"""

Unit tests for embedding models.

This module tests the OpenAIEmbeddingModel implementation,

including error handling and retry logic.

"""

import pytest

from unittest.mock import Mock, patch, MagicMock

import numpy as np

from src.embeddings.openai\_embeddings import OpenAIEmbeddingModel

from src.utils.exceptions import EmbeddingError, APIKeyError

class TestOpenAIEmbeddingModel:

"""Test suite for OpenAIEmbeddingModel."""

@pytest.fixture

def mock\_openai\_client(self):

"""Create a mock OpenAI client."""

mock\_client = Mock()

mock\_response = Mock()

mock\_response.data = [

Mock(embedding=[0.1, 0.2, 0.3, 0.4, 0.5])

]

mock\_client.embeddings.create.return\_value = mock\_response

return mock\_client

@pytest.fixture

def embedding\_model(self, mock\_openai\_client):

"""Create an embedding model with mocked client."""

with patch('src.embeddings.openai\_embeddings.OpenAI', return\_value=mock\_openai\_client):

model = OpenAIEmbeddingModel(

api\_key="test-key",

model\_name="text-embedding-ada-002"

)

model.client = mock\_openai\_client

return model

def test\_initialization\_with\_valid\_key(self):

"""Test initialization with valid API key."""

with patch('src.embeddings.openai\_embeddings.OpenAI') as mock\_openai:

model = OpenAIEmbeddingModel(api\_key="valid-key")

assert model.model\_name == "text-embedding-ada-002"

mock\_openai.assert\_called\_once\_with(api\_key="valid-key")

def test\_initialization\_without\_key(self):

"""Test initialization without API key raises error."""

with pytest.raises(APIKeyError, match="OpenAI API key is required"):

OpenAIEmbeddingModel(api\_key=None)

def test\_embed\_documents\_success(self, embedding\_model, mock\_openai\_client):

"""Test successful document embedding."""

texts = ["Document 1", "Document 2", "Document 3"]

# Configure mock to return different embeddings for each text

mock\_openai\_client.embeddings.create.side\_effect = [

Mock(data=[Mock(embedding=[0.1 \* i, 0.2 \* i, 0.3 \* i])])

for i in range(1, 4)

]

embeddings = embedding\_model.embed\_documents(texts)

assert len(embeddings) == 3

assert all(isinstance(emb, list) for emb in embeddings)

assert mock\_openai\_client.embeddings.create.call\_count == 3

def test\_embed\_documents\_empty\_list(self, embedding\_model):

"""Test embedding empty document list."""

embeddings = embedding\_model.embed\_documents([])

assert embeddings == []

def test\_embed\_documents\_with\_retry(self, embedding\_model, mock\_openai\_client):

"""Test retry logic on transient failures."""

# First call fails, second succeeds

mock\_openai\_client.embeddings.create.side\_effect = [

Exception("Temporary failure"),

Mock(data=[Mock(embedding=[0.1, 0.2, 0.3])])

]

with patch('time.sleep'): # Mock sleep to speed up test

embeddings = embedding\_model.embed\_documents(["Test document"])

assert len(embeddings) == 1

assert mock\_openai\_client.embeddings.create.call\_count == 2

def test\_embed\_documents\_max\_retries\_exceeded(self, embedding\_model, mock\_openai\_client):

"""Test failure after max retries."""

mock\_openai\_client.embeddings.create.side\_effect = Exception("Persistent failure")

with patch('time.sleep'): # Mock sleep to speed up test

with pytest.raises(EmbeddingError, match="Failed to generate embeddings"):

embedding\_model.embed\_documents(["Test document"])

# Should retry 3 times (default max\_retries)

assert mock\_openai\_client.embeddings.create.call\_count == 3

def test\_embed\_query\_success(self, embedding\_model, mock\_openai\_client):

"""Test successful query embedding."""

query = "What is RAG?"

embedding = embedding\_model.embed\_query(query)

assert isinstance(embedding, list)

assert len(embedding) == 5 # Based on mock data

mock\_openai\_client.embeddings.create.assert\_called\_once\_with(

input=[query],

model="text-embedding-ada-002"

)

def test\_embed\_query\_empty\_string(self, embedding\_model):

"""Test embedding empty query."""

with pytest.raises(ValueError, match="Query text cannot be empty"):

embedding\_model.embed\_query("")

def test\_embed\_documents\_batch\_processing(self, embedding\_model, mock\_openai\_client):

"""Test batch processing of documents."""

# Create 150 documents to test batching (batch size is typically 100)

texts = [f"Document {i}" for i in range(150)]

mock\_openai\_client.embeddings.create.return\_value = Mock(

data=[Mock(embedding=[0.1, 0.2, 0.3]) for \_ in range(100)]

)

embeddings = embedding\_model.embed\_documents(texts)

assert len(embeddings) == 150

# Should be called twice (100 + 50)

assert mock\_openai\_client.embeddings.create.call\_count == 2

def test\_embed\_documents\_with\_special\_characters(self, embedding\_model, mock\_openai\_client):

"""Test embedding documents with special characters."""

texts = [

"Document with émojis 😀",

"Document with \nnewlines",

"Document with unicode: 你好"

]

mock\_openai\_client.embeddings.create.return\_value = Mock(

data=[Mock(embedding=[0.1, 0.2, 0.3])]

)

embeddings = embedding\_model.embed\_documents(texts)

assert len(embeddings) == 3

assert mock\_openai\_client.embeddings.create.call\_count == 3

def test\_embedding\_dimension\_consistency(self, embedding\_model, mock\_openai\_client):

"""Test that all embeddings have consistent dimensions."""

texts = ["Doc 1", "Doc 2", "Doc 3"]

# Mock different dimension embeddings

mock\_openai\_client.embeddings.create.side\_effect = [

Mock(data=[Mock(embedding=[0.1, 0.2, 0.3, 0.4, 0.5])]),

Mock(data=[Mock(embedding=[0.2, 0.3, 0.4, 0.5, 0.6])]),

Mock(data=[Mock(embedding=[0.3, 0.4, 0.5, 0.6, 0.7])])

]

embeddings = embedding\_model.embed\_documents(texts)

# Check all embeddings have same dimension

dimensions = [len(emb) for emb in embeddings]

assert len(set(dimensions)) == 1

assert dimensions[0] == 5

def test\_rate\_limit\_handling(self, embedding\_model, mock\_openai\_client):

"""Test handling of rate limit errors."""

# Simulate rate limit error

rate\_limit\_error = Exception("Rate limit exceeded")

rate\_limit\_error.response = Mock(status\_code=429)

mock\_openai\_client.embeddings.create.side\_effect = [

rate\_limit\_error,

Mock(data=[Mock(embedding=[0.1, 0.2, 0.3])])

]

with patch('time.sleep'): # Mock sleep

embeddings = embedding\_model.embed\_documents(["Test"])

assert len(embeddings) == 1

assert mock\_openai\_client.embeddings.create.call\_count == 2

@patch('src.embeddings.openai\_embeddings.logger')

def test\_logging(self, mock\_logger, embedding\_model, mock\_openai\_client):

"""Test that appropriate logs are generated."""

embedding\_model.embed\_documents(["Test document"])

# Check info logs

assert mock\_logger.info.called

info\_messages = [call[0][0] for call in mock\_logger.info.call\_args\_list]

assert any("Embedding" in msg for msg in info\_messages)

assert any("Successfully embedded" in msg for msg in info\_messages)

def test\_embed\_function\_interface(self, embedding\_model):

"""Test that model can be used as LangChain embedding function."""

# Test the interface matches LangChain expectations

assert hasattr(embedding\_model, 'embed\_documents')

assert hasattr(embedding\_model, 'embed\_query')

assert callable(embedding\_model.embed\_documents)

assert callable(embedding\_model.embed\_query)