Large Language Models in Data Science

Week 5: Retrieval-Augmented Generation (RAG)

Sebastian Mueller
Aix-Marseille Université
2025-2026

Session Overview

Lecture (1h)

1. From LLMs to RAG
2. Retrieval and search basics
3. Augmenting prompts with context
4. Lexical, metadata, and embedding search
5. Putting it together: RAG architectures
6. Example: Marseille & data science assistant

Lab (2h)

- Ingest a small corpus about Marseille & AMU data science
- Build an embedding index with metadata
- Implement a simple RAG chain
- Compare answers with and without retrieval
- Optional: add citations & reranking

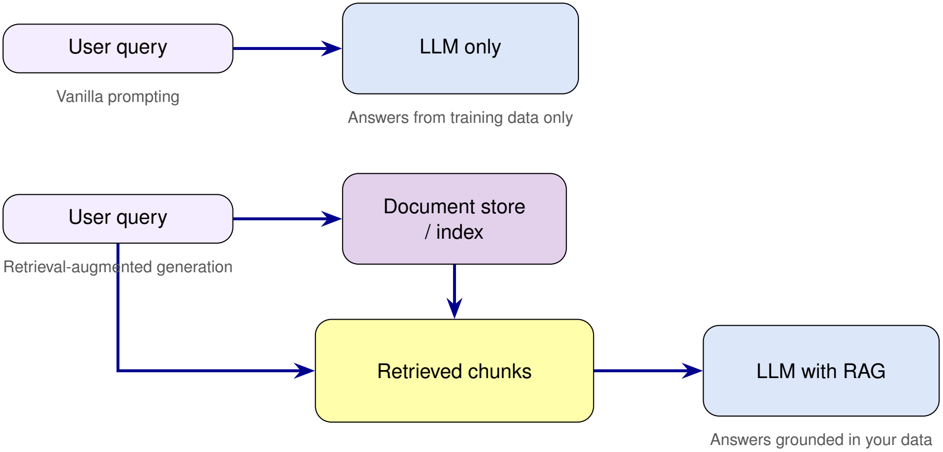
Where We Are in the Course

- Week 1: Tokens, embeddings, transformer architecture.
- Week 2: Using pretrained models via Hugging Face.
- Week 3: Effective LLM use for coding, research, ideation.
- Week 4: Text classification and intent routing for an AMU chatbot.
- **Week 5: RAG** combine all ingredients to build assistants grounded in your own data.

Why Do We Need RAG?

- LLMs are trained on huge but *fixed* corpora; they do not know your internal documents or freshest data.
- Direct prompting relies on the model "remembering" facts from pretraining, which leads to hallucinations and outdated answers.
- Many data science tasks need answers grounded in:
  - institutional documents (course catalog, policies),
  - project reports, notebooks, dashboards,
  - domain-specific knowledge bases.
- RAG attaches a *retrieval system* to the LLM so answers can depend on external, updatable data.

From Vanilla LLM to RAG



From Vanilla LLM to RAG

- Mathematically we move from $P(\text{answer} \mid \text{question})$ to

$$P(\text{answer} \mid \text{question}, \text{retrieved context}).$$

- The only thing the LLM sees is the *prompt*; RAG works by **augmenting the prompt** with relevant context.

Core Components of a RAG System

- **Document store**: where raw texts live (internet, files, database, data lake, etc.).
- **Chunking**: split documents into passages that are short enough to fit in the context window.
- **Metadata**: structured fields such as title, source, date, language, tags.
- **Index**: data structures to support fast search (inverted index, vector index, or both).
- **Retriever**: given a query, returns relevant chunks (possibly with scores).
- **LLM + prompt template**: turns query + retrieved chunks into an answer.

Documents, Chunks, and Metadata

- **Document**: a logically complete piece of text (course syllabus, FAQ page, PDF report).
- **Chunk**: a slice of a document (e.g., a few paragraphs) used as the atomic retrieval unit.
- **Metadata examples for AMU / Marseille:**
  - `source_type` ("course_catalog", "housing_info", "lab_sheet").
  - `campus` ("Luminy", "Saint-Charles", "Aix"), `city` ("Marseille", "Aix-en-Provence").
  - `program` ("MSc Data Science"), `year`, `language`.

Retrieval Basics: Three Signals

- **Lexical / keyword search**
  - Uses exact or fuzzy matches of words (e.g., BM25).
  - Very precise when user vocabulary matches the documents.
- **Embedding (semantic) search**
  - Encode query and chunks into vectors; retrieve nearest neighbors.
  - Captures semantics beyond exact words ("student card" $\approx$ "carte étudiant").
- **Metadata filtering**
  - Restrict the candidate set using structured filters (e.g., campus = "Marseille").
  - Cheap and interpretable; in our lab pipeline we apply it as a *final refinement* after lexical and embedding search.
  - In large production systems you might also use metadata as a coarse pre-filter before more expensive retrieval.

From Bag-of-Words to BM25

- **Bag-of-words (BoW)**:
  - represent each document as a vector of word counts,
  - simple and fast, but treats all terms and all documents equally.
- **Term frequency (TF)**: words that repeat many times in a document should matter more.
- **Inverse document frequency (IDF)**: rare words across the corpus carry more signal than very common ones.
- Naive TF or TF–IDF:
  - can over-favor very long documents (more opportunities to match),
  - can over-reward repeated words without saturation.
- BM25 refines this idea to give a stronger, length-aware lexical baseline.

BM25: Tunable Keyword Scoring

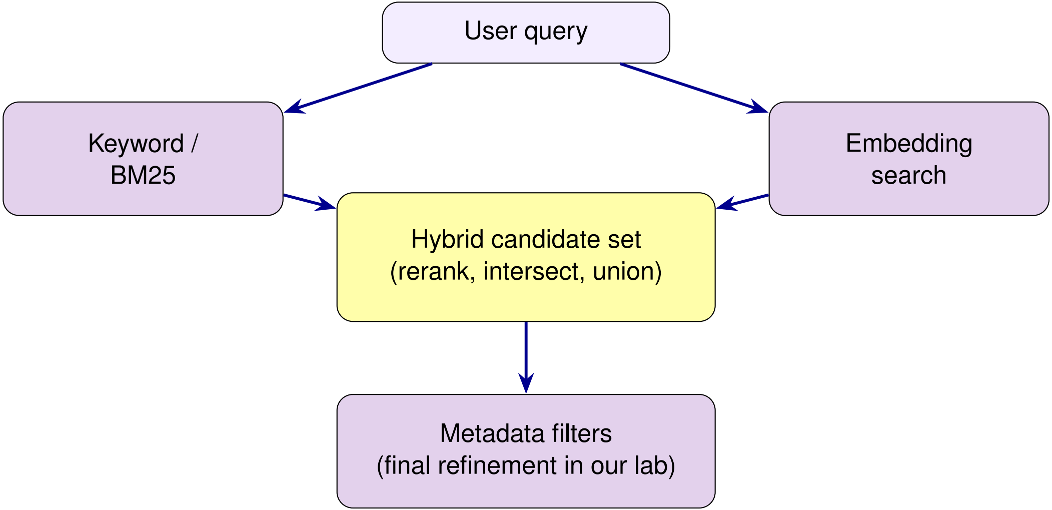
- **Definition**: BM25 scores a document d for a query term t roughly as:

$$\text{score}(t,d) \propto \text{IDF}(t)\,\frac{(k_1+1)\,\text{TF}(t,d)}{\text{TF}(t,d)+k_1\left(1-b+b\cdot\frac{\text{document length}}{\text{average document length}}\right)}.$$

  Best Matching 25 was named as the 25th variant in a series of scoring functions proposed by its creators.

- **Key intuitions**:
  - *Term frequency saturation*: repeating a word helps, but with diminishing returns.
  - *Document length normalization*: longer documents are penalized.
  - *IDF*: rare terms across the corpus contribute more to the score.
- **Tunable parameters**:
  - k_1 controls how fast the TF effect saturates,
  - b controls how strongly document length is normalized.
- In practice: a well-tuned BM25 is often the *best starting point* for keyword-based retrieval in RAG systems.

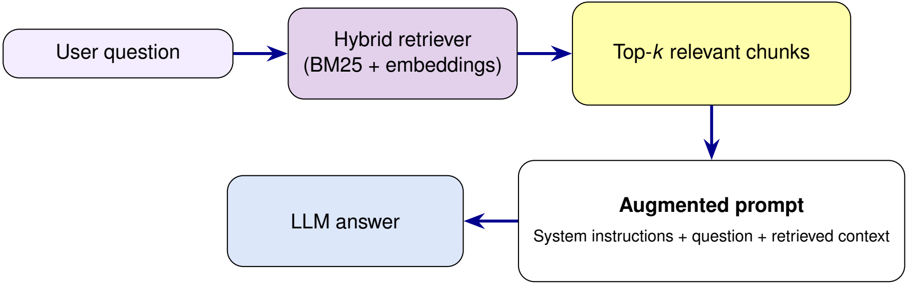
Combining Keywords, and Embeddings



Combining Keywords, and Embeddings

- Typical RAG uses a **hybrid retriever**:
  - run keyword search and embedding search,
  - combine candidates by intersection / union and rerank,
  - then apply metadata filters as a final refinement step (as in our lab).
- Choice of combination controls the trade-off between recall and precision.

From Retrieval to Augmented Prompt



- RAG is not a new model; it is a *prompting strategy* powered by a retriever.
- Quality hinges on:
  - what you retrieve (indexing, chunking, retriever),
  - how you format it in the prompt (templates, instructions, citations).

Prompt Template for Q&A

- Typical template (simplified):

```
You are a helpful assistant for data science students in Marseille.
Use only the information in the CONTEXT to answer.
If the answer is not in the CONTEXT, say you don't know.

CONTEXT:
{{ retrieved_chunks }}

QUESTION:
{{ user_question }}
```

- The retrieved chunks become part of the prompt; they *augment* the original question.
- Clear instructions reduce hallucinations and encourage citing the provided context.

Context Size and Ranking

- Context window is limited: we cannot paste the entire knowledge base into every prompt.
- We must choose:
  - **Top-k**: how many chunks to include?
  - **Max tokens**: truncate text to stay within the limit.
  - **Ranking strategy**: score by similarity, recency, or custom signals (e.g., student vs. admin).
- Trade-off:
  - too few chunks $\Rightarrow$ model misses key information;
  - too many chunks $\Rightarrow$ context becomes noisy and harder to use.

Use Case: AMU / Marseille Data Science Assistant

- Build an assistant for:
  - the MSc Data Science programme in Marseille,
  - practical life in the city for students.
- Example questions:
  - “Where are the data science lectures in Marseille usually held?”
  - “How can I access the computer labs on Sundays?”
  - “What are good quiet places to study data science in Marseille?”
- We want answers grounded in:
  - official programme pages,
  - campus maps and schedules,
  - curated local tips in our own documents.

Step 1: Ingest & Chunk

- Collect source documents:
  - programme descriptions (PDFs, HTML),
  - campus information and opening hours,
  - local guides curated by teaching staff.
- Normalize text (encoding, language detection, basic cleaning).
- Chunk into passages:
  - target e.g. 200–500 tokens per chunk,
  - keep semantic coherence (sections, headings).
- Attach metadata:
  - `campus=Marseille`, `program=Data Science`, `source_url`, `section_title`, ...

Step 2: Index & Retrieve

- Compute embeddings for each chunk using a multilingual sentence model.
- Store vectors in a vector index (e.g., FAISS, ScaNN, or a hosted service).
- At query time:
  1. Parse the user question (language, intent).
  2. Run hybrid search (embeddings + keywords) over all candidate chunks.
  3. Apply metadata filters as a final refinement (e.g., `campus=Marseille`, `program=Data Science`).
  4. Return top-k chunks with scores after filtering.
- This “search” stage is independent from the LLM and can be tuned separately (retriever, metadata filters, and ranking strategy).

Step 3: Build the Augmented Prompt - Template in code (pseudo-Python

```
chunks = retrieve_hybrid(query) # BM25 + embeddings
chunks = [ # Metadata-based refinement applied after retrieval
    c for c in chunks
    if c["campus"] == "Marseille"
    and c["program"] == "Data Science"
]
context = "\n\n".join(chunk["text"] for chunk in chunks)
prompt = f"""
You are an assistant for data science students in Marseille. Answer the ques-
tion using only the CONTEXT below. If the answer is not in the CONTEXT, say you
don't know.
CONTEXT:
{context}
QUESTION:
{query}"""
```

Step 4: Answer, Evaluate, Iterate

- Send the augmented prompt to the chosen LLM (OpenAI, local model, etc.).
- Log:
  - user question,
  - retrieved chunks and scores,
  - final answer and model parameters.
- Evaluate on a small test set:
  - correctness (does the answer match ground truth?),
  - grounding (can we see the supporting chunks?),
  - robustness (different phrasings, languages, levels of detail).
- Iteratively tune:
  - retriever (chunking, embeddings, ranking),
  - prompt (instructions, formatting, citations).

Precision and Recall Refresher

- **Precision**: among the retrieved items, how many are relevant?

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}.$$

- **Recall**: among all relevant items in the corpus, how many did we retrieve?

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}.$$

- Example (retrieval for a RAG system):
  - there are 20 truly relevant chunks in the corpus,
  - your retriever returns 10 chunks, 8 of which are relevant.

  Then:

$$\text{Precision} = \frac{8}{10} = 0.8, \quad \text{Recall} = \frac{8}{20} = 0.4.$$

- For RAG, we usually prefer *high recall* at the retrieval stage, then let the LLM focus on precision.

RAG vs Fine-Tuning

- **Fine-tuning**:
  - adapts model weights,
  - good for new behaviors or styles,
  - expensive to update, may still hallucinate facts.
- **RAG**:
  - keeps base model fixed,
  - plugs in external, updatable knowledge via retrieval,
  - easier to iterate on (update data, retriever, or prompts).
- In practice:
  - start with RAG for most data-centric use cases,
  - consider fine-tuning on top of a good RAG pipeline if behavior still needs shaping.

Common Pitfalls

- **Hallucinated citations**: model invents sources or mixes documents.
- **Shallow retrieval**: top-k chunks are off-topic or redundant.
- **Stale data**: document store not updated as the real world evolves.
- **Leaky prompts**: model uses prior knowledge instead of the provided context.
- **Evaluation gap**: no clear test set or feedback loop.

- Good engineering practice:
  - log everything,
  - add human review for early deployments,
  - maintain small, curated evaluation sets.

How the Pieces Fit Together

- Tokens, embeddings, and transformers (Weeks 1–2) explain how LLMs represent and process text.
- Effective prompting (Week 3) gives us control over model behavior.
- Classification (Week 4) is one example of using embeddings and LLMs for structured decisions.
- **RAG (Week 5)** combines all of these:
  - embeddings for retrieval,
  - prompts for grounding and style,
  - evaluation for reliability.
- This is the main pattern behind many modern data science assistants.

Looking Ahead

- In the lab you will:
  - build a minimal RAG pipeline over a small Marseille / AMU corpus,
  - experiment with keyword vs embedding vs hybrid retrieval,
  - observe how prompt design affects grounding.
- In the hackathon you can reuse these ideas:
  - many impactful projects are just: “a good RAG system + a good UI”.