RAG

Retrieval-Augmented Generation

Einführung in RAG (Retrieval-Augmented Generation)

Retrieval-Augmented Generation, kurz RAG, kombiniert zwei Welten:

Informationsabruf (Retrieval) und

Textgenerierung (Generation).

Ein Large Language Model (LLM) wird dabei mit externem Wissen versorgt, um fundierte, aktuelle und kontextrelevante Antworten zu erzeugen.

Beispiel aus der Praxis – Mittelständisches Unternehmen

Eine Maschinenbaufirma möchte ein Chat-System, das auf interne Handbücher, Wartungsanleitungen und Support-Tickets zugreift.

Ein reines LLM weiß nichts über diese Dokumente.

Mit RAG können Embeddings dieser Texte in einer Vektordatenbank wie ChromaDB gespeichert werden.

Das Modell erhält bei jeder Anfrage die passenden Abschnitte und liefert präzise Antworten.

Warum ein Chatbot – und nicht nur Suchergebnisse?

Eine reine semantische Suche in der Vektordatenbank liefert zwar relevante Textabschnitte, aber keine zusammenhängende, verständliche Antwort.

Der Chatbot (LLM) übernimmt genau diesen Schritt: Er fasst die gefundenen Informationen zusammen, formuliert sie natürlichsprachlich und passt sie an den Kontext oder die Fragestellung an.

So entsteht ein dialogfähiges, erklärendes System statt einer bloßen Suchmaschine.

Grundidee von RAG

1. Texte (z. B. Dokumente, PDFs, Support-Daten) werden in Embeddings umgewandelt.

2. Diese Vektoren werden in einer Datenbank wie ChromaDB gespeichert.

3. Bei einer Nutzeranfrage werden die ähnlichsten Vektoren gesucht.

4. Das LLM erhält den Originaltext als Kontext und generiert darauf basierend eine wohlklingende Antwort.

Alternativen zu RAG

RAG ist nicht die einzige Möglichkeit, Sprachmodelle mit Wissen zu erweitern. Es gibt mehrere Alternativen, jeweils mit eigenen Stärken und Grenzen:

- Fine-Tuning (PEFT): das Modell wird mit neuen, domänenspezifischen Daten nachtrainiert. Liefert sehr präzise Ergebnisse, ist aber teuer, zeitaufwendig und schwer zu aktualisieren. Klassisches Finetuning trainiert das ganze Model.

- Prompt Engineering: Wissen wird direkt in den Prompt eingebettet (One-shot oder Few-shot-Beispiele). Einfach umzusetzen, aber limitiert durch Tokenlänge und Kontextverlust.

- Tool-Augmentation: das Modell nutzt externe Systeme oder APIs (z. B. Suchmaschinen, Datenbanken, Rechner, MCP). Flexibel, aber erfordert Integration und Sicherheitskonzepte.

- RAG kombiniert die Vorteile: kein Training nötig, aber Zugriff auf strukturiertes, aktuelles Wissen: ideal für Unternehmensanwendungen.

Architekturüberblick

Retriever: durchsucht die Vektordatenbank (z. B. ChromaDB) nach relevanten Inhalten.

Generator: das LLM, das mit diesen Inhalten angereichert antwortet.

Speicher: Vektordatenbank zur effizienten Suche.

Diese Kombination sorgt für faktenbasierte Antworten, ohne das Modell neu trainieren zu müssen.

Rolle von ChromaDB im RAG-Prozess

ChromaDB dient als Wissensspeicher. Dort liegen Embeddings, Metadaten und Textabschnitte.

Bei jeder Anfrage werden per semantischer Suche die relevantesten Einträge gefunden.

Die Ergebnisse (z. B. Textauszüge mit Quelle) werden dann dem LLM als kontextuelle Eingabe übergeben.

Vorteile von RAG

- Keine Notwendigkeit für teures Fine-Tuning
- Zugriff auf aktuelles und domänenspezifisches Wissen
- Erklärbare Ergebnisse dank Quellenangabe
- Kosteneffizient und skalierbar
- Kombinierbar mit OpenAI, Hugging Face, Mistral oder lokalen Modellen

Beispielhafte Pipeline mit ChromaDB

1. Dokumente sammeln (z. B. PDFs, E-Mails, Wissensartikel)
2. Embeddings berechnen (OpenAI, SentenceTransformers etc.)
3. In ChromaDB speichern (mit Metadaten)
4. Benutzeranfrage → Embedding → Ähnlichkeitssuche
5. Passende Texte an LLM übergeben → Antwort generieren

Herausforderungen und Best Practices

- Gute Chunking-Strategie (Texte sinnvoll segmentieren, z.B. Overlapping um Kontextverlust zu vermeiden)
- sinnvolle Metadaten für Filterung (z. B. Abteilung, Sprache, Quelle)
- Caching von häufigen Anfragen (zb. Redis via Django)
- Zugriffskontrolle für vertrauliche Daten, z.B. Collections für verschiedene Benutzerrollen ("support", "intern", "hr").
- Qualität der Embeddings bestimmt die Qualität der Antworten

Qualitätssicherung der Embeddings

Die Güte eines RAG-Systems hängt direkt von der Qualität der Embeddings ab.

Deshalb sollten regelmäßig Stichproben geprüft werden, um sicherzustellen, dass ähnliche Texte auch wirklich ähnliche Bedeutungen ergeben.

Good Practice:

Führe Beispielabfragen durch und überprüfe, ob die zurückgelieferten Dokumente thematisch passen. Nutze Metriken wie Kosinus-Ähnlichkeit und manuelle Bewertungen.

So erkennst du frühzeitig Modell- oder Datenprobleme und kannst Embedding-Modelle gezielt austauschen oder nachtrainieren.

Beispiel: Qualität der Embeddings prüfen

```
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer("all-MiniLM-L6-v2")

sentences = [
    "I've seen things you people wouldn't believe.",
    "Attack ships on fire off the shoulder of Orion.",
    "May the Force be with you."
]

embeddings = model.encode(sentences, convert_to_tensor=True)

similarity_1_2 = util.cos_sim(embeddings[0], embeddings[1])
similarity_1_3 = util.cos_sim(embeddings[0], embeddings[2])

print("Ähnlichkeit (1 & 2):", similarity_1_2.item())
print("Ähnlichkeit (1 & 3):", similarity_1_3.item())
```

Erwartung: Die ersten beiden Zitate aus *Blade Runner* sollten eine höhere semantische Ähnlichkeit aufweisen als das dritte (*Star Wars*).

Strategien für das Einlesen und Aktualisieren von Embeddings

Beim Aufbau und der Pflege einer Vektordatenbank (z. B. ChromaDB) ist eine klare Strategie für das Einlesen und Aktualisieren entscheidend. Neue oder geänderte Dokumente müssen regelmäßig eingebettet werden, ohne alles neu zu berechnen.

- Initial-Import: Alle vorhandenen Texte einmalig einbetten und speichern.
- Incremental Updates: Nur neue oder geänderte Dokumente erkennen (z. B. per Timestamp, Hash) und gezielt neu einbetten.
- Versionierte Collections: Neue Embedding-Versionen in separaten Collections anlegen (z. B. docs_v2) - erleichtert Rücksprung und Vergleich.
- Batch-Verarbeitung: Große Datenmengen in Batches (z. B. 100–500 Texte) einbetten, um RAM zu schonen.
- Monitoring: Regelmäßig prüfen, ob neue Datenquellen oder Formate hinzugekommen sind, die noch nicht eingebettet wurden (zb. via Dashboard oder logs)