Classroom Task

Build a Mini RAG-Based Product Advisor for an eCommerce Electronics Store

(With Strict JSON Output Requirement)

Objective

In this task, you will build a small **Retrieval-Augmented Generation (RAG) system** for an online electronics marketplace.

By completing this exercise, you will:

- Implement chunking strategies
- Generate embeddings for product data
- Perform semantic similarity retrieval
- Dynamically assemble context
- Generate grounded responses using an LLM
- Enforce structured JSON output

You will simulate how enterprise eCommerce AI systems provide structured, machine-readable responses.

Business Scenario

You are building an AI Shopping Assistant for an online electronics store.

A customer asks:

> “Which laptop is best for video editing and heavy multitasking?”

Your system must:

- Retrieve relevant laptop product data
- Inject retrieved content into a structured prompt
- Generate a response strictly from retrieved context
- Avoid hallucinating specifications
- Return output strictly in JSON format

You are not allowed to answer using model memory alone.

Dataset — Electronics Product Catalog

Use the following product catalog:

```
product_catalog = [
  """
  Product Name: Dell XPS 15
```

Category: Laptop

Processor: Intel i7

RAM: 16GB

Storage: 1TB SSD

Use Case: Suitable for video editing and professional workloads.

Display: 4K UHD

""",

"""

Product Name: MacBook Air M2

Category: Laptop

Processor: Apple M2

RAM: 8GB

Storage: 512GB SSD

Use Case: Lightweight productivity and daily office tasks.

Display: Retina Display

""",

"""

Product Name: ASUS ROG Strix

Category: Laptop

Processor: Intel i9

RAM: 32GB

Storage: 1TB SSD

Use Case: Gaming and high-performance multitasking.

Graphics: Dedicated RTX GPU

""",

"""

Product Name: HP Pavilion 14

Category: Laptop

Processor: Intel i5

RAM: 8GB

Storage: 512GB SSD

Use Case: General office work and browsing.

Display: Full HD

"""

]

Part 1 — Implement Chunking

You must implement at least two chunking strategies:

1. Fixed-size chunking

2. Logical or sentence-based chunking

Your responsibilities:

- Generate chunks from each product document
- Print the generated chunks
- Compare chunking quality

You are preparing product specifications for semantic indexing.

Part 2 — Build the Vector Store

You must:

- Generate embeddings for each chunk
- Store embeddings in memory
- Implement cosine similarity using NumPy

You are building the semantic layer of the eCommerce platform.

Part 3 — Implement Retrieval

Customer Query:

> “Which laptop is best for video editing and heavy multitasking?”

You must:

- Convert the query to embedding
- Compute similarity against all chunk embeddings
- Retrieve top 2 or top 3 relevant chunks
- Print retrieved chunks before generation

You are implementing semantic product search.

Part 4 — Context Assembly and Grounded Generation (JSON Required)

You must construct the prompt in the following format:

Answer strictly from the context below.

If information is missing, say it is not available.

Return the response strictly in JSON format.

Required JSON structure:

{

 "recommended_product": "Product Name",

 "justification": "Explanation strictly from context",

"key_specifications": [
  "Specification 1",
  "Specification 2",
  "Specification 3"
]
}

Context:

[Retrieved Chunk 1]

[Retrieved Chunk 2]

[Retrieved Chunk 3]

Question:

Which laptop is best for video editing and heavy multitasking?

You must:

- Use temperature between 0.2–0.3
- Print only valid JSON output
- Ensure no text appears outside JSON

Expected JSON Output Example

Your system should return something similar to:

{

"recommended_product": "Dell XPS 15",

"justification": "Suitable for video editing and professional workloads with 16GB RAM and 1TB SSD.",

"key_specifications": [

"Intel i7 processor",

"16GB RAM",

"1TB SSD"

]

}

OR

{

"recommended_product": "ASUS ROG Strix",

"justification": "High-performance multitasking with Intel i9 processor and 32GB RAM.",

"key_specifications": [

"Intel i9 processor",

"32GB RAM",

"1TB SSD"

]

}

The justification must come only from retrieved context.

No invented specifications are allowed.

Evaluation Criteria

You will be evaluated on:

- Correct chunking implementation
- Accurate embedding generation
- Proper cosine similarity calculation
- Correct top-k retrieval
- Proper context injection
- Strict JSON compliance
- No hallucinated specifications

Reflection Questions (Mandatory)

After completing the task, answer:

1. Which chunking strategy retrieved the most accurate product?
2. Why did that chunking method perform better?
3. What happens if top-k is reduced to 1?

4. How does structured JSON output help downstream systems?

5. Why is strict grounding critical in eCommerce AI systems?

What You Are Learning

Through this exercise, you are building:

- A semantic product indexing system
- A vector-based search engine
- A context-controlled generation pipeline
- A structured AI response system

You are implementing the core intelligence layer of a production-grade eCommerce RAG architecture, capable of powering AI shopping assistants and structured recommendation APIs.

Solution

```
import numpy as np
import json
import os
from dotenv import load_dotenv
from openai import OpenAI

# ------------------------------------------------
```

```
# Environment Setup
# --------------------------------------------------

load_dotenv()
client = OpenAI()

# --------------------------------------------------
# Product Catalog (Electronics)
# --------------------------------------------------

product_catalog = [
    """
    Product Name: Dell XPS 15
    Category: Laptop
    Processor: Intel i7
    RAM: 16GB
    Storage: 1TB SSD
    Use Case: Suitable for video editing and professional workloads.
    Display: 4K UHD
    """,

    """
    Product Name: MacBook Air M2
    Category: Laptop
    Processor: Apple M2
    RAM: 8GB
    Storage: 512GB SSD
    Use Case: Lightweight productivity and daily office tasks.
    Display: Retina Display
    """,

    """
    Product Name: ASUS ROG Strix
    Category: Laptop
```

```
    Processor: Intel i9
    RAM: 32GB
    Storage: 1TB SSD
    Use Case: Gaming and high-performance multitasking.
    Graphics: Dedicated RTX GPU
    """,

    """
    Product Name: HP Pavilion 14
    Category: Laptop
    Processor: Intel i5
    RAM: 8GB
    Storage: 512GB SSD
    Use Case: General office work and browsing.
    Display: Full HD
    """
]

# --------------------------------------------------
# Chunking Methods
# --------------------------------------------------

def fixed_size_chunking(text, chunk_size=150):
    return [text[i:i+chunk_size] for i in range(0, len(text), chunk_size)]

def sentence_based_chunking(text):
    lines = text.split("\n")
    return [line.strip() for line in lines if line.strip()]

def prepare_chunks(method="fixed"):
    all_chunks = []
    for doc in product_catalog:
        if method == "fixed":
            chunks = fixed_size_chunking(doc)
```

```
        elif method == "sentence":
            chunks = sentence_based_chunking(doc)
        else:
            raise ValueError("Invalid chunking method")
        all_chunks.extend(chunks)
    return all_chunks

# -------------------------------------------------
# Embedding Functions
# -------------------------------------------------

def generate_embedding(text):
    response = client.embeddings.create(
        model="text-embedding-3-small",
        input=text
    )
    return response.data[0].embedding

def build_vector_store(chunks):
    embeddings = [generate_embedding(chunk) for chunk in chunks]
    return embeddings

def cosine_similarity(a, b):
    a = np.array(a)
    b = np.array(b)
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

# -------------------------------------------------
# Retrieval
# -------------------------------------------------

def retrieve_top_k(query, chunks, embeddings, k=3):
    query_embedding = generate_embedding(query)
    similarities = [
```

```
        cosine_similarity(query_embedding, emb)
        for emb in embeddings
    ]
    top_k_indices = np.argsort(similarities)[-k:][::-1]
    return [chunks[i] for i in top_k_indices]

# -------------------------------------------------
# Context Assembly + JSON Grounded Generation
# -------------------------------------------------

def generate_json_response(query, method="sentence"):

    chunks = prepare_chunks(method)
    embeddings = build_vector_store(chunks)
    top_chunks = retrieve_top_k(query, chunks, embeddings, k=3)

    context_block = "\n".join(top_chunks)

    prompt = f"""
You are an AI eCommerce Product Advisor.

Answer strictly from the provided context.
If information is missing, say it is not available.
Return ONLY valid JSON.
Do not include any text outside JSON.

Required JSON format:

{{
  "recommended_product": "Product Name",
  "justification": "Explanation strictly from context",
  "key_specifications": [
     "Specification 1",
     "Specification 2",
```

```
      "Specification 3"
  ]
}}

Context:
{context_block}

Question:
{query}
"""

    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "user", "content": prompt}],
        temperature=0.2
    )

    return response.choices[0].message.content

# --------------------------------------------------
# Main Execution
# --------------------------------------------------

if __name__ == "__main__":

    user_query = "Which laptop is best for video editing and heavy multitasking?"

    result = generate_json_response(user_query, method="sentence")

    print("Final JSON Output:")
    print(result)
```