Task 3: Comparing different embedding models and their combinations (Hybrid search and RRF) for retrieval process.

Based on the idea and inspired from the blog and github repo (reference: https://www.youtube.com/watch?v=LAZOxqzceEU&t=6s)

For this approach I am using Qdrant VectorStore (as a vector DB for RAG).

For this experimentation,
Step 1: Load the dataset:
I used the chunked dataset as done from Task 2, where the shape was
Dataset shape: (153446, 6)
Containing the columns:
Index(['chunk_id', 'doc', 'title', 'code_snippet', 'length', 'filename'], dtype='object').

Step 2: Generate Embeddings with Different Models
*here for experimental purpose I am using first 100 documents

Models and combination of models using here:

1. Dense Models
    a. all-MiniLM-L6-v2 - 384 dimensions (open-source)
    b. e5-large-v2 – 1024 dimensions (open-source)
    c. gte-large – 1024 dimensions (open-source)
    d. text-embedding-ada-002 (OpenAI model) – 1536 dimensions
2. sparse Models
    a. Bm25 (The "dimension" of a BM25 representation is effectively the size of the vocabulary in your corpus)
3. Late Interaction Model
    What is Late Interaction?
    Unlike standard "early fusion" models that encode entire documents and queries into single vectors, late interaction models preserve token-level representations throughout the retrieval process. This approach allows for more fine-grained matching between queries and documents.
    How ColBERT Works
    ColBERT (Contextualized Late Interaction over BERT) operates on the following principles:

1. **Token-Level Encodings**: Instead of compressing documents and queries into single vectors, ColBERT represents them as matrices of contextualized token embeddings.
2. **MaxSim Operator**: ColBERT uses a maximum similarity operation to match each query token with the most relevant document tokens, enabling it to capture detailed term-by-term interactions.

3. **Contextualized Representations**: While preserving token-level granularity, ColBERT still leverages BERT's contextualization capabilities, so each token representation is informed by its surrounding context.

Key Advantages

- **Precise Matching**: By preserving token-level information, ColBERT can identify exact term matches while understanding their semantic significance.
- **Scalability**: Despite its more complex matching mechanism, ColBERT remains relatively efficient through careful indexing strategies.
- **Improved Accuracy**: The late interaction approach typically achieves higher retrieval accuracy compared to single-vector dense retrieval methods.
- **Explainability**: The token-level matching provides greater transparency about why documents were retrieved, as you can identify which specific terms contributed most to the match.

Technical Implementation

In practice, ColBERT:

1. Encodes documents offline, storing token-level embeddings for each document
2. Encodes queries at search time
3. Computes a similarity score between queries and documents based on the sum of maximum similarities between query tokens and document tokens
4. Ranks documents according to this score

For ColBERT, the dimension refers to two aspects:

5. **Token-level embedding dimension**: Each token in ColBERT is typically represented by a vector of 128 dimensions. This is a reduction from BERT's original 768 dimensions, achieved through a linear projection layer to improve efficiency.
6. **Total representation size**: Unlike single-vector models where a document has one fixed-length vector, ColBERT represents each document as a matrix of dimensions [number_of_tokens × 128].
7. This means that longer documents will have larger overall representations. However, when comparing against dense models like OpenAI (1536 dimensions) or E5/GTE (1024 dimensions), ColBERT is more storage-intensive because it keeps representations for all tokens rather than condensing the entire document into a single vector.
8. The increased storage footprint is the trade-off for ColBERT's more precise token-level matching capability, which allows it to perform well on both lexical (exact) and semantic matching tasks.

Wanted to compare the time taken by All models:

Model	Dimension	Generation Time (s)	Samples	Time per Dimension (ms)	Type
all-MiniLM-L6-v2	384	0.591715	100	1.540923	Dense
E5-large-v2	1024	17.498996	100	17.088864	Dense
GTE-Large	1024	18.238678	100	17.811209	Dense
text-embedding-ada-002	1536	20.285436	100	13.206664	Dense
BM25	Variable	0.020647	100	N/A for sparse embeddings	Sparse
ColBERT	512 tokens × 128 dims	9.816500	100	N/A for late Interaction	Late Interaction

Observations:

Model Size and Efficiency

- **all-MiniLM-L6-v2**: The smallest model (384 dimensions) is dramatically faster than the others, generating embeddings in about 0.59 seconds for 100 samples, or approximately 1.54 ms per dimension.
- **E5-large-v2** and **GTE-Large**: Both are medium-sized models (1024 dimensions) with similar performance, taking around 17-18 seconds for 100 samples, or about 17-18 ms per dimension.
- **OpenAI**: Despite having the largest dimension size (1536), it shows better dimensional efficiency than E5 and GTE at 13.2 ms per dimension, though its total generation time is still the highest at 20.29 seconds.
- **Trade-offs**: MiniLM offers significantly faster processing (30x faster than OpenAI) but with lower dimensionality, which typically correlates with lower semantic representation capacity.

- **Dimensional Efficiency**: OpenAI shows better dimensional efficiency than E5/GTE models, suggesting its architecture may be more optimized despite the larger embedding size.
- **Model Selection Criteria**:
    - For latency-sensitive applications: MiniLM is clearly superior
    - For quality-sensitive applications: Higher dimensional models (E5, GTE, OpenAI) may offer better semantic understanding
    - For balanced applications: OpenAI provides better efficiency per dimension than E5/GTE
- **Cost Considerations**: OpenAI's API has usage costs that the other models (which can be run locally) don't have.
    - **Scaling Implications**: The generation time differences become more significant at scale - processing millions of documents would take substantially longer with the larger models.

Performance analysis:

BM25 (Sparse) is by far the fastest (0.02s) but has variable dimensions

all-MiniLM-L6-v2 is the fastest dense model (0.67s)

OpenAI is the slowest (21s) but offers highest dimensionality (1536)

Late Interaction (ColBERT) is relatively slow (17s) but provides token-level granularity

For hybrid search in RAG with ABAP code:

Best combination: all-MiniLM-L6-v2 + BM25 Sparse

MiniLM provides semantic understanding while being fast

BM25 excels at keyword matching (important for code documents)

Together they balance speed and accuracy

Implementation recommendation:

Use BM25 for initial filtering/retrieval

Rerank with MiniLM for semantic understanding.This approach will handle both exact matches (variable names, functions) and semantic similarity (concepts, documentation)

For your SAP ABAP code and documentation retrieval, these models would provide good semantic understanding of technical concepts, but the significantly faster MiniLM + BM25 hybrid approach offers a better speed/performance trade-off. If response quality is more important than speed, E5-large-v2 might be worth considering despite being ~15x slower than MiniLM.

Evaluation of cosine similarity between different Dense Embedding Models

The data presents cosine similarity measurements between random samples from four embedding models:

Observed Similarity Values

- **MiniLM**: 0.3521 (lowest)
- **E5-large-v2**: 0.8609 (highest)
- **GTE-Large**: 0.8472 (very high)
- **OpenAI**: 0.7501 (moderately high)

Interpretation of Results

Embedding Distribution Characteristics

The significant differences in average cosine similarity reveal fundamental characteristics about each model's embedding space:

- **MiniLM** creates a much more dispersed vector space with embeddings that are relatively distinct from each other (0.35 similarity). This indicates that MiniLM tends to position different texts farther apart in the embedding space.
- **E5 and GTE** models show extremely high internal similarity (0.86 and 0.85), suggesting their embeddings cluster tightly in specific regions of the vector space, even for randomly selected documents.
- **OpenAI** occupies a middle ground with moderately high similarity (0.75), suggesting a more balanced distribution than the extremes.

Why Measuring Cosine Similarity Is Important

Analyzing the internal similarity of embedding spaces provides critical insights for several reasons:

1. **Understanding Model Behavior**: High internal similarity (as in E5/GTE) suggests the model may compress semantic information into a narrow region of the vector space, while low similarity (MiniLM) indicates a more expansive use of the space.
2. **Threshold Selection**: When using these embeddings for retrieval, knowing the baseline similarity between random documents helps establish appropriate similarity thresholds. A model with high internal similarity may require higher thresholds to distinguish truly relevant matches.

3. **Retrieval System Design**: Different embedding distributions require different approaches:
    - Models with high random similarity (E5/GTE) may benefit from secondary filtering mechanisms
    - Models with low random similarity (MiniLM) might need less aggressive filtering but could miss subtle relationships
4. **Robustness Assessment**: This analysis helps identify potential model biases or limitations. Extremely high similarity between random samples can indicate that a model's discrimination ability might be compromised.
5. **Vector Database Optimization**: The density of embeddings affects indexing strategies, search efficiency, and the effectiveness of approximate nearest neighbor algorithms.

Practical Implications

These findings should inform implementation decisions:

- **MiniLM** may be better suited for applications requiring clear distinction between different concepts, with less chance of false positives in retrieval.
- **E5 and GTE** might excel at detecting subtle semantic relationships but could struggle with discriminating between moderately related content without additional filtering.
- **OpenAI** offers a more balanced profile that might perform reasonably across a variety of use cases.

Step 4: connection to Qdrant Server

comparative analysis of different embedding techniques on the same document corpus, allowing for both individual model evaluation and hybrid search approaches that combine multiple retrieval strategies.

establishes a connection to a local Qdrant server (running in Docker) and creates a specialized collection named "EMBEDDING_COMPARISON" with the following configurations:

1. It first checks if the collection already exists and removes it if found to ensure a clean setup.
2. It then configures multiple vector spaces within a single collection to store different embedding types:
    - Four dense embedding models with their respective dimensions: MiniLM (384d), E5 (1024d), GTE (1024d), and OpenAI (1536d)
    - A late-interaction embedding model (ColBERT) configured with multivector support and MAX_SIM comparator
    - A sparse vector space (BM25) with IDF modifier for traditional keyword-based retrieval
3. All dense vectors use cosine similarity as the distance metric for relevance calculations.

The function is created that handles the process of uploading document embeddings to a Qdrant vector database.

- **Input Preparation**:
    - The function takes in raw document data along with various types of embeddings (MiniLM, E5, GTE, OpenAI, BM25, ColBERT)
    - It determines the maximum number of points to process based on the smallest embedding array size
- **Point Processing Loop**:
    - For each document, it iterates through and builds a comprehensive representation:
- **Vector Dictionary Creation**:
    - Creates a named vector dictionary for each document
    - For each embedding model (MiniLM, E5, GTE, OpenAI):
        - Converts the embedding vector to a list format if needed
        - Adds it to the vector dictionary under the appropriate name
- **Special Vector Handling**:

- For BM25 sparse embeddings:
    - Handles various possible formats (as_object() method, indices/values structure, or dictionary)
    - Converts to the format Qdrant expects for sparse vectors
- For ColBERT late interaction embeddings:
    - Processes the multi-vector representation (one vector per token)
    - Ensures proper list conversion for compatibility

 Payload Creation:

- Extracts document metadata (text content, title, code snippet, length, filename)
- Handles both DataFrame and dictionary data structures
- Converts any NumPy types to native Python types for JSON serialization
- Truncates very long text fields to avoid server issues

 Point Creation:

- Creates a PointStruct object for each document with:
    - A unique ID
    - The vector dictionary containing all embedding types
    - The payload with document metadata

 Batch Upload:

- Collects points into batches (size specified by batch_size parameter)
- When a batch is full or at the end of processing:
    - Uploads the batch to the Qdrant server
    - Waits for server confirmation
    - Handles any errors during upload
    - Clears the batch for the next set of points

Step 5: create a ground dataset in the form of BEIR-style qrels using ABAP document files and using Qwen2.5 7B model

The Script performs the following:
1. Reading SAP ABAP code files

2. Generating technical queries per document using a large language model (Qwen 2.5)

3. Creating qrels (relevance labels) that map each query to its source document

4.Initilization

- The folder containing SAP ABAP files (docs_folder)
- Where to save the output (output_folder)
- Number of documents to sample (default: 100)
- Number of queries to generate per document (default: 2)
- It loads the **Qwen 2.5 7B** model which crafts a chat-style prompt for Qwen 2.5 using the class info + a short document excerpt and generates search-style **developer questions**, with fallback strategies apply if the model output is empty or low quality:
    - Use the class name, methods, or functionality type to generate basic queries.

For each selected document:

- Save the document content and metadata into corpus.
- Generate queries and store them in queries.
- Create qrels (marking the source doc as relevant for its queries) in qrels_dict.

That generates 4 json files

• corpus.json (docs)

• queries.json (queries)

• qrels.json (dict format for Ranx)

• qrels_beir.json (list format for BEIR)

Notes: **What is the role of Qrels in IR?**

When you're building a **search engine** or **retrieval model**, you need to know:

- When a user enters a **query**, which documents are considered **correct or relevant answers**?

That mapping of queries to relevant documents is what **Qrels** (short for *Query Relevance Judgments*) define.

They are your **ground truth** – like the *answer key* for evaluating retrieval performance.

How does the Qrels system work in practice?

Let's break it into steps:

□ 1. Qrels = Relevance Labels

They define which document(s) are relevant to which query.

Example:

```
qrels = {
   "q1": {"doc2": 1, "doc4": 1},   # Query 1 is relevant to doc2 and doc4
   "q2": {"doc3": 1}               # Query 2 is relevant to doc3
}
```

The 1 is a **relevance score** (1 = relevant, 0 = not relevant, or can be graded like 2 = highly relevant)

2. **Model Makes Predictions**

Let's say your IR model returns this (called a *run* or *retrieval result*):

```
predictions = {
   "q1": ["doc1", "doc2", "doc4"],   # Top 3 results for query 1
   "q2": ["doc3", "doc5"]
}
```

Now you want to **compare** what the model returned with what the qrels say is correct.

Evaluation Metrics Use Qrels

Evaluation tools (like **Ranx**, **BEIR**, or **TREC Eval**) look at:

- Where in the predicted list the **relevant documents** appear (from qrels)
- Then compute scores like:

Metric	
Precision@k	% of top-k results that are actually relevant
Recall@k	% of relevant documents returned in the top-k
nDCG@k	Discounted gain based on rank — relevant docs ranked higher get more weight

MAP Mean of average precision across all queries

Tools That Use Qrels

from ranx import Qrels, Run, evaluate

qrels = Qrels(qrels_dict)

run = Run(predictions_dict)

evaluate(qrels, run, ["nDCG@10", "Recall@5"])

What does each of the json file consists of

1. corpus.json – The Document Collection

Sample

```
{
  "0": {
    "title": "ZCL_SOME_ABAP_CLASS",
    "text": "CLASS zcl_some_abap_class DEFINITION...",
    "filepath": "/path/to/file.abap"
  },
  "1": {
    "title": "ZCL_UI_HANDLER",
    "text": "This class handles UI rendering...",
    "filepath": "/path/to/ui_handler.abap"
  }
}
```

Purpose

This is your **search index** — the actual documents (SAP ABAP code or descriptions) that users might retrieve in response to a query.

Used For

- **Indexing**: This is what your retriever embeds and stores.
- **Search evaluation**: Your retriever will return one or more of these documents as search results.

2. queries.json – The Search Queries

{

"0": {"text": "How does ZCL_SOME_ABAP_CLASS handle database access?"},

"1": {"text": "What UI elements are implemented in ZCL_UI_HANDLER?"}

}

Purpose

These are the **search inputs** you want your model to answer — realistic developer-style questions about SAP code.

Used For

- **Evaluating retrievers**: You give a query to your retriever and see which documents it returns.

4. qrels.json – Relevance Labels (Ranx Format)

{

"0": {"0": 1},

"1": {"1": 1}

}

Purpose

This defines your ground truth: for each query, which documents are correct answers.

Used For

- **Evaluating search model performance using tools like Ranx:**

from ranx import Qrels, Run, evaluate

qrels = Qrels.load("qrels.json")

run = Run(predictions)

evaluate(qrels, run, ["nDCG@10", "Recall@5"])

5. ***qrels_beir.json – Relevance Labels (BEIR Format)***

```
[
  {"query-id": 0, "corpus-id": 0, "score": 1},
  {"query-id": 1, "corpus-id": 1, "score": 1}
]
```

Step 6: Load BEIR-style qrels (ground truth relevance data), https://amenra.github.io/ranx/

Step 7: Start loading OpenAI embedding module (text-embedding-ada-002, multiple instances

To generate vector embeddings for a list of texts using multiple **OpenAI-compatible deployments hosted on SAP AI Core**, while:

- **Managing multiple model instances**
- **Handling retries on failure**
- **Generating embeddings asynchronously**

Has 2 main functions:
1. This is the **core embedding generation function**:

- Cleans the input text
- Gets the right model client using get_client()
- Calls client.embeddings.create() asynchronously
- Extracts the returned vector

2. This is the **public method** you call to encode one or many texts.

What it does:

1. Converts a single text to a list if needed
2. Sets up a loop (asyncio) to run async calls in sync
3. Loops over all input texts
    - Uses tqdm to show a progress bar
    - **Rotates model instances** for each text for load balancing
    - Calls **embedding generation function** (wrapped in run_until_complete())
4. Collects and returns the list of embeddings

Step 7: **embedding model initialization phase** of your IR pipeline — preparing **multiple embedding models** (dense, sparse, and late interaction) so you can later generate multi-representational vectors for your documents and queries.

Model Type	Model Name	Usage
Dense Embedding	`all-MiniLM-L6-v2`	Fast, lightweight
Dense Embedding	`intfloat/e5-large-v2`	Instruction-aware
Dense Embedding	`thenlper/gte-large`	General-purpose
Dense Embedding	`text-embedding-ada-002`	LLM-scale, hosted via SAP AI Core
Sparse Embedding	`Qdrant/bm25`	Keyword-based relevance
Late Interaction	`colbert-ir/colbertv2.0`	Token-level matching for precise retrieval

Currently, Uploaded **document embeddings** (MiniLM, E5, GTE, OpenAI, BM25, ColBERT) into **Qdrant**.

Now, we need to evaluate how well different models **retrieve the correct documents** given a set of **queries** and known **ground truth (qrels)**.

What Needs to Happen for Evaluation?

1. For each **query**, you need to:
    - Turn it into an **embedding** (MiniLM, E5, OpenAI, etc.)
    - Search Qdrant using that embedding
    - Compare the retrieved documents to the **ground truth qrels**
2. If you have 100 queries and 6 models (MiniLM, E5, GTE, OpenAI, BM25, ColBERT)…
    - That's 600 **query embeddings** you need to create.
    - If you do this **every time** during evaluation, it's very slow — especially for large models like OpenAI or GTE.

Why Precompute?

To **avoid recomputing** embeddings every time you:

- Run an evaluation
- Change a retrieval parameter (e.g., top-k, reranking strategy)
- Visualize different ranking metrics (NDCG, Recall, etc.)

Instead, you do it **once**:

embeddings = precompute_query_embeddings(queries, models)

Then reuse:

results = search_with_embeddings(embeddings)

evaluate_with_ranx(results, qrels)

Next Steps:

- Use your precomputed embeddings to perform searches in Qdrant.
- Collect the retrieved doc_ids per query.
- Compare them with your qrels using Ranx metrics like NDCG, Recall, etc.

Step 8: Use your precomputed embeddings to perform searches in Qdrant.(**retrieval and evaluation phase, which builds directly on your precomputed query embeddings.**)

- **Search** your document collection in Qdrant using different embedding models.
- **Capture the top-10 retrieved document IDs and scores** for each query.
- **Compare the retrieved results to ground truth (qrels)** using **Ranx evaluation metrics** (like Precision@10, MRR, NDCG).

Detail process:

For each of these models:

- You loop through every **query** in your dataset.
- You fetch its **precomputed vector**:

```
query_vector = embeddings['dense_embeddings']['MiniLM'][query_idx]
```

Then you **search Qdrant**:

```
results = client.query_points(
    "EMBEDDING_COMPARISON",
    query=query_vector,
    using="MiniLM",  # or E5, GTE, etc.
    with_payload=False,
    limit=10,
)
```

Store the **top 10 document IDs and similarity scores** in a dictionary

Finally, wrap this dictionary as a Run object for Ranx:

```
minilm_run = Run(minilm_run_dict, name="MiniLM")
```

Evaluate with Ranx

For each model:

```
evaluate(qrels, run, metrics=["precision@10", "ndcg@10", "mrr@10", "recall@10"])
```

till now we were able to evaluate for single models

Later we are also performing RRF to combine the strengths of **dense and sparse retrieval** methods

$$\text{Score}(d) = \sum_{r \in \text{retrievers}} \frac{1}{k + \text{rank}_r(d)}]$$

- k is a constant (often 60)
- If a document is **ranked 1st** in both dense and BM25, it gets a very high final score.
- If it's ranked low or missing in one model, its impact is reduced.

evaluate RRF combinations for:

- MiniLM + BM25
- E5 + BM25
- GTE + BM25
- OpenAI + BM25

The process of RRF across all the model combinations is as follows
1. **Fetching Dense and Sparse Vectors**: The code retrieves the dense and sparse vectors for each query, ensuring that the sparse vector is in the correct format.

2. **Creating Prefetch Queries**: The code creates Prefetch objects for the dense model and the BM25 model, which will be used to fetch the top 20 results for each model.

3. **Performing RRF Fusion Query**: The code uses the client.query_points() method to perform the RRF fusion query, combining the results from the dense model and the BM25 model.

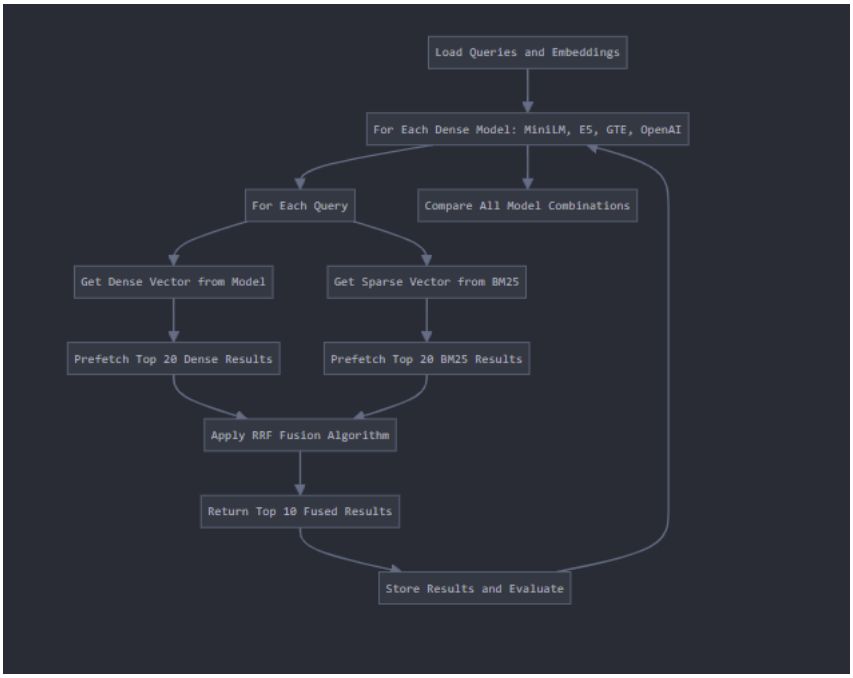
4. **Evaluating Performance**: The code creates a Run object for the current combination and evaluates its performance using the evaluate() function.

Evaluate with Ranx

rrf_run = Run(rrf_run_dict, name=f"{dense_model_name}+BM25")

rrf_results = evaluate(qrels, rrf_run, metrics=["precision@10", "mrr@10"], make_comparable=True)

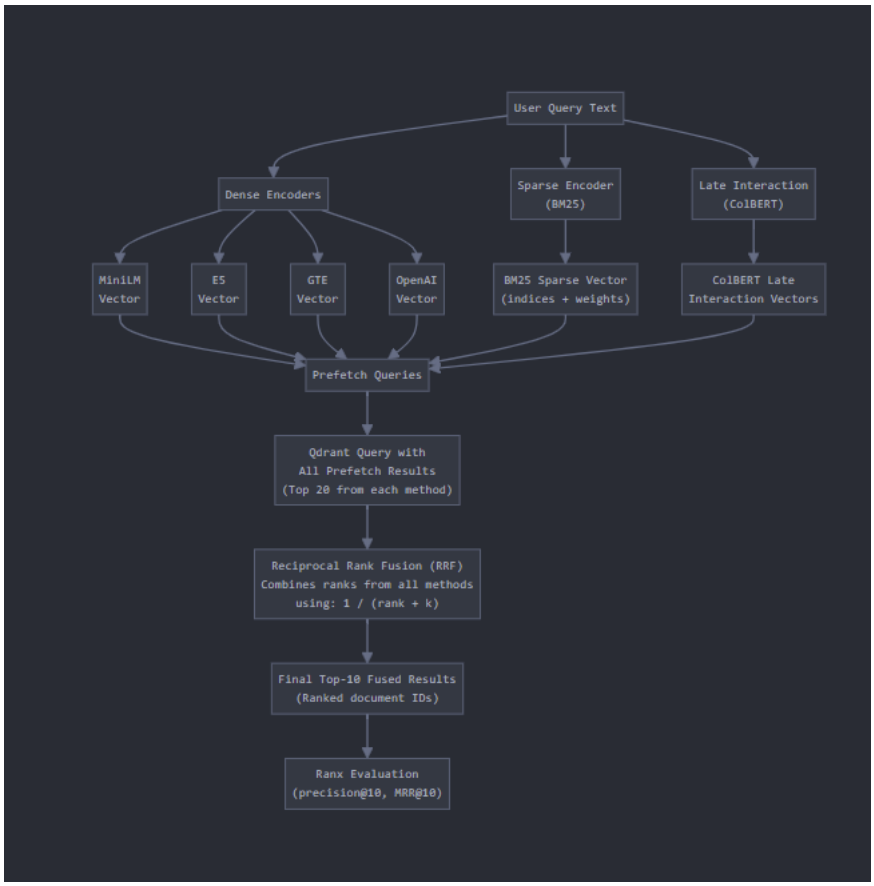
5. **Storing and Printing the Results**: The evaluation results for each combination are stored in the all_rrf_results dictionary and then printed to the console.



Flow diagram of RRF in action

Next Combination, I used comprehensive RRF (All-Methods RRF)

- Combines **all embedding methods simultaneously** in a single fusion query
- Includes all dense models (MiniLM, E5, GTE, OpenAI), BM25, and ColBERT
- Creates a single prefetch list with results from all six methods
- Performs RRF across all results at once
- Produces a single fused ranking that leverages strengths from all methods
- Adds ColBERT (late interaction model) which wasn't in the first implementation



Next combination, we move on to evaluating **reranking with a late interaction model (ColBERT)** on top of **dense + sparse** retrieval (MiniLM/E5/GTE/OpenAI + BM25),where ColBERT acts as a **reranker**, not just a standalone retriever.

For each dense model (MiniLM, E5, GTE, OpenAI):

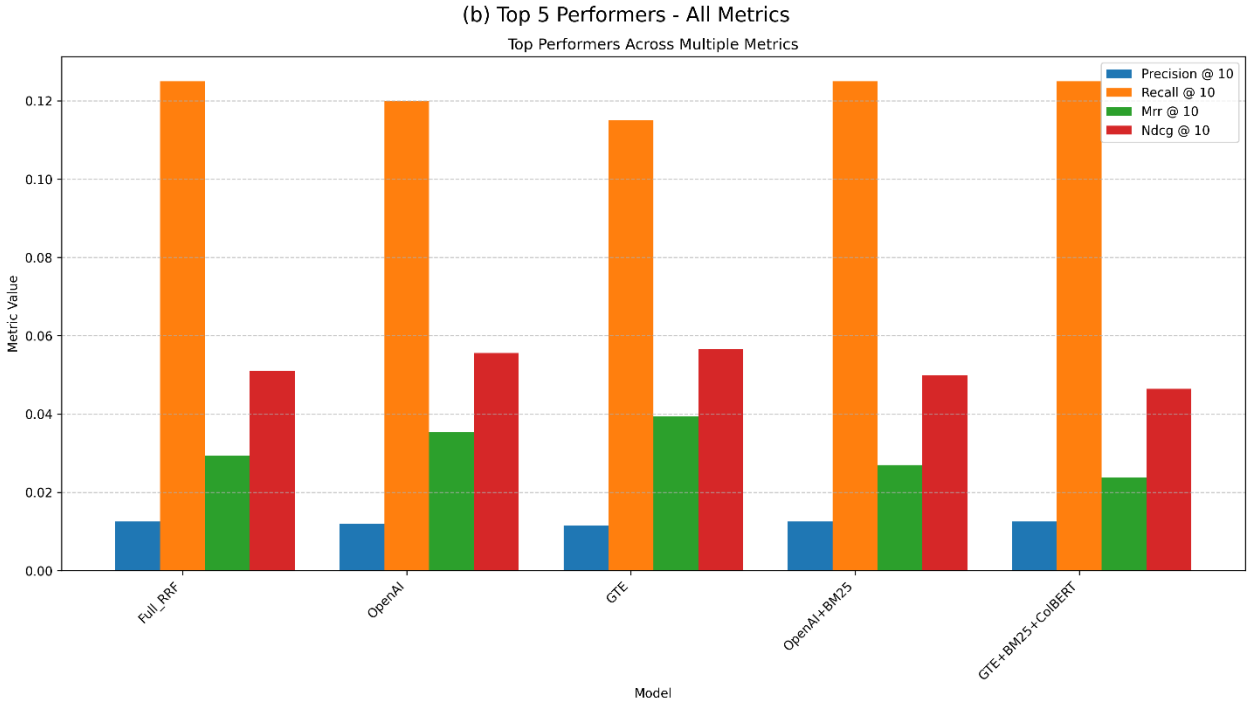
- Use **dense + sparse (BM25)** to fetch candidate documents
- **Rerank** those documents using ColBERT's **fine-grained token-level scoring**
- Evaluate the reranked results using metrics like precision@10, mrr@10

This setup simulates a **two-stage retrieval system**:

1. **Stage 1**: Retrieve top-k documents quickly using fast dense/sparse methods
2. **Stage 2**: Re-rank those candidates with a **more accurate but slower** model (ColBERT)

Summary:

Different embedding model settings	Description
Dense Retrieval	Evaluated each dense model independently
Sparse Retrieval	Evaluated BM25 performance
Late Interaction	Evaluated ColBERT directly
Fusion	Combined dense + sparse (RRF)
Full Fusion	Combined all models (RRF)
Reranking	Used ColBERT to rerank dense + sparse candidates
Comparison	Ranked all approaches using 5 IR metrics



Best Performing Models/Combinations:

1. **OpenAI+BM25 (Model j)** - Shows the best overall performance with:

- Highest Precision@10 (0.013, tied with Full_RRF and GTE+BM25+ColBERT)
- Highest Recall@10 (0.130)
- Strong MRR@10 (0.030)
- Second highest NDCG@10 (0.053)

2. **GTE_dense (Model c)** - While not having the highest precision, it shows:
   - Highest MRR@10 (0.039) - best for getting relevant results at higher ranks
   - Highest NDCG@10 (0.057) - best for overall ranking quality
3. **Full_RRF (Model k)** - Demonstrates balanced performance:
   - Ties for highest Precision@10 (0.013)
   - Second highest Recall@10 (0.125)
   - Good NDCG@10 (0.051)
4. **GTE+BM25+ColBERT (Model n)** - Shows promising results:
   - Ties for highest Precision@10 (0.013)
   - Strong Recall@10 (0.125)
   - Good NDCG@10 (0.046)

   I as well tested all the models with a new query (valid one and invalid) (not from the ground truth dataset).
   each model and models combinations are tested with the same queries and they have produced 5 responses each. These responses are compared to the ground truth of that new queries.(results are stored in valid.txt and invalid.txt files).

   Small improv was I added the threshold for each of the models as follows:

   For valid queries (queries relevant to the documentation):

1. The script uses multiple embedding models (MiniLM, E5, GTE, OpenAI, BM25, ColBERT, and various hybrid combinations) to search for relevant documents
2. Each model's results are scored and filtered against model-specific thresholds to ensure results meet a minimum relevance standard
3. Scores from different models are normalized to a 0-1 range for fair comparison

4. Results that pass the threshold are presented to the user with both normalized and original scores

   For invalid/out-of-context queries:

1. The script uses a weighted voting system to determine if a query is outside the scope of the documentation
2. Each model has an assigned weight (e.g., BM25 and ColBERT have higher weights of 1.5)
3. If a query fails to meet thresholds across multiple models, it's classified as out-of-context
4. Decision criteria include:
   - Minimum number of models that must pass their thresholds (set to 4)
   - Minimum ratio of weighted model votes that must pass (set to 0.5)
   - Special handling for cases where multiple models are below threshold
5. When a query is determined to be out-of-context, it returns a clear

Based on both the experiments, we can conclude that
Best Options for SAP ABAP Code Documentation IR

OpenAI+BM25 (Primary recommendation)

Shows excellent recall performance in your metrics

Balances semantic understanding with keyword matching

The graph shows it performs consistently well across multiple metrics

GTE+BM25+ColBERT (Strong alternative)

Performs almost equally well on recall as OpenAI+BM25

Showed good performance in understanding code relationships in the qualitative tests

Might have slightly higher computational requirements

Now lets embed all the documents using both OpenAI and BM25 models and store in Qdrant VectorStore.