# Data Quality Guidelines

## Overview

This document outlines the data quality standards and guidelines for the Model Drift Detection project, ensuring consistent and reliable analysis across different log sources.

## Dataset-Specific Guidelines

### 1. System Logs

#### HDFS Logs

- Format: `[Timestamp] [Severity] [Component] [Message]`
- Required Fields:
  - Timestamp: Must be in standard format
  - Severity: INFO/WARN/ERROR/FATAL
  - Component: DataNode/NameNode/FSNamesystem
  - Message: Must contain operation details
- Quality Thresholds:
  - Maximum 1% missing components
  - No missing timestamps
  - Valid severity levels only

#### Apache Logs

- Format: `[Timestamp] [Severity] [Client] [Request] [Status] [Size]`
- Required Fields:
  - Timestamp: Apache timestamp format
  - Client IP: Valid IP address
  - Request: HTTP method and path
  - Status: Valid HTTP status code
  - Size: Response size in bytes
- Quality Thresholds:
  - All fields must be present
  - Valid HTTP status codes only

    - Valid request methods only

### HealthApp Logs

- Format: `[Timestamp] [Component] [Severity] [UserID] [Action] [Status]`
- Required Fields:
    - Timestamp: ISO 8601 format
    - Component: Valid application module
    - UserID: Anonymized user identifier
    - Action: Valid system action
    - Status: Success/Failure/Warning
- Quality Thresholds:
    - 100% timestamp completeness
    - Valid user IDs only
    - Known component names only

## 2. Supercomputer Logs

### BGL Logs

- Format: `[Timestamp] [Location] [Severity] [Component] [Message]`
- Required Fields:
    - Timestamp: BGL standard format
    - Location: Valid node identifier
    - Severity: FATAL/ERROR/WARNING/INFO
    - Component: Hardware/Software component
- Quality Thresholds:
    - Valid location identifiers
    - Known component types only
    - Complete message field

### HPC Logs

- Format: `[Timestamp] [NodeID] [JobID] [Severity] [Message]`
- Required Fields:
    - Timestamp: System time format
    - NodeID: Valid compute node ID
    - JobID: Valid job identifier
    - Severity: Standard severity levels
- Quality Thresholds:

    - Valid node IDs only
    - Active job IDs only
    - Complete message content

# 3. Operating System Logs

## Linux Logs

- Format: `[Timestamp] [Hostname] [Process] [PID] [Message]`
- Required Fields:
    - Timestamp: Syslog format
    - Hostname: Valid system hostname
    - Process: Valid process name
    - PID: Process ID number
- Quality Thresholds:
    - Valid PIDs only
    - Known process names
    - Complete message field

## Mac Logs

- Format: `[Timestamp] [Sender] [Type] [Category] [Message]`
- Required Fields:
    - Timestamp: System log format
    - Sender: Valid process/application
    - Type: Default/Error/Debug
    - Category: System category
- Quality Thresholds:
    - Known sender processes only
    - Valid message types only
    - Complete category field

# Validation Methods for New Datasets

```
def validate_dataset_specific(logs, dataset_type):
    """
    Validate dataset-specific requirements

    Parameters:
    - logs: List of log entries
    - dataset_type: Type of dataset (HDFS/Apache/HealthApp/BGL/HPC/Linux/Mac)

    Returns:
    - Validation results for dataset-specific requirements
    """
    validators = {
        'HDFS': validate_hdfs_requirements,
        'Apache': validate_apache_requirements,
        'HealthApp': validate_health_requirements,
        'BGL': validate_bgl_requirements,
        'HPC': validate_hpc_requirements,
        'Linux': validate_linux_requirements,
        'Mac': validate_mac_requirements
    }

    if dataset_type not in validators:
        raise ValueError(f"Unknown dataset type: {dataset_type}")

    return validators[dataset_type](logs)

def validate_field_format(logs, field, format_checker):
    """
    Validate format of specific fields

    Parameters:
    - logs: List of log entries
    - field: Field to validate
    - format_checker: Function to check field format

    Returns:
    - Percentage of valid formats
    """
    total = len(logs)
    valid = sum(1 for log in logs if format_checker(log.get(field)))
    return (valid / total) * 100
```

# Data Quality Dimensions

## 1. Completeness

### Required Fields

- Timestamp (no missing values allowed)
- Component (maximum 1% missing)
- Severity (can be derived if missing)
- Message (no missing values allowed)

### Validation Methods

```
def analyze_log_structure(logs, name):
    """
    Analyze basic structure and completeness of logs

    Parameters:
    - logs: List of log entries
    - name: Name of the log source

    Returns:
    - Dictionary with analysis results
    """
    # Length statistics
    lengths = [len(log) for log in logs]

    results = {
        'min_length': min(lengths),
        'max_length': max(lengths),
        'mean_length': np.mean(lengths),
        'std_length': np.std(lengths),
        'completeness': {
            'timestamp': check_field_completeness(logs, 'timestamp'),
            'component': check_field_completeness(logs, 'component'),
            'severity': check_field_completeness(logs, 'severity'),
            'message': check_field_completeness(logs, 'message')
        }
    }

    return results

def check_field_completeness(logs, field):
    """Check completeness of a specific field"""
    total = len(logs)
    missing = sum(1 for log in logs if not log.get(field))
    return (total - missing) / total * 100
```

## 2. Consistency

### Format Standards

- Timestamps in ISO 8601

- Component names standardized
- Severity levels mapped to standard values
- Message formats validated

### Validation Methods

```python
def validate_consistency(data):
    """
    Validate data format consistency

    Parameters:
    - data: DataFrame with log entries

    Returns:
    - Dictionary with validation results
    """
    return {
        'timestamp_format': check_timestamp_format(data),
        'component_format': check_component_format(data),
        'severity_format': check_severity_format(data),
        'message_format': check_message_format(data)
    }
```

## 3. Accuracy

### Value Validation

- Timestamps within valid range
- Components match known list
- Severity levels valid
- Messages well-formed

### Validation Methods

```python
def validate_accuracy(data):
    """
    Validate data value accuracy

    Parameters:
    - data: DataFrame with log entries

    Returns:
    - Dictionary with validation results
    """
    return {
        'timestamp_valid': check_timestamp_validity(data),
        'component_valid': check_component_validity(data),
        'severity_valid': check_severity_validity(data),
        'message_valid': check_message_validity(data)
    }
```

## 4. Timeliness

### Temporal Requirements

- Logs in chronological order
- No future timestamps
- Reasonable time gaps
- Consistent timezone

### Validation Methods

```python
def validate_timeliness(data):
    """
    Validate temporal aspects of data

    Parameters:
    - data: DataFrame with log entries

    Returns:
    - Dictionary with validation results
    """
    return {
        'chronological_order': check_chronological_order(data),
        'timestamp_range': check_timestamp_range(data),
        'time_gaps': check_time_gaps(data),
        'timezone_consistency': check_timezone_consistency(data)
    }
```

# Quality Control Process

## 1. Initial Validation

```python
def perform_initial_validation(data):
    """
    Perform initial data quality checks

    Parameters:
    - data: Raw log data

    Returns:
    - Validation report
    """
    report = {
        'completeness': validate_completeness(data),
        'consistency': validate_consistency(data),
        'accuracy': validate_accuracy(data),
        'timeliness': validate_timeliness(data)
    }
    return report
```

## 2. Continuous Monitoring

```
def monitor_data_quality(data_stream):
    """
    Monitor data quality in real-time

    Parameters:
    - data_stream: Stream of log entries

    Returns:
    - Quality metrics
    """
    metrics = {
        'completeness_metrics': monitor_completeness(data_stream),
        'consistency_metrics': monitor_consistency(data_stream),
        'accuracy_metrics': monitor_accuracy(data_stream),
        'timeliness_metrics': monitor_timeliness(data_stream)
    }
    return metrics
```

## 3. Quality Improvement

```python
def improve_data_quality(data, validation_report):
    """
    Improve data quality based on validation results

    Parameters:
    - data: DataFrame with log entries
    - validation_report: Quality validation results

    Returns:
    - Improved data
    """
    improved_data = data.copy()

    # Fix completeness issues
    improved_data = fix_completeness_issues(improved_data, validation_report)

    # Fix consistency issues
    improved_data = fix_consistency_issues(improved_data, validation_report)

    # Fix accuracy issues
    improved_data = fix_accuracy_issues(improved_data, validation_report)

    # Fix timeliness issues
    improved_data = fix_timeliness_issues(improved_data, validation_report)

    return improved_data
```

# Quality Thresholds

## 1. Critical Requirements

- Timestamp completeness: 100%
- Message completeness: 100%
- Chronological order: 100%
- Format consistency: 100%

## 2. Warning Thresholds

- Component missing: > 1%
- Severity missing: > 5%
- Time gaps: > 1 hour
- Unknown components: > 0.1%

## 3. Monitoring Thresholds

- Error rate change: > 20%
- Component distribution change: > 10%
- Message pattern change: > 15%
- Performance metric change: > 25%

# Quality Reports

## 1. Validation Report

```python
def generate_validation_report(logs, name):
    """
    Generate comprehensive quality validation report

    Parameters:
    - logs: List of log entries
    - name: Name of the log source

    Returns:
    - Validation report with visualizations
    """
    # Structure analysis
    plt.figure(figsize=(10, 5))
    lengths = [len(log) for log in logs]
    plt.hist(lengths, bins=50)
    plt.title(f"{name} Log Length Distribution")
    plt.xlabel("Log Length")
    plt.ylabel("Frequency")

    report = {
        'structure_analysis': analyze_log_structure(logs, name),
        'completeness': validate_completeness(logs),
        'consistency': validate_consistency(logs),
        'accuracy': validate_accuracy(logs)
    }

    return report
```

## 2. Monitoring Report

```python
def generate_monitoring_report(data_stream):
    """
    Generate real-time quality monitoring report

    Parameters:
    - data_stream: Stream of log entries

    Returns:
    - Monitoring report
    """
    return {
        'current_metrics': calculate_current_metrics(data_stream),
        'trends': analyze_quality_trends(data_stream),
        'alerts': generate_quality_alerts(data_stream),
        'actions': recommend_actions(data_stream)
    }
```

# Best Practices

## 1. Data Collection

- Use standardized logging formats
- Implement proper error handling
- Maintain consistent timestamps
- Validate at source

## 2. Data Processing

- Handle missing values appropriately
- Standardize formats early
- Document transformations
- Maintain data lineage

## 3. Quality Monitoring

- Regular quality assessments
- Automated validation checks

- Alert on quality issues
- Track quality metrics

## 4. Quality Improvement

- Systematic issue resolution
- Root cause analysis
- Continuous monitoring
- Regular reviews