- Database Connector Documentation
  - Overview
    - Key Features
    - Integration Points
  - Table of Contents
  - Class Architecture
    - Class Purpose
    - Dependencies
    - Class Attributes
    - Thread Safety
    - Error Handling Strategy
  - Initialization
    - Constructor
    - Initialization Process
    - Error Handling During Initialization
    - Example with Error Handling
    - Configuration Options
    - Best Practices for Initialization
  - Core Methods
    - get_mill_data
      - Method Signature
      - Parameters
      - Return Value
      - Implementation Details
      - Example Usage
      - Performance Considerations
      - Common Issues and Solutions
      - Logging
    - get_ore_quality
      - Method Signature
      - Parameters
      - Return Value
      - Implementation Details
      - Example Usage
      - Performance Considerations
      - Common Issues and Solutions
      - Logging

        - Data Dictionary
    - process_dataframe
        - Method Signature
        - Parameters
        - Return Value
        - Processing Pipeline
        - Example Usage
        - Performance Considerations
        - Common Issues and Solutions
        - Logging
    - join_dataframes_on_timestamp
        - Method Signature
        - Parameters
        - Return Value
        - Implementation Details
        - Example Usage
        - Performance Considerations
        - Common Issues and Solutions
        - Logging
        - Advanced Usage
    - get_combined_data
        - Method Signature
        - Parameters
        - Return Value
        - Method Workflow
        - Example Usage
        - Performance Considerations
        - Error Handling
        - Logging
        - Best Practices
- Data Processing
- Error Handling
- Logging
- Usage Examples
    - Basic Usage
    - Advanced Usage with Custom Processing
- Best Practices
- Troubleshooting

- Common Issues

Database Connector Documentation

Overview

The `db_connector.py` module is a critical component of the Mills XGBoost application, serving as the primary interface between the application and the PostgreSQL database. It's specifically designed to handle time-series data from industrial mill operations, providing robust data retrieval, processing, and integration capabilities.

Key Features

- **Secure Database Connectivity**: Manages PostgreSQL connections with proper error handling and resource cleanup
- **Time-Series Data Handling**: Specialized methods for working with time-indexed industrial process data
- **Data Integrity**: Comprehensive handling of duplicate timestamps and data alignment issues
- **Flexible Data Processing**: Built-in methods for resampling, interpolation, and data cleaning
- **Logging and Monitoring**: Detailed logging for debugging and operational monitoring
- **Modular Design**: Clean separation of concerns between data retrieval and processing

Integration Points

The `MillsDataConnector` integrates with several key components of the application:

1. **Model Training Pipeline**: Provides clean, processed data for training XGBoost models
2. **API Endpoints**: Used by FastAPI endpoints to serve real-time and historical data
3. **Data Analysis Tools**: Supports data exploration and feature engineering tasks

4. **Testing Framework**: Includes test utilities for data quality validation

Table of Contents

1. Class Architecture
2. Initialization
3. Core Methods
    - get_mill_data
    - get_ore_quality
    - process_dataframe
    - join_dataframes_on_timestamp
    - get_combined_data
4. Data Processing Pipeline
5. Error Handling Strategy
6. Performance Considerations
7. Integration Guide
8. Testing and Validation
9. Usage Examples
10. Best Practices
11. Troubleshooting

Class Architecture

Class Purpose

The `MillsDataConnector` class serves as the primary data access layer for the application, abstracting away database interactions and providing a clean interface for retrieving and processing mill operation data. It's designed to be thread-safe and can be instantiated once and reused across the application.

Dependencies

- **SQLAlchemy**: For database connectivity and query building
- **Pandas**: For data manipulation and time-series processing
- **Python Logging**: For operational logging

- **Datetime**: For date/time handling

Class Attributes

- `connection_string`: Formatted PostgreSQL connection string
- `engine`: SQLAlchemy engine instance for database operations
- `logger`: Configured logger instance for the class

Thread Safety

The class is designed to be thread-safe, allowing multiple threads to use the same instance for database operations. The SQLAlchemy connection pool handles concurrent database connections efficiently.

Error Handling Strategy

The class implements comprehensive error handling with the following approach:

1. **Input Validation**: All public methods validate their inputs
2. **Graceful Degradation**: Methods return `None` or empty DataFrames rather than raising exceptions for expected edge cases
3. **Detailed Logging**: All errors are logged with sufficient context for debugging
4. **Resource Cleanup**: Database connections are properly managed and released

Initialization

Constructor

```
connector = MillsDataConnector(
    host="your_host",          # PostgreSQL server hostname or IP
    port=5432,                 # PostgreSQL server port
    dbname="your_database",    # Target database name
    user="your_username",      # Database username
    password="your_password"    # Database password
)
```

Initialization Process

1. **Connection String Formation**:
    - Combines credentials into a PostgreSQL connection string
    - Format: `postgresql://{user}:{password}@{host}:{port}/{dbname}`
    - Connection pooling is enabled by default for better performance
2. **Engine Creation**:
    - Initializes a SQLAlchemy engine with the connection string
    - Configures connection pooling with these settings:
        - `pool_size`: 5 (default)
        - `max_overflow`: 10
        - `pool_timeout`: 30 seconds
        - `pool_recycle`: 1800 seconds (30 minutes)
3. **Logging Setup**:
    - Configures logging for the module
    - Logs connection status and any initialization errors
    - Log level is set to INFO by default

Error Handling During Initialization

- Validates all connection parameters are provided
- Handles database connection errors gracefully
- Common issues caught during initialization:
    - Invalid credentials
    - Network connectivity issues
    - Database server not available
    - Insufficient permissions

Example with Error Handling

```
import logging
from datetime import datetime

try:
```

```
    # Initialize with production database credentials
    connector = MillsDataConnector(
        host="em-m-db4.ellatzite-med.com",
        port=5432,
        dbname="em_pulse_data",
        user="s.lyubenov",
        password="your_secure_password"
    )
    logging.info(f"Successfully connected to database at {datetime.now()}")

except Exception as e:
    logging.error(f"Failed to initialize database connection: {e}")
    # Handle the error appropriately (e.g., retry, use fallback, exit)
    raise
```

Configuration Options

For advanced configuration, you can modify these aspects after initialization:

```
# Adjust connection pool settings
connector.engine.dispose()  # Close existing connections
connector.engine = create_engine(
    connector.connection_string,
    pool_size=10,
    max_overflow=20,
    pool_timeout=60,
    pool_recycle=3600
)

# Adjust logging level
import logging
logging.getLogger('sqlalchemy.engine').setLevel(logging.WARNING)
```

Best Practices for Initialization

1. **Connection Management**:
    - Create a single instance of `MillsDataConnector` and reuse it
    - Let SQLAlchemy handle connection pooling
    - Avoid creating multiple instances unless necessary
2. **Security Considerations**:
    - Never hardcode credentials in source code
    - Use environment variables or secure configuration management

- Follow principle of least privilege for database users

3. **Performance Tips**:
    - Set appropriate pool sizes based on expected concurrency
    - Monitor connection usage in production
    - Consider using connection pooling middleware in web applications

Core Methods

get_mill_data

Retrieves operational data for a specific mill from the database. This is the primary method for accessing raw mill sensor data, which includes various measurements like motor current, pressure, temperature, and other process variables.

Method Signature

```
def get_mill_data(
    self,
    mill_number: int,
    start_date: Optional[Union[str, datetime]] = None,
    end_date: Optional[Union[str, datetime]] = None
) -> pd.DataFrame:
```

Parameters

- `mill_number` (int):
    - **Required**
    - The mill identifier (6, 7, or 8)
    - Determines which mill's data table to query (e.g., `MILL_06` for mill_number=6)
- `start_date` (str/datetime, optional):
    - Start of the date range (inclusive)
    - Can be a string in format 'YYYY-MM-DD' or 'YYYY-MM-DD HH:MM:SS'
    - If None, retrieves data from the earliest available record

- `end_date` (str/datetime, optional):
    - End of the date range (inclusive)
    - Same format as `start_date`
    - If None, retrieves data up to the most recent record

Return Value

- `pd.DataFrame`: A DataFrame containing mill sensor data with these characteristics:
    - **Index**: DateTimeIndex (timezone-naive)
    - **Columns**: Various sensor readings (e.g., 'Current', 'Pressure', 'Temperature')
    - **Data Types**: Numeric values (float64)
    - **Index Name**: 'TimeStamp'

Implementation Details

1. **Table Selection**:
    - Dynamically constructs table name as `MILL_{mill_number:02d}`
    - Example: For mill 8, queries table `MILL_08`
    - Validates that mill_number is between 1 and 99
2. **Query Construction**:
    - Uses SQLAlchemy's text() for safe SQL construction
    - Implements parameterized queries to prevent SQL injection
    - Adds date range filtering if provided
    - Example query:

```
SELECT * FROM mills."MILL_08"
WHERE "TimeStamp" BETWEEN :start_date AND :end_date
ORDER BY "TimeStamp"
```

3. **Data Retrieval**:
    - Uses `pd.read_sql_query` for efficient data loading
    - Sets 'TimeStamp' as the DataFrame index
    - Converts timestamp to datetime if needed
    - Drops any duplicate indices (keeps first occurrence)

- Sorts index in ascending order

4. **Error Handling**:
   - Validates mill_number is within allowed range
   - Handles database connection issues
   - Validates date formats
   - Logs warnings for empty results

Example Usage

```
# Get data for Mill 8 for a specific date range
mill_data = connector.get_mill_data(
    mill_number=8,
    start_date='2025-06-10 06:00:00',
    end_date='2025-06-11 18:00:00'
)

# Basic data inspection
print(f"Retrieved {len(mill_data)} records")
print(f"Date range: {mill_data.index.min()} to {mill_data.index.max()}")
print(f"Available columns: {list(mill_data.columns)}")

# Sample data
print("\nFirst 5 records:")
print(mill_data.head())
```

Performance Considerations

- **Indexing**: Ensure the 'TimeStamp' column is properly indexed in the database
- **Data Volume**: Be cautious with large date ranges as they can consume significant memory
- **Network Latency**: For large datasets, consider:
  - Using smaller date ranges
  - Processing data in chunks
  - Using the `chunksize` parameter in `read_sql_query`

Common Issues and Solutions

1. **No Data Returned**:
   - Verify the mill number is correct
   - Check that the date range contains data

    - Ensure database connection is active

2. **Incorrect Date Range**:
    - Dates are inclusive of both start and end dates
    - Time component defaults to 00:00:00 if not specified
    - Timezone is handled as UTC in the database

3. **Memory Errors**:
    - Reduce the date range
    - Select only required columns
    - Process data in smaller chunks

Logging

- **INFO**: Connection established, query executed
- **WARNING**: No data found for the specified criteria
- **ERROR**: Database connection issues, invalid parameters

get_ore_quality

Retrieves ore quality measurements from the database. This method provides access to laboratory analysis data that characterizes the ore being processed, which is essential for correlating material properties with mill performance.

Method Signature

```
def get_ore_quality(
    self,
    start_date: Optional[Union[str, datetime]] = None,
    end_date: Optional[Union[str, datetime]] = None
) -> pd.DataFrame:
```

Parameters

- `start_date` (str/datetime, optional):
    - Start of the date range (inclusive)
    - Can be a string in format 'YYYY-MM-DD' or 'YYYY-MM-DD HH:MM:SS'

    - If None, retrieves data from the earliest available record

- `end_date` (str/datetime, optional):
    - End of the date range (inclusive)
    - Same format as `start_date`
    - If None, retrieves data up to the most recent record

Return Value

- `pd.DataFrame`: A DataFrame containing ore quality measurements with these characteristics:
    - **Index**: Not set (use 'TimeStamp' column as index if needed)
    - **Columns**: Ore quality parameters (e.g., 'Fe_Content', 'SiO2_Content', 'Particle_Size')
    - **Data Types**: Mixed (numeric values converted to float64 where possible)

Implementation Details

1. **Query Construction**:
    - Queries the `mills.ore_quality` table
    - Implements parameterized queries for security
    - Adds date range filtering if provided
    - Example query:

```
SELECT * FROM mills.ore_quality
WHERE "TimeStamp" BETWEEN :start_date AND :end_date
ORDER BY "TimeStamp"
```

2. **Data Handling**:
    - Preserves the original timestamp column as 'TimeStamp'
    - Converts string representations of numbers to numeric types
    - Handles missing or malformed data gracefully
    - Drops any duplicate timestamps (keeps first occurrence)

3. **Error Handling**:
    - Validates date formats
    - Handles database connection issues

- Logs warnings for empty results or data quality issues

Example Usage

```
# Get ore quality data for a specific date range
ore_data = connector.get_ore_quality(
    start_date='2025-06-01',
    end_date='2025-06-30'
)

# Basic data inspection
print(f"Retrieved {len(ore_data)} ore quality samples")
print(f"Date range: {ore_data['TimeStamp'].min()} to
{ore_data['TimeStamp'].max()}")
print(f"Available measurements: {[c for c in ore_data.columns if c !=
'TimeStamp']}")

# Set timestamp as index if needed
ore_data_indexed = ore_data.set_index('TimeStamp')
print(f"\nFirst 5 samples:")
print(ore_data_indexed.head())
```

Performance Considerations

- **Sampling Frequency**: Ore quality data is typically less frequent than mill data (e.g., daily or per-shift)
- **Query Optimization**:
  - The table should have an index on the 'TimeStamp' column
  - Consider materialized views for complex queries
- **Memory Usage**:
  - Generally lower than mill data due to lower frequency
  - Still be mindful of very large date ranges

Common Issues and Solutions

1. **No Data Returned**:
   - Verify the date range contains data
   - Check for typos in column names if using direct SQL access
   - Ensure proper database permissions
2. **Data Quality Issues**:
   - Missing values are common in laboratory data

    - Some measurements might be reported as strings (e.g., "<0.5")
    - Consider implementing data validation rules

3. **Time Alignment**:
    - Ore quality data may not align perfectly with mill data timestamps
    - Use `process_dataframe` with `no_interpolation=True` for forward-fill behavior

Logging

- **INFO**: Query executed, number of records retrieved
- **WARNING**: No data found for the specified criteria
- **ERROR**: Database connection issues, invalid parameters

Data Dictionary

Column Name	Data Type	Description	Unit
TimeStamp	datetime	Timestamp of the measurement	
Fe_Content	float64	Iron content in the ore	%
SiO2_Content	float64	Silica content in the ore	%
Particle_Size	float64	Average particle size distribution	mm
Moisture	float64	Moisture content	%
...	...	...	...

Note: Actual columns may vary based on your database schema.

process_dataframe

Processes a time-series DataFrame to ensure consistent sampling, handle missing values, and prepare data for analysis. This method is essential for transforming raw mill or ore quality data into a clean, analysis-ready format with regular time intervals.

Method Signature

```python
def process_dataframe(
    self,
    df: pd.DataFrame,
    start_date: Optional[Union[str, datetime]] = None,
    end_date: Optional[Union[str, datetime]] = None,
    resample_freq: str = '1min',
    no_interpolation: bool = False
) -> pd.DataFrame:
```

Parameters

- `df` (pd.DataFrame):
  - **Required**
  - Input DataFrame containing time-series data
  - Must have a DateTimeIndex or a 'TimeStamp' column
- `start_date` (str/datetime, optional):
  - Filter data to include only records on or after this timestamp
  - If None, uses the earliest timestamp in the data
- `end_date` (str/datetime, optional):
  - Filter data to include only records on or before this timestamp
  - If None, uses the latest timestamp in the data
- `resample_freq` (str, default='1min'):
  - Target frequency for resampling
  - Uses pandas offset aliases (e.g., '1min', '5min', '1H', '1D')
  - See pandas documentation for all options
- `no_interpolation` (bool, default=False):
  - If True: Uses forward-fill (last observation carried forward) for missing values
  - If False: Uses linear interpolation followed by rolling mean smoothing

Return Value

- `pd.DataFrame`: Processed DataFrame with:
  - Regular time intervals as specified by `resample_freq`
  - Numeric data types for all columns

- Handled missing values (interpolated or forward-filled)
- Optional smoothing applied
- Index as DateTimeIndex

Processing Pipeline

1. **Input Validation**:
   - Checks for empty or None input
   - Validates timestamp index or column
   - Converts string timestamps to datetime objects
2. **Index Handling**:
   - Sets DateTimeIndex if not already set
   - Sorts index in ascending order
   - Removes duplicate indices (keeps first occurrence)
3. **Date Range Filtering**:
   - Applies start_date and end_date filters if provided
   - Handles timezone information consistently
4. **Resampling**:
   - Resamples data to the target frequency
   - For each interval, uses mean aggregation for numeric columns
   - Drops non-numeric columns with a warning
5. **Missing Data Handling**:
   - If `no_interpolation=True`:
     - Forward fills missing values (carries last observation forward)
   - If `no_interpolation=False` (default):
     - Applies linear interpolation to fill gaps
     - Applies a rolling mean with window=15 for smoothing
     - Forward/backward fills any remaining NAs at the edges

Example Usage

```
# Process mill data with default settings (1-minute resampling with interpolation)
processed_mill = connector.process_dataframe(
```

```
    df=mill_data,
    start_date='2025-06-10 08:00:00',
    end_date='2025-06-10 16:00:00',
    resample_freq='1min',
    no_interpolation=False
)

# Process ore quality data with forward-fill (keep values constant between samples)
processed_ore = connector.process_dataframe(
    df=ore_data,
    resample_freq='1H',  # Hourly samples
    no_interpolation=True  # Keep values constant between lab measurements
)

# Verify results
print(f"Original shape: {len(mill_data)} rows, Processed shape:
{len(processed_mill)} rows")
print(f"Date range: {processed_mill.index.min()} to {processed_mill.index.max()}")
print(f"Sample rate: {pd.infer_freq(processed_mill.index)}")
```

Performance Considerations

- **Resampling Frequency**:
  - Lower frequencies (e.g., '5min', '15min') reduce data volume
  - Higher frequencies increase memory usage and processing time
- **Interpolation Method**:
  - `no_interpolation=True` is faster but less accurate for continuous signals
  - Linear interpolation is more accurate but computationally intensive
- **Window Size**:
  - The rolling mean window size (15 samples) affects smoothing
  - Larger windows provide more smoothing but may obscure important features

Common Issues and Solutions

1. **Memory Errors**:
   - Reduce the date range
   - Increase resample_freq
   - Process in smaller chunks
2. **Unexpected Gaps**:

    - Check for large time gaps in the original data
    - Adjust the resample_freq if needed
    - Consider using `no_interpolation=True` for sparse data

3. **Non-Numeric Data**:

    - Non-numeric columns are dropped with a warning
    - Convert categorical data to numeric if needed before processing

Logging

- **INFO**: Processing started/completed, input/output shapes
- **WARNING**: Non-numeric columns dropped, empty result after filtering
- **DEBUG**: Detailed timing information for each processing step

join_dataframes_on_timestamp

Joins two time-series DataFrames on their timestamp indices with comprehensive error handling and validation. This method is essential for aligning mill operational data with ore quality measurements, ensuring temporal consistency for subsequent analysis.

Method Signature

```
def join_dataframes_on_timestamp(
    self,
    df1: pd.DataFrame,
    df2: pd.DataFrame
) -> pd.DataFrame:
```

Parameters

- `df1` (pd.DataFrame):

    - **Required**
    - First time-series DataFrame (typically mill sensor data)
    - Must have a DateTimeIndex or a 'TimeStamp' column

- `df2` (pd.DataFrame):

    - **Required**

    - Second time-series DataFrame (typically ore quality data)
    - Must have a DateTimeIndex or a 'TimeStamp' column

Return Value

- `pd.DataFrame`: A single DataFrame containing:
    - All columns from both input DataFrames
    - Rows aligned by timestamp (inner join)
    - Index as DateTimeIndex
    - Columns renamed if there are duplicates (with '_x' and '_y' suffixes)

Implementation Details

1. **Input Validation**:
    - Checks for None or empty DataFrames
    - Validates timestamp index or 'TimeStamp' column in both DataFrames
    - Converts string timestamps to datetime objects if needed
2. **Index Preparation**:
    - Ensures both DataFrames use DateTimeIndex
    - Sorts indices in ascending order
    - Removes duplicate indices (keeps first occurrence)
    - Converts timezones to UTC if timezone-aware
3. **Temporal Join**:
    - Performs an inner join on the timestamps
    - Handles different sampling frequencies between the DataFrames
    - Preserves all columns from both DataFrames
4. **Column Name Handling**:
    - Automatically renames duplicate column names with '_x' and '_y' suffixes
    - Logs warnings about any renamed columns
5. **Error Handling**:
    - Validates that there is temporal overlap between DataFrames
    - Handles empty results gracefully
    - Provides informative error messages for debugging

Example Usage

```
# Process mill and ore data first
processed_mill = connector.process_dataframe(mill_data, no_interpolation=False)
processed_ore = connector.process_dataframe(ore_data, no_interpolation=True)

# Join the dataframes
combined_data = connector.join_dataframes_on_timestamp(processed_mill,
processed_ore)

# Verify the result
print(f"Combined data shape: {combined_data.shape}")
print(f"Date range: {combined_data.index.min()} to {combined_data.index.max()}")
print(f"Columns: {list(combined_data.columns)}")

# Plot a sample of the combined data
combined_data[['Mill_Current', 'Fe_Content']].plot(
    secondary_y='Fe_Content',
    title='Mill Current vs. Iron Content',
    figsize=(12, 6)
)
```

Performance Considerations

- **Join Operation**:
    - Inner join ensures only matching timestamps are kept
    - Most efficient when both DataFrames are sorted by time
    - Performance degrades with very large DataFrames
- **Memory Usage**:
    - The result contains all columns from both DataFrames
    - Consider dropping unnecessary columns before joining
    - Process in smaller time windows if memory is a concern
- **Indexing**:
    - Ensure both DataFrames have proper DateTimeIndex
    - Pre-sorting can improve join performance

Common Issues and Solutions

1. **No Common Timestamps**:

- Check that the date ranges of both DataFrames overlap
- Verify timezone handling is consistent
- Consider using a merge_asof for nearest-neighbor matching

2. **Memory Errors**:
   - Reduce the date range before joining
   - Downsample to a lower frequency
   - Process in smaller chunks

3. **Column Name Conflicts**:
   - Columns with the same name will be suffixed with '_x' and '_y'
   - Rename columns before joining if specific naming is required
   - Use the `suffixes` parameter for custom suffixes

Logging

- **INFO**: Join operation started/completed, input/output shapes
- **WARNING**: Columns renamed due to naming conflicts
- **ERROR**: No temporal overlap between DataFrames, invalid inputs

Advanced Usage

For more control over the join operation, you can use pandas merge directly:

```
# Example of manual merge with more control
combined = pd.merge(
    left=df1.reset_index(),
    right=df2.reset_index(),
    on='TimeStamp',
    how='inner',
    suffixes=('_mill', '_ore')
).set_index('TimeStamp')
```

get_combined_data

High-level method that orchestrates the complete data retrieval and processing pipeline. This is the primary method used by most applications to get analysis-ready data that combines mill operational data with ore quality measurements.

Method Signature

```python
def get_combined_data(
    self,
    mill_number: int,
    start_date: Optional[Union[str, datetime]] = None,
    end_date: Optional[Union[str, datetime]] = None,
    resample_freq: str = '1min',
    save_to_logs: bool = True,
    no_interpolation: bool = False
) -> Optional[pd.DataFrame]:
```

Parameters

- `mill_number` (int):
  - **Required**
  - The mill identifier (6, 7, or 8)
  - Determines which mill's data to retrieve
- `start_date` (str/datetime, optional):
  - Start of the date range
  - Format: 'YYYY-MM-DD' or 'YYYY-MM-DD HH:MM:SS'
  - If None, retrieves all available data
- `end_date` (str/datetime, optional):
  - End of the date range
  - Same format as `start_date`
  - If None, retrieves up to the most recent data
- `resample_freq` (str, default='1min'):
  - Target frequency for resampling
  - Uses pandas offset aliases (e.g., '1min', '5min', '1H')
  - Affects both mill and ore quality data
- `save_to_logs` (bool, default=True):
  - If True, saves the combined data to a CSV file in the logs directory
  - Useful for debugging and reproducibility
  - Filename format: `combined_data_<mill>_<timestamp>.csv`

- `no_interpolation` (bool, default=False):
    - If True: Uses forward-fill for ore quality data
    - If False: Uses linear interpolation for ore quality data
    - Mill data always uses interpolation for continuous signals

Return Value

- `Optional[pd.DataFrame]`: A DataFrame containing:
    - All mill sensor data
    - All ore quality measurements
    - Indexed by timestamp
    - Returns None if there's an error during processing

Method Workflow

1. **Data Retrieval**:
    - Fetches mill data using `get_mill_data`
    - Fetches ore quality data using `get_ore_quality`
    - Handles cases where either dataset might be missing
2. **Data Processing**:
    - Processes mill data with interpolation (continuous signals)
    - Processes ore quality data according to `no_interpolation` parameter
    - Applies consistent date range filtering
    - Handles timezone information
3. **Data Integration**:
    - Joins the processed data using `join_dataframes_on_timestamp`
    - Handles cases with no overlapping timestamps
    - Validates the combined dataset
4. **Optional Logging**:
    - Saves the combined data to a timestamped CSV file
    - Logs summary statistics and data quality metrics

Example Usage

```python
# Get combined data for Mill 8 with default settings
data = connector.get_combined_data(
    mill_number=8,
    start_date='2025-06-10 06:00:00',
    end_date='2025-06-11 18:00:00',
    resample_freq='5min',
    save_to_logs=True,
    no_interpolation=True  # Keep ore quality values constant between measurements
)

if data is not None:
    print(f"Successfully retrieved {len(data)} samples")
    print(f"Features: {list(data.columns)}")

    # Basic analysis
    print("\nSummary statistics:")
    print(data.describe())

    # Plot key metrics
    import matplotlib.pyplot as plt

    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))

    # Plot mill current
    data['Mill_Current'].plot(ax=ax1, title='Mill Current', color='blue')
    ax1.set_ylabel('Current (A)')

    # Plot ore quality
    if 'Fe_Content' in data.columns:
        ax3 = ax1.twinx()
        data['Fe_Content'].plot(ax=ax3, color='red', alpha=0.5)
        ax3.set_ylabel('Fe Content (%)', color='red')

    # Plot another metric
    if 'Power' in data.columns:
        data['Power'].plot(ax=ax2, title='Power Consumption', color='green')
        ax2.set_ylabel('Power (kW)')

    plt.tight_layout()
    plt.show()
else:
    print("Failed to retrieve data")
```

Performance Considerations

- **Data Volume**:
  - Larger date ranges and higher frequencies increase memory usage
  - Consider using `resample_freq='5min'` or higher for long time periods
- **Processing Time**:

    - Interpolation and resampling are computationally intensive
    - For real-time applications, consider pre-processing data

- **I/O Operations**:
    - Setting `save_to_logs=False` can improve performance
    - Disk I/O can be a bottleneck for large datasets

Error Handling

- **Database Errors**:
    - Connection issues are logged and handled gracefully
    - Returns None if data retrieval fails

- **Data Quality Issues**:
    - Missing data is handled according to interpolation settings
    - Warnings are logged for potential data quality problems

- **No Data**:
    - Returns an empty DataFrame with appropriate columns if no data is found
    - Logs a warning with details

Logging

- **INFO**:
    - Processing start/end times
    - Data shapes before and after processing
    - File save locations (if save_to_logs=True)

- **WARNING**:
    - No data found for specified criteria
    - Data quality issues
    - Missing or incomplete data

- **ERROR**:
    - Database connection errors
    - Critical processing failures

Best Practices

1. **Parameter Selection**:
    - Choose an appropriate `resample_freq` for your analysis needs
    - Use `no_interpolation=True` for ore quality data to avoid artificial values
2. **Memory Management**:
    - Process data in smaller chunks for large date ranges
    - Consider using Dask for out-of-core processing if needed
3. **Reproducibility**:
    - Set `save_to_logs=True` to maintain a record of the processed data
    - Include the log file in your analysis for traceability
4. **Error Handling**:
    - Always check if the return value is None
    - Implement appropriate fallback behavior in your application

Data Processing

The module includes sophisticated data processing capabilities:

1. **Duplicate Handling**: Automatically detects and removes duplicate timestamps
2. **Resampling**: Converts data to a consistent time frequency
3. **Interpolation**: Fills missing values using linear interpolation or forward fill
4. **Smoothing**: Applies rolling window averaging to reduce noise
5. **Type Conversion**: Ensures all data columns are numeric

Error Handling

The module includes comprehensive error handling:

- Connection errors during database operations
- Missing or invalid data
- Duplicate timestamps

- Data alignment issues
- File I/O errors when saving logs

Logging

All operations are logged with different severity levels:

- INFO: Routine operations and status updates
- WARNING: Non-critical issues
- ERROR: Critical errors that might affect functionality

Logs include detailed information about data processing steps and any issues encountered.

Usage Examples

Basic Usage

```
from mills_xgboost.app.database.db_connector import MillsDataConnector

# Initialize connector
connector = MillsDataConnector(
    host="localhost",
    port="5432",
    dbname="mills_db",
    user="user",
    password="password"
)

# Get combined data for mill 6 for January 2023
data = connector.get_combined_data(
    mill_number=6,
    start_date='2023-01-01',
    end_date='2023-01-31'
)

# Display the first few rows
print(data.head())
```

Advanced Usage with Custom Processing

```
# Get raw mill data
mill_data = connector.get_mill_data(6, '2023-01-01', '2023-01-31')

# Get raw ore quality data
ore_data = connector.get_ore_quality('2023-01-01', '2023-01-31')

# Process with custom parameters
processed_mill = connector.process_dataframe(
    mill_data,
    resample_freq='5min',
    no_interpolation=True
)

processed_ore = connector.process_dataframe(
    ore_data,
    resample_freq='5min',
    no_interpolation=True  # Keep ore quality values constant between measurements
)

# Combine the data
combined = connector.join_dataframes_on_timestamp(processed_mill, processed_ore)
```

Best Practices

1. **Connection Management**:
   - Create a single instance of `MillsDataConnector` and reuse it
   - Handle exceptions when initializing the connector
2. **Data Retrieval**:
   - Always specify date ranges to limit data volume
   - Use appropriate resampling frequency based on your analysis needs
3. **Error Handling**:
   - Always wrap database calls in try-except blocks
   - Check for empty DataFrames after retrieval
4. **Performance**:
   - For large date ranges, consider processing data in chunks
   - Use `no_interpolation=True` for ore quality data to maintain discrete values

5. **Logging**:
    - Monitor logs for warnings about data quality
    - Use the saved CSV files for debugging data issues

Troubleshooting

Common Issues

1. **Connection Errors**:
    - Verify database credentials
    - Check network connectivity
    - Ensure the PostgreSQL server is running
2. **Empty Results**:
    - Verify the date range contains data
    - Check for typos in table or column names
3. **Performance Issues**:
    - Reduce the date range
    - Increase the resampling frequency
    - Process data in smaller chunks
4. **Data Quality Warnings**:
    - Review logs for details about missing or invalid data
    - Check for and handle duplicate timestamps

For additional assistance, refer to the logs or contact the development team with the relevant timestamps and error messages.