- Outside of academic benchmarks, **documents do not exist in a vacuum**. There's a lot of **metadata around them,** some of which can be very informative.
- Take this query:

Can you get me the cruise division financial report for Q4 2022?

- There is a lot of ways semantic search can fail here, the two main ones being:
  - The model must accurately represent all of "financial report", but also "cruise division", "Q4" and "2022", **into a single vector,** otherwise it will fetch documents **that look relevant but aren't meeting one or more of those criteria.**
  - If the number of documents you search for ("k") is set too high, you will be passing **irrelevant financial reports to your LLM**, hoping that it manages to figure out which set of numbers is correct.

# Metadata Filtering

- It's perfectly possible that vector search would succeed for this query, **but it's a lot more likely that it will fail in at least one way.**
- However, this is very easy to mitigate: there are entity detection models, such as GliNER, who can very easily extract zero-shot entity types from text:

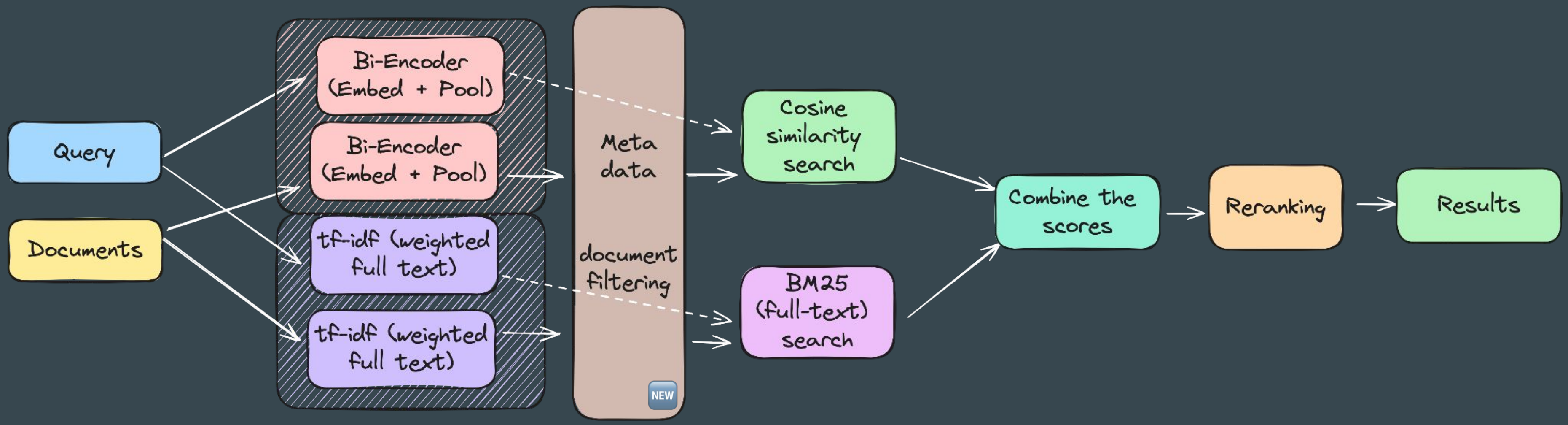

- All you need to do is ensure that **business/query-relevant information is stored alongside their associated documents**.

- You can then use the extracted entities to **pre-filter your documents**, ensuring you only **perform your search on documents whose attributes are related to the query.**

❤️ *GliNER demo from Tom Aarsen on HuggingFace Spaces, based on GliNER, introduced in GLiNER: Generalist Model for Named Entity Recognition using Bidirectional Transformer (2023), Zaratiana et al. (try it if you haven't already, it is a massive game-changer for any sort of pipeline that could use robust entity-detection with little overhead!)*

# The Final Compact MVP++

With this final additional component, this is what your MVP **Retrieval** pipeline should now look like:




This does look scarier (especially if you have to fit into a slide), but it's **very simple to implement.**

# The Final Compact MVP++

- This is the full implementation of all the tricks discussed.
- It might look slightly unfriendly, but there is actually very little to parse!
- Let's shed the data loading and see what's going on...

```python
# Fetch some text content in two different categories
from wikipediaapi import Wikipedia
wiki = Wikipedia('RAGBot/0.0', 'en')
docs = [{"text": x,
         "category": "person"}
        for x in wiki.page('Hayao_Miyazaki').text.split('\n\n')]
docs += [{"text": x,
          "category": "film"}
         for x in wiki.page('Spirited_Away').text.split('\n\n')]

# Enter LanceDB
import lancedb
from lancedb.pydantic import LanceModel, Vector
from lancedb.embeddings import get_registry

# Initialise the embedding model
model_registry = get_registry().get("sentence-transformers")
model = model_registry.create(name="BAAI/bge-small-en-v1.5")

# Create a Model to store attributes for filtering
class Document(LanceModel):
    text: str = model.SourceField()
    vector: Vector(384) = model.VectorField()
    category: str

db = lancedb.connect(".my_db")
tbl = db.create_table("my_table", schema=Document)

# Embed the documents and store them in the database
tbl.add(docs)

# Generate the full-text (tf-idf) search index
tbl.create_fts_index("text")

# Initialise a reranker -- here, Cohere's API one
from lancedb.rerankers import CohereReranker

reranker = CohereReranker()

query = "What is Chihiro's new name given to her by the witch?"

results = (tbl.search(query, query_type="hybrid") # Hybrid means text + vector
.where("category = 'film'", prefilter=True) # Restrict to only docs in the 'film' category
.limit(10) # Get 10 results from first-pass retrieval
.rerank(reranker=reranker) # For the reranker to compute the final ranking
          )
```

Load Bi-encoder

Define document metadata

Bi-Encoder (Embed + Pool)

tf-idf (weighted full text)

Load reranker

```python
# Enter LanceDB
import lancedb
from lancedb.pydantic import LanceModel, Vector
from lancedb.embeddings import get_registry
from lancedb.rerankers import CohereReranker

# Initialise the embedding model
model = get_registry().get("sentence-transformers").create(name="BAAI/bge-small-en-v1.5")

# Create a Model to store attributes for filtering
class Document(LanceModel):
    text: str = model.SourceField()
    vector: Vector(384) = model.VectorField()
    category: str
db = lancedb.connect(".my_db")
tbl = db.create_table("my_table", schema=Document)

# Embed the documents and store them in the database
tbl.add(docs)

# Generate the full-text (tf-idf) search index
tbl.create_fts_index("text")

# Initialise a reranker -- here, Cohere's API one
reranker = CohereReranker()

query = "What is Chihiro's new name given to her by the witch?"
results = (tbl.search(query, query_type="hybrid") # Hybrid means text + vector
.where("category = 'film'", prefilter=True) # Restrict to only docs in the 'film' category
.limit(10) # Get 10 results from first-pass retrieval
.rerank(reranker=reranker) # For the reranker to compute the final ranking
          )
```

# That’s all folks

- There’s a lot more to cover, but this is **your ideal quick MVP**!
- Most other improvements are **also very valuable, but will have decreasing cost-effort ratio.**
- It’s **definitely worth learning about** Sparse (like SPLADE) and multi-vector methods (like ColBERT) if you’re interested – feel free to bug me on the discord!
- You should watch Jason’s talk about RAG systems and Jo’s upcoming talk about retrieval evaluations!
- Any questions?