import os import json import time import logging import threading import pickle from datetime import datetime, timedelta from flask import Flask, request, jsonify from pymongo import MongoClient import redis import numpy as np import pandas as pd from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor from sklearn.linear_model import ElasticNet from sklearn.preprocessing import StandardScaler, OneHotEncoder from sklearn.compose import ColumnTransformer from sklearn.pipeline import Pipeline from sklearn.model_selection import train_test_split, cross_val_score from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score import joblib from dotenv import load_dotenv

Load environment variables

load_dotenv()

Configure logging

logging.basicConfig(level=logging.INFO, format=‘%(asctime)s - %(name)s - %(levelname)s - %(message)s’) logger = logging.getLogger(**name**)

Initialize Flask app

app = Flask(**name**)

Connect to MongoDB

mongo_uri = os.getenv(“MONGODB_URI”, “mongodb://localhost:27017/amgf”) mongo_client = MongoClient(mongo_uri) db = mongo_client.amgf opportunities_collection = db.opportunities predictions_collection = db.predictions historical_data_collection = db.historical_data metrics_collection = db.metrics

Connect to Redis

redis_uri = os.getenv(“REDIS_URI”, “redis://localhost:6379”) redis_client = redis.from_url(redis_uri)

Constants

MODEL_UPDATE_INTERVAL = int(os.getenv(“MODEL_UPDATE_INTERVAL”, “86400”)) # Default: update models daily PREDICTION_TTL = int(os.getenv(“PREDICTION_TTL”, “43200”)) # Default: 12 hours MODELS_DIR = os.getenv(“MODELS_DIR”, “./models”)

Ensure models directory exists

os.makedirs(MODELS_DIR, exist_ok=True)

class ROIPredictor: “““” Main class for predicting ROI of opportunities. “““”

```
def __init__(self):
    self.models = {
        "service": None,
        "service": None,
        "content": None,
        "marketplace": None
    }
    self.preprocessors = {
        "service": None,
        "content": None,
        "marketplace": None
    }
    self.feature_importances = {
        "service": None,
        "content": None,
        "marketplace": None
    }
    self.model_metrics = {
        "service": None,
        "content": None,
        "marketplace": None
    }
    self.last_update = {
        "service": None,
        "content": None,
        "marketplace": None
    }

    # Load existing models if available
    self.load_models()

def load_models(self):
    """Load existing models from disk."""
    for category in self.models.keys():
        model_path = os.path.join(MODELS_DIR, f"{category}_model.joblib")
        preprocessor_path = os.path.join(MODELS_DIR, f"
{category}_preprocessor.joblib")

        if os.path.exists(model_path) and
os.path.exists(preprocessor_path):
            try:
                self.models[category] = joblib.load(model_path)
                self.preprocessors[category] =
joblib.load(preprocessor_path)

                # Load model metadata
                metadata = metrics_collection.find_one({"metric": f"
{category}_model"})
                if metadata:
                    self.feature_importances[category] =
metadata.get("feature_importances")
                    self.model_metrics[category] = metadata.get("metrics")
                    self.last_update[category] =
metadata.get("last_update")

                logger.info(f"Loaded existing {category} model")
            except Exception as e:
                logger.error(f"Error loading {category} model: {str(e)}")
                self.models[category] = None
                self.preprocessors[category] = None
        else:
            logger.info(f"No existing {category} model found")

def update_models(self):
    """Update all prediction models."""
    logger.info("Updating prediction models")

    for category in self.models.keys():
        try:
            self.update_model(category)
        except Exception as e:
            logger.error(f"Error updating {category} model: {str(e)}")

    logger.info("Model update complete")

def update_model(self, category):
    """Update prediction model for a specific category."""
    logger.info(f"Updating {category} model")
    logger.info(f"Updating {category} model")

    # Get historical data
    historical_data = list(historical_data_collection.find({"category": 
category}))

    if len(historical_data) < 10:
        logger.warning(f"Not enough historical data to train {category} 
model (found {len(historical_data)} records)")
        return

    # Convert to DataFrame
    df = pd.DataFrame(historical_data)

    # Define features and target
    X = df.drop(["_id", "roi", "actual_revenue", "category"], axis=1, 
errors='ignore')
    y = df["roi"]

    # Define feature types
    categorical_features = [
        "platform", "opportunity_type", "competition_level",
        "source", "day_of_week", "time_of_day"
    ]
    categorical_features = [f for f in categorical_features if f in 
X.columns]

    numerical_features = [
        "estimated_value", "implementation_time", "skills_required_count",
        "hour_of_day", "day_of_month", "month", "trend_score"
    ]
    numerical_features = [f for f in numerical_features if f in X.columns]

    # Create preprocessor
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', StandardScaler(), numerical_features),
            ('cat', OneHotEncoder(handle_unknown='ignore'), 
categorical_features)
        ]
    )

    # Create model pipeline
    if category == "service":
        model = RandomForestRegressor(
            n_estimators=100,
            max_depth=10,
            min_samples_split=5,
            min_samples_leaf=2,
            random_state=42
        )
    elif category == "content":
        model = GradientBoostingRegressor(
            n_estimators=100,
            learning_rate=0.1,
            max_depth=5,
            random_state=42
        )
    else:  # marketplace
        model = ElasticNet(
            alpha=0.1,
            l1_ratio=0.5,
            random_state=42
        )

    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('model', model)
    ])

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(X, y, 
test_size=0.2, random_state=42)

    # Train model
    # Train model
    pipeline.fit(X_train, y_train)

    # Evaluate model
    y_pred = pipeline.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    # Cross-validation
    cv_scores = cross_val_score(pipeline, X, y, cv=5,
scoring='neg_mean_squared_error')
    cv_rmse = np.sqrt(-cv_scores.mean())

    # Get feature importances
    if hasattr(model, 'feature_importances_'):
        # For tree-based models
        feature_names = (
            numerical_features +
            list(pipeline.named_steps['preprocessor']
                .named_transformers_['cat']
                .get_feature_names_out(categorical_features))
        )
        feature_importances = dict(zip(feature_names,
model.feature_importances_))
    else:
        # For linear models
        feature_importances = None

    # Save model and preprocessor
    self.models[category] = model
    self.preprocessors[category] = preprocessor
    self.feature_importances[category] = feature_importances
    self.model_metrics[category] = {
        "mse": mse,
        "mae": mae,
        "r2": r2,
        "cv_rmse": cv_rmse
    }
    self.last_update[category] = datetime.utcnow().isoformat()

    # Save to disk
    joblib.dump(model, os.path.join(MODELS_DIR, f"
{category}_model.joblib"))
    joblib.dump(preprocessor, os.path.join(MODELS_DIR, f"
{category}_preprocessor.joblib"))

    # Update metadata in MongoDB
    metrics_collection.update_one(
        {"metric": f"{category}_model"},
        {"$set": {
            "feature_importances": feature_importances,
            "metrics": self.model_metrics[category],
            "last_update": self.last_update[category],
            "training_samples": len(X_train)
        }},
        upsert=True
    )

    logger.info(f"Updated {category} model: MSE={mse:.4f}, MAE={mae:.4f},
R2={r2:.4f}, CV_RMSE={cv_rmse:.4f}")

def predict_roi(self, opportunity):
    """Predict ROI for a single opportunity."""
    category = opportunity.get("category")

    if not category or category not in self.models:
        raise ValueError(f"Invalid category: {category}")

    if not self.models[category] or not self.preprocessors[category]:
        raise ValueError(f"Model for category {category} not available")

    # Prepare features
    features = self.prepare_features(opportunity)
    # Convert to DataFrame
    df = pd.DataFrame([features])

    # Make prediction
    try:
        # Transform features
        X = self.preprocessors[category].transform(df)

        # Predict
        prediction = self.models[category].predict(X)[0]

        # Calculate confidence interval
        # For simplicity, we'll use a fixed percentage of the prediction
        # In a real implementation, this would be more sophisticated
        confidence = 0.2  # 20% confidence interval
        lower_bound = prediction * (1 - confidence)
        upper_bound = prediction * (1 + confidence)

        # Calculate risk score
        risk_score = self.calculate_risk_score(opportunity, prediction)

        # Calculate implementation difficulty
        implementation_difficulty =
self.calculate_implementation_difficulty(opportunity)

        # Calculate time-to-revenue
        time_to_revenue = self.calculate_time_to_revenue(opportunity)

        result = {
            "opportunity_id": opportunity.get("id"),
            "predicted_roi": float(prediction),
            "confidence_interval": {
                "lower": float(lower_bound),
                "upper": float(upper_bound)
            },
            "risk_score": risk_score,
            "implementation_difficulty": implementation_difficulty,
            "time_to_revenue": time_to_revenue,
            "timestamp": datetime.utcnow().isoformat(),
            "ttl": int(time.time()) + PREDICTION_TTL
        }

        # Store prediction in MongoDB
        predictions_collection.update_one(
            {"opportunity_id": opportunity.get("id")},
            {"$set": result},
            upsert=True
        )

        return result
    except Exception as e:
        logger.error(f"Error making prediction: {str(e)}")
        raise

def prepare_features(self, opportunity):
    """Prepare features for prediction."""
    # Extract basic features
    features = {
        "platform": opportunity.get("platform", "unknown"),
        "opportunity_type": opportunity.get("opportunity_type",
"unknown"),
        "estimated_value": float(opportunity.get("estimated_value", 0)),
        "implementation_time":
float(opportunity.get("implementation_time", 0)),
        "competition_level": opportunity.get("competition_level",
"medium"),
        "source": opportunity.get("source", "unknown"),
        "skills_required_count": len(opportunity.get("skills_required",
[])),
    }

    # Add time-based features
    now = datetime.utcnow()
    features["hour_of_day"] = now.hour
```

```
        features["hour_of_day"] = now.hour
        features["day_of_week"] = now.weekday()
        features["day_of_month"] = now.day
        features["month"] = now.month

        # Simplify time of day
        if 5 <= now.hour < 12:
            features["time_of_day"] = "morning"
        elif 12 <= now.hour < 17:
            features["time_of_day"] = "afternoon"
        elif 17 <= now.hour < 22:
            features["time_of_day"] = "evening"
        else:
            features["time_of_day"] = "night"

        # Add trend score (simulated)
        features["trend_score"] = np.random.uniform(0, 1)

        return features

    def calculate_risk_score(self, opportunity, predicted_roi):
        """Calculate risk score for an opportunity."""
        # Base risk on competition level
        competition_level = opportunity.get("competition_level", "medium")
        if competition_level == "low":
            base_risk = 0.3
        elif competition_level == "medium":
            base_risk = 0.5
        else:  # high
            base_risk = 0.7

        # Adjust based on implementation time
        implementation_time = float(opportunity.get("implementation_time", 0))
        time_factor = min(implementation_time / 8, 1)  # Normalize to 0-1

        # Adjust based on predicted ROI
        roi_factor = 1 / (1 + predicted_roi)  # Higher ROI = lower risk

        # Calculate final risk score (0-1)
        risk_score = (base_risk * 0.5) + (time_factor * 0.3) + (roi_factor *
    0.2)
        risk_score = min(max(risk_score, 0), 1)  # Ensure between 0 and 1

        return float(risk_score)

    def calculate_implementation_difficulty(self, opportunity):
        """Calculate implementation difficulty for an opportunity."""
        # Base difficulty on implementation time
        implementation_time = float(opportunity.get("implementation_time", 0))

        # Normalize to 1-5 scale
        difficulty = 1 + min(implementation_time / 2, 4)

        # Adjust based on skills required
        skills_required = len(opportunity.get("skills_required", []))
        skills_factor = min(skills_required / 3, 1)  # Normalize to 0-1

        # Adjust difficulty
        difficulty += skills_factor

        # Ensure between 1 and 5
        difficulty = min(max(difficulty, 1), 5)

        return float(difficulty)

    def calculate_time_to_revenue(self, opportunity):
        """Calculate time to revenue for an opportunity."""
        # Base on implementation time
        implementation_time = float(opportunity.get("implementation_time", 0))

        # Different categories have different time-to-revenue patterns
        category = opportunity.get("category")
        if category == "service":
            # Services often have a delay after implementation
            time_to_revenue = implementation_time + 2
            time_to_revenue = implementation_time + 2
        elif category == "content":
            # Content can take longer to monetize
            time_to_revenue = implementation_time + 4
        else:  # marketplace
            # Marketplace items can sell quickly after listing
            time_to_revenue = implementation_time + 1

        return float(time_to_revenue)

    def batch_predict(self, opportunities):
        """Predict ROI for multiple opportunities."""
        results = []

        for opportunity in opportunities:
            try:
                prediction = self.predict_roi(opportunity)
                results.append(prediction)
            except Exception as e:
                logger.error(f"Error predicting ROI for opportunity
    {opportunity.get('id')}: {str(e)}")
                # Add error result
                results.append({
                    "opportunity_id": opportunity.get("id"),
                    "error": str(e),
                    "timestamp": datetime.utcnow().isoformat()
                })

        return results

    def get_prediction(self, opportunity_id):
        """Get existing prediction for an opportunity."""
        prediction = predictions_collection.find_one({"opportunity_id":
    opportunity_id})

        if prediction:
            # Convert ObjectId to string
            if "_id" in prediction:
                prediction["_id"] = str(prediction["_id"])

            # Check if prediction is still valid
            if "ttl" in prediction and prediction["ttl"] < int(time.time()):
                # Prediction has expired
                return None

            return prediction

        return None
```

Initialize predictor

predictor = ROIPredictor()

API Routes

@app.route(‘/health’, methods=[‘GET’]) def health_check(): “““Health check endpoint.”“” return jsonify({“status”: “healthy”, “timestamp”: datetime.utcnow().isoformat()})

@app.route(‘/predict’, methods=[‘POST’]) def predict_opportunity(): “““Predict ROI for a single opportunity.”“” try: opportunity = request.json

```
    if not opportunity:
        return jsonify({
            "status": "error",
            "message": "No opportunity data provided",
            "timestamp": datetime.utcnow().isoformat()
        }), 400

    # Check if prediction already exists
    existing_prediction = predictor.get_prediction(opportunity.get("id"))
    if existing_prediction:
        return jsonify({
            "status": "success",
            "prediction": existing_prediction,
            "source": "cache",
            "timestamp": datetime.utcnow().isoformat()
        })

    # Make prediction
    prediction = predictor.predict_roi(opportunity)

    return jsonify({
        "status": "success",
        "prediction": prediction,
        "source": "new",
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error predicting ROI: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/predict/batch’, methods=[‘POST’]) def predict_batch(): “““Predict ROI for multiple opportunities.”“” try: opportunities = request.json

```
    if not opportunities or not isinstance(opportunities, list):
        return jsonify({
            "status": "error",
            "message": "Invalid opportunities data",
            "timestamp": datetime.utcnow().isoformat()
        }), 400

    # Make predictions
    predictions = predictor.batch_predict(opportunities)

    return jsonify({
        "status": "success",
        "predictions": predictions,
        "count": len(predictions),
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error batch predicting ROI: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route(‘/models/update’, methods=[‘POST’]) def update_models(): “““Trigger model update.”“” try: predictor.update_models()

```
    return jsonify({
        "status": "success",
        "message": "Model update triggered",
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error updating models: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

@app.route('/models/status', methods=['GET']) def get_model_status(): “““Get status of prediction models.”“” try: status = {}

```
    for category in predictor.models.keys():
        status[category] = {
            "available": predictor.models[category] is not None,
            "last_update": predictor.last_update[category],
            "metrics": predictor.model_metrics[category],
            "feature_importances": predictor.feature_importances[category]
        }

    return jsonify({
        "status": "success",
        "models": status,
        "timestamp": datetime.utcnow().isoformat()
    })
except Exception as e:
    logger.error(f"Error getting model status: {str(e)}")
    return jsonify({
        "status": "error",
        "message": str(e),
        "timestamp": datetime.utcnow().isoformat()
    }), 500
```

def run_model_updater(): “““Run the model updater in a separate thread.”“” logger.info(“Starting model updater”)

```
while True:
    try:
        # Update models
        predictor.update_models()

        # Sleep until next update
        time.sleep(MODEL_UPDATE_INTERVAL)
    except Exception as e:
        logger.error(f"Error in model updater: {str(e)}")
        time.sleep(60)  # Sleep for a minute before retrying
```

if **name** == '**main**': # Start model updater in a separate thread updater_thread = threading.Thread(target=run_model_updater) updater_thread.daemon = True updater_thread.start()

```
# Start Flask app
app.run(host='0.0.0.0', port=5000)
```