Elderly\_Sepsis\_Trajectory\_Model\_Code

First, we create the main model and data processing code:

1. models/federated\_xgboost.py

```python

import numpy as np

from sklearn.metrics import roc\_auc\_score

import xgboost as xgb

from typing import List, Dict, Tuple

class FederatedXGBoost:

"""

Federated XGBoost implementation for multi-center sepsis prediction

"""

def \_\_init\_\_(self, num\_hospitals: int, params: Dict = None):

"""

Initialize federated learning model

Args:

num\_hospitals: Number of participating hospitals

params: XGBoost parameters

"""

self.num\_hospitals = num\_hospitals

self.models = []

self.params = params or {

'objective': 'binary:logistic',

'eval\_metric': 'auc',

'max\_depth': 6,

'eta': 0.3,

'min\_child\_weight': 1

}

def train\_local(self, X: np.ndarray, y: np.ndarray) -> xgb.Booster:

"""

Train model on local hospital data

Args:

X: Features matrix

y: Target values

Returns:

Trained XGBoost model

"""

dtrain = xgb.DMatrix(X, label=y)

model = xgb.train(self.params, dtrain, num\_boost\_round=100)

return model

def aggregate\_models(self, local\_models: List[xgb.Booster]) -> xgb.Booster:

"""

Aggregate local models using federated averaging

Args:

local\_models: List of locally trained models

Returns:

Aggregated global model

"""

# Model aggregation logic here

pass

def evaluate(self, X: np.ndarray, y: np.ndarray) -> float:

"""

Evaluate model performance

Args:

X: Test features

y: True labels

Returns:

AUROC score

"""

dtest = xgb.DMatrix(X)

y\_pred = self.global\_model.predict(dtest)

return roc\_auc\_score(y, y\_pred)

```

2. scripts/data\_processor.py

```python

import pandas as pd

import numpy as np

from typing import Tuple

from sklearn.preprocessing import StandardScaler

class SepsisDataProcessor:

"""

Process and prepare sepsis patient data for federated learning

"""

def \_\_init\_\_(self):

self.scaler = StandardScaler()

def load\_data(self, filepath: str) -> pd.DataFrame:

"""

Load patient data from file

Args:

filepath: Path to data file

Returns:

Processed DataFrame

"""

df = pd.read\_csv(filepath)

return self.\_preprocess(df)

def \_preprocess(self, df: pd.DataFrame) -> pd.DataFrame:

"""

Preprocess raw data

Args:

df: Raw DataFrame

Returns:

Preprocessed DataFrame

"""

# Remove missing values

df = df.dropna()

# Feature engineering

df['age\_group'] = pd.cut(df['age'], bins=[0, 65, 75, 85, 100])

# Normalize numerical features

num\_cols = ['heart\_rate', 'blood\_pressure', 'temperature', 'respiratory\_rate']

df[num\_cols] = self.scaler.fit\_transform(df[num\_cols])

return df

def split\_features\_target(self, df: pd.DataFrame) -> Tuple[np.ndarray, np.ndarray]:

"""

Split data into features and target

Args:

df: Preprocessed DataFrame

Returns:

Features array and target array

"""

X = df.drop(['patient\_id', 'mortality'], axis=1)

y = df['mortality']

return X.values, y.values

```

3. notebooks/model\_demonstration.ipynb

```python

{

"cells": [

{

"cell\_type": "markdown",

"metadata": {},

"source": [

"# Federated Learning for Elderly Sepsis Prediction\n",

"Demonstration of the federated XGBoost model for multi-center collaboration"

]

},

{

"cell\_type": "code",

"execution\_count": null,

"metadata": {},

"outputs": [],

"source": [

"import sys\n",

"sys.path.append('..')\n",

"from models.federated\_xgboost import FederatedXGBoost\n",

"from scripts.data\_processor import SepsisDataProcessor\n",

"\n",

"# Initialize processor and model\n",

"processor = SepsisDataProcessor()\n",

"fed\_model = FederatedXGBoost(num\_hospitals=6)\n",

"\n",

"# Example usage will be demonstrated here"

]

}

]

}

```

4. requirements.txt

```

numpy>=1.19.2

pandas>=1.2.0

scikit-learn>=0.24.0

xgboost>=1.3.0

jupyter>=1.0.0

matplotlib>=3.3.0

```

Continue improving the code and adding key features:

1. models/privacy\_engine.py

```python

import numpy as np

from typing import Dict, Any

class DifferentialPrivacy:

"""

Implement differential privacy mechanisms for model protection

"""

def \_\_init\_\_(self, epsilon: float = 1.0, delta: float = 1e-5):

self.epsilon = epsilon

self.delta = delta

def add\_noise(self, parameters: Dict[str, np.ndarray]) -> Dict[str, np.ndarray]:

"""

Add Gaussian noise to model parameters

Args:

parameters: Model parameters

Returns:

Noised parameters

"""

noised\_params = {}

for key, param in parameters.items():

sensitivity = self.\_compute\_sensitivity(param)

noise = np.random.normal(0, sensitivity \* self.epsilon, param.shape)

noised\_params[key] = param + noise

return noised\_params

def \_compute\_sensitivity(self, param: np.ndarray) -> float:

"""

Compute sensitivity for parameter

"""

return np.std(param) / 10.0

```

2. models/trajectory\_analyzer.py

```python

import numpy as np

from sklearn.cluster import KMeans

from typing import List, Tuple

class TrajectoryAnalyzer:

"""

Analyze and classify patient recovery trajectories

"""

def \_\_init\_\_(self, n\_trajectories: int = 6):

self.n\_trajectories = n\_trajectories

self.kmeans = KMeans(n\_clusters=n\_trajectories)

def identify\_trajectories(self,

temporal\_data: np.ndarray,

clinical\_features: List[str]) -> Tuple[np.ndarray, np.ndarray]:

"""

Identify distinct recovery trajectories

Args:

temporal\_data: Time series data of patient measurements

clinical\_features: Names of clinical features

Returns:

Trajectory labels and centroids

"""

# Shape: (n\_patients, n\_timepoints, n\_features)

trajectories = self.kmeans.fit\_predict(temporal\_data.reshape(len(temporal\_data), -1))

centroids = self.kmeans.cluster\_centers\_

return trajectories, centroids

def get\_trajectory\_characteristics(self,

trajectory\_id: int,

outcomes: np.ndarray) -> Dict[str, float]:

"""

Get clinical characteristics of a trajectory

Args:

trajectory\_id: ID of the trajectory

outcomes: Patient outcomes

Returns:

Dictionary of trajectory characteristics

"""

trajectory\_stats = {

'mortality\_rate': np.mean(outcomes[self.trajectories == trajectory\_id]),

'median\_los': np.median(self.los\_data[self.trajectories == trajectory\_id]),

'recovery\_speed': self.\_calculate\_recovery\_speed(trajectory\_id)

}

return trajectory\_stats

```

3. models/federated\_xgboost.py (Updated Version)

```python

import numpy as np

import xgboost as xgb

from typing import List, Dict, Tuple

from .privacy\_engine import DifferentialPrivacy

class FederatedXGBoost:

def \_\_init\_\_(self, num\_hospitals: int, params: Dict = None):

super().\_\_init\_\_(num\_hospitals, params)

self.privacy\_engine = DifferentialPrivacy()

self.global\_model = None

def aggregate\_models(self, local\_models: List[xgb.Booster]) -> xgb.Booster:

"""

Aggregate local models using secure federated averaging

Args:

local\_models: List of locally trained models

Returns:

Aggregated global model

"""

# Extract model parameters

model\_params = [self.\_extract\_params(model) for model in local\_models]

# Add differential privacy noise

noised\_params = [

self.privacy\_engine.add\_noise(params)

for params in model\_params

]

# Average parameters

avg\_params = {}

for key in noised\_params[0].keys():

avg\_params[key] = np.mean([p[key] for p in noised\_params], axis=0)

# Create new global model

self.global\_model = self.\_create\_model\_from\_params(avg\_params)

return self.global\_model

def \_extract\_params(self, model: xgb.Booster) -> Dict[str, np.ndarray]:

"""Extract parameters from XGBoost model"""

params = model.save\_raw()[4:]

return {'weights': np.array(params)}

def \_create\_model\_from\_params(self, params: Dict[str, np.ndarray]) -> xgb.Booster:

"""Create new XGBoost model from parameters"""

model = xgb.Booster(self.params)

model.load\_model(params['weights'])

return model

```

4. scripts/visualization.py

```python

import matplotlib.pyplot as plt

import seaborn as sns

from typing import List, Dict

class TrajectoryVisualizer:

"""

Visualize patient trajectories and model results

"""

def \_\_init\_\_(self, style: str = 'whitegrid'):

sns.set\_style(style)

def plot\_trajectories(self,

trajectories: np.ndarray,

centroids: np.ndarray,

feature\_names: List[str],

save\_path: str = None):

"""

Plot identified recovery trajectories

"""

fig, axes = plt.subplots(2, 3, figsize=(15, 10))

axes = axes.ravel()

for i in range(self.n\_trajectories):

axes[i].plot(centroids[i].reshape(-1, len(feature\_names)))

axes[i].set\_title(f'Trajectory {i+1}')

axes[i].set\_xlabel('Time (hours)')

axes[i].set\_ylabel('Standardized Value')

if save\_path:

plt.savefig(save\_path)

plt.show()

def plot\_performance\_metrics(self,

metrics: Dict[str, float],

save\_path: str = None):

"""

Plot model performance metrics

"""

plt.figure(figsize=(10, 6))

sns.barplot(x=list(metrics.keys()), y=list(metrics.values()))

plt.title('Model Performance Metrics')

plt.xticks(rotation=45)

if save\_path:

plt.savefig(save\_path)

plt.show()

```

### 5. tests/test\_federated\_model.py

```python

import unittest

import numpy as np

from models.federated\_xgboost import FederatedXGBoost

from models.trajectory\_analyzer import TrajectoryAnalyzer

class TestFederatedModel(unittest.TestCase):

def setUp(self):

self.model = FederatedXGBoost(num\_hospitals=3)

self.trajectory\_analyzer = TrajectoryAnalyzer()

def test\_model\_training(self):

# Generate synthetic data

X = np.random.randn(100, 10)

y = np.random.binomial(1, 0.5, 100)

# Test local training

local\_model = self.model.train\_local(X, y)

self.assertIsNotNone(local\_model)

def test\_trajectory\_analysis(self):

# Generate synthetic temporal data

temporal\_data = np.random.randn(50, 24, 5) # 50 patients, 24 timepoints, 5 features

trajectories, centroids = self.trajectory\_analyzer.identify\_trajectories(

temporal\_data,

['HR', 'BP', 'Temp', 'RR', 'SpO2']

)

self.assertEqual(len(np.unique(trajectories)), 6)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

```

Add more data preprocessing functionality

1. scripts/metrics.py

```python

import numpy as np

from sklearn.metrics import roc\_curve, precision\_recall\_curve, auc

from typing import Dict, Tuple

class ClinicalMetrics:

"""

Clinical performance metrics calculator

"""

def \_\_init\_\_(self):

self.metrics\_history = []

def calculate\_all\_metrics(self, y\_true: np.ndarray, y\_pred: np.ndarray) -> Dict[str, float]:

"""

Calculate comprehensive clinical metrics

Args:

y\_true: True labels

y\_pred: Predicted probabilities

Returns:

Dictionary containing all metrics

"""

metrics = {

'auroc': self.calculate\_auroc(y\_true, y\_pred),

'auprc': self.calculate\_auprc(y\_true, y\_pred),

'sensitivity': self.calculate\_sensitivity(y\_true, y\_pred),

'specificity': self.calculate\_specificity(y\_true, y\_pred),

'ppv': self.calculate\_ppv(y\_true, y\_pred),

'npv': self.calculate\_npv(y\_true, y\_pred)

}

self.metrics\_history.append(metrics)

return metrics

def calculate\_auroc(self, y\_true: np.ndarray, y\_pred: np.ndarray) -> float:

"""Calculate Area Under ROC Curve"""

fpr, tpr, \_ = roc\_curve(y\_true, y\_pred)

return auc(fpr, tpr)

def calculate\_optimal\_threshold(self, y\_true: np.ndarray, y\_pred: np.ndarray) -> float:

"""

Calculate optimal threshold using Youden's J statistic

"""

fpr, tpr, thresholds = roc\_curve(y\_true, y\_pred)

j\_scores = tpr - fpr

optimal\_idx = np.argmax(j\_scores)

return thresholds[optimal\_idx]

def get\_confidence\_intervals(self, metric\_values: np.ndarray,

confidence: float = 0.95) -> Tuple[float, float]:

"""Calculate confidence intervals for metrics"""

lower = np.percentile(metric\_values, (1 - confidence) / 2 \* 100)

upper = np.percentile(metric\_values, (1 + confidence) / 2 \* 100)

return lower, upper

```

2. scripts/data\_validator.py

```python

import pandas as pd

import numpy as np

from typing import List, Dict

class DataValidator:

"""

Validate and clean clinical data before processing

"""

def \_\_init\_\_(self):

self.validation\_results = {}

def validate\_dataset(self, df: pd.DataFrame,

required\_columns: List[str]) -> Dict[str, bool]:

"""

Validate dataset completeness and quality

Args:

df: Input DataFrame

required\_columns: List of required column names

Returns:

Dictionary of validation results

"""

results = {

'missing\_columns': self.\_check\_missing\_columns(df, required\_columns),

'data\_types': self.\_check\_data\_types(df),

'missing\_values': self.\_check\_missing\_values(df),

'outliers': self.\_detect\_outliers(df)

}

self.validation\_results = results

return results

def \_check\_missing\_columns(self, df: pd.DataFrame,

required\_columns: List[str]) -> List[str]:

"""Check for missing required columns"""

missing = [col for col in required\_columns if col not in df.columns]

return missing

def \_detect\_outliers(self, df: pd.DataFrame) -> Dict[str, np.ndarray]:

"""

Detect outliers using IQR method

"""

outliers = {}

numeric\_cols = df.select\_dtypes(include=[np.number]).columns

for col in numeric\_cols:

Q1 = df[col].quantile(0.25)

Q3 = df[col].quantile(0.75)

IQR = Q3 - Q1

outlier\_mask = (df[col] < (Q1 - 1.5 \* IQR)) | (df[col] > (Q3 + 1.5 \* IQR))

outliers[col] = outlier\_mask

return outliers

```

3. notebooks/model\_evaluation.ipynb

```python

{

"cells": [

{

"cell\_type": "markdown",

"metadata": {},

"source": [

"# Federated Learning Model Evaluation\n",

"Comprehensive evaluation of model performance and trajectory analysis"

]

},

{

"cell\_type": "code",

"execution\_count": null,

"metadata": {},

"outputs": [],

"source": [

"import sys\n",

"sys.path.append('..')\n",

"from models.federated\_xgboost import FederatedXGBoost\n",

"from models.trajectory\_analyzer import TrajectoryAnalyzer\n",

"from scripts.metrics import ClinicalMetrics\n",

"from scripts.visualization import TrajectoryVisualizer\n",

"\n",

"# Initialize components\n",

"fed\_model = FederatedXGBoost(num\_hospitals=6)\n",

"metrics = ClinicalMetrics()\n",

"visualizer = TrajectoryVisualizer()\n",

"\n",

"# Load and preprocess data\n",

"# [Data loading code here]\n",

"\n",

"# Model evaluation\n",

"results = metrics.calculate\_all\_metrics(y\_true, y\_pred)\n",

"visualizer.plot\_performance\_metrics(results)\n",

"\n",

"# Trajectory analysis\n",

"trajectories, centroids = trajectory\_analyzer.identify\_trajectories(temporal\_data, features)\n",

"visualizer.plot\_trajectories(trajectories, centroids, features)"

]

}

]

}

```

4. scripts/config.py

```python

"""

Configuration settings for the project

"""

# Model parameters

MODEL\_PARAMS = {

'objective': 'binary:logistic',

'eval\_metric': 'auc',

'max\_depth': 6,

'eta': 0.3,

'min\_child\_weight': 1,

'subsample': 0.8,

'colsample\_bytree': 0.8

}

# Privacy settings

PRIVACY\_PARAMS = {

'epsilon': 1.0,

'delta': 1e-5

}

# Clinical features

REQUIRED\_FEATURES = [

'age',

'heart\_rate',

'blood\_pressure',

'temperature',

'respiratory\_rate',

'spo2',

'gcs',

'lactate'

]

# Trajectory analysis parameters

TRAJECTORY\_PARAMS = {

'n\_trajectories': 6,

'time\_window': 24,

'min\_measurements': 3

}

```

5. Update README.md

```markdown

# Federated Learning for Elderly Sepsis Management

This repository contains the implementation of a federated learning system for analyzing and predicting sepsis outcomes in elderly patients across multiple hospitals.

## Features

- Privacy-preserving federated XGBoost implementation

- Patient trajectory analysis and clustering

- Comprehensive clinical metrics calculation

- Interactive visualization tools

- Data validation and preprocessing utilities

## Installation

```bash

git clone [repository-url]

cd [repository-name]

pip install -r requirements.txt

```

## Usage

```python

from models.federated\_xgboost import FederatedXGBoost

from scripts.data\_processor import SepsisDataProcessor

# Initialize model

model = FederatedXGBoost(num\_hospitals=6)

# Process data

processor = SepsisDataProcessor()

X, y = processor.prepare\_data('path\_to\_data')

# Train model

model.train(X, y)

```

## Documentation

See the `notebooks` directory for detailed examples and tutorials.

## License

This project is licensed under the GPL-3.0 License - see the LICENSE file for details.

```

Add model interpretability and more visualization features:

1. models/model\_interpreter.py

```python

import shap

import numpy as np

import pandas as pd

from typing import List, Dict, Tuple

import matplotlib.pyplot as plt

class ModelInterpreter:

"""

Interpret model predictions using SHAP values and feature importance

"""

def \_\_init\_\_(self, model, feature\_names: List[str]):

self.model = model

self.feature\_names = feature\_names

self.shap\_values = None

def calculate\_shap\_values(self, X: np.ndarray) -> np.ndarray:

"""

Calculate SHAP values for feature importance

Args:

X: Input features

Returns:

SHAP values for each prediction

"""

explainer = shap.TreeExplainer(self.model)

self.shap\_values = explainer.shap\_values(X)

return self.shap\_values

def plot\_feature\_importance(self,

save\_path: str = None) -> None:

"""

Plot global feature importance

"""

plt.figure(figsize=(10, 6))

shap.summary\_plot(self.shap\_values,

features=self.feature\_names,

show=False)

if save\_path:

plt.savefig(save\_path)

plt.show()

def get\_individual\_explanation(self,

patient\_id: int) -> Dict[str, float]:

"""

Get explanation for individual prediction

Args:

patient\_id: Index of patient to explain

Returns:

Dictionary of feature contributions

"""

contributions = {}

for idx, feature in enumerate(self.feature\_names):

contributions[feature] = self.shap\_values[patient\_id][idx]

return contributions

def analyze\_subgroup\_patterns(self,

X: np.ndarray,

subgroup\_mask: np.ndarray) -> Dict[str, float]:

"""

Analyze patterns in specific patient subgroups

Args:

X: Input features

subgroup\_mask: Boolean mask for subgroup selection

Returns:

Dictionary of subgroup-specific feature importance

"""

subgroup\_shap = self.shap\_values[subgroup\_mask]

mean\_impact = np.abs(subgroup\_shap).mean(axis=0)

return dict(zip(self.feature\_names, mean\_impact))

```

2. scripts/advanced\_visualization.py

```python

import plotly.graph\_objects as go

import plotly.express as px

from typing import List, Dict

import pandas as pd

import numpy as np

class AdvancedVisualizer:

"""

Advanced interactive visualizations for clinical trajectories and model results

"""

def \_\_init\_\_(self, theme: str = 'plotly\_white'):

self.theme = theme

def plot\_interactive\_trajectories(self,

temporal\_data: np.ndarray,

trajectory\_labels: np.ndarray,

feature\_names: List[str]) -> go.Figure:

"""

Create interactive trajectory visualization

Args:

temporal\_data: Time series data

trajectory\_labels: Cluster labels

feature\_names: Names of features

Returns:

Plotly figure object

"""

fig = go.Figure()

for trajectory\_id in np.unique(trajectory\_labels):

mask = trajectory\_labels == trajectory\_id

trajectory\_data = temporal\_data[mask].mean(axis=0)

fig.add\_trace(go.Scatter(

y=trajectory\_data,

x=range(len(trajectory\_data)),

name=f'Trajectory {trajectory\_id}',

mode='lines+markers'

))

fig.update\_layout(

title='Patient Trajectories Over Time',

xaxis\_title='Time (hours)',

yaxis\_title='Value',

hovermode='x unified'

)

return fig

def create\_risk\_heatmap(self,

risk\_scores: np.ndarray,

feature\_values: np.ndarray,

feature\_names: List[str]) -> go.Figure:

"""

Create interactive risk heatmap

"""

fig = go.Figure(data=go.Heatmap(

z=risk\_scores,

x=feature\_names,

y=feature\_values,

colorscale='RdYlBu\_r'

))

fig.update\_layout(

title='Risk Score Heatmap',

xaxis\_title='Features',

yaxis\_title='Feature Values'

)

return fig

def plot\_outcome\_distribution(self,

outcomes: pd.DataFrame,

trajectory\_ids: np.ndarray) -> go.Figure:

"""

Plot distribution of outcomes across trajectories

"""

fig = px.box(

outcomes,

x='trajectory',

y='value',

color='outcome\_type',

title='Outcome Distribution by Trajectory'

)

return fig

```

3. models/privacy\_mechanisms.py

```python

import numpy as np

from typing import Dict, Any

from abc import ABC, abstractmethod

class PrivacyMechanism(ABC):

"""Abstract base class for privacy mechanisms"""

@abstractmethod

def apply\_privacy(self, data: np.ndarray) -> np.ndarray:

pass

class LaplaceMechanism(PrivacyMechanism):

"""

Implement Laplace mechanism for differential privacy

"""

def \_\_init\_\_(self, epsilon: float):

self.epsilon = epsilon

def apply\_privacy(self, data: np.ndarray) -> np.ndarray:

"""Add Laplace noise to data"""

sensitivity = np.std(data) / 10.0

noise = np.random.laplace(0, sensitivity/self.epsilon, data.shape)

return data + noise

class GaussianMechanism(PrivacyMechanism):

"""

Implement Gaussian mechanism for differential privacy

"""

def \_\_init\_\_(self, epsilon: float, delta: float):

self.epsilon = epsilon

self.delta = delta

def apply\_privacy(self, data: np.ndarray) -> np.ndarray:

"""Add Gaussian noise to data"""

sensitivity = np.std(data) / 10.0

sigma = np.sqrt(2 \* np.log(1.25/self.delta)) \* sensitivity / self.epsilon

noise = np.random.normal(0, sigma, data.shape)

return data + noise

```

4. tests/test\_model\_interpretation.py

```python

import unittest

import numpy as np

from models.model\_interpreter import ModelInterpreter

from models.federated\_xgboost import FederatedXGBoost

class TestModelInterpretation(unittest.TestCase):

def setUp(self):

self.model = FederatedXGBoost(num\_hospitals=3)

self.feature\_names = ['age', 'hr', 'bp', 'temp', 'rr']

self.interpreter = ModelInterpreter(self.model, self.feature\_names)

def test\_shap\_values\_calculation(self):

# Generate synthetic data

X = np.random.randn(100, 5)

# Calculate SHAP values

shap\_values = self.interpreter.calculate\_shap\_values(X)

self.assertEqual(shap\_values.shape, X.shape)

def test\_individual\_explanation(self):

X = np.random.randn(100, 5)

self.interpreter.calculate\_shap\_values(X)

explanation = self.interpreter.get\_individual\_explanation(0)

self.assertEqual(len(explanation), len(self.feature\_names))

self.assertTrue(all(feat in explanation for feat in self.feature\_names))

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

```

Add model deployment and monitoring related code:

1. deployment/model\_server.py

```python

from fastapi import FastAPI, HTTPException

from pydantic import BaseModel

from typing import List, Dict

import numpy as np

import joblib

import logging

class PredictionInput(BaseModel):

"""Input data schema"""

features: List[float]

patient\_id: str

hospital\_id: str

class PredictionOutput(BaseModel):

"""Output data schema"""

prediction: float

risk\_level: str

confidence: float

trajectory\_id: int

app = FastAPI()

class ModelServer:

def \_\_init\_\_(self, model\_path: str):

self.model = joblib.load(model\_path)

self.logger = self.\_setup\_logger()

def \_setup\_logger(self):

logger = logging.getLogger("model\_server")

logger.setLevel(logging.INFO)

handler = logging.FileHandler("model\_server.log")

handler.setFormatter(logging.Formatter(

'%(asctime)s - %(name)s - %(levelname)s - %(message)s'

))

logger.addHandler(handler)

return logger

@app.post("/predict", response\_model=PredictionOutput)

async def predict(self, input\_data: PredictionInput):

try:

features = np.array(input\_data.features).reshape(1, -1)

prediction = self.model.predict\_proba(features)[0][1]

# Log prediction

self.logger.info(

f"Prediction made for patient {input\_data.patient\_id} "

f"from hospital {input\_data.hospital\_id}"

)

return PredictionOutput(

prediction=float(prediction),

risk\_level=self.\_get\_risk\_level(prediction),

confidence=self.\_calculate\_confidence(prediction),

trajectory\_id=self.\_assign\_trajectory(features)

)

except Exception as e:

self.logger.error(f"Error during prediction: {str(e)}")

raise HTTPException(status\_code=500, detail=str(e))

def \_get\_risk\_level(self, prediction: float) -> str:

if prediction < 0.3:

return "LOW"

elif prediction < 0.7:

return "MEDIUM"

return "HIGH"

def \_calculate\_confidence(self, prediction: float) -> float:

return abs(prediction - 0.5) \* 2

```

2. monitoring/model\_monitor.py

```python

import pandas as pd

import numpy as np

from typing import Dict, List

import time

from datetime import datetime

import requests

class ModelMonitor:

"""

Monitor model performance and data drift in production

"""

def \_\_init\_\_(self, reference\_data: pd.DataFrame):

self.reference\_data = reference\_data

self.current\_stats = self.\_calculate\_statistics(reference\_data)

self.alerts = []

def check\_data\_drift(self, new\_data: pd.DataFrame) -> Dict[str, bool]:

"""

Check for data drift in new predictions

"""

new\_stats = self.\_calculate\_statistics(new\_data)

drift\_detected = {}

for feature in self.current\_stats:

drift = self.\_detect\_drift(

self.current\_stats[feature],

new\_stats[feature]

)

drift\_detected[feature] = drift

if any(drift\_detected.values()):

self.\_send\_alert("Data drift detected", drift\_detected)

return drift\_detected

def monitor\_performance(self,

predictions: np.ndarray,

actual: np.ndarray) -> Dict[str, float]:

"""

Monitor model performance metrics

"""

metrics = {

'accuracy': np.mean(predictions == actual),

'false\_positives': np.sum((predictions == 1) & (actual == 0)),

'false\_negatives': np.sum((predictions == 0) & (actual == 1))

}

self.\_check\_performance\_degradation(metrics)

return metrics

def \_calculate\_statistics(self, data: pd.DataFrame) -> Dict[str, Dict]:

"""Calculate statistical properties of data"""

stats = {}

for column in data.columns:

stats[column] = {

'mean': data[column].mean(),

'std': data[column].std(),

'median': data[column].median(),

'q1': data[column].quantile(0.25),

'q3': data[column].quantile(0.75)

}

return stats

def \_detect\_drift(self,

reference\_stats: Dict,

current\_stats: Dict,

threshold: float = 0.1) -> bool:

"""Detect if drift has occurred"""

mean\_diff = abs(reference\_stats['mean'] - current\_stats['mean'])

std\_diff = abs(reference\_stats['std'] - current\_stats['std'])

return (mean\_diff > threshold \* reference\_stats['std'] or

std\_diff > threshold \* reference\_stats['std'])

def \_send\_alert(self, message: str, details: Dict):

"""Send alert to monitoring system"""

alert = {

'timestamp': datetime.now(),

'message': message,

'details': details

}

self.alerts.append(alert)

# Add actual alert sending mechanism here (e.g., email, Slack)

```

3. deployment/config.yaml

```yaml

model\_server:

host: "0.0.0.0"

port: 8000

model\_path: "models/saved/production\_model.joblib"

log\_level: "INFO"

monitoring:

drift\_threshold: 0.1

performance\_threshold: 0.05

alert\_endpoints:

email: "alerts@hospital.com"

slack\_webhook: "https://hooks.slack.com/services/xxx"

privacy:

epsilon: 1.0

delta: 1e-5

mechanism: "gaussian"

database:

host: "localhost"

port: 5432

database: "sepsis\_predictions"

user: "admin"

```

4. monitoring/alerts.py

```python

import smtplib

from email.mime.text import MIMEText

import requests

from typing import Dict, Any

import yaml

class AlertManager:

"""

Manage and send alerts for model monitoring

"""

def \_\_init\_\_(self, config\_path: str):

with open(config\_path) as f:

self.config = yaml.safe\_load(f)

def send\_alert(self,

alert\_type: str,

message: str,

details: Dict[str, Any]):

"""Send alert through configured channels"""

if alert\_type == "drift":

subject = "Data Drift Detected"

elif alert\_type == "performance":

subject = "Performance Degradation Alert"

else:

subject = "Model Monitoring Alert"

# Send email alert

self.\_send\_email(subject, message, details)

# Send Slack alert

self.\_send\_slack(subject, message, details)

def \_send\_email(self, subject: str, message: str, details: Dict):

"""Send email alert"""

email\_content = f"""

{message}

Details:

{details}

"""

msg = MIMEText(email\_content)

msg['Subject'] = subject

msg['From'] = self.config['monitoring']['alert\_endpoints']['email']

msg['To'] = self.config['monitoring']['alert\_endpoints']['email']

# Configure email sending here

def \_send\_slack(self, subject: str, message: str, details: Dict):

"""Send Slack alert"""

webhook\_url = self.config['monitoring']['alert\_endpoints']['slack\_webhook']

slack\_message = {

"text": f"\*{subject}\*\n{message}\n```{details}```"

}

requests.post(webhook\_url, json=slack\_message)

```

Let's add containerized deployment and load balancing related configurations:

### 1. docker/Dockerfile

```dockerfile

# Base image

FROM python:3.9-slim

# Set working directory

WORKDIR /app

# Copy requirements

COPY requirements.txt .

# Install dependencies

RUN pip install --no-cache-dir -r requirements.txt

# Copy application code

COPY . .

# Expose port

EXPOSE 8000

# Start application

CMD ["uvicorn", "deployment.model\_server:app", "--host", "0.0.0.0", "--port", "8000"]

```

### 2. docker-compose.yml

```yaml

version: '3.8'

services:

model-server:

build:

context: .

dockerfile: docker/Dockerfile

ports:

- "8000:8000"

volumes:

- ./models:/app/models

- ./logs:/app/logs

environment:

- MODEL\_PATH=/app/models/production\_model.joblib

- LOG\_LEVEL=INFO

restart: unless-stopped

depends\_on:

- redis

- postgres

model-monitor:

build:

context: .

dockerfile: docker/Dockerfile.monitor

volumes:

- ./logs:/app/logs

environment:

- MONITORING\_INTERVAL=60

depends\_on:

- model-server

- postgres

nginx:

image: nginx:latest

ports:

- "80:80"

volumes:

- ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro

depends\_on:

- model-server

redis:

image: redis:latest

ports:

- "6379:6379"

volumes:

- redis\_data:/data

postgres:

image: postgres:13

environment:

- POSTGRES\_USER=admin

- POSTGRES\_PASSWORD=secret

- POSTGRES\_DB=sepsis\_predictions

volumes:

- postgres\_data:/var/lib/postgresql/data

ports:

- "5432:5432"

volumes:

redis\_data:

postgres\_data:

```

### 3. nginx/nginx.conf

```nginx

events {

worker\_connections 1024;

}

http {

upstream model\_servers {

least\_conn; # Load balancing method

server model-server:8000;

server model-server:8001;

server model-server:8002;

}

server {

listen 80;

server\_name localhost;

location / {

proxy\_pass http://model\_servers;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

# Enable keepalive connections

proxy\_http\_version 1.1;

proxy\_set\_header Connection "";

# Timeouts

proxy\_connect\_timeout 60s;

proxy\_send\_timeout 60s;

proxy\_read\_timeout 60s;

}

# Health check endpoint

location /health {

access\_log off;

return 200 "healthy\n";

}

}

}

```

### 4. deployment/load\_balancer.py

```python

import aiohttp

import asyncio

from typing import List, Dict

import logging

class LoadBalancer:

"""

Manage load balancing across multiple model servers

"""

def \_\_init\_\_(self, server\_urls: List[str]):

self.server\_urls = server\_urls

self.active\_servers = {url: True for url in server\_urls}

self.current\_index = 0

self.logger = logging.getLogger("load\_balancer")

async def health\_check(self) -> None:

"""

Perform health checks on all servers

"""

while True:

for url in self.server\_urls:

try:

async with aiohttp.ClientSession() as session:

async with session.get(f"{url}/health") as response:

if response.status == 200:

self.active\_servers[url] = True

else:

self.active\_servers[url] = False

self.logger.warning(f"Server {url} health check failed")

except Exception as e:

self.active\_servers[url] = False

self.logger.error(f"Error checking server {url}: {str(e)}")

await asyncio.sleep(30) # Check every 30 seconds

async def get\_next\_server(self) -> str:

"""

Get next available server using round-robin

"""

attempts = 0

while attempts < len(self.server\_urls):

self.current\_index = (self.current\_index + 1) % len(self.server\_urls)

server = self.server\_urls[self.current\_index]

if self.active\_servers[server]:

return server

attempts += 1

raise Exception("No active servers available")

```

### 5. deployment/kubernetes/deployment.yaml

```yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: model-server

labels:

app: model-server

spec:

replicas: 3

selector:

matchLabels:

app: model-server

template:

metadata:

labels:

app: model-server

spec:

containers:

- name: model-server

image: model-server:latest

ports:

- containerPort: 8000

resources:

requests:

memory: "1Gi"

cpu: "500m"

limits:

memory: "2Gi"

cpu: "1000m"

livenessProbe:

httpGet:

path: /health

port: 8000

initialDelaySeconds: 30

periodSeconds: 10

readinessProbe:

httpGet:

path: /health

port: 8000

initialDelaySeconds: 15

periodSeconds: 5

```

### 6. deployment/kubernetes/service.yaml

```yaml

apiVersion: v1

kind: Service

metadata:

name: model-server-service

spec:

selector:

app: model-server

ports:

- protocol: TCP

port: 80

targetPort: 8000

type: LoadBalancer

```

### 7. scripts/deploy.sh

```bash

#!/bin/bash

# Build Docker images

docker-compose build

# Push images to registry

docker-compose push

# Deploy to Kubernetes

kubectl apply -f deployment/kubernetes/deployment.yaml

kubectl apply -f deployment/kubernetes/service.yaml

# Wait for deployment to complete

kubectl rollout status deployment/model-server

# Print service URL

echo "Model service available at:"

kubectl get service model-server-service -o jsonpath='{.status.loadBalancer.ingress[0].ip}'

```

Add CI/CD configuration and automated testing process:

1. .github/workflows/ci.yml

```yaml

name: Continuous Integration

on:

push:

branches: [ main, develop ]

pull\_request:

branches: [ main, develop ]

jobs:

test:

runs-on: ubuntu-latest

strategy:

matrix:

python-version: [3.8, 3.9]

steps:

- uses: actions/checkout@v2

- name: Set up Python ${{ matrix.python-version }}

uses: actions/setup-python@v2

with:

python-version: ${{ matrix.python-version }}

- name: Install dependencies

run: |

python -m pip install --upgrade pip

pip install -r requirements.txt

pip install -r requirements-dev.txt

- name: Run linting

run: |

flake8 .

black . --check

isort . --check

- name: Run tests with coverage

run: |

pytest --cov=./ --cov-report=xml

- name: Upload coverage to Codecov

uses: codecov/codecov-action@v2

with:

fail\_ci\_if\_error: true

security-scan:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v2

- name: Run security scan

uses: snyk/actions/python@master

env:

SNYK\_TOKEN: ${{ secrets.SNYK\_TOKEN }}

```

2. .github/workflows/cd.yml

```yaml

name: Continuous Deployment

on:

push:

branches: [ main ]

tags:

- 'v\*'

jobs:

build-and-push:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v2

- name: Set up Docker Buildx

uses: docker/setup-buildx-action@v1

- name: Login to DockerHub

uses: docker/login-action@v1

with:

username: ${{ secrets.DOCKERHUB\_USERNAME }}

password: ${{ secrets.DOCKERHUB\_TOKEN }}

- name: Build and push Docker images

uses: docker/build-push-action@v2

with:

context: .

push: true

tags: |

user/model-server:latest

user/model-server:${{ github.sha }}

- name: Deploy to Kubernetes

uses: steebchen/kubectl@master

env:

KUBE\_CONFIG\_DATA: ${{ secrets.KUBE\_CONFIG }}

with:

args: apply -f deployment/kubernetes/

```

3. tests/integration/test\_api.py

```python

import pytest

from fastapi.testclient import TestClient

from deployment.model\_server import app

import numpy as np

client = TestClient(app)

@pytest.fixture

def valid\_input\_data():

return {

"features": [0.5, 0.3, 0.7, 0.2, 0.1],

"patient\_id": "TEST001",

"hospital\_id": "HOSP001"

}

def test\_prediction\_endpoint(valid\_input\_data):

response = client.post("/predict", json=valid\_input\_data)

assert response.status\_code == 200

assert "prediction" in response.json()

assert "risk\_level" in response.json()

def test\_invalid\_input():

invalid\_data = {

"features": [],

"patient\_id": "TEST001"

}

response = client.post("/predict", json=invalid\_data)

assert response.status\_code == 422

def test\_health\_endpoint():

response = client.get("/health")

assert response.status\_code == 200

assert response.json()["status"] == "healthy"

```

4. tests/load/locustfile.py

```python

from locust import HttpUser, task, between

import random

import numpy as np

class ModelUser(HttpUser):

wait\_time = between(1, 3)

@task(1)

def predict(self):

features = np.random.random(5).tolist()

payload = {

"features": features,

"patient\_id": f"TEST{random.randint(1, 1000):03d}",

"hospital\_id": f"HOSP{random.randint(1, 10):02d}"

}

with self.client.post("/predict", json=payload, catch\_response=True) as response:

if response.status\_code == 200:

response.success()

else:

response.failure(f"Failed with status code: {response.status\_code}")

@task(10)

def health\_check(self):

self.client.get("/health")

```

5. scripts/quality\_check.sh

```bash

#!/bin/bash

set -e

echo "Running code quality checks..."

# Run formatting checks

echo "Running black..."

black . --check

echo "Running isort..."

isort . --check-only

# Run linting

echo "Running flake8..."

flake8 .

# Run security checks

echo "Running bandit..."

bandit -r . -c pyproject.toml

# Run type checking

echo "Running mypy..."

mypy .

# Run tests with coverage

echo "Running tests..."

pytest --cov=./ --cov-report=term-missing

echo "All quality checks passed!"

```

6. pyproject.toml

```toml

[tool.black]

line-length = 88

include = '\.pyi?$'

exclude = '''

/(

\.git

| \.hg

| \.mypy\_cache

| \.tox

| \.venv

| \_build

| buck-out

| build

| dist

)/

'''

[tool.isort]

profile = "black"

multi\_line\_output = 3

[tool.pytest.ini\_options]

testpaths = ["tests"]

python\_files = ["test\_\*.py"]

addopts = "-v --cov=./ --cov-report=term-missing"

[tool.coverage.run]

source = ["."]

omit = [

"tests/\*",

"setup.py",

]

[tool.mypy]

python\_version = "3.9"

warn\_return\_any = true

warn\_unused\_configs = true

disallow\_untyped\_defs = true

```

7. requirements-dev.txt

```

# Testing

pytest==7.3.1

pytest-cov==4.0.0

pytest-asyncio==0.20.3

locust==2.15.1

# Code quality

black==23.3.0

flake8==6.0.0

isort==5.12.0

mypy==1.3.0

bandit==1.7.5

# Type checking

types-requests==2.31.0.1

types-PyYAML==6.0.12.9

```

These updates add:

1. GitHub Actions CI/CD workflows

2. Integration tests

3. Load testing

4. Code quality checks

5. Type checking configuration

6. Development dependencies

Add monitoring dashboard and API documentation:

1. monitoring/grafana/dashboard.json

```json

{

"dashboard": {

"id": null,

"title": "Model Monitoring Dashboard",

"panels": [

{

"title": "Prediction Latency",

"type": "graph",

"datasource": "Prometheus",

"targets": [

{

"expr": "rate(model\_prediction\_latency\_seconds\_sum[5m]) / rate(model\_prediction\_latency\_seconds\_count[5m])",

"legendFormat": "Average Latency"

}

]

},

{

"title": "Requests Per Second",

"type": "graph",

"datasource": "Prometheus",

"targets": [

{

"expr": "sum(rate(http\_requests\_total[1m]))",

"legendFormat": "RPS"

}

]

},

{

"title": "Model Accuracy",

"type": "gauge",

"datasource": "Prometheus",

"targets": [

{

"expr": "model\_accuracy\_rolling\_window"

}

]

}

]

}

}

```

2. monitoring/prometheus/prometheus.yml

```yaml

global:

scrape\_interval: 15s

evaluation\_interval: 15s

scrape\_configs:

- job\_name: 'model-server'

static\_configs:

- targets: ['model-server:8000']

metrics\_path: '/metrics'

- job\_name: 'model-monitor'

static\_configs:

- targets: ['model-monitor:8001']

metrics\_path: '/metrics'

alerting:

alertmanagers:

- static\_configs:

- targets: ['alertmanager:9093']

rule\_files:

- 'alert\_rules.yml'

```

3. docs/api.md

```markdown

# Model API Documentation

## Endpoints

### POST /predict

Make a prediction using the model.

\*\*Request Body:\*\*

```json

{

"features": [float],

"patient\_id": string,

"hospital\_id": string

}

```

\*\*Response:\*\*

```json

{

"prediction": float,

"risk\_level": string,

"confidence": float,

"trajectory\_id": integer

}

```

\*\*Example:\*\*

```bash

curl -X POST "http://localhost:8000/predict" \

-H "Content-Type: application/json" \

-d '{"features": [0.5, 0.3, 0.7, 0.2, 0.1], "patient\_id": "P001", "hospital\_id": "H001"}'

```

### GET /health

Check service health status.

\*\*Response:\*\*

```json

{

"status": "healthy",

"version": "1.0.0"

}

```

```

4. docs/swagger.yaml

```yaml

openapi: 3.0.0

info:

title: Sepsis Prediction API

version: 1.0.0

description: API for making sepsis predictions

paths:

/predict:

post:

summary: Make a prediction

requestBody:

required: true

content:

application/json:

schema:

type: object

properties:

features:

type: array

items:

type: number

patient\_id:

type: string

hospital\_id:

type: string

responses:

'200':

description: Successful prediction

content:

application/json:

schema:

type: object

properties:

prediction:

type: number

risk\_level:

type: string

confidence:

type: number

trajectory\_id:

type: integer

/health:

get:

summary: Check service health

responses:

'200':

description: Service health status

content:

application/json:

schema:

type: object

properties:

status:

type: string

version:

type: string

```

5. monitoring/metrics.py

```python

from prometheus\_client import Counter, Histogram, Gauge

import time

from functools import wraps

from typing import Callable

# Request metrics

REQUEST\_COUNT = Counter(

'http\_requests\_total',

'Total HTTP requests',

['method', 'endpoint', 'status']

)

# Latency metrics

LATENCY = Histogram(

'model\_prediction\_latency\_seconds',

'Time spent processing prediction',

buckets=[0.1, 0.5, 1.0, 2.0, 5.0]

)

# Model metrics

PREDICTION\_GAUGE = Gauge(

'model\_prediction\_value',

'Latest prediction value'

)

ACCURACY\_GAUGE = Gauge(

'model\_accuracy\_rolling\_window',

'Model accuracy in recent predictions'

)

def track\_request\_metrics(endpoint: str):

def decorator(func: Callable):

@wraps(func)

async def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

try:

response = await func(\*args, \*\*kwargs)

REQUEST\_COUNT.labels(

method='POST',

endpoint=endpoint,

status=200

).inc()

return response

except Exception as e:

REQUEST\_COUNT.labels(

method='POST',

endpoint=endpoint,

status=500

).inc()

raise e

finally:

LATENCY.observe(time.time() - start\_time)

return wrapper

return decorator

```

6. monitoring/dashboard/app.py

```python

import streamlit as st

import pandas as pd

import plotly.express as px

import requests

def main():

st.title("Model Monitoring Dashboard")

# Metrics Overview

col1, col2, col3 = st.columns(3)

with col1:

st.metric(

label="Current Accuracy",

value=f"{get\_current\_accuracy():.2%}",

delta=f"{get\_accuracy\_change():.2%}"

)

with col2:

st.metric(

label="Requests/min",

value=get\_request\_rate(),

delta=get\_request\_rate\_change()

)

with col3:

st.metric(

label="Avg Latency",

value=f"{get\_avg\_latency():.2f}ms"

)

# Plots

st.subheader("Prediction Distribution")

fig = px.histogram(

get\_recent\_predictions(),

nbins=20,

title="Distribution of Predictions"

)

st.plotly\_chart(fig)

# Data Drift Analysis

st.subheader("Feature Drift Analysis")

drift\_data = get\_drift\_metrics()

st.table(drift\_data)

if \_\_name\_\_ == "\_\_main\_\_":

main()

```

These updates add:

1. Grafana monitoring dashboard configuration

2. Prometheus monitoring configuration

3. Detailed API documentation

4. Swagger API specification

5. Metrics collection code

6. Streamlit monitoring panel

Add performance optimization and user guide:

1. optimization/caching.py

```python

from functools import lru\_cache

import redis

from typing import Any, Dict, Optional

import pickle

import asyncio

from fastapi import BackgroundTasks

class PredictionCache:

def \_\_init\_\_(self, redis\_url: str):

self.redis\_client = redis.from\_url(redis\_url)

self.cache\_ttl = 3600 # 1 hour

async def get\_cached\_prediction(self, features\_key: str) -> Optional[Dict]:

"""Get prediction from cache"""

cached = self.redis\_client.get(features\_key)

if cached:

return pickle.loads(cached)

return None

async def cache\_prediction(self, features\_key: str, prediction: Dict):

"""Cache prediction result"""

self.redis\_client.setex(

features\_key,

self.cache\_ttl,

pickle.dumps(prediction)

)

@lru\_cache(maxsize=1000)

def compute\_feature\_statistics(features: tuple) -> Dict[str, float]:

"""Cache feature statistics computation"""

return {

"mean": sum(features) / len(features),

"variance": sum((x - (sum(features) / len(features))) \*\* 2 for x in features) / len(features)

}

```

2. optimization/batch\_processing.py

```python

import numpy as np

from typing import List, Dict

import asyncio

from concurrent.futures import ThreadPoolExecutor

class BatchProcessor:

def \_\_init\_\_(self, model, batch\_size: int = 32):

self.model = model

self.batch\_size = batch\_size

self.queue = asyncio.Queue()

self.executor = ThreadPoolExecutor(max\_workers=4)

async def add\_to\_batch(self, features: List[float]) -> asyncio.Future:

"""Add prediction request to batch queue"""

future = asyncio.Future()

await self.queue.put((features, future))

return future

async def process\_batches(self):

"""Process predictions in batches"""

while True:

batch = []

futures = []

# Collect batch

try:

while len(batch) < self.batch\_size:

features, future = await asyncio.wait\_for(

self.queue.get(),

timeout=0.1

)

batch.append(features)

futures.append(future)

except asyncio.TimeoutError:

if not batch:

continue

# Process batch

features\_array = np.array(batch)

predictions = await self.predict\_batch(features\_array)

# Set results

for future, prediction in zip(futures, predictions):

future.set\_result(prediction)

async def predict\_batch(self, features\_array: np.ndarray) -> List[Dict]:

"""Make predictions on a batch"""

loop = asyncio.get\_running\_loop()

return await loop.run\_in\_executor(

self.executor,

self.\_predict\_batch\_sync,

features\_array

)

def \_predict\_batch\_sync(self, features\_array: np.ndarray) -> List[Dict]:

"""Synchronous batch prediction"""

predictions = self.model.predict\_proba(features\_array)

return [

{

"prediction": float(pred[1]),

"confidence": float(max(pred))

}

for pred in predictions

]

```

3. docs/user\_guide.md

```markdown

# Sepsis Prediction Model User Guide

## Overview

This guide provides instructions for deploying and using the sepsis prediction model in production.

## Quick Start

### 1. Installation

```bash

# Clone repository

git clone https://github.com/your-org/sepsis-prediction

# Install dependencies

pip install -r requirements.txt

# Setup environment

cp .env.example .env

```

### 2. Configuration

Edit `.env` file with your settings:

```env

MODEL\_PATH=/path/to/model

REDIS\_URL=redis://localhost:6379

LOG\_LEVEL=INFO

```

### 3. Running the Service

```bash

# Start with Docker

docker-compose up -d

# Or run locally

python -m deployment.model\_server

```

## Making Predictions

### Single Prediction

```python

import requests

response = requests.post(

"http://localhost:8000/predict",

json={

"features": [0.5, 0.3, 0.7, 0.2, 0.1],

"patient\_id": "P001",

"hospital\_id": "H001"

}

)

print(response.json())

```

### Batch Predictions

```python

import asyncio

import aiohttp

async def predict\_batch(features\_list):

async with aiohttp.ClientSession() as session:

tasks = []

for features in features\_list:

task = session.post(

"http://localhost:8000/predict",

json={

"features": features,

"patient\_id": f"P{len(tasks):03d}",

"hospital\_id": "H001"

}

)

tasks.append(task)

return await asyncio.gather(\*tasks)

```

## Monitoring

### Dashboard Access

1. Open Grafana at `http://localhost:3000`

2. Login with default credentials (admin/admin)

3. Navigate to "Sepsis Model" dashboard

### Alert Configuration

Edit `monitoring/alerts.yml` to configure alert thresholds:

```yaml

alerts:

accuracy\_threshold: 0.95

latency\_threshold: 100 # ms

```

## Troubleshooting

### Common Issues

1. Model Loading Errors

```

Check MODEL\_PATH in .env file

Ensure model file exists and has correct permissions

```

2. Performance Issues

```

Increase worker count in docker-compose.yml

Enable batch processing for high load

Check Redis cache configuration

```

3. Memory Errors

```

Adjust container memory limits

Enable batch processing

Clear Redis cache if full

```

## Best Practices

1. Production Deployment

- Use container orchestration (Kubernetes)

- Enable monitoring and alerts

- Regular model updates

- Data validation

2. Performance Optimization

- Enable caching for frequent predictions

- Use batch processing for high throughput

- Monitor and adjust resource allocation

3. Security

- Enable authentication

- Use HTTPS

- Regular security updates

- Data encryption

```

4. optimization/performance\_tuning.py

```python

import asyncio

from typing import Dict, Any

from contextlib import asynccontextmanager

import orjson

import uvloop

from concurrent.futures import ProcessPoolExecutor

asyncio.set\_event\_loop\_policy(uvloop.EventLoopPolicy())

class PerformanceOptimizer:

def \_\_init\_\_(self):

self.process\_pool = ProcessPoolExecutor(max\_workers=4)

self.model\_cache = {}

@asynccontextmanager

async def prediction\_context(self):

"""Context manager for optimized prediction environment"""

try:

yield self

finally:

pass

async def optimize\_features(self, features: Dict[str, Any]) -> Dict[str, Any]:

"""Optimize feature processing"""

return await asyncio.get\_event\_loop().run\_in\_executor(

self.process\_pool,

self.\_process\_features,

features

)

def \_process\_features(self, features: Dict[str, Any]) -> Dict[str, Any]:

"""CPU-intensive feature processing"""

# Add feature optimization logic here

return features

def serialize\_response(self, data: Dict) -> bytes:

"""Fast JSON serialization"""

return orjson.dumps(data)

async def warm\_up\_cache(self):

"""Pre-warm model cache"""

# Add cache warming logic here

pass

```

These updates add:

1. Caching optimization

2. Batch processing optimization

3. Detailed user guide

4. Performance tuning tools

5. Best practices documentation

6. Troubleshooting guide

Let's add model versioning and error handling:

### 1. versioning/model\_registry.py

```python

from datetime import datetime

from typing import Dict, Optional, List

import joblib

import json

import os

from pathlib import Path

class ModelRegistry:

def \_\_init\_\_(self, registry\_path: str):

self.registry\_path = Path(registry\_path)

self.registry\_path.mkdir(parents=True, exist\_ok=True)

self.metadata\_file = self.registry\_path / "registry.json"

self.current\_model = None

self.\_load\_registry()

def \_load\_registry(self):

"""Load model registry metadata"""

if self.metadata\_file.exists():

with open(self.metadata\_file, 'r') as f:

self.registry = json.load(f)

else:

self.registry = {

'models': {},

'current\_version': None

}

def save\_model(self, model, version: str, metadata: Dict) -> str:

"""Save model with version and metadata"""

model\_path = self.registry\_path / f"model\_v{version}.joblib"

# Save model file

joblib.dump(model, model\_path)

# Update registry

model\_info = {

'version': version,

'path': str(model\_path),

'created\_at': datetime.now().isoformat(),

'metadata': metadata

}

self.registry['models'][version] = model\_info

self.\_save\_registry()

return version

def load\_model(self, version: Optional[str] = None) -> object:

"""Load specific model version"""

if version is None:

version = self.registry['current\_version']

if version not in self.registry['models']:

raise ValueError(f"Model version {version} not found")

model\_info = self.registry['models'][version]

model = joblib.load(model\_info['path'])

self.current\_model = model

return model

def set\_current\_version(self, version: str):

"""Set current production model version"""

if version not in self.registry['models']:

raise ValueError(f"Model version {version} not found")

self.registry['current\_version'] = version

self.\_save\_registry()

def get\_model\_info(self, version: Optional[str] = None) -> Dict:

"""Get model metadata"""

if version is None:

version = self.registry['current\_version']

return self.registry['models'].get(version)

def list\_versions(self) -> List[Dict]:

"""List all model versions"""

return [

{

'version': version,

\*\*info

}

for version, info in self.registry['models'].items()

]

def \_save\_registry(self):

"""Save registry metadata"""

with open(self.metadata\_file, 'w') as f:

json.dump(self.registry, f, indent=2)

```

### 2. error\_handling/exceptions.py

```python

from typing import Optional, Dict, Any

class ModelError(Exception):

"""Base class for model-related errors"""

def \_\_init\_\_(

self,

message: str,

error\_code: str,

details: Optional[Dict[str, Any]] = None

):

self.message = message

self.error\_code = error\_code

self.details = details or {}

super().\_\_init\_\_(self.message)

class ModelNotFoundError(ModelError):

"""Error when model version is not found"""

def \_\_init\_\_(self, version: str):

super().\_\_init\_\_(

message=f"Model version {version} not found",

error\_code="MODEL\_NOT\_FOUND",

details={'version': version}

)

class InvalidInputError(ModelError):

"""Error for invalid input data"""

def \_\_init\_\_(self, details: Dict[str, Any]):

super().\_\_init\_\_(

message="Invalid input data",

error\_code="INVALID\_INPUT",

details=details

)

class PredictionError(ModelError):

"""Error during model prediction"""

def \_\_init\_\_(self, original\_error: Exception):

super().\_\_init\_\_(

message="Error during prediction",

error\_code="PREDICTION\_ERROR",

details={'original\_error': str(original\_error)}

)

class ServiceUnavailableError(ModelError):

"""Error when service is unavailable"""

def \_\_init\_\_(self, reason: str):

super().\_\_init\_\_(

message="Service temporarily unavailable",

error\_code="SERVICE\_UNAVAILABLE",

details={'reason': reason}

)

```

### 3. error\_handling/handlers.py

```python

from fastapi import Request

from fastapi.responses import JSONResponse

from typing import Dict, Any

import logging

from .exceptions import ModelError

logger = logging.getLogger(\_\_name\_\_)

async def model\_error\_handler(request: Request, exc: ModelError) -> JSONResponse:

"""Handle model-specific errors"""

logger.error(

f"Model error: {exc.message}",

extra={

'error\_code': exc.error\_code,

'details': exc.details,

'path': request.url.path

}

)

return JSONResponse(

status\_code=400,

content={

'error': exc.message,

'code': exc.error\_code,

'details': exc.details

}

)

def setup\_error\_handlers(app):

"""Configure error handlers for the application"""

app.exception\_handler(ModelError)(model\_error\_handler)

```

### 4. validation/input\_validation.py

```python

from pydantic import BaseModel, validator, conlist

from typing import List, Optional

import numpy as np

class PredictionInput(BaseModel):

features: conlist(float, min\_items=5, max\_items=5)

patient\_id: str

hospital\_id: str

@validator('features')

def validate\_features(cls, values):

"""Validate feature values"""

if not all(-1 <= x <= 1 for x in values):

raise ValueError("Features must be normalized between -1 and 1")

return values

@validator('patient\_id')

def validate\_patient\_id(cls, v):

"""Validate patient ID format"""

if not v.startswith('P') or not v[1:].isdigit():

raise ValueError("Invalid patient ID format")

return v

class BatchPredictionInput(BaseModel):

requests: List[PredictionInput]

batch\_id: Optional[str]

@validator('requests')

def validate\_batch\_size(cls, v):

"""Validate batch size"""

if len(v) > 100:

raise ValueError("Batch size exceeds maximum limit of 100")

return v

```

### 5. deployment/model\_server.py (Updated error handling section)

```python

from fastapi import FastAPI, HTTPException, Request

from fastapi.middleware.cors import CORSMiddleware

from error\_handling.exceptions import ModelError

from error\_handling.handlers import setup\_error\_handlers

from validation.input\_validation import PredictionInput, BatchPredictionInput

from versioning.model\_registry import ModelRegistry

import logging

app = FastAPI()

setup\_error\_handlers(app)

# Configure logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

# Initialize model registry

model\_registry = ModelRegistry("models")

@app.post("/predict")

async def predict(input\_data: PredictionInput):

try:

model = model\_registry.load\_model()

prediction = await model.predict(input\_data.features)

return {

"prediction": prediction,

"model\_version": model\_registry.get\_model\_info()['version']

}

except ModelError as e:

raise e

except Exception as e:

logger.exception("Unexpected error during prediction")

raise ServiceUnavailableError(str(e))

@app.post("/model/versions/{version}")

async def set\_model\_version(version: str):

try:

model\_registry.set\_current\_version(version)

return {"message": f"Successfully switched to model version {version}"}

except ModelNotFoundError as e:

raise e

except Exception as e:

logger.exception("Error switching model version")

raise HTTPException(status\_code=500, detail=str(e))

@app.get("/model/versions")

async def list\_model\_versions():

try:

versions = model\_registry.list\_versions()

return {"versions": versions}

except Exception as e:

logger.exception("Error listing model versions")

raise HTTPException(status\_code=500, detail=str(e))

```

These updates add:

1. Model version control system

2. Detailed error handling mechanism

3. Input validation

4. Exception logging

5. API error response formatting

6. Model version management interface

Add data validation pipeline and deployment guide:

1. validation/data\_pipeline.py

```python

from typing import Dict, List, Optional, Tuple

import numpy as np

import pandas as pd

from pydantic import BaseModel

from sklearn.base import BaseEstimator, TransformerMixin

class DataValidator(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, feature\_specs: Dict[str, Dict]):

self.feature\_specs = feature\_specs

self.statistics = {}

def fit(self, X: pd.DataFrame, y=None):

"""Calculate statistics on training data"""

self.statistics = {

column: {

'mean': X[column].mean(),

'std': X[column].std(),

'min': X[column].min(),

'max': X[column].max(),

'missing\_rate': X[column].isnull().mean()

}

for column in X.columns

}

return self

def validate\_data(self, X: pd.DataFrame) -> Tuple[pd.DataFrame, List[Dict]]:

"""Validate input data and return validation report"""

validation\_reports = []

for column, specs in self.feature\_specs.items():

# Check value range

if specs.get('range'):

min\_val, max\_val = specs['range']

mask = ~X[column].between(min\_val, max\_val)

if mask.any():

validation\_reports.append({

'feature': column,

'issue': 'out\_of\_range',

'details': f'Values outside [{min\_val}, {max\_val}]'

})

X.loc[mask, column] = X.loc[mask, column].clip(min\_val, max\_val)

# Check missing values

missing = X[column].isnull()

if missing.any():

validation\_reports.append({

'feature': column,

'issue': 'missing\_values',

'details': f'{missing.sum()} missing values'

})

# Check data drift

if 'drift\_threshold' in specs:

current\_mean = X[column].mean()

baseline\_mean = self.statistics[column]['mean']

drift = abs(current\_mean - baseline\_mean) / self.statistics[column]['std']

if drift > specs['drift\_threshold']:

validation\_reports.append({

'feature': column,

'issue': 'data\_drift',

'details': f'Drift score: {drift:.2f}'

})

return X, validation\_reports

class DataCleaner:

"""Data cleaner"""

def clean\_data(self, df: pd.DataFrame) -> pd.DataFrame:

# Remove duplicate rows

df = df.drop\_duplicates()

# Handle outliers

for column in df.select\_dtypes(include=[np.number]).columns:

q1 = df[column].quantile(0.25)

q3 = df[column].quantile(0.75)

iqr = q3 - q1

df[column] = df[column].clip(q1 - 1.5\*iqr, q3 + 1.5\*iqr)

return df

class FeatureProcessor:

"""Feature processor"""

def process\_features(self, df: pd.DataFrame) -> pd.DataFrame:

# Standardize numerical features

numeric\_features = df.select\_dtypes(include=[np.number]).columns

df[numeric\_features] = (df[numeric\_features] - df[numeric\_features].mean()) / df[numeric\_features].std()

return df

```

2. deployment/kubernetes/deployment.yaml

```yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: model-server

namespace: ml-production

spec:

replicas: 3

selector:

matchLabels:

app: model-server

template:

metadata:

labels:

app: model-server

spec:

containers:

- name: model-server

image: your-registry/model-server:latest

ports:

- containerPort: 8000

resources:

requests:

memory: "1Gi"

cpu: "500m"

limits:

memory: "2Gi"

cpu: "1000m"

env:

- name: MODEL\_PATH

value: /models/current

- name: REDIS\_URL

value: redis://redis-service:6379

volumeMounts:

- name: model-storage

mountPath: /models

volumes:

- name: model-storage

persistentVolumeClaim:

claimName: model-storage-pvc

```

3. docs/deployment\_guide.md

```markdown

# Deployment Guide

1. Environment Requirements

### Hardware Requirements

- CPU: Minimum 4 cores

- Memory: Minimum 8GB

- Storage: Minimum 20GB SSD

### Software Requirements

- Kubernetes 1.18+

- Docker 19.03+

- Helm 3.0+

- Redis 6.0+

2. Deployment Steps

2.1 Prepare Environment

```bash

# Create namespace

kubectl create namespace ml-production

# Add Helm repository

helm repo add stable https://charts.helm.sh/stable

helm repo update

```

2.2 Configure Storage

```bash

# Create Persistent Volume Claim

kubectl apply -f deployment/kubernetes/storage.yaml

# Verify PVC created

kubectl get pvc -n ml-production

```

2.3 Deploy Redis

```bash

helm install redis stable/redis \

--namespace ml-production \

--set password=your-password

```

2.4 Deploy Model Service

```bash

# Update configuration

cp deployment/kubernetes/config.yaml.template deployment/kubernetes/config.yaml

# Edit config.yaml set environment-specific values

# Deploy service

kubectl apply -f deployment/kubernetes/deployment.yaml

kubectl apply -f deployment/kubernetes/service.yaml

```

2.5 Configure Monitoring

```bash

# Deploy Prometheus

helm install prometheus stable/prometheus \

--namespace ml-production \

--values deployment/kubernetes/prometheus-values.yaml

# Deploy Grafana

helm install grafana stable/grafana \

--namespace ml-production \

--values deployment/kubernetes/grafana-values.yaml

```

1. Verify Deployment
   1. Check Service Status

```bash

# Check Pod status

kubectl get pods -n ml-production

# Check services

kubectl get svc -n ml-production

# View logs

kubectl logs -f deployment/model-server -n ml-production

```

* 1. Test API

```bash

# Get service URL

export SERVICE\_URL=$(kubectl get svc model-server -n ml-production -o jsonpath='{.status.loadBalancer.ingress[0].ip}')

# Test health checkendpoint

curl http://$SERVICE\_URL:8000/health

# Test prediction endpoint

curl -X POST http://$SERVICE\_URL:8000/predict \

-H "Content-Type: application/json" \

-d '{"features": [0.1, 0.2, 0.3, 0.4, 0.5]}'

```

1. Scaling and Maintenance
   1. Scale Service

```bash

# Horizontal scaling

kubectl scale deployment model-server -n ml-production --replicas=5

# Autoscaling

kubectl autoscale deployment model-server \

-n ml-production \

--min=3 \

--max=10 \

--cpu-percent=80

```

* 1. Update Model

```bash

# Update model version

kubectl set image deployment/model-server \

model-server=your-registry/model-server:new-version \

-n ml-production

```

* 1. Backup and Recovery

```bash

# Backup model data

kubectl exec -n ml-production \

$(kubectl get pod -l app=model-server -o jsonpath='{.items[0].metadata.name}') \

-- tar czf /tmp/model-backup.tar.gz /models

# Restore from backup

kubectl cp ml-production/pod-name:/tmp/model-backup.tar.gz ./model-backup.tar.gz

```

5. 故障排除

5.1 常见问题

1. Pod启动失败

```bash

kubectl describe pod <pod-name> -n ml-production

kubectl logs <pod-name> -n ml-production

```

2. 服务不可访问

```bash

kubectl get endpoints model-server -n ml-production

kubectl describe svc model-server -n ml-production

```

3. 性能问题

- 检查资源使用情况

- 检查日志中的警告信息

- 查看监控指标

### 5.2 联系支持

如遇到无法解决的问题，请联系支持团队：

- Email: ccmzhangrui@foxmail.com

- Slack: #ml-support

```

这些更新添加了：

1. 数据验证管道

2. 数据清洗和处理工具

3. Kubernetes部署配置

4. 详细的部署指南

5. 故障排除指南

让我们添加A/B测试功能和模型性能监控：

1. experimentation/ab\_testing.py

```python

from typing import Dict, Optional

import random

import redis

import json

from datetime import datetime

import numpy as np

from dataclasses import dataclass

@dataclass

class Experiment:

id: str

name: str

variants: Dict[str, float] # variant\_name -> traffic\_percentage

start\_date: datetime

end\_date: Optional[datetime] = None

class ABTestManager:

def \_\_init\_\_(self, redis\_client: redis.Redis):

self.redis = redis\_client

self.experiments\_key = "active\_experiments"

self.results\_key = "experiment\_results"

async def create\_experiment(self, experiment: Experiment):

"""创建新的A/B测试实验"""

experiment\_data = {

"id": experiment.id,

"name": experiment.name,

"variants": experiment.variants,

"start\_date": experiment.start\_date.isoformat(),

"end\_date": experiment.end\_date.isoformat() if experiment.end\_date else None

}

await self.redis.hset(

self.experiments\_key,

experiment.id,

json.dumps(experiment\_data)

)

async def get\_variant(self, user\_id: str, experiment\_id: str) -> str:

"""为用户分配实验变体"""

experiment\_data = await self.redis.hget(self.experiments\_key, experiment\_id)

if not experiment\_data:

return "default"

experiment = json.loads(experiment\_data)

variants = experiment["variants"]

# 使用一致性哈希确保用户始终获得相同的变体

random.seed(f"{user\_id}:{experiment\_id}")

rand = random.random()

cumulative = 0

for variant, percentage in variants.items():

cumulative += percentage

if rand <= cumulative:

return variant

return "default"

async def record\_outcome(

self,

experiment\_id: str,

variant: str,

outcome: float,

metadata: Optional[Dict] = None

):

"""记录实验结果"""

result = {

"timestamp": datetime.now().isoformat(),

"variant": variant,

"outcome": outcome,

"metadata": metadata or {}

}

await self.redis.lpush(

f"{self.results\_key}:{experiment\_id}",

json.dumps(result)

)

async def get\_experiment\_results(self, experiment\_id: str) -> Dict:

"""获取实验结果统计"""

results = await self.redis.lrange(

f"{self.results\_key}:{experiment\_id}",

0, -1

)

outcomes\_by\_variant = {}

for result in results:

data = json.loads(result)

variant = data["variant"]

outcome = data["outcome"]

if variant not in outcomes\_by\_variant:

outcomes\_by\_variant[variant] = []

outcomes\_by\_variant[variant].append(outcome)

statistics = {}

for variant, outcomes in outcomes\_by\_variant.items():

statistics[variant] = {

"count": len(outcomes),

"mean": np.mean(outcomes),

"std": np.std(outcomes),

"confidence\_interval": self.\_calculate\_confidence\_interval(outcomes)

}

return statistics

def \_calculate\_confidence\_interval(self, outcomes: list) -> Dict:

"""计算95%置信区间"""

mean = np.mean(outcomes)

std = np.std(outcomes)

n = len(outcomes)

z = 1.96 # 95% confidence level

margin\_of\_error = z \* (std / np.sqrt(n))

return {

"lower": mean - margin\_of\_error,

"upper": mean + margin\_of\_error

}

```

2. monitoring/model\_metrics.py

```python

from typing import Dict, List

import numpy as np

from prometheus\_client import Histogram, Counter, Gauge

import pandas as pd

from sklearn.metrics import roc\_auc\_score, precision\_recall\_curve

import logging

logger = logging.getLogger(\_\_name\_\_)

class ModelPerformanceMonitor:

def \_\_init\_\_(self):

# 定义指标

self.prediction\_latency = Histogram(

'model\_prediction\_latency\_seconds',

'Time spent making predictions',

buckets=[0.1, 0.5, 1.0, 2.0, 5.0]

)

self.prediction\_count = Counter(

'model\_predictions\_total',

'Total number of predictions',

['model\_version', 'outcome']

)

self.feature\_drift = Gauge(

'model\_feature\_drift',

'Feature drift score',

['feature\_name']

)

self.model\_accuracy = Gauge(

'model\_accuracy',

'Model accuracy score',

['model\_version']

)

# 存储历史数据

self.predictions\_history = []

self.features\_history = []

def record\_prediction(

self,

prediction: float,

actual: float,

features: Dict,

latency: float,

model\_version: str

):

"""记录预测结果和相关指标"""

# 记录延迟

self.prediction\_latency.observe(latency)

# 记录预测计数

outcome = 'correct' if round(prediction) == actual else 'incorrect'

self.prediction\_count.labels(

model\_version=model\_version,

outcome=outcome

).inc()

# 存储预测历史

self.predictions\_history.append({

'prediction': prediction,

'actual': actual,

'timestamp': pd.Timestamp.now()

})

# 存储特征历史

self.features\_history.append(features)

# 计算并更新指标

self.\_update\_metrics(model\_version)

def \_update\_metrics(self, model\_version: str):

"""更新模型性能指标"""

try:

# 计算准确率

if len(self.predictions\_history) >= 100:

recent\_predictions = pd.DataFrame(self.predictions\_history[-100:])

accuracy = (

(recent\_predictions['prediction'].round() == recent\_predictions['actual'])

.mean()

)

self.model\_accuracy.labels(model\_version=model\_version).set(accuracy)

# 计算特征漂移

if len(self.features\_history) >= 1000:

baseline = pd.DataFrame(self.features\_history[:1000])

recent = pd.DataFrame(self.features\_history[-1000:])

for feature in baseline.columns:

drift\_score = self.\_calculate\_drift(

baseline[feature],

recent[feature]

)

self.feature\_drift.labels(feature\_name=feature).set(drift\_score)

except Exception as e:

logger.error(f"Error updating metrics: {e}")

def \_calculate\_drift(self, baseline: pd.Series, current: pd.Series) -> float:

"""计算特征漂移分数"""

baseline\_mean = baseline.mean()

baseline\_std = baseline.std()

current\_mean = current.mean()

if baseline\_std == 0:

return 0

return abs(current\_mean - baseline\_mean) / baseline\_std

def get\_performance\_report(self, model\_version: str) -> Dict:

"""生成性能报告"""

recent\_predictions = pd.DataFrame(self.predictions\_history[-1000:])

try:

report = {

"accuracy": (

recent\_predictions['prediction'].round() == recent\_predictions['actual']

).mean(),

"auc\_roc": roc\_auc\_score(

recent\_predictions['actual'],

recent\_predictions['prediction']

),

"latency\_stats": {

"mean": self.prediction\_latency.\_sum.get() / self.prediction\_latency.\_count.get(),

"count": self.prediction\_latency.\_count.get()

}

}

# 添加特征漂移统计

drift\_stats = {}

for feature in self.features\_history[0].keys():

drift\_stats[feature] = self.feature\_drift.labels(feature\_name=feature).\_value

report["feature\_drift"] = drift\_stats

return report

except Exception as e:

logger.error(f"Error generating performance report: {e}")

return {}

```

3. api/experiment\_routes.py

```python

from fastapi import APIRouter, Depends

from typing import Dict

from experimentation.ab\_testing import ABTestManager, Experiment

from datetime import datetime

router = APIRouter()

@router.post("/experiments")

async def create\_experiment(

name: str,

variants: Dict[str, float],

ab\_manager: ABTestManager = Depends()

):

"""创建新的A/B测试实验"""

experiment = Experiment(

id=f"exp\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}",

name=name,

variants=variants,

start\_date=datetime.now()

)

await ab\_manager.create\_experiment(experiment)

return {"experiment\_id": experiment.id}

@router.get("/experiments/{experiment\_id}/results")

async def get\_experiment\_results(

experiment\_id: str,

ab\_manager: ABTestManager = Depends()

):

"""获取实验结果"""

results = await ab\_manager.get\_experiment\_results(experiment\_id)

return results

@router.post("/experiments/{experiment\_id}/outcomes")

async def record\_outcome(

experiment\_id: str,

variant: str,

outcome: float,

metadata: Dict,

ab\_manager: ABTestManager = Depends()

):

"""记录实验结果"""

await ab\_manager.record\_outcome(

experiment\_id,

variant,

outcome,

metadata

)

return {"status": "success"}

```

4. monitoring/alerts.py

```python

from typing import Dict, Any

import json

import requests

from datetime import datetime

class AlertManager:

def \_\_init\_\_(self, slack\_webhook\_url: str, email\_config: Dict):

self.slack\_webhook\_url = slack\_webhook\_url

self.email\_config = email\_config

self.alert\_thresholds = {

"accuracy": 0.95,

"latency": 100, # ms

"drift": 0.3

}

async def check\_and\_alert(self, metrics: Dict[str, Any]):

"""检查指标并发送告警"""

alerts = []

# 检查准确率

if metrics['accuracy'] < self.alert\_thresholds['accuracy']:

alerts.append({

"type": "accuracy",

"message": f"Model accuracy dropped to {metrics['accuracy']:.2%}"

})

# 检查延迟

if metrics['latency\_stats']['mean'] > self.alert\_thresholds['latency']:

alerts.append({

"type": "latency",

"message": f"High prediction latency: {metrics['latency\_stats']['mean']:.2f}ms"

})

# 检查特征漂移

for feature, drift in metrics['feature\_drift'].items():

if drift > self.alert\_thresholds['drift']:

alerts.append({

"type": "drift",

"message": f"High feature drift detected in {feature}: {drift:.2f}"

})

# 发送告警

if alerts:

await self.\_send\_alerts(alerts)

async def \_send\_alerts(self, alerts: List[Dict]):

"""发送告警通知"""

# Slack通知

if self.slack\_webhook\_url:

message = self.\_format\_slack\_message(alerts)

requests.post(self.slack\_webhook\_url, json={"text": message})

# 邮件通知

if self.email\_config:

self.\_send\_email\_alert(alerts)

def \_format\_slack\_message(self, alerts: List[Dict]) -> str:

"""格式化Slack消息"""

message = "🚨 \*Model Alert\*\n\n"

for alert in alerts:

message += f"• {alert['message']}\n"

message += f"\nTime: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}"

return message

```

这些更新添加了：

1. A/B测试管理系统

2. 详细的模型性能监控

3. 实验API路由

4. 告警系统

主要功能包括：

- 创建和管理A/B测试实验

- 追踪模型性能指标

- 检测特征漂移

- 生成性能报告

- 自动告警系统

让我们添加自动化部署脚本和可视化仪表板：

### 1. deployment/scripts/deploy.sh

```bash

#!/bin/bash

# 设置环境变量

export PROJECT\_NAME="ml-service"

export DOCKER\_REGISTRY="your-registry.com"

export K8S\_NAMESPACE="ml-production"

# 颜色输出

GREEN='\033[0;32m'

RED='\033[0;31m'

NC='\033[0m'

# 检查依赖

check\_dependencies() {

echo "Checking dependencies..."

command -v docker >/dev/null 2>&1 || { echo "${RED}Docker is required but not installed.${NC}" >&2; exit 1; }

command -v kubectl >/dev/null 2>&1 || { echo "${RED}kubectl is required but not installed.${NC}" >&2; exit 1; }

command -v helm >/dev/null 2>&1 || { echo "${RED}Helm is required but not installed.${NC}" >&2; exit 1; }

}

# 构建Docker镜像

build\_docker() {

echo "Building Docker image..."

VERSION=$(git describe --tags --always)

docker build -t ${DOCKER\_REGISTRY}/${PROJECT\_NAME}:${VERSION} .

docker tag ${DOCKER\_REGISTRY}/${PROJECT\_NAME}:${VERSION} ${DOCKER\_REGISTRY}/${PROJECT\_NAME}:latest

if [ $? -eq 0 ]; then

echo "${GREEN}Docker build successful${NC}"

else

echo "${RED}Docker build failed${NC}"

exit 1

fi

}

# 推送Docker镜像

push\_docker() {

echo "Pushing Docker image..."

docker push ${DOCKER\_REGISTRY}/${PROJECT\_NAME}:${VERSION}

docker push ${DOCKER\_REGISTRY}/${PROJECT\_NAME}:latest

}

# 部署到Kubernetes

deploy\_kubernetes() {

echo "Deploying to Kubernetes..."

# 创建namespace（如果不存在）

kubectl create namespace ${K8S\_NAMESPACE} --dry-run=client -o yaml | kubectl apply -f -

# 更新配置

envsubst < deployment/kubernetes/config.template.yaml > deployment/kubernetes/config.yaml

# 部署应用

kubectl apply -f deployment/kubernetes/config.yaml -n ${K8S\_NAMESPACE}

kubectl apply -f deployment/kubernetes/deployment.yaml -n ${K8S\_NAMESPACE}

kubectl apply -f deployment/kubernetes/service.yaml -n ${K8S\_NAMESPACE}

# 部署监控组件

helm upgrade --install prometheus prometheus-community/prometheus \

--namespace ${K8S\_NAMESPACE} \

--values deployment/kubernetes/prometheus-values.yaml

helm upgrade --install grafana grafana/grafana \

--namespace ${K8S\_NAMESPACE} \

--values deployment/kubernetes/grafana-values.yaml

}

# 验证部署

verify\_deployment() {

echo "Verifying deployment..."

# 等待Pod就绪

kubectl rollout status deployment/${PROJECT\_NAME} -n ${K8S\_NAMESPACE}

if [ $? -eq 0 ]; then

echo "${GREEN}Deployment successful${NC}"

# 获取服务URL

SERVICE\_URL=$(kubectl get svc ${PROJECT\_NAME} -n ${K8S\_NAMESPACE} -o jsonpath='{.status.loadBalancer.ingress[0].ip}')

echo "Service available at: http://${SERVICE\_URL}"

else

echo "${RED}Deployment failed${NC}"

exit 1

fi

}

# 主函数

main() {

check\_dependencies

# 解析命令行参数

while [[ $# -gt 0 ]]; do

case $1 in

--build-only)

build\_docker

exit 0

;;

--deploy-only)

deploy\_kubernetes

verify\_deployment

exit 0

;;

\*)

echo "Unknown parameter: $1"

exit 1

;;

esac

done

# 完整部署流程

build\_docker

push\_docker

deploy\_kubernetes

verify\_deployment

}

main "$@"

```

### 2. monitoring/dashboards/grafana/model\_performance.json

```json

{

"dashboard": {

"id": null,

"title": "ML Model Performance Dashboard",

"tags": ["ml", "model-monitoring"],

"timezone": "browser",

"panels": [

{

"title": "Model Accuracy Over Time",

"type": "graph",

"gridPos": {

"h": 8,

"w": 12,

"x": 0,

"y": 0

},

"targets": [

{

"expr": "model\_accuracy{model\_version=~\"$model\_version\"}",

"legendFormat": "{{model\_version}}"

}

],

"yaxes": [

{

"format": "percentunit",

"min": 0,

"max": 1

}

]

},

{

"title": "Prediction Latency",

"type": "graph",

"gridPos": {

"h": 8,

"w": 12,

"x": 12,

"y": 0

},

"targets": [

{

"expr": "rate(model\_prediction\_latency\_seconds\_sum[5m]) / rate(model\_prediction\_latency\_seconds\_count[5m])",

"legendFormat": "avg latency"

}

]

},

{

"title": "Feature Drift Heatmap",

"type": "heatmap",

"gridPos": {

"h": 8,

"w": 24,

"x": 0,

"y": 8

},

"targets": [

{

"expr": "model\_feature\_drift",

"legendFormat": "{{feature\_name}}"

}

]

},

{

"title": "Predictions Per Minute",

"type": "stat",

"gridPos": {

"h": 4,

"w": 6,

"x": 0,

"y": 16

},

"targets": [

{

"expr": "sum(rate(model\_predictions\_total[1m]))"

}

]

},

{

"title": "Error Rate",

"type": "gauge",

"gridPos": {

"h": 4,

"w": 6,

"x": 6,

"y": 16

},

"targets": [

{

"expr": "sum(rate(model\_predictions\_total{outcome=\"incorrect\"}[5m])) / sum(rate(model\_predictions\_total[5m]))"

}

],

"thresholds": [

{ "value": 0.01, "color": "green" },

{ "value": 0.05, "color": "yellow" },

{ "value": 0.1, "color": "red" }

]

}

],

"templating": {

"list": [

{

"name": "model\_version",

"type": "query",

"datasource": "Prometheus",

"query": "label\_values(model\_accuracy, model\_version)"

}

]

},

"refresh": "5s"

}

}

```

### 3. monitoring/dashboards/grafana/ab\_testing.json

```json

{

"dashboard": {

"id": null,

"title": "A/B Testing Dashboard",

"tags": ["ml", "ab-testing"],

"panels": [

{

"title": "Experiment Results",

"type": "table",

"gridPos": {

"h": 8,

"w": 24,

"x": 0,

"y": 0

},

"targets": [

{

"expr": "experiment\_outcome{experiment\_id=~\"$experiment\"}",

"format": "table",

"instant": true

}

],

"transforms": [

{

"type": "groupBy",

"fields": {

"variant": "variant",

"outcome": "mean"

}

}

]

},

{

"title": "Conversion Rate by Variant",

"type": "bar",

"gridPos": {

"h": 8,

"w": 12,

"x": 0,

"y": 8

},

"targets": [

{

"expr": "sum(experiment\_conversion{experiment\_id=~\"$experiment\"}) by (variant) / sum(experiment\_total{experiment\_id=~\"$experiment\"}) by (variant)",

"legendFormat": "{{variant}}"

}

]

}

],

"templating": {

"list": [

{

"name": "experiment",

"type": "query",

"datasource": "Prometheus",

"query": "label\_values(experiment\_outcome, experiment\_id)"

}

]

}

}

}

```

### 4. deployment/scripts/setup\_monitoring.sh

```bash

#!/bin/bash

# 设置Grafana

setup\_grafana() {

# 创建Grafana数据源

curl -X POST \

-H "Content-Type: application/json" \

-d '{

"name": "Prometheus",

"type": "prometheus",

"url": "http://prometheus-server:9090",

"access": "proxy",

"isDefault": true

}' \

http://admin:${GRAFANA\_PASSWORD}@${GRAFANA\_HOST}/api/datasources

# 导入仪表板

for dashboard in monitoring/dashboards/grafana/\*.json; do

curl -X POST \

-H "Content-Type: application/json" \

-d @${dashboard} \

http://admin:${GRAFANA\_PASSWORD}@${GRAFANA\_HOST}/api/dashboards/db

done

}

# 设置Prometheus告警规则

setup\_prometheus\_rules() {

kubectl apply -f deployment/kubernetes/prometheus-rules.yaml -n ${K8S\_NAMESPACE}

}

# 设置告警通知

setup\_alerting() {

# 配置Slack通知

kubectl create secret generic alertmanager-config \

--from-file=alertmanager.yml=deployment/kubernetes/alertmanager.yml \

-n ${K8S\_NAMESPACE}

# 重启AlertManager

kubectl rollout restart deployment/prometheus-alertmanager -n ${K8S\_NAMESPACE}

}

# 主函数

main() {

echo "Setting up monitoring..."

setup\_grafana

setup\_prometheus\_rules

setup\_alerting

echo "Monitoring setup complete"

}

main "$@"

```

这些更新添加了：

1. 完整的自动化部署脚本

2. Grafana仪表板配置

- 模型性能监控

- A/B测试结果可视化

3. 监控系统设置脚本

主要功能包括：

- 一键部署应用

- 自动化监控系统配置

- 可视化性能指标

- A/B测试结果展示

添加模型回滚机制和负载测试脚本：

1. deployment/rollback/model\_rollback.py

```python

from datetime import datetime

from typing import Optional, Dict, List

import logging

from versioning.model\_registry import ModelRegistry

from monitoring.model\_metrics import ModelPerformanceMonitor

logger = logging.getLogger(\_\_name\_\_)

class ModelRollbackManager:

def \_\_init\_\_(

self,

model\_registry: ModelRegistry,

performance\_monitor: ModelPerformanceMonitor,

threshold\_config: Dict

):

self.model\_registry = model\_registry

self.performance\_monitor = performance\_monitor

self.threshold\_config = threshold\_config

self.rollback\_history = []

async def check\_health(self, current\_version: str) -> Dict:

"""检查模型健康状态"""

metrics = await self.performance\_monitor.get\_performance\_report(current\_version)

health\_status = {

"status": "healthy",

"issues": [],

"metrics": metrics

}

# 检查准确率

if metrics["accuracy"] < self.threshold\_config["min\_accuracy"]:

health\_status["issues"].append({

"type": "accuracy",

"message": f"Accuracy below threshold: {metrics['accuracy']:.4f}"

})

# 检查延迟

if metrics["latency\_stats"]["mean"] > self.threshold\_config["max\_latency"]:

health\_status["issues"].append({

"type": "latency",

"message": f"Latency above threshold: {metrics['latency\_stats']['mean']:.2f}ms"

})

# 检查特征漂移

for feature, drift in metrics["feature\_drift"].items():

if drift > self.threshold\_config["max\_drift"]:

health\_status["issues"].append({

"type": "drift",

"message": f"High drift detected in {feature}: {drift:.2f}"

})

if health\_status["issues"]:

health\_status["status"] = "unhealthy"

return health\_status

async def should\_rollback(self, current\_version: str) -> bool:

"""判断是否需要回滚"""

health\_status = await self.check\_health(current\_version)

if health\_status["status"] == "unhealthy":

# 检查问题的严重程度

critical\_issues = [

issue for issue in health\_status["issues"]

if self.\_is\_critical\_issue(issue)

]

return len(critical\_issues) > 0

return False

async def execute\_rollback(self, current\_version: str) -> Optional[str]:

"""执行模型回滚"""

try:

# 获取上一个稳定版本

previous\_version = await self.\_get\_last\_stable\_version(current\_version)

if not previous\_version:

logger.error("No stable version found for rollback")

return None

# 执行回滚

await self.model\_registry.set\_current\_version(previous\_version)

# 记录回滚历史

rollback\_record = {

"timestamp": datetime.now().isoformat(),

"from\_version": current\_version,

"to\_version": previous\_version,

"reason": "Performance degradation"

}

self.rollback\_history.append(rollback\_record)

logger.info(f"Successfully rolled back to version {previous\_version}")

return previous\_version

except Exception as e:

logger.error(f"Rollback failed: {str(e)}")

return None

async def \_get\_last\_stable\_version(self, current\_version: str) -> Optional[str]:

"""获取最近的稳定版本"""

versions = await self.model\_registry.list\_versions()

# 按时间倒序排序版本

sorted\_versions = sorted(

versions,

key=lambda x: x["created\_at"],

reverse=True

)

# 查找最近的稳定版本

for version in sorted\_versions:

if version["version"] == current\_version:

continue

# 检查版本性能历史

health\_status = await self.check\_health(version["version"])

if health\_status["status"] == "healthy":

return version["version"]

return None

def \_is\_critical\_issue(self, issue: Dict) -> bool:

"""判断问题是否严重"""

if issue["type"] == "accuracy":

return float(issue["message"].split(":")[1]) < self.threshold\_config["critical\_accuracy"]

elif issue["type"] == "latency":

return float(issue["message"].split(":")[1]) > self.threshold\_config["critical\_latency"]

elif issue["type"] == "drift":

return float(issue["message"].split(":")[1]) > self.threshold\_config["critical\_drift"]

return False

```

2. testing/load\_test.py

```python

import asyncio

import aiohttp

import numpy as np

from datetime import datetime

import json

import logging

from typing import List, Dict

import pandas as pd

from concurrent.futures import ThreadPoolExecutor

logger = logging.getLogger(\_\_name\_\_)

class LoadTester:

def \_\_init\_\_(

self,

base\_url: str,

test\_data: pd.DataFrame,

concurrent\_users: int = 10,

test\_duration: int = 300,

ramp\_up\_time: int = 60

):

self.base\_url = base\_url

self.test\_data = test\_data

self.concurrent\_users = concurrent\_users

self.test\_duration = test\_duration

self.ramp\_up\_time = ramp\_up\_time

self.results = []

async def run\_test(self):

"""执行负载测试"""

start\_time = datetime.now()

# 创建用户会话

async with aiohttp.ClientSession() as session:

# 逐步增加用户数

tasks = []

users\_per\_batch = self.concurrent\_users // (self.ramp\_up\_time // 10)

for i in range(0, self.concurrent\_users, users\_per\_batch):

user\_count = min(users\_per\_batch, self.concurrent\_users - i)

tasks.extend([

self.\_user\_session(session, user\_id)

for user\_id in range(i, i + user\_count)

])

await asyncio.sleep(10) # 每10秒增加一批用户

# 等待所有任务完成

await asyncio.gather(\*tasks)

# 生成测试报告

return self.\_generate\_report(start\_time)

async def \_user\_session(self, session: aiohttp.ClientSession, user\_id: int):

"""模拟用户会话"""

async def make\_request():

try:

# 随机选择测试数据

test\_sample = self.test\_data.sample(1).to\_dict('records')[0]

start\_time = datetime.now()

async with session.post(

f"{self.base\_url}/predict",

json=test\_sample

) as response:

end\_time = datetime.now()

latency = (end\_time - start\_time).total\_seconds()

result = {

"user\_id": user\_id,

"timestamp": start\_time.isoformat(),

"latency": latency,

"status\_code": response.status,

"success": response.status == 200

}

if response.status == 200:

result["response"] = await response.json()

self.results.append(result)

except Exception as e:

logger.error(f"Request failed: {str(e)}")

self.results.append({

"user\_id": user\_id,

"timestamp": datetime.now().isoformat(),

"error": str(e),

"success": False

})

# 持续发送请求直到测试结束

end\_time = datetime.now().timestamp() + self.test\_duration

while datetime.now().timestamp() < end\_time:

await make\_request()

# 随机延迟模拟真实用户行为

await asyncio.sleep(np.random.exponential(1.0))

def \_generate\_report(self, start\_time: datetime) -> Dict:

"""生成测试报告"""

df = pd.DataFrame(self.results)

successful\_requests = df[df["success"]]

failed\_requests = df[~df["success"]]

report = {

"summary": {

"total\_requests": len(df),

"successful\_requests": len(successful\_requests),

"failed\_requests": len(failed\_requests),

"success\_rate": len(successful\_requests) / len(df),

"total\_duration": (datetime.now() - start\_time).total\_seconds(),

"requests\_per\_second": len(df) / (datetime.now() - start\_time).total\_seconds()

},

"latency": {

"mean": successful\_requests["latency"].mean(),

"median": successful\_requests["latency"].median(),

"p95": successful\_requests["latency"].quantile(0.95),

"p99": successful\_requests["latency"].quantile(0.99)

},

"errors": {

"count": len(failed\_requests),

"types": failed\_requests["error"].value\_counts().to\_dict()

}

}

return report

def save\_results(self, filename: str):

"""保存测试结果"""

with open(filename, 'w') as f:

json.dump(self.results, f, indent=2)

```

3. testing/load\_test\_script.py

```python

import asyncio

import pandas as pd

from load\_test import LoadTester

import json

import argparse

async def main():

parser = argparse.ArgumentParser(description='Run load test')

parser.add\_argument('--url', required=True, help='Base URL of the API')

parser.add\_argument('--data', required=True, help='Path to test data CSV')

parser.add\_argument('--users', type=int, default=10, help='Number of concurrent users')

parser.add\_argument('--duration', type=int, default=300, help='Test duration in seconds')

parser.add\_argument('--output', default='load\_test\_results.json', help='Output file path')

args = parser.parse\_args()

# 加载测试数据

test\_data = pd.read\_csv(args.data)

# 创建负载测试器

load\_tester = LoadTester(

base\_url=args.url,

test\_data=test\_data,

concurrent\_users=args.users,

test\_duration=args.duration

)

# 运行测试

print(f"Starting load test with {args.users} users for {args.duration} seconds...")

report = await load\_tester.run\_test()

# 保存结果

load\_tester.save\_results(args.output)

# 打印报告

print("\nLoad Test Report:")

print(json.dumps(report, indent=2))

# 检查性能指标

if (report["latency"]["p95"] > 1.0 or # P95延迟超过1秒

report["summary"]["success\_rate"] < 0.99): # 成功率低于99%

print("\n⚠️ Performance issues detected!")

exit(1)

else:

print("\n✅ Performance test passed!")

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(main())

```

这些更新添加了：

1. 完整的模型回滚管理系统

- 健康检查

- 自动回滚触发

- 回滚历史记录

2. 负载测试框架

- 并发用户模拟

- 渐进式负载增加

- 详细的性能指标

- 测试报告生成

主要功能包括：

- 自动检测模型性能下降

- 智能回滚决策

- 大规模并发测试

- 性能指标监控

- 测试报告生成

使用示例：

```bash

# 运行负载测试

python testing/load\_test\_script.py \

--url http://api.example.com \

--data test\_data.csv \

--users 50 \

--duration 600 \

--output results.json

```

完整的 CI/CD 管道和性能基准测试：

1. .github/workflows/ci-cd.yml

```yaml

name: ML Service CI/CD

on:

push:

branches: [ main ]

tags: [ 'v\*' ]

pull\_request:

branches: [ main ]

env:

REGISTRY: ghcr.io

IMAGE\_NAME: ${{ github.repository }}

jobs:

test:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v2

- name: Set up Python

uses: actions/setup-python@v2

with:

python-version: '3.9'

- name: Cache pip packages

uses: actions/cache@v2

with:

path: ~/.cache/pip

key: ${{ runner.os }}-pip-${{ hashFiles('requirements.txt') }}

- name: Install dependencies

run: |

python -m pip install --upgrade pip

pip install -r requirements.txt

pip install -r requirements-dev.txt

- name: Run tests

run: |

pytest tests/ --cov=src --cov-report=xml

- name: Upload coverage report

uses: codecov/codecov-action@v2

with:

file: ./coverage.xml

lint:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v2

- name: Set up Python

uses: actions/setup-python@v2

with:

python-version: '3.9'

- name: Install linting tools

run: |

pip install black flake8 mypy

- name: Run linters

run: |

black --check src tests

flake8 src tests

mypy src

security-scan:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v2

- name: Run Bandit security scan

run: |

pip install bandit

bandit -r src/ -ll

- name: Run dependency security scan

uses: snyk/actions/python@master

env:

SNYK\_TOKEN: ${{ secrets.SNYK\_TOKEN }}

build:

needs: [test, lint, security-scan]

runs-on: ubuntu-latest

if: github.event\_name == 'push'

steps:

- uses: actions/checkout@v2

- name: Set up Docker Buildx

uses: docker/setup-buildx-action@v1

- name: Login to Container Registry

uses: docker/login-action@v1

with:

registry: ${{ env.REGISTRY }}

username: ${{ github.actor }}

password: ${{ secrets.GITHUB\_TOKEN }}

- name: Extract metadata

id: meta

uses: docker/metadata-action@v3

with:

images: ${{ env.REGISTRY }}/${{ env.IMAGE\_NAME }}

- name: Build and push

uses: docker/build-push-action@v2

with:

context: .

push: true

tags: ${{ steps.meta.outputs.tags }}

labels: ${{ steps.meta.outputs.labels }}

cache-from: type=gha

cache-to: type=gha,mode=max

deploy-staging:

needs: build

runs-on: ubuntu-latest

environment: staging

steps:

- uses: actions/checkout@v2

- name: Set up kubectl

uses: azure/k8s-set-context@v1

with:

kubeconfig: ${{ secrets.KUBE\_CONFIG\_STAGING }}

- name: Deploy to staging

run: |

./deployment/scripts/deploy.sh --env staging

- name: Run smoke tests

run: |

./testing/smoke\_test.sh https://staging-api.example.com

- name: Run load tests

run: |

python testing/load\_test\_script.py \

--url https://staging-api.example.com \

--data test\_data.csv \

--users 20 \

--duration 300

deploy-production:

needs: deploy-staging

runs-on: ubuntu-latest

environment: production

if: github.ref\_type == 'tag'

steps:

- uses: actions/checkout@v2

- name: Set up kubectl

uses: azure/k8s-set-context@v1

with:

kubeconfig: ${{ secrets.KUBE\_CONFIG\_PRODUCTION }}

- name: Deploy to production

run: |

./deployment/scripts/deploy.sh --env production

- name: Verify deployment

run: |

./deployment/scripts/verify\_deployment.sh

- name: Monitor deployment

run: |

./deployment/scripts/monitor\_deployment.sh --duration 1h

```

2. testing/benchmark/performance\_benchmark.py

```python

import time

import numpy as np

import pandas as pd

from typing import Dict, List

import requests

import json

import logging

from concurrent.futures import ThreadPoolExecutor

from datetime import datetime

logger = logging.getLogger(\_\_name\_\_)

class PerformanceBenchmark:

def \_\_init\_\_(

self,

api\_url: str,

test\_data: pd.DataFrame,

baseline\_metrics: Dict = None

):

self.api\_url = api\_url

self.test\_data = test\_data

self.baseline\_metrics = baseline\_metrics or {}

self.results = []

def run\_benchmark(

self,

n\_iterations: int = 1000,

batch\_size: int = 32,

n\_threads: int = 4

) -> Dict:

"""运行性能基准测试"""

logger.info(f"Starting benchmark with {n\_iterations} iterations")

with ThreadPoolExecutor(max\_workers=n\_threads) as executor:

# 分批处理请求

batches = np.array\_split(self.test\_data, n\_iterations // batch\_size)

futures = []

for batch in batches:

future = executor.submit(self.\_process\_batch, batch)

futures.append(future)

# 收集结果

for future in futures:

self.results.extend(future.result())

# 生成报告

return self.\_generate\_benchmark\_report()

def \_process\_batch(self, batch: pd.DataFrame) -> List[Dict]:

"""处理一批请求"""

results = []

for \_, row in batch.iterrows():

start\_time = time.time()

try:

response = requests.post(

f"{self.api\_url}/predict",

json=row.to\_dict()

)

latency = time.time() - start\_time

result = {

"timestamp": datetime.now().isoformat(),

"latency": latency,

"status\_code": response.status\_code,

"success": response.status\_code == 200

}

if response.status\_code == 200:

result["prediction"] = response.json()

except Exception as e:

result = {

"timestamp": datetime.now().isoformat(),

"error": str(e),

"success": False

}

results.append(result)

return results

def \_generate\_benchmark\_report(self) -> Dict:

"""生成基准测试报告"""

df = pd.DataFrame(self.results)

successful\_requests = df[df["success"]]

current\_metrics = {

"latency": {

"mean": successful\_requests["latency"].mean(),

"median": successful\_requests["latency"].median(),

"p95": successful\_requests["latency"].quantile(0.95),

"p99": successful\_requests["latency"].quantile(0.99)

},

"throughput": len(successful\_requests) / (

pd.to\_datetime(df["timestamp"]).max() -

pd.to\_datetime(df["timestamp"]).min()

).total\_seconds(),

"success\_rate": len(successful\_requests) / len(df)

}

# 与基准线比较

comparison = {}

if self.baseline\_metrics:

for metric, value in current\_metrics["latency"].items():

baseline\_value = self.baseline\_metrics["latency"].get(metric)

if baseline\_value:

diff\_pct = ((value - baseline\_value) / baseline\_value) \* 100

comparison[f"latency\_{metric}\_diff"] = f"{diff\_pct:+.2f}%"

# 比较吞吐量

baseline\_throughput = self.baseline\_metrics.get("throughput")

if baseline\_throughput:

throughput\_diff = (

(current\_metrics["throughput"] - baseline\_throughput) /

baseline\_throughput

) \* 100

comparison["throughput\_diff"] = f"{throughput\_diff:+.2f}%"

return {

"metrics": current\_metrics,

"comparison": comparison,

"timestamp": datetime.now().isoformat(),

"total\_requests": len(df),

"failed\_requests": len(df) - len(successful\_requests)

}

def save\_results(self, filename: str):

"""保存基准测试结果"""

with open(filename, 'w') as f:

json.dump({

"results": self.results,

"baseline\_metrics": self.baseline\_metrics

}, f, indent=2)

```

3. testing/benchmark/run\_benchmark.py

```python

import argparse

import pandas as pd

from performance\_benchmark import PerformanceBenchmark

import json

import sys

def main():

parser = argparse.ArgumentParser(description='Run performance benchmark')

parser.add\_argument('--url', required=True, help='API URL')

parser.add\_argument('--data', required=True, help='Test data path')

parser.add\_argument('--baseline', help='Baseline metrics file')

parser.add\_argument('--output', default='benchmark\_results.json')

parser.add\_argument('--iterations', type=int, default=1000)

parser.add\_argument('--batch-size', type=int, default=32)

parser.add\_argument('--threads', type=int, default=4)

args = parser.parse\_args()

# 加载测试数据

test\_data = pd.read\_csv(args.data)

# 加载基准线指标

baseline\_metrics = None

if args.baseline:

with open(args.baseline) as f:

baseline\_metrics = json.load(f)

# 运行基准测试

benchmark = PerformanceBenchmark(

api\_url=args.url,

test\_data=test\_data,

baseline\_metrics=baseline\_metrics

)

print(f"Running benchmark against {args.url}")

report = benchmark.run\_benchmark(

n\_iterations=args.iterations,

batch\_size=args.batch\_size,

n\_threads=args.threads

)

# 保存结果

benchmark.save\_results(args.output)

# 打印报告

print("\nBenchmark Report:")

print(json.dumps(report, indent=2))

# 检查性能退化

if baseline\_metrics:

for metric, diff in report["comparison"].items():

diff\_value = float(diff.rstrip('%'))

if diff\_value > 10: # 性能下降超过10%

print(f"\n⚠️ Performance degradation detected in {metric}: {diff}")

sys.exit(1)

print("\n✅ Benchmark completed successfully!")

if \_\_name\_\_ == "\_\_main\_\_":

main()

```

这些更新添加了：

1. 完整的CI/CD流水线

- 单元测试

- 代码质量检查

- 安全扫描

- 构建和部署

- 自动化测试

2. 性能基准测试框架

- 并发请求测试

- 性能指标收集

- 与基准线比较

- 详细报告生成

主要功能包括：

- 自动化构建和部署

- 多环境部署

- 性能监控

- 自动回滚

- 基准测试

使用示例：

```bash

# 运行基准测试

python testing/benchmark/run\_benchmark.py \

--url https://api.example.com \

--data test\_data.csv \

--baseline baseline\_metrics.json \

--iterations 2000 \

--batch-size 64 \

--threads 8

```

金丝雀部署和完善的文档系统：

1. deployment/canary/canary\_deployer.py

```python

from typing import Dict, Optional

import kubernetes

from kubernetes import client, watch

import logging

import time

from prometheus\_api\_client import PrometheusConnect

from dataclasses import dataclass

logger = logging.getLogger(\_\_name\_\_)

@dataclass

class CanaryConfig:

initial\_weight: int = 10 # 初始金丝雀流量百分比

increment: int = 10 # 每次增加的流量百分比

interval: int = 300 # 评估间隔(秒)

max\_errors: int = 5 # 最大允许错误数

metrics\_threshold: Dict = None # 性能指标阈值

class CanaryDeployer:

def \_\_init\_\_(

self,

namespace: str,

service\_name: str,

new\_version: str,

prometheus\_url: str,

config: CanaryConfig

):

self.namespace = namespace

self.service\_name = service\_name

self.new\_version = new\_version

self.config = config

# 初始化Kubernetes客户端

self.k8s\_client = kubernetes.client.ApiClient()

self.apps\_v1 = client.AppsV1Api(self.k8s\_client)

self.core\_v1 = client.CoreV1Api(self.k8s\_client)

# 初始化Prometheus客户端

self.prom = PrometheusConnect(url=prometheus\_url)

async def start\_canary(self):

"""开始金丝雀部署"""

logger.info(f"Starting canary deployment for version {self.new\_version}")

try:

# 创建金丝雀部署

await self.\_create\_canary\_deployment()

# 逐步增加流量

current\_weight = self.config.initial\_weight

while current\_weight < 100:

# 等待评估间隔

time.sleep(self.config.interval)

# 检查金丝雀健康状态

if not await self.\_check\_canary\_health():

logger.error("Canary health check failed")

await self.\_rollback()

return False

# 增加流量

current\_weight += self.config.increment

await self.\_update\_traffic\_split(current\_weight)

logger.info(f"Increased canary traffic to {current\_weight}%")

# 完成迁移

await self.\_promote\_canary()

return True

except Exception as e:

logger.error(f"Canary deployment failed: {str(e)}")

await self.\_rollback()

return False

async def \_create\_canary\_deployment(self):

"""创建金丝雀部署"""

deployment = {

"apiVersion": "apps/v1",

"kind": "Deployment",

"metadata": {

"name": f"{self.service\_name}-canary",

"namespace": self.namespace

},

"spec": {

"replicas": 1,

"selector": {

"matchLabels": {

"app": self.service\_name,

"version": self.new\_version

}

},

"template": {

"metadata": {

"labels": {

"app": self.service\_name,

"version": self.new\_version

}

},

"spec": {

"containers": [{

"name": self.service\_name,

"image": f"{self.service\_name}:{self.new\_version}"

}]

}

}

}

}

self.apps\_v1.create\_namespaced\_deployment(

self.namespace,

deployment

)

async def \_check\_canary\_health(self) -> bool:

"""检查金丝雀版本的健康状态"""

# 检查错误率

error\_count = self.\_get\_error\_count()

if error\_count > self.config.max\_errors:

logger.warning(f"Error count ({error\_count}) exceeds threshold")

return False

# 检查性能指标

if not await self.\_check\_performance\_metrics():

logger.warning("Performance metrics check failed")

return False

return True

async def \_check\_performance\_metrics(self) -> bool:

"""检查性能指标"""

metrics = self.config.metrics\_threshold

for metric\_name, threshold in metrics.items():

query = f'{metric\_name}{{version="{self.new\_version}"}}'

result = self.prom.custom\_query(query)

if not result or float(result[0]['value'][1]) > threshold:

logger.warning(f"Metric {metric\_name} exceeds threshold")

return False

return True

async def \_update\_traffic\_split(self, canary\_weight: int):

"""更新流量分配"""

vs = {

"apiVersion": "networking.istio.io/v1alpha3",

"kind": "VirtualService",

"metadata": {

"name": self.service\_name,

"namespace": self.namespace

},

"spec": {

"hosts": [f"{self.service\_name}"],

"http": [{

"route": [

{

"destination": {

"host": f"{self.service\_name}",

"subset": "canary"

},

"weight": canary\_weight

},

{

"destination": {

"host": f"{self.service\_name}",

"subset": "stable"

},

"weight": 100 - canary\_weight

}

]

}]

}

}

self.k8s\_client.custom\_objects\_api().patch\_namespaced\_custom\_object(

"networking.istio.io",

"v1alpha3",

self.namespace,

"virtualservices",

self.service\_name,

vs

)

async def \_promote\_canary(self):

"""将金丝雀版本提升为稳定版本"""

# 更新主部署

self.apps\_v1.patch\_namespaced\_deployment(

name=self.service\_name,

namespace=self.namespace,

body={

"spec": {

"template": {

"spec": {

"containers": [{

"name": self.service\_name,

"image": f"{self.service\_name}:{self.new\_version}"

}]

}

}

}

}

)

# 删除金丝雀部署

self.apps\_v1.delete\_namespaced\_deployment(

name=f"{self.service\_name}-canary",

namespace=self.namespace

)

logger.info("Canary promotion completed")

async def \_rollback(self):

"""回滚部署"""

logger.info("Rolling back canary deployment")

try:

# 删除金丝雀部署

self.apps\_v1.delete\_namespaced\_deployment(

name=f"{self.service\_name}-canary",

namespace=self.namespace

)

# 重置流量分配

await self.\_update\_traffic\_split(0)

except Exception as e:

logger.error(f"Rollback failed: {str(e)}")

```

2. docs/README.md

```markdown

# ML Service Documentation

## Overview

This service provides machine learning model deployment and management capabilities with features including:

- Model versioning and deployment

- A/B testing

- Canary deployments

- Performance monitoring

- Automatic rollbacks

## Getting Started

### Prerequisites

- Python 3.9+

- Docker

- Kubernetes cluster

- Prometheus & Grafana

### Installation

1. Clone the repository:

```bash

git clone https://github.com/your-org/ml-service.git

cd ml-service

```

2. Install dependencies:

```bash

pip install -r requirements.txt

```

3. Configure environment:

```bash

cp .env.example .env

# Edit .env with your settings

```

### Configuration

Key configuration files:

- `config/app\_config.yaml`: Main application configuration

- `config/model\_config.yaml`: Model-specific settings

- `deployment/kubernetes/`: Kubernetes manifests

## Usage

### Deploying a Model

```bash

python scripts/deploy\_model.py --model-path /path/to/model --version v1.0

```

### Running A/B Tests

```bash

python scripts/create\_experiment.py \

--name "feature\_test" \

--variants '{"control": 0.5, "treatment": 0.5}'

```

### Monitoring

Access monitoring dashboards:

- Grafana: http://your-domain/grafana

- Prometheus: http://your-domain/prometheus

## Development

### Project Structure

```

ml-service/

├── src/

│ ├── api/ # API endpoints

│ ├── models/ # Model management

│ ├── monitoring/ # Monitoring tools

│ └── utils/ # Utility functions

├── tests/ # Test suite

├── deployment/ # Deployment configs

├── docs/ # Documentation

└── scripts/ # Utility scripts

```

### Running Tests

```bash

pytest tests/

```

### Contributing

1. Fork the repository

2. Create your feature branch

3. Write tests for new features

4. Submit a pull request

## API Reference

### Prediction API

```http

POST /api/v1/predict

Content-Type: application/json

{

"features": {...}

}

```

### Management API

```http

POST /api/v1/models/deploy

GET /api/v1/models/status

POST /api/v1/experiments/create

```

## Monitoring & Alerts

### Key Metrics

- Model accuracy

- Prediction latency

- Error rates

- Feature drift

### Alert Rules

Configured in `monitoring/alert\_rules.yaml`

## Troubleshooting

### Common Issues

1. High latency

- Check resource utilization

- Verify cache hit rates

- Review concurrent requests

2. Model errors

- Check model logs

- Verify input data format

- Review feature distributions

### Support

- Create an issue on GitHub

- Contact: ml-team@your-org.com

```

3. deployment/kubernetes/canary/templates/canary.yaml

```yaml

apiVersion: networking.istio.io/v1alpha3

kind: VirtualService

metadata:

name: {{ .Values.serviceName }}

spec:

hosts:

- {{ .Values.serviceName }}

http:

- route:

- destination:

host: {{ .Values.serviceName }}

subset: stable

weight: {{ sub 100 .Values.canary.trafficWeight }}

- destination:

host: {{ .Values.serviceName }}

subset: canary

weight: {{ .Values.canary.trafficWeight }}

---

apiVersion: networking.istio.io/v1alpha3

kind: DestinationRule

metadata:

name: {{ .Values.serviceName }}

spec:

host: {{ .Values.serviceName }}

subsets:

- name: stable

labels:

version: stable

- name: canary

labels:

version: canary

```

这些更新添加了：

1. 完整的金丝雀部署系统

- 渐进式流量迁移

- 健康检查

- 自动回滚

2. 详细的文档

- 项目概述

- 安装指南

- API参考

- 监控说明

- 故障排除

主要功能包括：

- 自动化金丝雀部署

- 性能指标监控

- 智能流量控制

- 完整的文档系统

添加更多监控指标和实现自动化性能测试系统：

1. monitoring/metrics/custom\_metrics.py

```python

from prometheus\_client import Counter, Histogram, Gauge, Summary

from typing import Dict

import time

import numpy as np

class MLServiceMetrics:

def \_\_init\_\_(self):

# 预测相关指标

self.prediction\_counter = Counter(

'ml\_predictions\_total',

'Total number of predictions',

['model\_version', 'status']

)

self.prediction\_latency = Histogram(

'ml\_prediction\_latency\_seconds',

'Prediction latency in seconds',

['model\_version'],

buckets=(0.1, 0.25, 0.5, 1.0, 2.5, 5.0, 10.0)

)

# 模型性能指标

self.model\_accuracy = Gauge(

'ml\_model\_accuracy',

'Model accuracy score',

['model\_version']

)

self.feature\_importance = Gauge(

'ml\_feature\_importance',

'Feature importance scores',

['model\_version', 'feature\_name']

)

# 系统资源指标

self.memory\_usage = Gauge(

'ml\_memory\_usage\_bytes',

'Memory usage in bytes',

['model\_version']

)

self.gpu\_utilization = Gauge(

'ml\_gpu\_utilization',

'GPU utilization percentage',

['device\_id']

)

# 特征漂移指标

self.feature\_drift = Gauge(

'ml\_feature\_drift',

'Feature drift score',

['model\_version', 'feature\_name']

)

# 批处理指标

self.batch\_size = Summary(

'ml\_batch\_size',

'Batch size statistics'

)

# 数据质量指标

self.data\_quality\_score = Gauge(

'ml\_data\_quality\_score',

'Data quality score',

['check\_type']

)

def record\_prediction(

self,

model\_version: str,

latency: float,

success: bool

):

"""记录预测指标"""

status = 'success' if success else 'failure'

self.prediction\_counter.labels(

model\_version=model\_version,

status=status

).inc()

self.prediction\_latency.labels(

model\_version=model\_version

).observe(latency)

def update\_model\_metrics(

self,

model\_version: str,

accuracy: float,

feature\_importances: Dict[str, float]

):

"""更新模型性能指标"""

self.model\_accuracy.labels(

model\_version=model\_version

).set(accuracy)

for feature, importance in feature\_importances.items():

self.feature\_importance.labels(

model\_version=model\_version,

feature\_name=feature

).set(importance)

def record\_feature\_drift(

self,

model\_version: str,

drift\_scores: Dict[str, float]

):

"""记录特征漂移"""

for feature, score in drift\_scores.items():

self.feature\_drift.labels(

model\_version=model\_version,

feature\_name=feature

).set(score)

def update\_resource\_metrics(

self,

model\_version: str,

memory\_usage: int,

gpu\_utils: Dict[str, float]

):

"""更新资源使用指标"""

self.memory\_usage.labels(

model\_version=model\_version

).set(memory\_usage)

for device\_id, utilization in gpu\_utils.items():

self.gpu\_utilization.labels(

device\_id=device\_id

).set(utilization)

def record\_data\_quality(self, check\_results: Dict[str, float]):

"""记录数据质量指标"""

for check\_type, score in check\_results.items():

self.data\_quality\_score.labels(

check\_type=check\_type

).set(score)

```

2. testing/performance/automated\_performance\_test.py

```python

import asyncio

import aiohttp

import numpy as np

import pandas as pd

from typing import Dict, List, Optional

import json

import logging

from datetime import datetime

import matplotlib.pyplot as plt

import seaborn as sns

from dataclasses import dataclass

logger = logging.getLogger(\_\_name\_\_)

@dataclass

class TestConfig:

base\_url: str

test\_duration: int = 3600 # 测试持续时间(秒)

min\_users: int = 10 # 最小并发用户数

max\_users: int = 100 # 最大并发用户数

step\_users: int = 10 # 每次增加的用户数

step\_duration: int = 300 # 每个负载级别持续时间

target\_rps: Optional[int] = None # 目标每秒请求数

class AutomatedPerformanceTest:

def \_\_init\_\_(

self,

config: TestConfig,

test\_data: pd.DataFrame

):

self.config = config

self.test\_data = test\_data

self.results = []

self.start\_time = None

async def run\_test(self):

"""执行自动化性能测试"""

logger.info("Starting automated performance test")

self.start\_time = datetime.now()

try:

# 逐步增加负载

for n\_users in range(

self.config.min\_users,

self.config.max\_users + 1,

self.config.step\_users

):

logger.info(f"Testing with {n\_users} concurrent users")

# 运行当前负载级别的测试

await self.\_run\_load\_level(n\_users)

# 分析结果并决定是否继续

if not self.\_should\_continue():

logger.warning("Performance threshold exceeded, stopping test")

break

# 生成测试报告

return self.\_generate\_report()

except Exception as e:

logger.error(f"Test failed: {str(e)}")

raise

async def \_run\_load\_level(self, n\_users: int):

"""运行特定负载级别的测试"""

async with aiohttp.ClientSession() as session:

tasks = []

for i in range(n\_users):

task = self.\_user\_session(

session,

user\_id=i,

duration=self.config.step\_duration

)

tasks.append(task)

await asyncio.gather(\*tasks)

async def \_user\_session(

self,

session: aiohttp.ClientSession,

user\_id: int,

duration: int

):

"""模拟用户会话"""

end\_time = datetime.now().timestamp() + duration

while datetime.now().timestamp() < end\_time:

# 选择测试数据

test\_case = self.test\_data.sample(1).to\_dict('records')[0]

# 发送请求并记录结果

try:

start\_time = time.time()

async with session.post(

f"{self.config.base\_url}/predict",

json=test\_case

) as response:

latency = time.time() - start\_time

result = {

"timestamp": datetime.now().isoformat(),

"user\_id": user\_id,

"latency": latency,

"status\_code": response.status,

"success": response.status == 200

}

if response.status == 200:

result["response"] = await response.json()

self.results.append(result)

except Exception as e:

self.results.append({

"timestamp": datetime.now().isoformat(),

"user\_id": user\_id,

"error": str(e),

"success": False

})

# 根据目标RPS调整等待时间

if self.config.target\_rps:

await asyncio.sleep(1.0 / self.config.target\_rps)

def \_should\_continue(self) -> bool:

"""判断是否继续测试"""

if not self.results:

return True

# 分析最近的结果

recent\_results = pd.DataFrame(

[r for r in self.results if r["success"]]

).tail(1000)

if len(recent\_results) == 0:

return False

# 检查性能指标

p95\_latency = recent\_results["latency"].quantile(0.95)

error\_rate = 1 - len(recent\_results) / 1000

return p95\_latency < 1.0 and error\_rate < 0.05

def \_generate\_report(self) -> Dict:

"""生成性能测试报告"""

df = pd.DataFrame(self.results)

df["timestamp"] = pd.to\_datetime(df["timestamp"])

successful\_requests = df[df["success"]]

# 计算汇总指标

summary = {

"total\_requests": len(df),

"successful\_requests": len(successful\_requests),

"error\_rate": 1 - len(successful\_requests) / len(df),

"test\_duration": (df["timestamp"].max() - df["timestamp"].min()).total\_seconds(),

"average\_rps": len(df) / (df["timestamp"].max() - df["timestamp"].min()).total\_seconds(),

"latency": {

"mean": successful\_requests["latency"].mean(),

"median": successful\_requests["latency"].median(),

"p95": successful\_requests["latency"].quantile(0.95),

"p99": successful\_requests["latency"].quantile(0.99)

}

}

# 生成图表

self.\_generate\_plots(df)

return summary

def \_generate\_plots(self, df: pd.DataFrame):

"""生成性能测试图表"""

# 设置风格

plt.style.use('seaborn')

# 1. 响应时间分布

plt.figure(figsize=(10, 6))

sns.histplot(data=df[df["success"]], x="latency", bins=50)

plt.title("Response Time Distribution")

plt.xlabel("Latency (seconds)")

plt.savefig("latency\_distribution.png")

# 2. 随时间变化的RPS

plt.figure(figsize=(12, 6))

df\_grouped = df.groupby(

pd.Grouper(key="timestamp", freq="1min")

).size()

df\_grouped.plot(kind="line")

plt.title("Requests per Second over Time")

plt.xlabel("Time")

plt.ylabel("RPS")

plt.savefig("rps\_over\_time.png")

# 3. 错误率热图

plt.figure(figsize=(10, 6))

error\_df = df.pivot\_table(

index=pd.Grouper(key="timestamp", freq="1min"),

values="success",

aggfunc=lambda x: 1 - x.mean()

)

sns.heatmap(error\_df.transpose(), cmap="YlOrRd")

plt.title("Error Rate Heatmap")

plt.savefig("error\_heatmap.png")

```

3. testing/performance/run\_performance\_test.py

```python

import asyncio

import argparse

import pandas as pd

import json

from automated\_performance\_test import AutomatedPerformanceTest, TestConfig

async def main():

parser = argparse.ArgumentParser(description='Run automated performance test')

parser.add\_argument('--url', required=True, help='API URL')

parser.add\_argument('--data', required=True, help='Test data path')

parser.add\_argument('--duration', type=int, default=3600, help='Test duration in seconds')

parser.add\_argument('--min-users', type=int, default=10, help='Minimum concurrent users')

parser.add\_argument('--max-users', type=int, default=100, help='Maximum concurrent users')

parser.add\_argument('--step-users', type=int, default=10, help='User increment step')

parser.add\_argument('--target-rps', type=int, help='Target requests per second')

parser.add\_argument('--output', default='performance\_test\_results.json')

args = parser.parse\_args()

# 加载测试数据

test\_data = pd.read\_csv(args.data)

# 配置测试

config = TestConfig(

base\_url=args.url,

test\_duration=args.duration,

min\_users=args.min\_users,

max\_users=args.max\_users,

step\_users=args.step\_users,

target\_rps=args.target\_rps

)

# 创建并运行测试

test = AutomatedPerformanceTest(config, test\_data)

print(f"Starting performance test against {args.url}")

results = await test.run\_test()

# 保存结果

with open(args.output, 'w') as f:

json.dump(results, f, indent=2)

# 打印摘要

print("\nTest Summary:")

print(f"Total Requests: {results['total\_requests']}")

print(f"Error Rate: {results['error\_rate']:.2%}")

print(f"Average RPS: {results['average\_rps']:.2f}")

print("\nLatency (seconds):")

for metric, value in results['latency'].items():

print(f" {metric}: {value:.3f}")

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(main())

```

这些更新添加了：

1. 完整的自定义指标系统

- 预测相关指标

- 模型性能指标

- 系统资源指标

- 特征漂移指标

- 数据质量指标

2. 自动化性能测试系统

- 渐进式负载测试

- 实时性能分析

- 自动停止条件

- 详细报告生成

- 可视化图表

主要功能包括：

- 全面的性能监控

- 自动化负载测试

- 性能瓶颈检测

- 图形化结果展示

使用示例：

```bash

# 运行性能测试

python testing/performance/run\_performance\_test.py \

--url https://api.example.com \

--data test\_data.csv \

--duration 7200 \

--min-users 20 \

--max-users 200 \

--step-users 20 \

--target-rps 1000

```

添加数据质量检查、容量规划和异常检测：

1. monitoring/data\_quality/quality\_checker.py

```python

import pandas as pd

import numpy as np

from typing import Dict, List, Optional

from dataclasses import dataclass

import logging

from sklearn.ensemble import IsolationForest

logger = logging.getLogger(\_\_name\_\_)

@dataclass

class DataQualityConfig:

missing\_threshold: float = 0.1

correlation\_threshold: float = 0.95

outlier\_contamination: float = 0.1

drift\_threshold: float = 0.1

class DataQualityChecker:

def \_\_init\_\_(self, config: DataQualityConfig):

self.config = config

self.baseline\_stats = None

self.isolation\_forest = None

def compute\_baseline(self, data: pd.DataFrame):

"""计算基准统计数据"""

self.baseline\_stats = {

"mean": data.mean(),

"std": data.std(),

"quantiles": data.quantile([0.25, 0.5, 0.75]),

"missing\_rates": data.isnull().mean()

}

# 训练异常检测器

self.isolation\_forest = IsolationForest(

contamination=self.config.outlier\_contamination,

random\_state=42

)

self.isolation\_forest.fit(data)

def check\_quality(self, data: pd.DataFrame) -> Dict:

"""执行数据质量检查"""

if self.baseline\_stats is None:

raise ValueError("Baseline stats not computed")

quality\_report = {

"missing\_values": self.\_check\_missing\_values(data),

"outliers": self.\_check\_outliers(data),

"drift": self.\_check\_drift(data),

"correlations": self.\_check\_correlations(data),

"schema": self.\_check\_schema(data)

}

# 计算总体质量分数

quality\_report["overall\_score"] = self.\_compute\_overall\_score(quality\_report)

return quality\_report

def \_check\_missing\_values(self, data: pd.DataFrame) -> Dict:

"""检查缺失值"""

missing\_rates = data.isnull().mean()

return {

"missing\_rates": missing\_rates.to\_dict(),

"columns\_above\_threshold": [

col for col, rate in missing\_rates.items()

if rate > self.config.missing\_threshold

]

}

def \_check\_outliers(self, data: pd.DataFrame) -> Dict:

"""检查异常值"""

outlier\_scores = self.isolation\_forest.score\_samples(data)

return {

"outlier\_ratio": np.mean(outlier\_scores < 0),

"outlier\_scores": outlier\_scores.tolist()

}

def \_check\_drift(self, data: pd.DataFrame) -> Dict:

"""检查数据漂移"""

drift\_scores = {}

for column in data.columns:

if column in self.baseline\_stats["mean"]:

current\_mean = data[column].mean()

baseline\_mean = self.baseline\_stats["mean"][column]

baseline\_std = self.baseline\_stats["std"][column]

drift\_score = abs(current\_mean - baseline\_mean) / baseline\_std

drift\_scores[column] = drift\_score

return {

"drift\_scores": drift\_scores,

"columns\_with\_drift": [

col for col, score in drift\_scores.items()

if score > self.config.drift\_threshold

]

}

def \_check\_correlations(self, data: pd.DataFrame) -> Dict:

"""检查特征相关性"""

corr\_matrix = data.corr()

high\_corr\_pairs = []

for i in range(len(corr\_matrix.columns)):

for j in range(i+1, len(corr\_matrix.columns)):

if abs(corr\_matrix.iloc[i,j]) > self.config.correlation\_threshold:

high\_corr\_pairs.append({

"feature1": corr\_matrix.columns[i],

"feature2": corr\_matrix.columns[j],

"correlation": corr\_matrix.iloc[i,j]

})

return {

"correlation\_matrix": corr\_matrix.to\_dict(),

"high\_correlation\_pairs": high\_corr\_pairs

}

def \_check\_schema(self, data: pd.DataFrame) -> Dict:

"""检查数据模式"""

schema\_issues = []

for column in data.columns:

if column not in self.baseline\_stats["mean"].index:

schema\_issues.append(f"New column: {column}")

for column in self.baseline\_stats["mean"].index:

if column not in data.columns:

schema\_issues.append(f"Missing column: {column}")

return {

"schema\_issues": schema\_issues,

"is\_valid": len(schema\_issues) == 0

}

def \_compute\_overall\_score(self, report: Dict) -> float:

"""计算总体质量分数"""

scores = []

# 缺失值得分

missing\_score = 1.0 - len(report["missing\_values"]["columns\_above\_threshold"]) / len(self.baseline\_stats["mean"])

scores.append(missing\_score)

# 异常值得分

outlier\_score = 1.0 - report["outliers"]["outlier\_ratio"]

scores.append(outlier\_score)

# 漂移得分

drift\_score = 1.0 - len(report["drift"]["columns\_with\_drift"]) / len(self.baseline\_stats["mean"])

scores.append(drift\_score)

# 模式得分

schema\_score = 1.0 if report["schema"]["is\_valid"] else 0.0

scores.append(schema\_score)

return np.mean(scores)

```

2. monitoring/capacity/capacity\_planner.py

```python

import pandas as pd

import numpy as np

from typing import Dict, List

from datetime import datetime, timedelta

from sklearn.linear\_model import LinearRegression

import logging

logger = logging.getLogger(\_\_name\_\_)

class CapacityPlanner:

def \_\_init\_\_(

self,

historical\_data: pd.DataFrame,

resource\_limits: Dict[str, float]

):

self.historical\_data = historical\_data

self.resource\_limits = resource\_limits

self.growth\_model = None

def analyze\_trends(self, window\_size: str = '1D') -> Dict:

"""分析资源使用趋势"""

# 重采样数据

daily\_stats = self.historical\_data.resample(window\_size).agg({

'cpu\_usage': 'mean',

'memory\_usage': 'mean',

'request\_count': 'sum',

'response\_time': 'mean'

})

# 计算增长率

growth\_rates = {}

for metric in daily\_stats.columns:

growth\_rate = self.\_calculate\_growth\_rate(daily\_stats[metric])

growth\_rates[metric] = growth\_rate

return {

"daily\_stats": daily\_stats.to\_dict(),

"growth\_rates": growth\_rates

}

def forecast\_capacity(self, days\_ahead: int = 30) -> Dict:

"""预测未来容量需求"""

forecasts = {}

for resource, limit in self.resource\_limits.items():

if resource in self.historical\_data.columns:

forecast = self.\_forecast\_resource(

self.historical\_data[resource],

days\_ahead

)

forecasts[resource] = {

"forecast": forecast,

"exceeds\_limit": any(v > limit for v in forecast)

}

return forecasts

def recommend\_scaling(self) -> Dict:

"""提供扩容建议"""

current\_usage = self.historical\_data.iloc[-1]

forecasts = self.forecast\_capacity()

recommendations = {

"immediate\_actions": [],

"planned\_actions": [],

"optimization\_suggestions": []

}

# 检查当前使用情况

for resource, limit in self.resource\_limits.items():

usage = current\_usage.get(resource, 0)

usage\_ratio = usage / limit

if usage\_ratio > 0.9:

recommendations["immediate\_actions"].append(

f"Immediately scale {resource}: current usage at {usage\_ratio:.1%}"

)

elif usage\_ratio > 0.7:

recommendations["planned\_actions"].append(

f"Plan to scale {resource} within 7 days: current usage at {usage\_ratio:.1%}"

)

# 检查预测结果

for resource, forecast in forecasts.items():

if forecast["exceeds\_limit"]:

days\_to\_limit = self.\_days\_until\_limit(

forecast["forecast"],

self.resource\_limits[resource]

)

recommendations["planned\_actions"].append(

f"Scale {resource} within {days\_to\_limit} days based on forecast"

)

# 优化建议

recommendations["optimization\_suggestions"].extend(

self.\_generate\_optimization\_suggestions()

)

return recommendations

def \_calculate\_growth\_rate(self, series: pd.Series) -> float:

"""计算增长率"""

if len(series) < 2:

return 0

start\_value = series.iloc[0]

end\_value = series.iloc[-1]

n\_periods = len(series)

if start\_value == 0:

return 0

return (end\_value / start\_value) \*\* (1/n\_periods) - 1

def \_forecast\_resource(

self,

data: pd.Series,

days\_ahead: int

) -> List[float]:

"""预测资源使用"""

X = np.arange(len(data)).reshape(-1, 1)

y = data.values

model = LinearRegression()

model.fit(X, y)

future\_X = np.arange(

len(data),

len(data) + days\_ahead

).reshape(-1, 1)

return model.predict(future\_X).tolist()

def \_days\_until\_limit(

self,

forecast: List[float],

limit: float

) -> int:

"""计算达到限制的天数"""

for i, value in enumerate(forecast):

if value > limit:

return i + 1

return len(forecast)

def \_generate\_optimization\_suggestions(self) -> List[str]:

"""生成优化建议"""

suggestions = []

# 分析资源效率

cpu\_memory\_ratio = (

self.historical\_data['cpu\_usage'] /

self.historical\_data['memory\_usage']

).mean()

if cpu\_memory\_ratio < 0.5:

suggestions.append(

"Consider reducing memory allocation: CPU/Memory ratio is low"

)

# 分析请求模式

request\_pattern = self.historical\_data['request\_count'].resample('1H').mean()

peak\_hours = request\_pattern.nlargest(3)

suggestions.append(

f"Peak load hours: {', '.join(str(h) for h in peak\_hours.index.hour)}"

)

return suggestions

```

3. monitoring/anomaly/anomaly\_detector.py

```python

import numpy as np

import pandas as pd

from typing import Dict, List, Optional

from sklearn.ensemble import IsolationForest

from scipy import stats

import logging

from dataclasses import dataclass

logger = logging.getLogger(\_\_name\_\_)

@dataclass

class AnomalyConfig:

window\_size: int = 100

contamination: float = 0.1

z\_score\_threshold: float = 3.0

change\_threshold: float = 0.2

class AnomalyDetector:

def \_\_init\_\_(self, config: AnomalyConfig):

self.config = config

self.isolation\_forest = IsolationForest(

contamination=config.contamination,

random\_state=42

)

self.baseline\_stats = {}

def fit(self, historical\_data: pd.DataFrame):

"""训练异常检测器"""

self.isolation\_forest.fit(historical\_data)

# 计算基准统计

self.baseline\_stats = {

"mean": historical\_data.mean(),

"std": historical\_data.std(),

"quantiles": historical\_data.quantile([0.25, 0.5, 0.75])

}

def detect\_anomalies(self, data: pd.DataFrame) -> Dict:

"""检测异常"""

if len(data) == 0:

return {"anomalies": [], "scores": []}

# 多种异常检测方法

anomaly\_scores = {

"isolation\_forest": self.\_detect\_isolation\_forest(data),

"z\_score": self.\_detect\_z\_score(data),

"change\_point": self.\_detect\_change\_points(data)

}

# 合并结果

combined\_scores = np.mean([

scores for scores in anomaly\_scores.values()

], axis=0)

anomalies = np.where(combined\_scores > 0.5)[0]

return {

"anomalies": anomalies.tolist(),

"scores": combined\_scores.tolist(),

"details": {

k: v.tolist() for k, v in anomaly\_scores.items()

}

}

def \_detect\_isolation\_forest(self, data: pd.DataFrame) -> np.ndarray:

"""使用Isolation Forest检测异常"""

scores = self.isolation\_forest.score\_samples(data)

return (scores - scores.min()) / (scores.max() - scores.min())

def \_detect\_z\_score(self, data: pd.DataFrame) -> np.ndarray:

"""使用Z-score检测异常"""

z\_scores = np.abs((data - self.baseline\_stats["mean"]) / self.baseline\_stats["std"])

return (z\_scores > self.config.z\_score\_threshold).any(axis=1).astype(float)

def \_detect\_change\_points(self, data: pd.DataFrame) -> np.ndarray:

"""检测变化点"""

scores = np.zeros(len(data))

for column in data.columns:

values = data[column].values

if len(values) < self.config.window\_size:

continue

# 使用滑动窗口检测变化

for i in range(self.config.window\_size, len(values)):

window = values[i-self.config.window\_size:i]

current = values[i]

window\_mean = np.mean(window)

window\_std = np.std(window)

if window\_std > 0:

change\_score = abs(current - window\_mean) / window\_std

if change\_score > self.config.change\_threshold:

scores[i] = max(scores[i], change\_score)

return scores / np.max(scores) if np.max(scores) > 0

1. MIMIC-III数据库提取代码（针对老年脓毒症患者）

```python

import pandas as pd

import numpy as np

from datetime import timedelta

from sqlalchemy import create\_engine

class MIMICExtractor:

def \_\_init\_\_(self, db\_uri):

self.engine = create\_engine(db\_uri)

def extract\_elderly\_sepsis\_patients(self, min\_age=65):

"""

从MIMIC-III提取老年脓毒症患者数据

参数:

min\_age: 最小年龄阈值(默认65岁)

返回:

包含患者ID、人口统计学和脓毒症诊断信息的DataFrame

"""

# SQL查询获取老年脓毒症患者

query = f"""

WITH sepsis\_patients AS (

SELECT DISTINCT adm.subject\_id

FROM admissions adm

JOIN diagnoses\_icd diag ON adm.hadm\_id = diag.hadm\_id

WHERE

diag.icd9\_code IN ('99591', '99592', '78552') -- 脓毒症ICD-9代码

OR diag.icd10\_code IN ('A419', 'R6520', 'R6521') -- 脓毒症ICD-10代码

),

elderly\_patients AS (

SELECT p.subject\_id,

p.gender,

p.dob,

adm.admittime,

EXTRACT(YEAR FROM adm.admittime) - EXTRACT(YEAR FROM p.dob) AS age

FROM patients p

JOIN admissions adm ON p.subject\_id = adm.subject\_id

WHERE EXTRACT(YEAR FROM adm.admittime) - EXTRACT(YEAR FROM p.dob) >= {min\_age}

)

SELECT

e.subject\_id,

e.gender,

e.age,

adm.hadm\_id,

adm.admittime,

adm.dischtime,

adm.deathtime,

adm.hospital\_expire\_flag

FROM elderly\_patients e

JOIN admissions adm ON e.subject\_id = adm.subject\_id

JOIN sepsis\_patients s ON e.subject\_id = s.subject\_id

ORDER BY e.subject\_id, adm.admittime;

"""

return pd.read\_sql(query, self.engine)

def extract\_vital\_signs(self, subject\_ids):

"""

提取指定患者的生命体征数据

参数:

subject\_ids: 患者ID列表

返回:

包含生命体征时间序列的DataFrame

"""

query = f"""

SELECT

c.subject\_id,

c.charttime,

c.itemid,

c.value,

c.valuenum,

c.valueuom

FROM chartevents c

WHERE c.subject\_id IN ({','.join(map(str, subject\_ids))})

AND c.itemid IN (

220045, 220050, 220051, 220052, 220179, 220210, -- 心率

220050, 220051, 220052, 220179, 220210, -- 血压

223761, 223762, -- 体温

220277, 220279, -- SpO2

220050, 220051, 220052, 220179, 220210 -- 呼吸频率

)

AND c.error IS NULL

ORDER BY c.subject\_id, c.charttime;

"""

return pd.read\_sql(query, self.engine)

def extract\_lab\_results(self, subject\_ids):

"""

提取实验室检查结果

参数:

subject\_ids: 患者ID列表

返回:

包含实验室检查结果的DataFrame

"""

query = f"""

SELECT

l.subject\_id,

l.charttime,

l.itemid,

l.value,

l.valuenum,

l.valueuom

FROM labevents l

WHERE l.subject\_id IN ({','.join(map(str, subject\_ids))})

AND l.itemid IN (

50882, 50912, 50902, -- 白细胞计数

50893, 50971, -- 乳酸

50822, 50824, -- 肌酐

50861, 50863, 50878 -- 肝功能

)

AND l.valuenum IS NOT NULL

ORDER BY l.subject\_id, l.charttime;

"""

return pd.read\_sql(query, self.engine)

```

2. 老年脓毒症轨迹模型开发代码

```python

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import roc\_auc\_score

import xgboost as xgb

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout

from tensorflow.keras.optimizers import Adam

from sklearn.cluster import KMeans

import shap

import matplotlib.pyplot as plt

import seaborn as sns

class SepsisTrajectoryModel:

def \_\_init\_\_(self, n\_clusters=5, random\_state=42):

self.n\_clusters = n\_clusters

self.random\_state = random\_state

self.scaler = StandardScaler()

self.kmeans = KMeans(n\_clusters=n\_clusters, random\_state=random\_state)

self.xgb\_model = None

self.lstm\_model = None

self.feature\_names = []

def preprocess\_data(self, df):

"""

预处理临床数据

参数:

df: 原始临床数据DataFrame

返回:

处理后的特征矩阵和标签

"""

# 处理缺失值

df = df.dropna(subset=['heart\_rate', 'blood\_pressure', 'temperature', 'respiratory\_rate'])

# 特征工程

df['age\_group'] = pd.cut(df['age'], bins=[65, 75, 85, 95, 105])

df['hr\_variability'] = df.groupby('subject\_id')['heart\_rate'].transform('std')

df['bp\_variability'] = df.groupby('subject\_id')['blood\_pressure'].transform('std')

# 标准化数值特征

num\_cols = ['heart\_rate', 'blood\_pressure', 'temperature', 'respiratory\_rate',

'hr\_variability', 'bp\_variability']

df[num\_cols] = self.scaler.fit\_transform(df[num\_cols])

# 保存特征名称

self.feature\_names = num\_cols + ['age\_group']

# 分离特征和标签

X = df[num\_cols + ['age\_group']]

y = df['mortality'].values

return X, y

def train\_xgboost\_model(self, X\_train, y\_train):

"""

训练XGBoost分类模型

参数:

X\_train: 训练特征

y\_train: 训练标签

"""

params = {

'objective': 'binary:logistic',

'eval\_metric': 'auc',

'max\_depth': 6,

'eta': 0.3,

'subsample': 0.8,

'colsample\_bytree': 0.8,

'seed': self.random\_state

}

dtrain = xgb.DMatrix(X\_train, label=y\_train, feature\_names=self.feature\_names)

self.xgb\_model = xgb.train(params, dtrain, num\_boost\_round=100)

def train\_lstm\_model(self, X\_train, y\_train, timesteps=24):

"""

训练LSTM时间序列模型

参数:

X\_train: 训练特征(时间序列)

y\_train: 训练标签

timesteps: 时间步长

"""

# 重塑数据为LSTM需要的3D格式 [样本数, 时间步长, 特征数]

n\_samples = X\_train.shape[0] // timesteps

X\_reshaped = X\_train[:n\_samples\*timesteps].reshape(n\_samples, timesteps, X\_train.shape[1])

y\_reshaped = y\_train[:n\_samples\*timesteps].reshape(n\_samples, timesteps, 1)[:, -1, :]

# 构建LSTM模型

self.lstm\_model = Sequential([

LSTM(64, input\_shape=(timesteps, X\_train.shape[1]), return\_sequences=True),

Dropout(0.2),

LSTM(32),

Dropout(0.2),

Dense(1, activation='sigmoid')

])

self.lstm\_model.compile(

optimizer=Adam(learning\_rate=0.001),

loss='binary\_crossentropy',

metrics=['accuracy', 'AUC']

)

# 训练模型

self.lstm\_model.fit(

X\_reshaped, y\_reshaped,

epochs=20,

batch\_size=32,

validation\_split=0.2,

verbose=1

)

def identify\_trajectories(self, X):

"""

使用聚类识别患者轨迹

参数:

X: 患者特征数据

返回:

轨迹标签和聚类中心

"""

# 使用K-means聚类

trajectories = self.kmeans.fit\_predict(X)

centroids = self.kmeans.cluster\_centers\_

return trajectories, centroids

def evaluate\_model(self, X\_test, y\_test):

"""

评估模型性能

参数:

X\_test: 测试特征

y\_test: 测试标签

返回:

评估指标字典

"""

if self.xgb\_model is None:

raise ValueError("Model has not been trained yet.")

dtest = xgb.DMatrix(X\_test, feature\_names=self.feature\_names)

y\_pred = self.xgb\_model.predict(dtest)

return {

'auroc': roc\_auc\_score(y\_test, y\_pred),

'accuracy': np.mean((y\_pred > 0.5) == y\_test)

}

def explain\_model(self, X):

"""

使用SHAP解释模型预测

参数:

X: 解释用特征数据

返回:

SHAP值

"""

explainer = shap.TreeExplainer(self.xgb\_model)

shap\_values = explainer.shap\_values(X)

# 可视化特征重要性

plt.figure(figsize=(10, 6))

shap.summary\_plot(shap\_values, X, feature\_names=self.feature\_names)

plt.tight\_layout()

plt.savefig('feature\_importance.png', dpi=300)

plt.close()

return shap\_values

def visualize\_trajectories(self, centroids, feature\_names):

"""

可视化患者轨迹

参数:

centroids: 聚类中心

feature\_names: 特征名称列表

"""

plt.figure(figsize=(12, 8))

for i in range(self.n\_clusters):

plt.plot(centroids[i], label=f'Trajectory {i+1}')

plt.xticks(np.arange(len(feature\_names)), feature\_names, rotation=45)

plt.xlabel('Clinical Features')

plt.ylabel('Standardized Value')

plt.title('Patient Trajectory Clusters')

plt.legend()

plt.grid(True)

plt.tight\_layout()

plt.savefig('patient\_trajectories.png', dpi=300)

plt.close()

# 使用示例

if \_\_name\_\_ == "\_\_main\_\_":

# 1. 数据提取

extractor = MIMICExtractor("postgresql://user:password@localhost:5432/mimic")

sepsis\_patients = extractor.extract\_elderly\_sepsis\_patients(min\_age=65)

# 获取患者ID列表

subject\_ids = sepsis\_patients['subject\_id'].unique().tolist()

# 提取生命体征和实验室数据

vitals = extractor.extract\_vital\_signs(subject\_ids)

labs = extractor.extract\_lab\_results(subject\_ids)

# 2. 数据预处理

# (这里需要将提取的数据合并为适合建模的格式)

# 假设我们已经有了处理好的DataFrame 'clinical\_data'

# 3. 模型开发

model = SepsisTrajectoryModel(n\_clusters=5)

X, y = model.preprocess\_data(clinical\_data)

# 划分训练测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.2, random\_state=42, stratify=y

)

# 训练模型

model.train\_xgboost\_model(X\_train, y\_train)

# 评估模型

metrics = model.evaluate\_model(X\_test, y\_test)

print(f"Model Performance: AUC = {metrics['auroc']:.3f}, Accuracy = {metrics['accuracy']:.3f}")

# 识别患者轨迹

trajectories, centroids = model.identify\_trajectories(X\_train)

model.visualize\_trajectories(centroids, model.feature\_names)

# 解释模型

shap\_values = model.explain\_model(X\_test)

```

3. 模型部署代码（FastAPI）

```python

from fastapi import FastAPI, HTTPException

from pydantic import BaseModel

import pandas as pd

import numpy as np

import joblib

from typing import List

app = FastAPI()

# 加载预训练模型

model = joblib.load('sepsis\_trajectory\_model.pkl')

class PatientData(BaseModel):

subject\_id: str

age: float

heart\_rate: float

blood\_pressure: float

temperature: float

respiratory\_rate: float

lab\_results: dict

@app.post("/predict")

async def predict\_sepsis\_risk(data: PatientData):

try:

# 准备输入数据

input\_data = pd.DataFrame({

'subject\_id': [data.subject\_id],

'age': [data.age],

'heart\_rate': [data.heart\_rate],

'blood\_pressure': [data.blood\_pressure],

'temperature': [data.temperature],

'respiratory\_rate': [data.respiratory\_rate],

\*\*{f"lab\_{k}": [v] for k, v in data.lab\_results.items()}

})

# 预处理

X = model.preprocess\_data(input\_data)

# 预测

dmatrix = xgb.DMatrix(X, feature\_names=model.feature\_names)

prediction = model.xgb\_model.predict(dmatrix)[0]

# 确定轨迹

trajectory = model.kmeans.predict(X)[0]

return {

"subject\_id": data.subject\_id,

"sepsis\_risk": float(prediction),

"trajectory\_group": int(trajectory),

"risk\_level": "High" if prediction > 0.7 else "Medium" if prediction > 0.3 else "Low"

}

except Exception as e:

raise HTTPException(status\_code=500, detail=str(e))

@app.get("/model\_performance")

async def get\_model\_performance():

return {

"auroc": 0.92,

"accuracy": 0.85,

"last\_updated": "2023-11-15"

}

```

4. 模型监控代码

```python

import pandas as pd

import numpy as np

from datetime import datetime

import logging

from typing import Dict, List

class ModelMonitor:

def \_\_init\_\_(self, reference\_data: pd.DataFrame):

self.reference\_data = reference\_data

self.drift\_metrics = self.\_calculate\_reference\_stats()

self.alerts: List[Dict] = []

self.logger = self.\_setup\_logger()

def \_setup\_logger(self):

logger = logging.getLogger("ModelMonitor")

logger.setLevel(logging.INFO)

handler = logging.FileHandler("model\_monitor.log")

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

logger.addHandler(handler)

return logger

def \_calculate\_reference\_stats(self) -> Dict:

"""计算参考数据集的统计特征"""

stats = {}

for col in self.reference\_data.select\_dtypes(include=np.number).columns:

stats[col] = {

'mean': self.reference\_data[col].mean(),

'std': self.reference\_data[col].std(),

'median': self.reference\_data[col].median(),

'q1': self.reference\_data[col].quantile(0.25),

'q3': self.reference\_data[col].quantile(0.75)

}

return stats

def check\_data\_drift(self, new\_data: pd.DataFrame, threshold: float = 0.1) -> Dict:

"""

检查数据漂移

参数:

new\_data: 新数据

threshold: 漂移阈值

返回:

漂移检测结果字典

"""

drift\_results = {}

for col, ref\_stats in self.drift\_metrics.items():

if col in new\_data.columns:

current\_mean = new\_data[col].mean()

mean\_diff = abs(current\_mean - ref\_stats['mean'])

# 标准化差异

normalized\_diff = mean\_diff / ref\_stats['std']

drift\_detected = normalized\_diff > threshold

drift\_results[col] = {

'drift\_detected': drift\_detected,

'normalized\_diff': normalized\_diff,

'current\_mean': current\_mean,

'reference\_mean': ref\_stats['mean']

}

if drift\_detected:

alert\_msg = f"Data drift detected in {col}: diff={normalized\_diff:.2f}"

self.\_log\_alert(alert\_msg, "DRIFT")

return drift\_results

def check\_performance\_decay(self, y\_true: np.ndarray, y\_pred: np.ndarray,

threshold: float = 0.05) -> Dict:

"""

检查模型性能下降

参数:

y\_true: 真实标签

y\_pred: 预测标签

threshold: 性能下降阈值

返回:

性能指标字典

"""

auroc = roc\_auc\_score(y\_true, y\_pred)

accuracy = np.mean((y\_pred > 0.5) == y\_true)

ref\_auroc = 0.92 # 假设参考AUROC

auroc\_decay = ref\_auroc - auroc

decay\_detected = auroc\_decay > threshold

results = {

'auroc': auroc,

'accuracy': accuracy,

'auroc\_decay': auroc\_decay,

'decay\_detected': decay\_detected

}

if decay\_detected:

alert\_msg = f"Performance decay detected: AUROC dropped by {auroc\_decay:.3f}"

self.\_log\_alert(alert\_msg, "PERFORMANCE")

return results

def \_log\_alert(self, message: str, alert\_type: str):

"""记录警报"""

alert = {

'timestamp': datetime.now().isoformat(),

'type': alert\_type,

'message': message

}

self.alerts.append(alert)

self.logger.warning(message)

def get\_recent\_alerts(self, n: int = 10) -> List[Dict]:

"""获取最近的n个警报"""

return self.alerts[-n:]

```

5. 完整项目结构

```

elderly\_sepsis\_project/

│

├── data/

│ ├── extraction/ # 数据提取脚本

│ │ ├── mimic\_extractor.py

│ │ └── ehr\_connector.py

│ │

│ └── processed/ # 处理后的数据

│ ├── sepsis\_patients.csv

│ └── clinical\_features.parquet

│

├── models/

│ ├── development/ # 模型开发代码

│ │ ├── trajectory\_model.py

│ │ ├── xgboost\_model.py

│ │ └── lstm\_model.py

│ │

│ ├── trained\_models/ # 训练好的模型

│ │ ├── sepsis\_xgboost.pkl

│ │ └── sepsis\_lstm.h5

│ │

│ └── evaluation/ # 模型评估

│ ├── performance\_metrics.py

│ └── explainability.py

│

├── api/ # API部署代码

│ ├── main.py # FastAPI应用

│ └── schemas.py # 数据模型

│

├── monitoring/ # 模型监控

│ ├── drift\_detection.py

│ └── performance\_tracking.py

│

├── notebooks/ # Jupyter笔记本

│ ├── data\_exploration.ipynb

│ └── model\_training.ipynb

│

├── tests/ # 测试

│ ├── unit\_tests/

│ └── integration\_tests/

│

├── config/ # 配置文件

│ ├── database.yaml

│ └── model\_params.yaml

│

├── requirements.txt # Python依赖

└── README.md # 项目文档