Source: https://www.pinecone.io/learn/openai-gen-qa/

In [ ]:
```
%pip install -qU openai pinecone datasets cohere tiktoken --upgrade
```

In [ ]:
```
# Get the openai secret key
import getpass

OPENAI_API_KEY = getpass.getpass('Please enter your openai key: ')
```

In [ ]:
```
from openai import OpenAI

# Get API key from top-right dropdown on OpenAI website
client = OpenAI(api_key=OPENAI_API_KEY)
MODEL = "gpt-4.1-mini"
```

In [ ]:
```
query = "who was the 12th person on the moon and when did they land?"

# Now query WITHOUT context
res = client.chat.completions.create(
    model=MODEL,
    messages=[
        {
            "role": "user",
            "content": query,
        }
    ]
)

res.choices[0].message.content
```

In [ ]:
```
# First let's make it simpler to get answers
def complete(prompt):
    response = client.chat.completions.create(
        model=MODEL,
        messages=[
            {
                "role": "user",
                "content": prompt
            }
        ]
    )
    return response.choices[0].message.content

query = (
    "Which training method should I use for sentence transformers when " +
    "I only have pairs of related sentences?"
)

complete(query)
```

In [ ]:
```
# Use OpenAI's text embedding model
embed_model = "text-embedding-3-small"

res = client.embeddings.create(
    input=[
        "Sample document text goes here",
        "there will be several phrases in each batch"
    ],
    model=embed_model
)

# Vector embeddings for each document
res.data
```

In [ ]:
```
# We have created two vectors (one for each sentence input)
len(res.data)
```

In [ ]:
```
# We have created two 1536-dimensional vectors
len(res.data[0].embedding), len(res.data[1].embedding)
```

In [ ]:
```
# We can also get the vector for a single sentence
res.data[0].embedding
```

In [ ]:
```
from datasets import load_dataset

data = load_dataset('jamescalam/youtube-transcriptions', split='train')
data
```

In [ ]:
```
data[0]
```

In [ ]:
```
from tqdm.auto import tqdm

new_data = []

window = 20  # number of sentences to combine
stride = 4  # number of sentences to 'stride' over, used to create overlap

for i in tqdm(range(0, len(data), stride)):
    i_end = min(len(data)-1, i+window)
    if data[i]['title'] != data[i_end]['title']:
        # in this case we skip this entry as we have start/end of two videos
        continue
    text = ' '.join(data[i:i_end]['text'])
    # create the new merged dataset
    new_data.append({
        'start': data[i]['start'],
        'end': data[i_end]['end'],
        'title': data[i]['title'],
        'text': text,
        'id': data[i]['id'],
        'url': data[i]['url'],
```

```
        'published': data[i]['published'],
        'channel_id': data[i]['channel_id']
    })
```

In [ ]:
```
new_data[0]
```

In [ ]:
```
from pinecone import Pinecone, ServerlessSpec

import os

PINECONE_API_KEY = getpass.getpass("Please enter your pinecone key: ")

# Initialize connection (get API key at app.pinecone.io):
os.environ["PINECONE_API_KEY"] = PINECONE_API_KEY
```

In [ ]:
```
index_name = "employee-handbook"
pc = Pinecone()  # This reads the PINECONE_API_KEY env var

if index_name not in pc.list_indexes().names():
    pc.create_index(
        name=index_name,
        dimension=1536,  # Using the same vector dimensions as text-embedding-ada-002
        metric="cosine",
        spec=ServerlessSpec(
            cloud="aws",
            region="us-east-1"
        )
    )
```

In [ ]:
```
# Connect to Index:
index = pc.Index(name=index_name)
```

In [ ]:
```
# Describe the Index:
description = pc.describe_index(name=index_name)
print(description)
```

In [ ]:
```
from tqdm.auto import tqdm
import datetime
import time
from time import sleep

# Wait for the index to be ready
while not pc.describe_index(index_name).status['ready']:
    time.sleep(1)

batch_size = 100  # how many embeddings we create and insert at once

for i in tqdm(range(0, len(new_data), batch_size)):
    # find end of batch
    i_end = min(len(new_data), i+batch_size)
    meta_batch = new_data[i:i_end]
```

```
    # get texts to encode
    texts = [x['text'] for x in meta_batch]

    # create embeddings (try-except added to avoid RateLimitError)
    try:
        res = client.embeddings.create(input=texts, model=embed_model)
    except:
        done = False
        while not done:
            sleep(5)
            try:
                res = client.embeddings.create(input=texts, model=embed_model)
                done = True
            except:
                pass

    # prepare vectors for upsert
    vectors = []
    for data, embedding in zip(meta_batch, res.data):
        vectors.append({
            "id": data['id'],
            "values": embedding.embedding,
            "metadata": {
                'start': data['start'],
                'end': data['end'],
                'title': data['title'],
                'text': data['text'],
                'url': data['url'],
                'published': data['published'],
                'channel_id': data['channel_id']
            }
        })

    # Upsert to Pinecone
    index.upsert(vectors=vectors, namespace="ns1")
```

In [ ]:

```
res = client.embeddings.create(
    input=[query],
    model=embed_model
)

# retrieve from Pinecone
xq = res.data[0].embedding

# get relevant contexts (including the questions)
res = index.query(namespace="ns1", vector=xq, top_k=2, include_metadata=True)

# Print search results in a readable format
print("Search results:")
print("-" * 80)
for i, match in enumerate(res['matches'], 1):
    print(f"\nMatch {i} (Score: {match['score']:.3f})")
    print(f"ID: {match['id']}")
    print("\nMetadata:")
    print(f"  Title: {match['metadata']['title']}")
    print(f"  Time: {match['metadata']['start']:.1f}s - {match['metadata']['end']:.1f
    print(f"  URL: {match['metadata']['url']}")
    print(f"  Published: {match['metadata']['published']}")
```

```
    print("\nText:")
    print("  " + match['metadata']['text'])
```

In [ ]:

```
limit = 3750

def retrieve(query):
    res = client.embeddings.create(
        input=[query],
        model=embed_model
    )

    # retrieve from Pinecone
    xq = res.data[0].embedding

    # get relevant contexts
    res = index.query(namespace="ns1", vector=xq, top_k=3, include_metadata=True)
    contexts = [
        x['metadata']['text'] for x in res['matches']
    ]

    # build our prompt with the retrieved contexts included
    prompt_start = (
        "Answer the question based on the context below.\n\n"+
        "Context:\n"
    )
    prompt_end = (
        f"\n\nQuestion: {query}\nAnswer:"
    )

    # Initialize prompt with all contexts
    prompt = (
        prompt_start +
        "\n\n---\n\n".join(contexts) +
        prompt_end
    )

    # If total length exceeds limit, reduce contexts one by one
    for i in range(len(contexts)-1, 0, -1):
        if len("\n\n---\n\n".join(contexts[:i])) < limit:
            prompt = (
                prompt_start +
                "\n\n---\n\n".join(contexts[:i]) +
                prompt_end
            )
            break

    return prompt

# First we retrieve relevant items from Pinecone
query_with_contexts = retrieve(query)
print(query_with_contexts)
```

In [ ]:

```
# Then we complete the context-infused query
print(complete(query_with_contexts))
```