Retrieval-Augmented Generation for Policy Question Answering

Tuba Seker

2025-05-28

1. Introduction

Retrieval-Augmented Generation (RAG) combines the strengths of two powerful paradigms:

- **Retrieval**: Bringing in relevant external knowledge
- **Generation**: Producing language responses based on retrieved content

This project demonstrates how to build a functional RAG system that answers policy-related questions using a dataset of 200 policy statements.

2. Theoretical Foundation

2.1 Sentence Embeddings

Sentence embeddings are vector representations of text that capture semantic meaning.

Why Embedding?

In NLP, embedding refers to the mathematical representation of language. Whether it's a sentence, paragraph, or word, the goal is to produce a vector (a numerical array) that represents meaning.

For example:

"Carbon tax was introduced to reduce emissions."

"To fight climate change, a carbon levy was applied."

These two sentences use different words but have similar meanings. A good embedding model can capture this similarity numerically.

An output of embedding is seen as for instance: [0.123, -0.554, 0.768,…,-0.023]

This vector encodes multiple semantic dimensions of the sentence, including:

- **Topic** (e.g., environment, taxation, technology…)
- **Tone** (e.g., supportive, critical…)
- **Focus** (e.g., solution, cause, consequence…)

Therefore, by comparing these vectors using similarity measures such as **cosine similarity**, we can identify the most semantically similar sentences.

Using `all-MiniLM-L6-v2` from `sentence-transformers`, we generate 384-dimensional vectors.

What's Our Goal in This Lesson?

- We will vectorize 200 sentences
  (using **embeddings** generated by a **transformer-based model**)
- We will store these vectors in **FAISS**
  (FAISS = Facebook AI Similarity Search, a high-speed vector search library)
- When a user query is received:
  - We'll compute its embedding
  - Then retrieve the **top 5–10 most similar sentences** using FAISS

Tools We Will Use

Tool	Purpose
`sentence-transformers`	Generate embeddings from sentences (using BERT-based models)
`FAISS`	Index and search these embeddings efficiently
`pandas`	Read, store, and visualize the data
`scikit-learn` (optional)	May be used to test and evaluate similarity or clustering metrics

2.2 Semantic Search with FAISS

(Facebook AI Similarity Search)

Why FAISS?

The Problem

We have a set of 200 sentence embeddings, each represented as a **384-dimensional vector**.

When a user asks a question, we want to find:
"Which existing sentences are semantically most similar to this query?"

This is known as a **semantic similarity search** problem: finding the vectors that are *closest* to a given vector, **quickly**.

FAISS (Facebook AI Similarity Search) is one of the most powerful tools designed specifically for this task.

How Does FAISS Work?

- We provide FAISS with a dataset of vectors (our sentence embeddings).
- FAISS indexes this dataset — meaning it builds an internal structure that enables **fast approximate nearest-neighbor search**.
- Then, when we supply a **query embedding** (e.g., from a user question),
  FAISS retrieves the most similar vectors from the indexed dataset.

THEORY: Cosine Similarity vs. Euclidean Distance

In NLP applications, **cosine similarity** is typically preferred because:

- It measures the **angle** between vectors,
  regardless of their **magnitude** (length).
- This better captures **semantic similarity** between sentences.

However, FAISS by default uses **L2 distance** (Euclidean distance).
To reconcile this:

We **normalize** all embeddings (i.e., ensure their **L2 norm = 1**)
This makes **cosine similarity ≈ dot product**, which FAISS can efficiently compute.

1. What is Cosine Similarity?

It measures the **similarity between the directions** of two vectors. It is independent of their magnitudes.

$$\text{cosine similarity}(A, B) = \frac{A \cdot B}{\|A\| \|B\|}$$

- $A \cdot B \rightarrow$ Dot product (scalar product)
- $\|A\|, \|B\| \rightarrow$ L2 norms (magnitudes) of the vectors

This returns a value between **0 and 1**, where 1 means the vectors are pointing in exactly the same direction.

---

2. What is Dot Product?

It simply computes:

$$A \cdot B = \sum_{i=1}^{n} A_i B_i$$

That is, it multiplies each component of vector A with the corresponding component in vector B and sums the result.

This value reflects **both the direction and magnitude** of the vectors.

Critical Point: Why Do We Normalize?

FAISS's default search method is: **Dot Product (inner product)**

But we want to measure **semantic similarity** → which requires **cosine similarity**.

Problem:

Dot product also considers magnitude. A larger vector might get a higher similarity score — even if it's **not semantically similar**.

In other words, if we **don't normalize**:

FAISS, when searching using `dot product`, gives an **advantage to larger vectors**.

Example:

- Sentence A → [5.0, 5.0, 5.0] → has large magnitude
- Sentence B → [0.5, 0.5, 0.5] → smaller magnitude

→ Even if both point in the **same direction**, A will always have a **higher inner product**.

So:

The system "prefers" larger vectors — but these may not be semantically similar!

Solution:

If we normalize all vectors to have **unit L2 norm**, that is:

$$\|A\| = 1, \quad \|B\| = 1$$

Then:

$$\text{cosine similarity}(A, B) = A \cdot B$$

✔ Because when $\|A\| = \|B\| = 1$, the denominator of the cosine similarity formula becomes 1 and disappears.

Now, each vector has **unit length**. This ensures:

- Only **direction (i.e., meaning)** is considered
- FAISS returns more accurate similarity results

Mental Model: Think on a Circle

- If all embeddings are normalized, they lie on the surface of a **unit circle**.
- This means **only the direction is compared** — so we truly focus on "how semantically similar" they are.
- Since FAISS uses dot product, in this case it starts to behave like **cosine similarity**.

Note: Sources of Non-Semantic Magnitude in NLP

Magnitude of sentence embeddings can be affected by:

- Length of the sentence

- Very frequent or very rare words
- Overactivation of certain tokens by the model

These factors can influence **vector magnitude** but have **nothing to do with semantic similarity**.
➡ This is why **normalization is critical**.

Summary Table

Condition	Result
Not normalized	Dot product → influenced by both meaning and magnitude → ✖ flawed ranking
Normalized	Dot product = cosine similarity → ✅ correct, meaning-based ranking

Term	Meaning	Effect in NLP
Vector magnitude	Numeric distance, energy	Not meaningful; may cause bias in similarity ranking
Normalized vector	Length = 1, only direction matters	Allows accurate **semantic similarity** computation

---

Question: If We Normalize Vectors in FAISS to make similar to Cosine Similarity , why not to use directly the Cosine Similarity?

Yes, it's true that if we normalize vectors in FAISS, we effectively obtain **cosine similarity**.

So, what's the point of using **dot product**?

Answer: Performance, Speed, and Hardware Optimization

- **Cosine similarity** is not directly compatible with FAISS's C++/GPU optimizations.
- Formula for cosine similarity:

$$\text{cosine}(A, B) = \frac{A \cdot B}{\|A\|\|B\|}$$

→ Requires norm calculation for every search — **computationally expensive!**

FAISS Instead:

- Uses only the **inner product** (dot product)
- If vectors are **pre-normalized**, dot product behaves like cosine similarity:

$$\|A\| = \|B\| = 1 \Rightarrow \text{cosine}(A, B) = A \cdot B$$

This way, we get cosine similarity **in a much faster and FAISS-friendly** manner.

Comparison Table

Feature	Dot Product	Cosine Similarity
Affected by magnitude?	✅ Yes	✖ No
Measures direction?	✅ Yes, but affected by magnitude	✅ Only measures direction
What happens if normalized?	Becomes cosine similarity	It already is cosine similarity
Why preferred in FAISS?	For GPU speed and optimization reasons	Too slow to compute in real time

---

3: Answer Generation with LLMs

This is the **"Generation" step** of a Retrieval-Augmented Generation (RAG) system.

Our Goal:

1. Retrieve top 5–10 relevant sentences using FAISS
2. Insert them into a prompt and pass to the LLM
3. LLM generates a **contextual, meaningful answer**

What Does RAG Do?

RAG = Retrieval-Augmented Generation

LLMs don't know everything, especially not **recent**, **custom**, or **domain-specific** data.

RAG says:
> “I’ll find the relevant info — LLM, you focus on generating the answer.”

→ This is **“open-book NLP”**.

RAG System Flow

```
[Question] → [Semantic Search] → [Relevant Sentences] → [LLM Prompt] → [Answer]
```

Our LLM: `google/flan-t5-base`

Why this?

- Lightweight (Colab-friendly)
- Good zero-shot QA performance
- Transformer-based model

Code Steps

1. Get the user’s question

```
question = "What are some common government responses to unemployment?"
```

2. Search with FAISS

```
query_embedding = model.encode([question])
faiss.normalize_L2(query_embedding)
D, I = index.search(query_embedding, 5)
retrieved = [sentences[i] for i in I[0]]
```

3. Format as prompt

```
context = "\n".join(retrieved)
prompt = f"Answer the following question based on the given context:\n\nContext:\n{context}\n\nQuestion: {questio
n}\nAnswer:"
```

4. Feed prompt to LLM

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-base")
rag_model = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-base")

inputs = tokenizer(prompt, return_tensors="pt", truncation=True)
outputs = rag_model.generate(**inputs, max_new_tokens=100)
answer = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(answer)
```

What You’ll Learn

Step	What You Learn
Prompt design	How to craft prompts for LLMs
Truncation	Handle token length limits
Retrieval + Generation combo	Retrieval is not enough — guide the model with correct info

---

4. Implementation Steps

4.1 Dataset and Embedding

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')
sentences = df['Diverse Policy Statements'].tolist()
embeddings = model.encode(sentences)
```

We encode each policy sentence into a fixed-length semantic vector.

4.2 Indexing with FAISS

```
import faiss
import numpy as np
embeddings = np.array(embeddings).astype('float32')
faiss.normalize_L2(embeddings)
index = faiss.IndexFlatIP(embeddings.shape[1])
index.add(embeddings)
```

Vectors are normalized and indexed for fast cosine-similar search.

Concept	Description
`normalize_L2`	Normalizes vectors to unit length (L2 norm = 1) to enable cosine similarity
`IndexFlatIP`	Uses inner product (IP) in FAISS to simulate cosine similarity
`index.add(...)`	Adds the embeddings to the FAISS index (i.e., saves the database)

4.3 Query Retrieval

```
query = "What are the government's responses to unemployment?"
query_embedding = model.encode([query])
faiss.normalize_L2(query_embedding)
D, I = index.search(query_embedding, 10)
retrieved = [sentences[i] for i in I[0]]
```

The system retrieves the most relevant policy statements to the user's query.

5. Answer Generation with LLM

5.1 Prompt Design and Generation

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

context = "\n".join(retrieved[:10])
prompt = f"""
You are a policy assistant. Based on the following policy statements, summarize how governments have responded to
unemployment.

Policy Statements:
{context}

Question: What are the government's responses to unemployment?
Answer in 1-2 sentences:
"""

tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-base")
model = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-base")
inputs = tokenizer(prompt, return_tensors="pt", truncation=True)
outputs = model.generate(**inputs, max_new_tokens=100)
answer = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(answer)
```

This step uses a pretrained FLAN-T5 model to generate an answer based on retrieved content.

6. Insights and Practical Notes

6.1 Why not include all 200 sentences in the prompt?

1. Token Limit Issue (Context Window)

Models like `google/flan-t5-base` typically have a **maximum input limit of ~512–1024 tokens**.

If you use `top_k = 200`, the **total context length becomes too large** → it exceeds the model's input capacity.

Result: - May throw an error - Or it silently truncates the input and **ignores the first part of the context**

2. Focus Issue

Using **200 sentences** is also problematic in terms of cognitive load:

- The LLM cannot focus properly

- It may fail to identify what's truly important
- Risk of **hallucinations** increases

Result: Answers become generic, vague, and less reliable.

Recommended Strategy: Top-k + Filtering + Compact Prompt

To get **accurate and focused** answers, follow this workflow:

Step	Description
`top_k = 10 or 15`	Select only the most **semantically similar** 10–15 sentences
Filter duplicates	Remove repeated or redundant sentences
Build compact context	Join them using: `context = "\n".join(retrieved)`
Validate truncation	Check token count using tokenizer to avoid overflow

---

Pro Tip: For a More Advanced RAG Pipeline

To enhance answer quality further:

- ✅ **Summarize** retrieved results before sending to LLM
  → Give fewer but more informative chunks
- ✅ Use **multi-hop RAG**
  → First retrieve titles or sections, then drill down into details
- Use a **reranker model** to re-sort the top-k sentences based on contextual relevance

---

Summary Table

Scenario	Outcome
`top_k = 200` and send full context	❌ Token overflow, model loses focus, poor answer quality
`top_k = 10–15` with cleaned context	✅ Efficient, focused, and higher-quality responses

---

6.2 What if retrieved sentences are repetitive?

Filter duplicates using Python `set()` or apply clustering techniques before prompting.

6.3 How to improve the output?

- Use clearer prompt instructions
- Employ reranking models
- Switch to larger models like `flan-t5-large` or `GPT-4`

7. Conclusion

We implemented a basic yet effective RAG pipeline: - Generated sentence embeddings - Indexed and searched semantically relevant content - Prompted an LLM with real context

Future work may include: - Retrieval + summarization - Reranking with cross-encoders - Multilingual QA