Mills XGBoost System - Architecture and Implementation Details

Project Overview

This document provides a comprehensive explanation of the Mills XGBoost System architecture, implementation details, and the relationships between components.

The project implements a production-ready system for:

1. Training XGBoost regression models to predict mill performance metrics (PSI80, FR200)
2. Applying Bayesian optimization techniques to find optimal mill parameter settings
3. Exposing these capabilities through a FastAPI web service

System Architecture

Directory Structure

```
mills-xgboost/
├── app/
│   ├── database/
│   │   ├── __init__.py
│   │   └── db_connector.py     # PostgreSQL database connector
│   ├── models/
│   │   ├── __init__.py
│   │   ├── data_processor.py   # Data preprocessing pipeline
│   │   └── xgboost_model.py    # XGBoost model implementation
│   ├── optimization/
│   │   ├── __init__.py
│   │   └── bayesian_opt.py     # Bayesian optimization module
```

```
│   ├── api/
│   │   ├── __init__.py
│   │   ├── endpoints.py        # API endpoint definitions
│   │   └── schemas.py          # Pydantic models for request/response
│   ├── __init__.py
│   └── main.py                 # FastAPI application
├── config/
│   └── settings.py             # Configuration settings
├── logs/                       # Directory for log files
├── models/                     # Directory for saved models
├── optimization_results/       # Directory for optimization results
├── requirements.txt            # Project dependencies
├── README.md                   # Project documentation
├── run_api.py                  # Script to run the FastAPI server
└── test_system.py              # Test script for system validation
```

Component Relationships and Data Flow

1. **Database Connection**: `db_connector.py` connects to PostgreSQL to retrieve mill sensor data and ore quality data.
2. **Data Processing**: `data_processor.py` preprocesses the raw data for model training and prediction.
3. **Model Training**: `xgboost_model.py` trains XGBoost models using the processed data.
4. **Parameter Optimization**: `bayesian_opt.py` uses trained models to find optimal mill parameters.
5. **API Interface**: FastAPI endpoints in `endpoints.py` expose the functionality through RESTful API.

```
[PostgreSQL DB] → [MillsDataConnector] → [DataProcessor] → [MillsXGBoostModel]
                                                                    ↓
[FastAPI] ← [API Endpoints] ← [MillBayesianOptimizer] ← [Trained Model]
```

Detailed Component Explanations

1. Database Connector (`db_connector.py`)

Purpose

Connect to PostgreSQL database to retrieve mill sensor data and ore quality data, process the data, and join them on timestamps.

Key Classes and Methods

- `MillsDataConnector`: Main class for database connections and data retrieval.
  - `__init__(self, host, port, dbname, user, password)`: Initialize database connection parameters.
  - `connect(self)`: Create SQLAlchemy engine and connection.
  - `get_mill_data(self, mill_number, start_date, end_date)`: Fetch mill sensor data.
  - `get_ore_quality_data(self, start_date, end_date)`: Fetch ore quality lab data.
  - `process_mill_data(self, df, resample_freq='1min')`: Process mill data (resampling, smoothing).
  - `process_ore_quality_data(self, df, resample_freq='1min')`: Process ore quality data.
  - `join_mill_and_ore_data(self, mill_df, ore_df)`: Join the two datasets on timestamp.
  - `get_combined_data(self, mill_number, start_date, end_date, resample_freq='1min')`: Get completely processed data ready for modeling.

Key Implementation Details

- Uses SQLAlchemy for database connections
- Handles case-sensitive column names from PostgreSQL
- Implements data smoothing with rolling window averages
- Properly resamples data to 1-minute frequency
- Joins mill sensor data with ore quality data based on timestamps

2. Data Processor (`data_processor.py`)

Purpose

Preprocess mill data for XGBoost modeling, including filtering, cleaning, scaling, and feature selection.

Key Classes and Methods

- `DataProcessor`: Encapsulates data preprocessing steps.
  - `preprocess(self, df, features, target_col)`: Main method to preprocess data.
  - `_filter_data(self, df, features, target_col)`: Filter data for required features and target.
  - `_clean_data(self, df)`: Clean data by removing null values and outliers.
  - `_scale_features(self, X)`: Scale features using StandardScaler.
  - `transform_new_data(self, df, scaler)`: Transform new data for prediction.

Key Implementation Details

- Uses scikit-learn's StandardScaler for feature scaling
- Handles missing values and outliers
- Provides methods for both training preprocessing and inference preprocessing

3. XGBoost Model (`xgboost_model.py`)

Purpose

Implement a production-ready XGBoost regression model for mill data with methods for training, prediction, evaluation, and model persistence.

Key Classes and Methods

- `MillsXGBoostModel`: Main class for XGBoost modeling.
  - `__init__(self, features=None, target_col='PSI80')`: Initialize model with features and target column.
  - `train(self, X_train, X_test, y_train, y_test, scaler=None, params=None)`: Train the model.
  - `predict(self, data)`: Make predictions with the model.
  - `calculate_metrics(self, y_true, y_pred)`: Calculate performance metrics.

- `save_model(self, directory='models')`: Save model, scaler, and metadata to disk.
- `load_model(self, model_path, scaler_path)`: Load model and scaler from disk.
- `get_feature_importance(self)`: Get and format feature importance.

Key Implementation Details

- Uses XGBoost's early stopping to prevent overfitting
- Logs training metrics and feature importance without plots
- Supports prediction from both DataFrame and dictionary inputs
- Serializes model and scaler for persistence
- Provides detailed metrics calculation

4. Bayesian Optimization (`bayesian_opt.py`)

Purpose

Implement Bayesian optimization to tune mill parameters for optimal performance using a trained XGBoost model.

Key Classes and Methods

- `MillBayesianOptimizer`: Main class for Bayesian optimization.
  - `__init__(self, xgboost_model, target_col='PSI80', maximize=True)`: Initialize with model and objective.
  - `_black_box_function(self, **kwargs)`: Black box function that predicts outcome using XGBoost model.
  - `set_parameter_bounds(self, pbounds=None, data=None)`: Set parameter search bounds.
  - `set_constraints(self, constraints=None)`: Set constraints on parameter combinations.
  - `_check_constraints(self, params)`: Check if parameters meet all constraints.
  - `optimize(self, init_points=5, n_iter=25, acq='ei', kappa=2.5, xi=0.0, save_dir=None)`: Run optimization.
  - `_save_optimization_results(self, directory)`: Save optimization results to disk.

- `recommend_parameters(self, n_recommendations=3)`: Get top N parameter recommendations.

Key Implementation Details

- Uses the bayesian-optimization library for Gaussian Process optimization
- Supports both maximization and minimization objectives
- Allows parameter bounds to be derived from data min/max values
- Supports parameter constraints (e.g., WaterMill ≥ 1.5 × WaterZumpf)
- Records optimization history for analysis
- Provides parameter recommendations with predicted performance

5. FastAPI Application (`main.py`, `endpoints.py`, `schemas.py`)

Purpose

Expose model training, prediction, and optimization capabilities through a RESTful API interface.

Key Components

API Schemas (`schemas.py`)

- Defines Pydantic models for request/response validation:
  - `DatabaseConfig`: Database connection parameters
  - `TrainingParameters`: XGBoost training parameters
  - `TrainingRequest`: Model training request
  - `ModelMetrics`: Model performance metrics
  - `TrainingResponse`: Model training response
  - `PredictionRequest`: Prediction request
  - `PredictionResponse`: Prediction response
  - `OptimizationRequest`: Parameter optimization request
  - `ParameterRecommendation`: Single parameter recommendation
  - `OptimizationResponse`: Optimization results response

API Endpoints (`endpoints.py`)

- Implements API endpoints:

    - `POST /train`: Train a new XGBoost model
    - `POST /predict`: Make predictions with a trained model
    - `POST /optimize`: Run Bayesian optimization on mill parameters
    - `GET /models`: List all available models

FastAPI Application (`main.py`)

- Configures the FastAPI application:
    - CORS middleware for cross-origin requests
    - Request timing middleware
    - API router inclusion
    - Health check and information endpoints
    - Exception handlers for graceful error responses

Key Implementation Details

- Uses FastAPI for high-performance API with automatic schema validation
- Implements in-memory model storage (can be replaced with database in production)
- Provides detailed error handling and logging
- Includes timing middleware for performance monitoring

6. Configuration (`settings.py`)

Purpose

Manage application configuration settings.

Key Components

- `Settings` class with:
    - Application information
    - API settings
    - Database connection parameters
    - Default paths for models, logs, etc.
    - Data processing settings
    - Feature sets for different target variables
    - Logging settings

Key Implementation Details

- Uses Pydantic for configuration validation
- Supports environment variables override
- Sets case_sensitive=True for proper handling of PostgreSQL case-sensitive columns

7. Testing and Deployment

Test Script (`test_system.py`)

- Tests all components of the system:
  - Database connection and data retrieval
  - Model training
  - Prediction
  - Bayesian optimization
- Includes detailed logging of test progress

API Runner (`run_api.py`)

- Script to start the FastAPI server
- Configures command-line arguments
- Sets up logging
- Creates necessary directories

Data Flow and Integration

Training Flow

1. User sends a training request via the API
2. API endpoint calls the database connector to fetch data
3. Data processor prepares the data for modeling
4. XGBoost model is trained and evaluated
5. Model, scaler, and metadata are saved
6. Training results are returned via API

Prediction Flow

1. User sends a prediction request with model ID and input data
2. API endpoint retrieves the model from storage
3. Input data is transformed using the stored scaler
4. Model makes predictions
5. Results are returned via API

Optimization Flow

1. User sends an optimization request with model ID and parameter bounds
2. API endpoint retrieves the model from storage
3. Bayesian optimizer is initialized with the model as objective function
4. Optimizer runs through specified iterations
5. Optimization results and recommendations are returned via API

Design Decisions and Implementation Highlights

1. Modular Architecture

- Separation of concerns between database access, data processing, modeling, optimization, and API
- Each component has a clear responsibility and interface
- Easy to maintain, test, and extend

2. Production Readiness

- No extraneous plotting in production code
- Comprehensive logging instead of plots
- Proper error handling and validation
- Configuration management
- Scalable directory structure

3. PostgreSQL Integration

- Direct connection to PostgreSQL database
- Proper handling of case-sensitive column names
- Efficient data retrieval and processing
- Joining mill sensor data with ore quality lab data

4. Model Persistence

- Serialization of model, scaler, and metadata
- Support for loading and saving models
- Model version tracking

5. Parameter Optimization

- Flexible Bayesian optimization framework
- Support for parameter bounds and constraints
- Multiple optimization strategies (maximize/minimize)
- Parameter recommendations with predicted performance

6. API Design

- RESTful API with clear endpoints
- Proper request/response validation using Pydantic
- Comprehensive error handling
- Performance monitoring with timing middleware

Conclusion

The Mills XGBoost System is a comprehensive solution for mill performance prediction and parameter optimization. The modular architecture and clean separation of concerns make it easy to maintain and extend. The system is designed for production deployment with proper error handling, logging, and no extraneous plotting during production runs.

The key highlights of the implementation are:

1. Direct PostgreSQL integration with proper handling of case-sensitive columns

2. Production-ready XGBoost modeling with comprehensive logging
3. Flexible Bayesian optimization framework for parameter tuning
4. Clean API design with proper validation and error handling
5. Modular architecture with clear separation of concerns