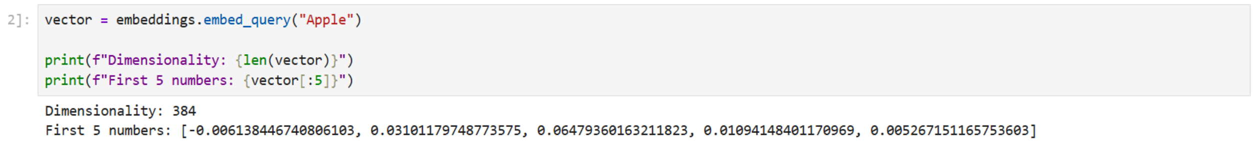
Gen ai :

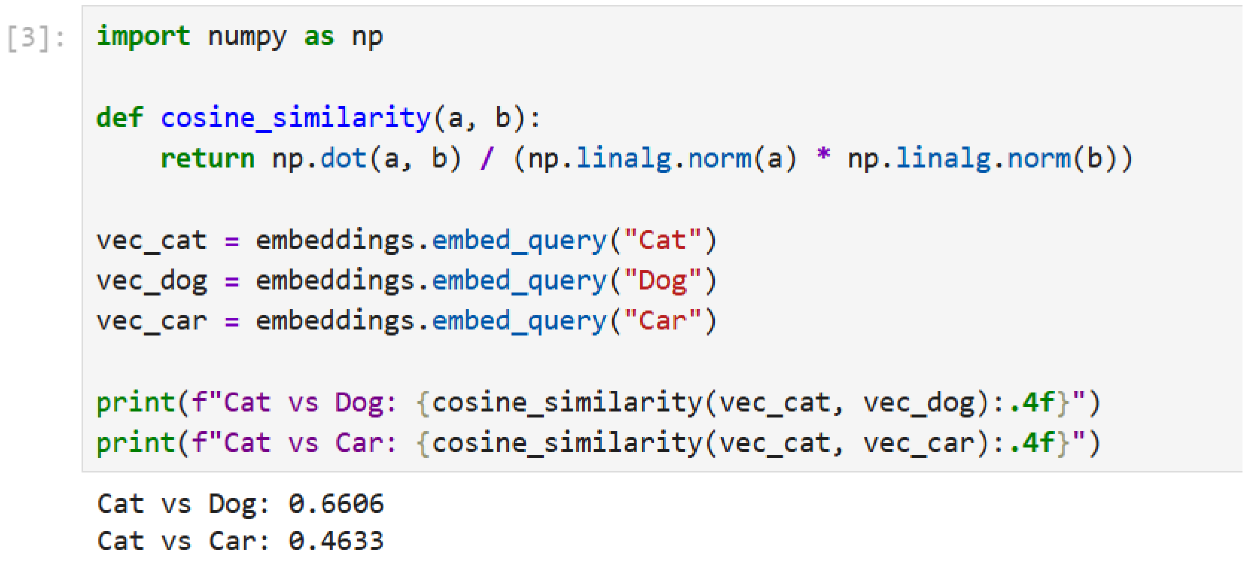
RAG :



```
2]: vector = embeddings.embed_query("Apple")

    print(f"Dimensionality: {len(vector)}")
    print(f"First 5 numbers: {vector[:5]}")
```

```
Dimensionality: 384
First 5 numbers: [-0.006138446740806103, 0.03101179748773575, 0.06479360163211823, 0.01094148401170969, 0.005267151165753603]
```



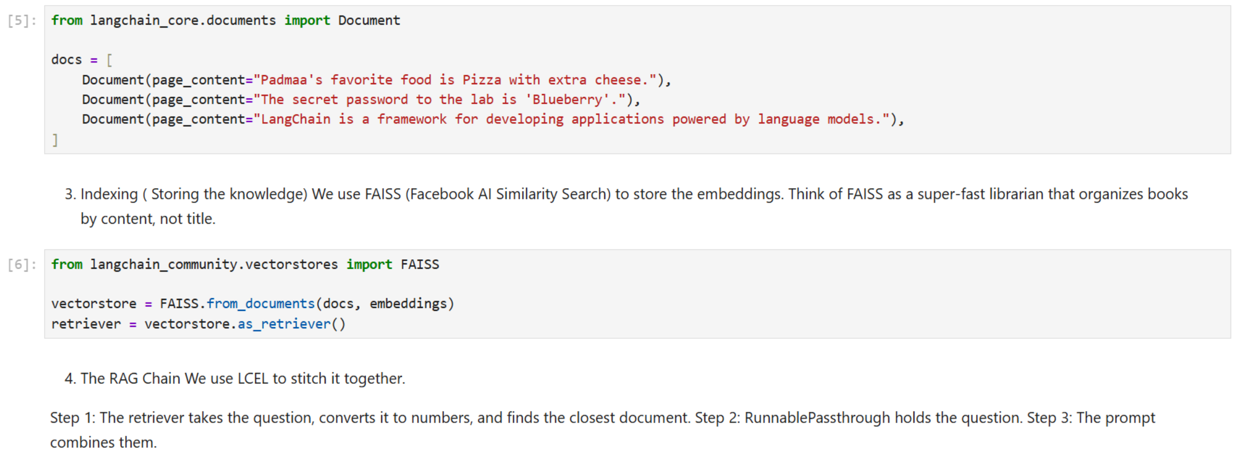
```
[3]: import numpy as np

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

     vec_cat = embeddings.embed_query("Cat")
     vec_dog = embeddings.embed_query("Dog")
     vec_car = embeddings.embed_query("Car")

     print(f"Cat vs Dog: {cosine_similarity(vec_cat, vec_dog):.4f}")
     print(f"Cat vs Car: {cosine_similarity(vec_cat, vec_car):.4f}")
```

```
Cat vs Dog: 0.6606
Cat vs Car: 0.4633
```



```
[5]: from langchain_core.documents import Document

     docs = [
         Document(page_content="Padmaa's favorite food is Pizza with extra cheese."),
         Document(page_content="The secret password to the lab is 'Blueberry'."),
         Document(page_content="LangChain is a framework for developing applications powered by language models."),
     ]
```

3. Indexing (Storing the knowledge) We use FAISS (Facebook AI Similarity Search) to store the embeddings. Think of FAISS as a super-fast librarian that organizes books by content, not title.

```
[6]: from langchain_community.vectorstores import FAISS

     vectorstore = FAISS.from_documents(docs, embeddings)
     retriever = vectorstore.as_retriever()
```

4. The RAG Chain We use LCEL to stitch it together.

Step 1: The retriever takes the question, converts it to numbers, and finds the closest document. Step 2: RunnablePassthrough holds the question. Step 3: The prompt combines them.

```
[7]: from langchain_core.prompts import ChatPromptTemplate
     from langchain_core.output_parsers import StrOutputParser
     from langchain_core.runnables import RunnablePassthrough

     template = """
     Answer based ONLY on the context below:
     {context}

     Question: {question}
     """
     prompt = ChatPromptTemplate.from_template(template)

     chain = (
         {"context": retriever, "question": RunnablePassthrough()}
         | prompt
         | llm
         | StrOutputParser()
     )

     result = chain.invoke("What is the secret password?")
     print(result)
```

```
The secret password to the lab is 'Blueberry'.
```

```
[11]: import faiss
      import numpy as np

      # Mock Data: 10,000 vectors of size 128
      d = 128
      nb = 10000
      xb = np.random.random((nb, d)).astype('float32')
```

```
[12]: vector_0 = index.reconstruct(0)
      print(vector_0[:10])
```

```
[0.7394615  0.8480146  0.26458973 0.7516232  0.3828365  0.9568646
 0.26083246 0.4199663  0.693522   0.01274962]
```

2. Flat Index (Brute Force) Concept: Check every single item.

Algo: IndexFlatL2 Pros: 100% Accuracy (Gold Standard). Cons: Slow (O(N)). Unusable at 1M+ vectors.

```python
index = faiss.IndexFlatL2(d)
index.add(xb)
print(f"Flat Index contains {index.ntotal} vectors")
xq = np.random.random((1, d)).astype('float32')
k = 5
D, I = index.search(xq, k)

print("Nearest vector indices:", I)
print("Distances:", D)
```

```
Flat Index contains 10000 vectors
Nearest vector indices: [[9371 6834 5573 7741 5525]]
Distances: [[13.302937 13.452788 13.783545 14.07968  14.10849 ]]
```

[13]:
```python
nlist = 100 # How many 'zip codes' (clusters) we want
quantizer = faiss.IndexFlatL2(d) # The calculator for distance
index_ivf = faiss.IndexIVFFlat(quantizer, d, nlist)

# We MUST train it first so it learns where the clusters are
index_ivf.train(xb)
index_ivf.add(xb)
```

[14]:
```python
print("Is index trained?", index_ivf.is_trained)
print("Total vectors in index:", index_ivf.ntotal)
print("Number of clusters (nlist):", index_ivf.nlist)
```

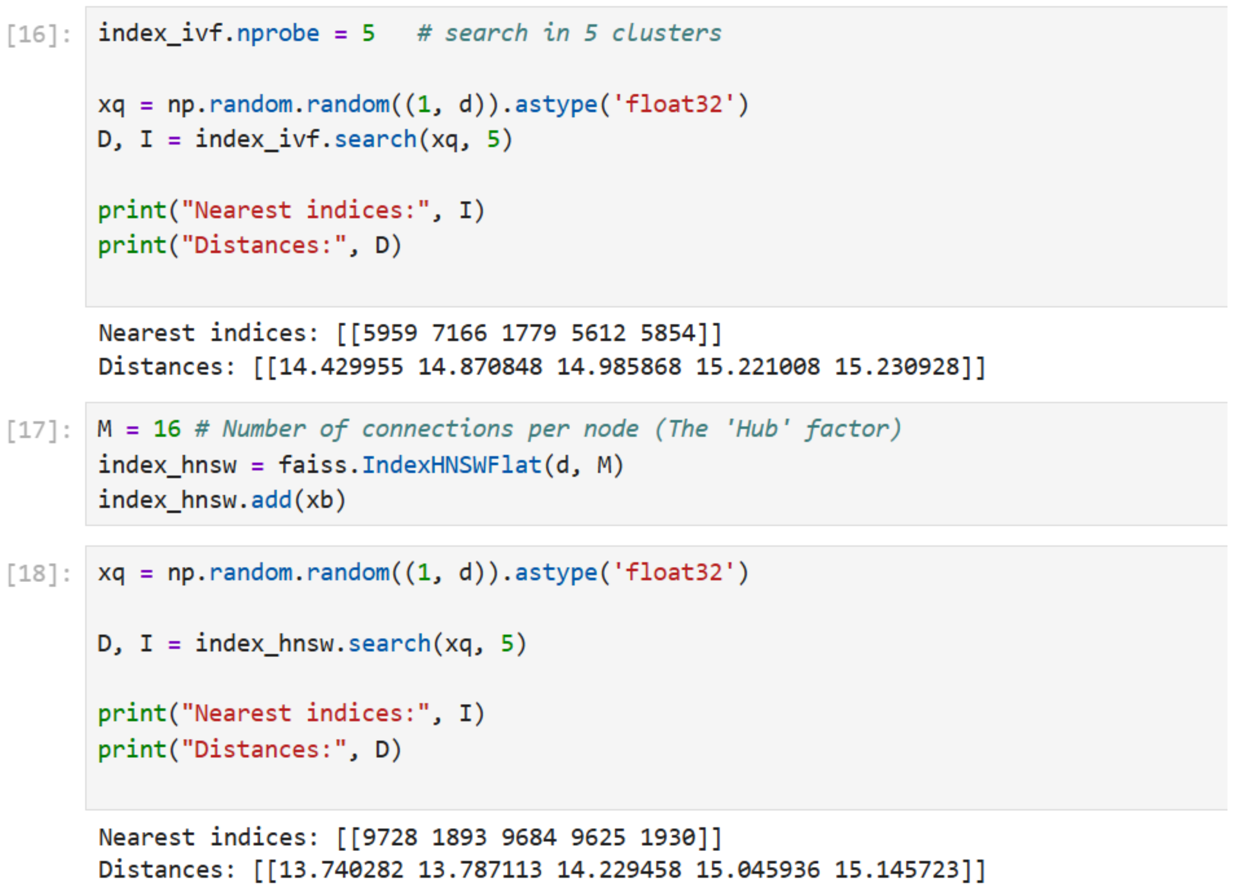
```
Is index trained? True
Total vectors in index: 10000
Number of clusters (nlist): 100
```

[15]:
```python
index_ivf.nprobe = 5   # search in 5 clusters

xq = np.random.random((1, d)).astype('float32')
D, I = index_ivf.search(xq, 5)

print("Nearest indices:", I)
print("Distances:", D)
```

```
Nearest indices: [[1568 2913 2344  692 2590]]
Distances: [[12.611797 12.6929   13.726816 14.824322 15.027815]]
```



[16]:
```python
index_ivf.nprobe = 5   # search in 5 clusters

xq = np.random.random((1, d)).astype('float32')
D, I = index_ivf.search(xq, 5)

print("Nearest indices:", I)
print("Distances:", D)
```

```
Nearest indices: [[5959 7166 1779 5612 5854]]
Distances: [[14.429955 14.870848 14.985868 15.221008 15.230928]]
```

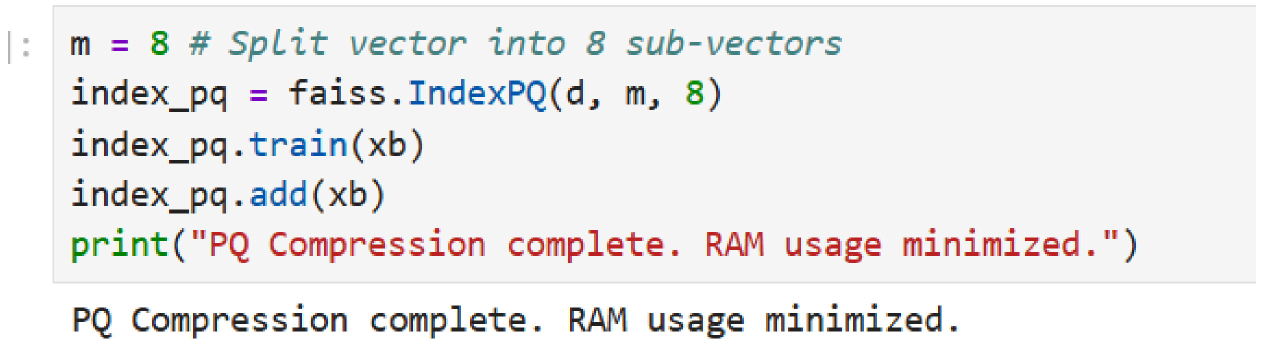
[17]:
```python
M = 16 # Number of connections per node (The 'Hub' factor)
index_hnsw = faiss.IndexHNSWFlat(d, M)
index_hnsw.add(xb)
```

[18]:
```python
xq = np.random.random((1, d)).astype('float32')

D, I = index_hnsw.search(xq, 5)

print("Nearest indices:", I)
print("Distances:", D)
```

```
Nearest indices: [[9728 1893 9684 9625 1930]]
Distances: [[13.740282 13.787113 14.229458 15.045936 15.145723]]
```



[ ]:
```python
m = 8 # Split vector into 8 sub-vectors
index_pq = faiss.IndexPQ(d, m, 8)
index_pq.train(xb)
index_pq.add(xb)
print("PQ Compression complete. RAM usage minimized.")
```

```
PQ Compression complete. RAM usage minimized.
```