"""

Unit tests for RAG evaluation functionality.

This module tests the RagasEvaluator implementation for

evaluating RAG pipeline performance.

"""

import pytest

from unittest.mock import Mock, patch, MagicMock

import pandas as pd

from datasets import Dataset

from src.evaluation.ragas\_evaluator import RagasEvaluator

from src.utils.exceptions import EvaluationError

from langchain.schema import Document

class TestRagasEvaluator:

"""Test suite for RagasEvaluator."""

@pytest.fixture

def mock\_qa\_chain(self):

"""Create a mock QA chain."""

mock\_chain = Mock()

mock\_chain.run.return\_value = {

"query": "Test question?",

"answer": "Test answer.",

"source\_documents": [

Document(page\_content="Source content 1", metadata={"page": 1}),

Document(page\_content="Source content 2", metadata={"page": 2})

]

}

return mock\_chain

@pytest.fixture

def evaluator(self, mock\_qa\_chain):

"""Create an evaluator instance."""

with patch.dict('os.environ', {'OPENAI\_API\_KEY': 'test-key'}):

return RagasEvaluator(

qa\_chain=mock\_qa\_chain,

api\_key="test-key"

)

@pytest.fixture

def sample\_evaluation\_data(self):

"""Create sample evaluation data."""

return [

{

"question": "What is RAG?",

"ground\_truth\_answer": "RAG stands for Retrieval-Augmented Generation.",

"relevant\_context": "RAG is a technique that combines retrieval and generation."

},

{

"question": "How does chunking work?",

"ground\_truth\_answer": "Chunking splits documents into smaller pieces.",

"relevant\_context": "Documents are divided into chunks for processing."

}

]

def test\_initialization(self, mock\_qa\_chain):

"""Test evaluator initialization."""

with patch.dict('os.environ', {'OPENAI\_API\_KEY': 'test-key'}):

evaluator = RagasEvaluator(qa\_chain=mock\_qa\_chain, api\_key="test-key")

assert evaluator.qa\_chain == mock\_qa\_chain

assert evaluator.api\_key == "test-key"

assert evaluator.results\_df is None

def test\_initialization\_no\_api\_key(self, mock\_qa\_chain):

"""Test initialization without API key."""

with patch.dict('os.environ', {}, clear=True):

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

RagasEvaluator(qa\_chain=mock\_qa\_chain, api\_key=None)

@patch('src.evaluation.ragas\_evaluator.evaluate')

@patch('src.evaluation.ragas\_evaluator.Dataset')

def test\_evaluate\_success(self, mock\_dataset\_class, mock\_ragas\_evaluate,

evaluator, sample\_evaluation\_data):

"""Test successful evaluation."""

# Setup mocks

mock\_dataset = Mock()

mock\_dataset\_class.from\_list.return\_value = mock\_dataset

mock\_result = Mock()

mock\_result.to\_pandas.return\_value = pd.DataFrame({

'question': ['Q1', 'Q2'],

'answer': ['A1', 'A2'],

'context\_precision': [0.8, 0.9],

'context\_recall': [0.7, 0.85],

'faithfulness': [0.9, 0.95],

'answer\_relevancy': [0.85, 0.9]

})

mock\_ragas\_evaluate.return\_value = mock\_result

# Run evaluation

results\_df = evaluator.evaluate(sample\_evaluation\_data)

# Verify dataset creation

mock\_dataset\_class.from\_list.assert\_called\_once()

dataset\_arg = mock\_dataset\_class.from\_list.call\_args[0][0]

assert len(dataset\_arg) == 2

# Verify evaluation was called with correct metrics

mock\_ragas\_evaluate.assert\_called\_once()

call\_kwargs = mock\_ragas\_evaluate.call\_args.kwargs

assert call\_kwargs['dataset'] == mock\_dataset

assert len(call\_kwargs['metrics']) == 4

# Verify results

assert isinstance(results\_df, pd.DataFrame)

assert len(results\_df) == 2

assert evaluator.results\_df is not None

def test\_evaluate\_empty\_data(self, evaluator):

"""Test evaluation with empty data."""

with pytest.raises(ValueError, match="Evaluation data cannot be empty"):

evaluator.evaluate([])

@patch('src.evaluation.ragas\_evaluator.evaluate')

@patch('src.evaluation.ragas\_evaluator.Dataset')

def test\_evaluate\_with\_qa\_chain\_error(self, mock\_dataset\_class, mock\_ragas\_evaluate,

evaluator, sample\_evaluation\_data):

"""Test evaluation when QA chain fails."""

# Make QA chain raise an error

evaluator.qa\_chain.run.side\_effect = Exception("QA chain error")

with pytest.raises(EvaluationError, match="Evaluation failed"):

evaluator.evaluate(sample\_evaluation\_data)

def test\_prepare\_ragas\_data(self, evaluator):

"""Test data preparation for Ragas."""

eval\_item = {

"question": "Test question?",

"ground\_truth\_answer": "Ground truth answer.",

"relevant\_context": "Relevant context."

}

# Mock QA chain response

evaluator.qa\_chain.run.return\_value = {

"answer": "Generated answer.",

"source\_documents": [

Document(page\_content="Retrieved content 1"),

Document(page\_content="Retrieved content 2")

]

}

# Prepare data

ragas\_item = evaluator.\_prepare\_ragas\_data(eval\_item)

# Verify structure

assert ragas\_item["question"] == "Test question?"

assert ragas\_item["answer"] == "Generated answer."

assert ragas\_item["reference"] == "Ground truth answer."

assert ragas\_item["ground\_truth\_answer"] == "Ground truth answer."

assert len(ragas\_item["retrieved\_contexts"]) == 2

assert ragas\_item["retrieved\_contexts"][0] == "Retrieved content 1"

def test\_get\_metrics\_before\_evaluation(self, evaluator):

"""Test getting metrics before evaluation."""

metrics = evaluator.get\_metrics()

assert metrics == {}

def test\_get\_metrics\_after\_evaluation(self, evaluator):

"""Test getting metrics after evaluation."""

# Set mock results

evaluator.results\_df = pd.DataFrame({

'context\_precision': [0.8, 0.9, 0.7],

'context\_recall': [0.7, 0.85, 0.75],

'faithfulness': [0.9, 0.95, 0.85],

'answer\_relevancy': [0.85, 0.9, 0.8]

})

metrics = evaluator.get\_metrics()

# Verify averages

assert 'context\_precision\_avg' in metrics

assert 'context\_recall\_avg' in metrics

assert 'faithfulness\_avg' in metrics

assert 'answer\_relevancy\_avg' in metrics

# Verify variances

assert 'context\_precision\_var' in metrics

assert 'context\_recall\_var' in metrics

assert 'faithfulness\_var' in metrics

assert 'answer\_relevancy\_var' in metrics

# Check calculation

assert metrics['context\_precision\_avg'] == pytest.approx(0.8, rel=1e-2)

assert metrics['faithfulness\_avg'] == pytest.approx(0.9, rel=1e-2)

def test\_get\_detailed\_results(self, evaluator):

"""Test getting detailed results."""

# No results yet

assert evaluator.get\_detailed\_results() is None

# Set results

evaluator.results\_df = pd.DataFrame({

'question': ['Q1'],

'answer': ['A1'],

'score': [0.9]

})

detailed = evaluator.get\_detailed\_results()

assert detailed.equals(evaluator.results\_df)

@patch('src.evaluation.ragas\_evaluator.logger')

def test\_logging(self, mock\_logger, evaluator, sample\_evaluation\_data):

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

with patch('src.evaluation.ragas\_evaluator.evaluate') as mock\_evaluate:

with patch('src.evaluation.ragas\_evaluator.Dataset'):

mock\_result = Mock()

mock\_result.to\_pandas.return\_value = pd.DataFrame({

'question': ['Q1'],

'score': [0.9]

})

mock\_evaluate.return\_value = mock\_result

evaluator.evaluate(sample\_evaluation\_data)

# Check logging

assert mock\_logger.info.called

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

assert any("Starting evaluation" in msg for msg in log\_messages)

assert any("Evaluation completed" in msg for msg in log\_messages)

def test\_custom\_metrics\_selection(self, mock\_qa\_chain):

"""Test evaluator with custom metric selection."""

with patch.dict('os.environ', {'OPENAI\_API\_KEY': 'test-key'}):

evaluator = RagasEvaluator(

qa\_chain=mock\_qa\_chain,

api\_key="test-key",

metrics=["faithfulness", "answer\_relevancy"] # Only 2 metrics

)

assert len(evaluator.metrics) == 2

assert any("faithfulness" in str(m) for m in evaluator.metrics)

assert any("answer\_relevancy" in str(m) for m in evaluator.metrics)

@patch('src.evaluation.ragas\_evaluator.evaluate')

@patch('src.evaluation.ragas\_evaluator.Dataset')

def test\_evaluation\_with\_missing\_fields(self, mock\_dataset\_class, mock\_ragas\_evaluate,

evaluator):

"""Test evaluation with incomplete data."""

incomplete\_data = [

{

"question": "Test question?",

# Missing ground\_truth\_answer and relevant\_context

}

]

# Should use empty strings for missing fields

mock\_dataset = Mock()

mock\_dataset\_class.from\_list.return\_value = mock\_dataset

mock\_result = Mock()

mock\_result.to\_pandas.return\_value = pd.DataFrame({'score': [0.5]})

mock\_ragas\_evaluate.return\_value = mock\_result

evaluator.evaluate(incomplete\_data)

# Verify dataset was created with defaults

dataset\_arg = mock\_dataset\_class.from\_list.call\_args[0][0]

assert dataset\_arg[0]["reference"] == "" # Default for missing ground truth

def test\_save\_results(self, evaluator):

"""Test saving evaluation results."""

evaluator.results\_df = pd.DataFrame({

'question': ['Q1'],

'score': [0.9]

})

with patch('pandas.DataFrame.to\_csv') as mock\_to\_csv:

evaluator.save\_results("results.csv")

mock\_to\_csv.assert\_called\_once\_with("results.csv", index=False)

def test\_save\_results\_no\_data(self, evaluator):

"""Test saving results when no evaluation has been run."""

with pytest.raises(ValueError, match="No evaluation results to save"):

evaluator.save\_results("results.csv")