- Mills XGBoost System - Architecture and Implementation Details
  - Project Overview
  - System Architecture
    - Directory Structure
    - Main API Integration
    - Component Relationships and Data Flow
  - Detailed Component Explanations
    - 1. Database Connector (db_connector.py)
      - Purpose
      - Key Classes and Methods
      - Key Implementation Details
    - 2. Data Processor (data_processor.py)
      - Purpose
      - Key Classes and Methods
      - Key Implementation Details
    - 3. XGBoost Model (xgboost_model.py)
      - Purpose
      - Key Classes and Methods
      - Key Implementation Details
    - 4. Bayesian Optimization with Optuna (endpoints.py)
      - Purpose
      - Key Classes and Methods
      - Code Example: BlackBoxFunction Implementation
      - Code Example: Optuna Optimization
      - Key Implementation Details
    - 5. FastAPI Application (main.py, endpoints.py, schemas.py)
      - Purpose
      - Key Components
      - Key Schema Implementations
      - Example Endpoint Implementation: /predict
      - Example Endpoint Implementation: /optimize (Partial)
      - Key Implementation Details
    - 6. Configuration (settings.py)
      - Purpose
      - Key Components
      - Key Implementation Details
    - 7. Testing and Deployment

- Data Flow and Integration
  - Training Flow
  - Prediction Flow
  - Optimization Flow
- Design Decisions and Implementation Highlights
  - 1. Modular Architecture
  - 2. Production Readiness
  - 3. PostgreSQL Integration
  - 4. Model Persistence
  - 5. Parameter Optimization
  - 6. API Design
- Conclusion
- Frontend Integration
  - React Frontend Integration
  - Key Frontend Components
  - API Integration Hooks
    - Prediction Hook Implementation
  - Component Usage Example
- Conclusion
  - Key System Highlights

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 integrated with the main API

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 fixed log files
(mills_xgboost_server.log)
├── models/                     # Directory for saved models with mill-specific
names
├── optimization_results/       # Directory for optimization results
├── requirements.txt            # Project dependencies
├── README.md                   # Project documentation
├── run_api.py                  # Script to run the standalone FastAPI server
└── test_system.py              # Test script for system validation
```

Main API Integration

The mills-xgboost system is cleanly integrated into the main API via the `mills_ml_router.py` adapter:

```
ok_dashboard/
├── python/
│   ├── api.py                  # Main API with minimal integration code
│   ├── mills_ml_router.py      # Integration adapter for mills-xgboost
│   └── mills-xgboost/          # Mills XGBoost system (as detailed above)
```

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 with time-ordered train-test split.
3. **Model Training**: `xgboost_model.py` trains XGBoost models using the processed data and saves models with mill-specific filenames.
4. **Parameter Optimization**: `bayesian_opt.py` uses trained models to find optimal mill parameters.
5. **API Integration**: `mills_ml_router.py` integrates the ML functionality with the main API at `/api/v1/ml/*` prefix.

```
[PostgreSQL DB] → [MillsDataConnector] → [DataProcessor] → [MillsXGBoostModel]
                                                                    ↓
[Main API] ← [mills_ml_router.py] ← [API Endpoints] ← [Trained Model/Optimizer]
```

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 with Optuna (`endpoints.py`)

Purpose

Implement Bayesian optimization using Optuna to find optimal mill parameters that maximize or minimize a target performance metric using a trained XGBoost model.

Key Classes and Methods

- `BlackBoxFunction`: Main class representing the model prediction function to optimize.
  - `__init__(self, model_id: str, xgb_model=None, maximize: bool = True)`: Initialize with model and objective direction.
  - `_load_model(self)`: Load XGBoost model, scaler, and metadata from disk.
  - `set_parameter_bounds(self, parameter_bounds: Dict[str, List[float]])`: Set min/max bounds for each parameter.
  - `__call__(self, **features) -> float`: Predict outcome using XGBoost model for given parameters.
- `optimize_with_optuna`: Function that orchestrates the Optuna optimization process.
  - Parameters: `black_box_func, n_trials=100, timeout=None`
  - Returns: `Tuple[Dict[str, float], float, optuna.study.Study]` (best parameters, best value, study object)

Code Example: BlackBoxFunction Implementation

```
class BlackBoxFunction:
    """A black box function that loads an XGBoost model and predicts output based
on input features."""

    def __init__(self, model_id: str, xgb_model=None, maximize: bool = True):
        self.model_id = model_id
        self.maximize = maximize
        self.xgb_model = xgb_model
        self.parameter_bounds = None
```

```
        # If model is not provided, load it
        if self.xgb_model is None:
            self._load_model()

    def set_parameter_bounds(self, parameter_bounds: Dict[str, List[float]]):
        """Set bounds for the parameters to optimize."""
        self.parameter_bounds = parameter_bounds

        # Validate that bounds are provided for features in the model
        for feature in parameter_bounds:
            if feature not in self.features:
                logger.warning(f"Parameter bound provided for feature '{feature}'
which is not in the model features.")

    def __call__(self, **features) -> float:
        """Predict the target value based on the provided features."""
        # Create a dictionary with all features
        input_data = {feature: features.get(feature, 0.0) for feature in
self.features}

        # Make prediction
        prediction = self.xgb_model.predict(input_data)[0]

        # Return prediction (negated if minimizing)
        return prediction if self.maximize else -prediction
```

Code Example: Optuna Optimization

```
def optimize_with_optuna(
    black_box_func: BlackBoxFunction,
    n_trials: int = 100,
    timeout: int = None
) -> Tuple[Dict[str, float], float, optuna.study.Study]:
    """Optimize the black box function using Optuna."""
    # Define the objective function for Optuna
    def objective(trial):
        # Suggest values for each parameter within bounds
        params = {}
        for feature, bounds in black_box_func.parameter_bounds.items():
            params[feature] = trial.suggest_float(feature, bounds[0], bounds[1])

        # Call the black box function
        return black_box_func(**params)

    # Create and run the study
    direction = "maximize" if black_box_func.maximize else "minimize"
    study = optuna.create_study(direction=direction)
    study.optimize(objective, n_trials=n_trials, timeout=timeout)

    # Get best parameters and value
    best_params = study.best_params
    best_value = study.best_value
```

```
    return best_params, best_value, study
```

Key Implementation Details

- Uses Optuna for Bayesian optimization with Tree-structured Parzen Estimator (TPE)
- Supports both maximization and minimization objectives
- Provides multiple recommendations ranked by performance, not just the best solution
- Automatically exports optimization trial history to CSV files
- Returns best parameters, performance value, and the full optimization study object
- Dynamically loads models from memory or disk based on availability
- Handles parameter bound validation against model features

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
  - `PredictionRequest`: Request model for model predictions
  - `PredictionResponse`: Response model with prediction results
  - `OptimizationRequest`: Request model for parameter optimization
  - `OptimizationResponse`: Response model with optimization results and recommendations
  - `Recommendation`: Model for individual parameter recommendations

Key Schema Implementations

```python
class PredictionRequest(BaseModel):
    model_id: str
    data: Dict[str, float]

class PredictionResponse(BaseModel):
    model_id: str
    prediction: float
    target_col: str

class OptimizationRequest(BaseModel):
    model_id: str
    parameter_bounds: Optional[Dict[str, List[float]]] = None
    n_iter: Optional[int] = 25
    init_points: Optional[int] = 5
    maximize: bool = True

class Recommendation(BaseModel):
    params: Dict[str, float]
    predicted_value: float

class OptimizationResponse(BaseModel):
    best_params: Dict[str, float]
    best_target: float
    target_col: str
    maximize: bool
    recommendations: List[Recommendation]
    model_id: str
```

- `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 with mill-specific naming
    - Models are saved with mill number in the filename (e.g., `xgboost_PSI80_mill6_model.json`)
  - `POST /predict`: Make predictions using a trained model
    - Takes model_id and feature data as input

        - Returns predicted target value
    - `POST /optimize`: Optimize mill parameters using Bayesian Optimization
        - Uses Optuna to find optimal parameters that maximize/minimize target metric
        - Returns best parameters, predicted performance, and multiple recommendations
    - `GET /models`: List all available models
    - `GET /info`: API system information

Example Endpoint Implementation: `/predict`

```
@router.post("/predict", response_model=PredictionResponse)
async def predict(request: PredictionRequest):
    """Make predictions with a trained XGBoost model"""
    try:
        # Log prediction request
        logger.info(f"Prediction request received for model {request.model_id}")

        # Check if model exists
        if request.model_id not in models_store:
            # Try to load from disk
            try:
                model_name = request.model_id.split('.')[0]
                project_root =
os.path.abspath(os.path.join(os.path.dirname(__file__), '..', '..'))
                models_dir = os.path.join(project_root, 'models')
                model_path = os.path.join(models_dir, f"{model_name}_model.json")
                scaler_path = os.path.join(models_dir, f"{model_name}_scaler.pkl")
                metadata_path = os.path.join(models_dir, f"
{model_name}_metadata.json")

                # Create model and load from disk
                model = MillsXGBoostModel()
                model.load_model(model_path, scaler_path, metadata_path if
os.path.exists(metadata_path) else None)

                # Get target column from metadata or use default
                target_col = "PSI80"
                if os.path.exists(metadata_path):
                    with open(metadata_path, 'r') as f:
                        metadata = json.load(f)
                        target_col = metadata.get('target_col', target_col)

                # Store in memory for future use
                models_store[request.model_id] = {
                    "model": model,
                    "target_col": target_col
                }
                logger.info(f"Successfully loaded model {request.model_id} from
disk")
            except Exception as e:
```

```
                logger.error(f"Failed to load model from disk: {str(e)}")
                raise HTTPException(status_code=404, detail="Model not found")

        # Get model and make prediction
        model_info = models_store[request.model_id]
        model = model_info["model"]
        target_col = model_info.get("target_col", "PSI80")

        # Make prediction
        prediction = model.predict(request.data)[0]

        # Return prediction
        return PredictionResponse(
            model_id=request.model_id,
            prediction=float(prediction),
            target_col=target_col
        )
    except Exception as e:
        logger.error(f"Error during prediction: {str(e)}")
        raise HTTPException(status_code=500, detail=f"Prediction failed: {str(e)}")
```

Example Endpoint Implementation: `/optimize` (Partial)

```
@router.post("/optimize", response_model=OptimizationResponse)
async def optimize_parameters(request: OptimizationRequest):
    """Optimize XGBoost hyperparameters using Bayesian Optimization"""
    try:
        # Load model (from memory or disk)
        # ...

        # Create black box function with the loaded model
        black_box = BlackBoxFunction(
            model_id=request.model_id,
            xgb_model=model,
            maximize=request.maximize
        )

        # Set parameter bounds
        black_box.set_parameter_bounds(parameter_bounds)

        # Run optimization
        best_params, best_value, study = optimize_with_optuna(
            black_box_func=black_box,
            n_trials=n_trials
        )

        # Generate recommendations from top trials
        recommendations = []
        for trial in sorted(study.trials, key=lambda t: t.value,
reverse=request.maximize)[:5]:
            value = trial.value if request.maximize else -trial.value
            recommendations.append({
                "params": trial.params,
                "predicted_value": float(value)
```

```
            })

        # Return optimized parameters with recommendations
        return OptimizationResponse(
            best_params=best_params,
            best_target=float(best_value),
            target_col=target_col,
            maximize=request.maximize,
            recommendations=recommendations,
            model_id=request.model_id
        )
    except Exception as e:
        logger.error(f"Error during optimization: {str(e)}")
        raise HTTPException(status_code=500, detail=f"Optimization failed:
{str(e)}")
```

```
- Uses time-ordered train-test split (no shuffling) for time series data
- Stores trained models both on disk and in memory for immediate use
```

- `POST /predict`: Make predictions with a trained model
  - Can load models from disk if not found in memory
  - Supports full model name including mill number (e.g., `xgboost_PSI80_mill6_model`)
- `POST /optimize`: Run Bayesian optimization to find optimal mill parameters
- `GET /models`: List available trained models
- `GET /status`: Health check endpoint
- `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

Frontend Integration

React Frontend Integration

The frontend integrates with the ML API through a set of custom hooks and components.

Key Frontend Components

- **XGBoost Simulation Dashboard**: Main dashboard component for interacting with ML models
- **Parameter optimization UI**: Component for setting parameter bounds and running optimization
- **Model training interface**: Component for selecting data ranges and training parameters

API Integration Hooks

Prediction Hook Implementation

The prediction functionality is implemented in `use-predict-target.ts` hook which handles API calls to the ML prediction endpoint:

```
// Type definitions for prediction API
export interface PredictionResponse {
  model_id: string;
  prediction: number;
  target_col: string;
}

// Hook for making prediction calls
export const usePredictTarget = () => {
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);
  const [prediction, setPrediction] = useState<number | null>(null);

  const predictTarget = useCallback(
    async (modelId: string, parameters: Record<string, number>) => {
      setIsLoading(true);
      setError(null);

      try {
        const response = await fetch('/api/v1/ml/predict', {
          method: 'POST',
          headers: {
            'Content-Type': 'application/json',
          },
          body: JSON.stringify({
            model_id: modelId,  // Use correct field name to match API schema
            data: parameters,   // Use correct field name to match API schema
          }),
        });

        if (!response.ok) {
          throw new Error(`Error ${response.status}: ${response.statusText}`);
        }

        const result: PredictionResponse = await response.json();
        setPrediction(result.prediction);
        return result.prediction;
      } catch (err) {
        setError(err instanceof Error ? err.message : 'An unknown error occurred');
        return null;
      } finally {
        setIsLoading(false);
      }
    },
    []
  );

  return { predictTarget, prediction, isLoading, error };
};
```

Component Usage Example

Example of using the prediction hook in the XGBoost dashboard component:

```
const XGBoostSimulationDashboard = () => {
  const { predictTarget, prediction, isLoading, error } = usePredictTarget();

  const handlePrediction = async () => {
    const parameters = {
      Ore: oreValue,
      WaterMill: waterMillValue,
      WaterZumpf: waterZumpfValue,
      // Other parameters
    };

    await predictTarget('xgboost_PSI80_mill8', parameters);
  };

  // Component render code
};
```

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.

Key System Highlights

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