Real Time Data Ingestion Platform

User Documentation

Real Time Data Ingestion Platform

AMOS Group 1

Table of contents

1. Installation

The RTDIP SDK is a PyPi package which can be found here, to install it run the following command:

```
pip install rtdip-sdk
```

1.1 Data Manipulation

1.2 `NormalizationBaseClass`

Bases: `DataManipulationBaseInterface`, `InputValidator`

A base class for applying normalization techniques to multiple columns in a PySpark DataFrame. This class serves as a framework to support various normalization methods (e.g., Z-Score, Min-Max, and Mean), with specific implementations in separate subclasses for each normalization type.

Subclasses should implement specific normalization and denormalization methods by inheriting from this base class.

Example

```
from src.sdk.python.rtdip_sdk.pipelines.data_wranglers import NormalizationZScore
from pyspark.sql import SparkSession
from pyspark.sql.dataframe import DataFrame

normalization = NormalizationZScore(df, column_names=["value_column_1", "value_column_2"], in_place=False)
normalized_df = normalization.filter()
```

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	PySpark DataFrame to be normalized.	*required*
`column_names`	`List[str]`	List of columns in the DataFrame to be normalized.	*required*
`in_place`	`bool`	If true, then result of normalization is stored in the same column.	`False`

NORMALIZATION_NAME_POSTFIX : str Suffix added to the column name if a new column is created for normalized values.

1.2.1 `denormalize(input_df)`

Denormalizes the input DataFrame. Intended to be used by the denormalization component.

Parameters:

Name	Type	Description	Default
`input_df`	`DataFrame`	Dataframe containing the current data.	*required*

1.2.2 `normalize()`

Applies the specified normalization to each column in column_names.

Returns:

Name	Type	Description
`DataFrame`	`DataFrame`	A PySpark DataFrame with the normalized values.

1.2.3 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.3 `NormalizationZScore`

Bases: `NormalizationBaseClass`

1.4 `NormalizationMinMax`

Bases: `NormalizationBaseClass`

1.5 `NormalizationMinMax`

Bases: `NormalizationBaseClass`

1.6 `Denormalization`

Bases: `DataManipulationBaseInterface`, `InputValidator`

Applies the appropriate denormalization method to revert values to their original scale.

Example

```
from src.sdk.python.rtdip_sdk.pipelines.data_wranglers import Denormalization
from pyspark.sql import SparkSession
from pyspark.sql.dataframe import DataFrame

denormalization = Denormalization(normalized_df, normalization)
denormalized_df = denormalization.filter()
```

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	PySpark DataFrame to be reverted to its original scale.	*required*
`normalization_to_revert`	`NormalizationBaseClass`	An instance of the specific normalization subclass (NormalizationZScore, NormalizationMinMax, NormalizationMean) that was originally used to normalize the data.	*required*

1.6.1 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.7 `DuplicateDetection`

Bases: `DataManipulationBaseInterface`, `InputValidator`

Cleanses a PySpark DataFrame from duplicates.

Example

```
from rtdip_sdk.pipelines.monitoring.spark.data_manipulation.duplicate_detection import DuplicateDetection
from pyspark.sql import SparkSession
```

```
from pyspark.sql.dataframe import DataFrame

duplicate_detection_monitor = DuplicateDetection(df, primary_key_columns=["TagName", "EventTime"])

result = duplicate_detection_monitor.filter()
```

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	PySpark DataFrame to be cleansed.	*required*
`primary_key_columns`	`list`	List of column names that serve as primary key for duplicate detection.	*required*

1.7.1 `filter()`

Returns:

Name	Type	Description
`DataFrame`	`DataFrame`	A cleansed PySpark DataFrame from all duplicates based on primary key columns.

1.7.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.8 `IntervalFiltering`

Bases: `DataManipulationBaseInterface`, `InputValidator`

Cleanses a DataFrame by removing rows outside a specified interval window. Supported time stamp columns are DateType and StringType.

Parameters:

Name	Type	Description	Default
`spark`	`SparkSession`	A SparkSession object.	*required*
`df`	`DataFrame`	PySpark DataFrame to be converted	*required*
`interval`	`int`	The interval length for cleansing.	*required*
`interval_unit`	`str`	'hours', 'minutes', 'seconds' or 'milliseconds' to specify the unit of the interval.	*required*
`time_stamp_column_name`	`str`	The name of the column containing the time stamps. Default is 'EventTime'.	`None`
`tolerance`	`int`	The tolerance for the interval. Default is None.	`None`

1.8.1 `filter()`

Filters the DataFrame based on the interval

1.8.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.9 `MissingValueImputation`

Bases: `DataManipulationBaseInterface`, `InputValidator`

Imputes missing values in a univariate time series creating a continuous curve of data points. For that, the time intervals of each individual source is calculated, to then insert empty records at the missing timestamps with NaN values. Through spline interpolation the missing NaN values are calculated resulting in a consistent data set and thus enhance your data quality.

Example

```
from pyspark.sql import SparkSession
from pyspark.sql.dataframe import DataFrame
from pyspark.sql.types import StructType, StructField, StringType
from src.sdk.python.rtdip_sdk.pipelines.data_wranglers.spark.data_manipulation.missing_value_imputation import (
    MissingValueImputation,
)

spark = spark_session()

schema = StructType([
    StructField("TagName", StringType(), True),
    StructField("EventTime", StringType(), True),
    StructField("Status", StringType(), True),
    StructField("Value", StringType(), True)
])

data = [
    ("A2PS64V0J.:ZUX09R", "2024-01-01 03:29:21.000", "Good", "1.0"),
    ("A2PS64V0J.:ZUX09R", "2024-01-01 07:32:55.000", "Good", "2.0"),
    ("A2PS64V0J.:ZUX09R", "2024-01-01 11:36:29.000", "Good", "3.0"),
    ("A2PS64V0J.:ZUX09R", "2024-01-01 15:39:03.000", "Good", "4.0"),
    ("A2PS64V0J.:ZUX09R", "2024-01-01 19:42:37.000", "Good", "5.0"),
    #("A2PS64V0J.:ZUX09R", "2024-01-01 23:46:11.000", "Good", "6.0"), # Test values
    #("A2PS64V0J.:ZUX09R", "2024-01-02 03:49:45.000", "Good", "7.0"),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 07:53:11.000", "Good", "8.0"),
]
df = spark.createDataFrame(data, schema=schema)

missing_value_imputation = MissingValueImputation(spark, df)
result = missing_value_imputation.filter()
```

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	Dataframe containing the raw data.	*required*
`tolerance_percentage`	`int`	Percentage value that indicates how much the time series data points may vary in each interval	`5`

1.9.1 `filter()`

Imputate missing values based on [Spline Interpolation,]

1.9.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.10 `DimensionalityReduction`

Bases: `DataManipulationBaseInterface`

Detects and combines columns based on correlation or exact duplicates.

Example

```
from rtdip_sdk.pipelines.monitoring.spark.data_manipulation.column_correlation import ColumnCorrelationDetection
from pyspark.sql import SparkSession

column_correlation_monitor = ColumnCorrelationDetection(
    df,
    columns_to_check=['column1', 'column2'],
    threshold=0.95,
    combination_method='mean'
)

result = column_correlation_monitor.process()
```

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	PySpark DataFrame to be analyzed and transformed.	*required*
`columns`	`list`	List of column names to check for correlation. Only two columns are supported.	*required*
`threshold`	`float`	Correlation threshold for column combination [0-1]. If the absolute value of the correlation is equal or bigger, than the columns are combined. Defaults to 0.9.	`0.9`
`combination_method`	`str`	Method to combine correlated columns. Supported methods: - 'mean': Average the values of both columns and write the result to the first column (New value = (column1 + column2) / 2) - 'sum': Sum the values of both columns and write the result to the first column (New value = column1 + column2) - 'first': Keep the first column, drop the second column - 'second': Keep the second column, drop the first column - 'delete': Remove both columns entirely from the DataFrame Defaults to 'mean'.	`'mean'`

1.10.1 `filter()`

Process DataFrame by detecting and combining correlated columns.

Returns:

Name	Type	Description
`DataFrame`	`DataFrame`	Transformed PySpark DataFrame

1.10.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.11 `KSigmaAnomalyDetection`

Bases: `DataManipulationBaseInterface`, `InputValidator`

Anomaly detection with the k-sigma method. This method either computes the mean and standard deviation, or the median and the median absolute deviation (MAD) of the data. The k-sigma method then filters out all data points that are k times the standard deviation away from the mean, or k times the MAD away from the median. Assuming a normal distribution, this method keeps around 99.7% of the data points when k=3 and use_median=False.

Example

```
from src.sdk.python.rtdip_sdk.pipelines.data_wranglers.spark.data_manipulation.k_sigma_anomaly_detection import KSigmaAnom

spark = ... # SparkSession
df = ... # Get a PySpark DataFrame

filtered_df = KSigmaAnomalyDetection(
    spark, df, ["<column to filter>"]
).filter()

filtered_df.show()
```

Parameters:

Name	Type	Description	Default
`spark`	`SparkSession`	A SparkSession object.	*required*
`df`	`DataFrame`	Dataframe containing the raw data.	*required*
`column_names`	`list[str]`	The names of the columns to be filtered (currently only one column is supported).	*required*
`k_value`	`float`	The number of deviations to build the threshold.	`3.0`
`use_median`	`book`	If True the median and the median absolute deviation (MAD) are used, instead of the mean and standard deviation.	`False`

1.11.1 `filter()`

Filter anomalies based on the k-sigma rule

1.11.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.12 `OutOfRangeValueFilter`

Bases: `DataManipulationBaseInterface`

Filters data in a DataFrame by checking the 'Value' column against expected ranges for specified TagNames. Logs events when 'Value' exceeds the defined ranges for any TagName and deletes the rows.

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	The DataFrame to monitor.	*required*
`tag_ranges`	`dict`	A dictionary where keys are TagNames and values are dictionaries specifying 'min' and/or 'max', and optionally 'inclusive_bounds' values. Example: { 'A2PS64V0J.:ZUX09R': {'min': 0, 'max': 100, 'inclusive_bounds': True}, 'B3TS64V0K.:ZUX09R': {'min': 10, 'max': 200, 'inclusive_bounds': False}, }	*required*

• **Example**

```
from pyspark.sql import SparkSession
from rtdip_sdk.pipelines.data_manipulation.spark.data_quality.check_value_ranges import DeleteOutOfRangeValues

spark = SparkSession.builder.master("local[1]").appName("DeleteOutOfRangeValuesExample").getOrCreate()

data = [
    ("A2PS64V0J.:ZUX09R", "2024-01-02 03:49:45.000", "Good", 25.0),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 07:53:11.000", "Good", -5.0),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 11:56:42.000", "Good", 50.0),
    ("B3TS64V0K.:ZUX09R", "2024-01-02 16:00:12.000", "Good", 80.0),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 20:03:46.000", "Good", 100.0),
]

columns = ["TagName", "EventTime", "Status", "Value"]

df = spark.createDataFrame(data, columns)

tag_ranges = {
    "A2PS64V0J.:ZUX09R": {"min": 0, "max": 50, "inclusive_bounds": True},
    "B3TS64V0K.:ZUX09R": {"min": 50, "max": 100, "inclusive_bounds": False},
}

delete_out_of_range_values = DeleteOutOfRangeValues(
    df=df,
    tag_ranges=tag_ranges,
)

result_df = delete_out_of_range_values.filter()
```

1.12.1 `filter()`

Executes the value range checking logic for the specified TagNames. Identifies, logs and deletes any rows where 'Value' exceeds the defined ranges for each TagName.

Returns:

Type	Description
`DataFrame`	pyspark.sql.DataFrame: Returns a PySpark DataFrame without the rows that were out of range.

1.12.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.13 `FlatlineFilter`

Bases: `DataManipulationBaseInterface`

Removes and logs rows with flatlining detected in specified columns of a PySpark DataFrame.

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	The input DataFrame to process.	*required*
`watch_columns`	`list`	List of column names to monitor for flatlining (null or zero values).	*required*
`tolerance_timespan`	`int`	Maximum allowed consecutive flatlining period. Rows exceeding this period are removed.	*required*

- **Example**

```
from pyspark.sql import SparkSession
from rtdip_sdk.pipelines.data_manipulation.spark.data_quality.flatline_filter import FlatlineFilter

spark = SparkSession.builder.master("local[1]").appName("FlatlineFilterExample").getOrCreate()

# Example DataFrame
data = [
    (1, "2024-01-02 03:49:45.000", 0.0),
    (1, "2024-01-02 03:50:45.000", 0.0),
    (1, "2024-01-02 03:51:45.000", 0.0),
    (2, "2024-01-02 03:49:45.000", 5.0),
]
columns = ["TagName", "EventTime", "Value"]
df = spark.createDataFrame(data, columns)

filter_flatlining_rows = FlatlineFilter(
    df=df,
    watch_columns=["Value"],
    tolerance_timespan=2,
)

result_df = filter_flatlining_rows.filter()
result_df.show()
```

1.13.1 `filter()`

Removes rows with flatlining detected.

Returns:

Type	Description
`DataFrame`	pyspark.sql.DataFrame: A DataFrame without rows with flatlining detected.

1.14 Data Monitoring

1.15 `CheckValueRanges`

Bases: `MonitoringBaseInterface`, `InputValidator`

Monitors data in a DataFrame by checking the 'Value' column against expected ranges for specified TagNames. Logs events when 'Value' exceeds the defined ranges for any TagName.

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	The DataFrame to monitor.	*required*
`tag_ranges`	`dict`	A dictionary where keys are TagNames and values are dictionaries specifying 'min' and/or 'max', and optionally 'inclusive_bounds' values. Example: { 'A2PS64V0J.:ZUX09R': {'min': 0, 'max': 100, 'inclusive_bounds': True}, 'B3TS64V0K.:ZUX09R': {'min': 10, 'max': 200, 'inclusive_bounds': False}, }	*required*

- **Example**

```
from pyspark.sql import SparkSession
from rtdip_sdk.pipelines.monitoring.spark.data_quality.check_value_ranges import CheckValueRanges

spark = SparkSession.builder.master("local[1]").appName("CheckValueRangesExample").getOrCreate()

data = [
    ("A2PS64V0J.:ZUX09R", "2024-01-02 03:49:45.000", "Good", 25.0),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 07:53:11.000", "Good", -5.0),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 11:56:42.000", "Good", 50.0),
    ("B3TS64V0K.:ZUX09R", "2024-01-02 16:00:12.000", "Good", 80.0),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 20:03:46.000", "Good", 100.0),
]

columns = ["TagName", "EventTime", "Status", "Value"]

df = spark.createDataFrame(data, columns)

tag_ranges = {
    "A2PS64V0J.:ZUX09R": {"min": 0, "max": 50, "inclusive_bounds": True},
    "B3TS64V0K.:ZUX09R": {"min": 50, "max": 100, "inclusive_bounds": False},
}

check_value_ranges = CheckValueRanges(
    df=df,
    tag_ranges=tag_ranges,
)

result_df = check_value_ranges.check()
```

1.15.1 `check()`

Executes the value range checking logic for the specified TagNames. Identifies and logs any rows where 'Value' exceeds the defined ranges for each TagName.

Returns:

Type	Description
`DataFrame`	pyspark.sql.DataFrame: Returns the original PySpark DataFrame without changes.

1.15.2 `check_for_out_of_range()`

Identifies rows where 'Value' exceeds defined ranges.

Returns: pyspark.sql.DataFrame: A DataFrame containing rows with out-of-range values.

1.15.3 `log_out_of_range_values(out_of_range_df)`

Logs out-of-range values for all TagNames.

1.15.4 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.16 `FlatlineDetection`

Bases: `MonitoringBaseInterface`, `InputValidator`

Detects flatlining in specified columns of a PySpark DataFrame and logs warnings.

Flatlining occurs when a column contains consecutive null or zero values exceeding a specified tolerance period. This class identifies such occurrences and logs the rows where flatlining is detected.

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	The input DataFrame to monitor for flatlining.	*required*
`watch_columns`	`list`	List of column names to monitor for flatlining (null or zero values).	*required*
`tolerance_timespan`	`int`	Maximum allowed consecutive flatlining period. If exceeded, a warning is logged.	*required*

- **Example**

```
from rtdip_sdk.pipelines.monitoring.spark.data_manipulation.flatline_detection import FlatlineDetection
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[1]").appName("FlatlineDetectionExample").getOrCreate()

# Example DataFrame
data = [
    (1, 1),
    (2, 0),
    (3, 0),
    (4, 0),
    (5, 5),
]
columns = ["ID", "Value"]
df = spark.createDataFrame(data, columns)

# Initialize FlatlineDetection
flatline_detection = FlatlineDetection(
    df,
    watch_columns=["Value"],
    tolerance_timespan=2
)

# Detect flatlining
flatline_detection.check()
```

1.16.1 `check()`

Detects flatlining and logs relevant rows.

Returns:

Type	Description
`DataFrame`	pyspark.sql.DataFrame: The original DataFrame with additional flatline detection metadata.

1.16.2 `check_for_flatlining()`

Identifies rows with flatlining based on the specified columns and tolerance.

Returns:

Type	Description
`DataFrame`	pyspark.sql.DataFrame: A DataFrame containing rows with flatlining detected.

1.16.3 `log_flatlining_rows(flatlined_rows)`

Logs flatlining rows for all monitored columns.

Parameters:

Name	Type	Description	Default
`flatlined_rows`	`DataFrame`	The DataFrame containing rows with flatlining detected.	*required*

1.16.4 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.17 `IdentifyMissingDataInterval`

Bases: `MonitoringBaseInterface`, `InputValidator`

Detects missing data intervals in a DataFrame by identifying time differences between consecutive measurements that exceed a specified tolerance or a multiple of the Median Absolute Deviation (MAD). Logs the start and end times of missing intervals along with their durations.

Parameters:

Name	Type	Description	Default
`df`	`Dataframe`	DataFrame containing at least the 'EventTime' column.	*required*
`interval`	`str`	Expected interval between data points (e.g., '10ms', '500ms'). If not specified, the median of time differences is used.	`None`
`tolerance`	`str`	Tolerance time beyond which an interval is considered missing (e.g., '10ms'). If not specified, it defaults to 'mad_multiplier' times the Median Absolute Deviation (MAD) of time differences.	`None`
`mad_multiplier`	`float`	Multiplier for MAD to calculate tolerance. Default is 3.	`3`
`min_tolerance`	`str`	Minimum tolerance for pattern-based detection (e.g., '100ms'). Default is '10ms'.	`'10ms'`

Returns:

Name	Type	Description
`df`	`Dataframe`	Returns the original PySparkDataFrame without changes.

Example

```python from rtdip_sdk.pipelines.monitoring.spark.data_manipulation import IdentifyMissingDataInterval from pyspark.sql import SparkSession

```
missing_data_monitor = IdentifyMissingDataInterval(
    df=df,
    interval='100ms',
    tolerance='10ms',
)

df_result = missing_data_monitor.check()
```
```

1.17.1 `check()`

Executes the identify missing data logic.

Returns:

Type	Description
`DataFrame`	pyspark.sql.DataFrame: Returns the original PySpark DataFrame without changes.

1.17.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.18 `IdentifyMissingDataPattern`

Bases: `MonitoringBaseInterface`, `InputValidator`

Identifies missing data in a DataFrame based on specified time patterns. Logs the expected missing times.

Parameters:

Name	Type	Description	Default
`df`	`Dataframe`	DataFrame containing at least the 'EventTime' column.	*required*
`patterns`	`list of dict`	List of dictionaries specifying the time patterns. - For 'minutely' frequency: Specify 'second' and optionally 'millisecond'. Example: [{'second': 0}, {'second': 13}, {'second': 49}] - For 'hourly' frequency: Specify 'minute', 'second', and optionally 'millisecond'. Example: [{'minute': 0, 'second': 0}, {'minute': 30, 'second': 30}]	*required*
`frequency`	`str`	Frequency of the patterns. Must be either 'minutely' or 'hourly'. - 'minutely': Patterns are checked every minute at specified seconds. - 'hourly': Patterns are checked every hour at specified minutes and seconds.	`'minutely'`
`tolerance`	`str`	Maximum allowed deviation from the pattern (e.g., '1s', '500ms'). Default is '10ms'.	`'10ms'`

- **Example**

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[1]").appName("IdentifyMissingDataPatternExample").getOrCreate()

patterns = [
    {"second": 0},
    {"second": 20},
]

frequency = "minutely"
tolerance = "1s"

identify_missing_data = IdentifyMissingDataPattern(
    df=df,
    patterns=patterns,
    frequency=frequency,
    tolerance=tolerance,
)

identify_missing_data.check()
```

1.18.1 `check()`

Executes the missing pattern detection logic. Identifies and logs any missing patterns based on the provided patterns and frequency within the specified tolerance.

Returns:

Type	Description
`DataFrame`	pyspark.sql.DataFrame: Returns the original PySpark DataFrame without changes.

1.18.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.19 `MovingAverage`

Bases: `MonitoringBaseInterface`, `InputValidator`

Computes and logs the moving average over a specified window size for a given PySpark DataFrame.

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	The DataFrame to process.	*required*
`window_size`	`int`	The size of the moving window.	*required*

- **Example**

```
from pyspark.sql import SparkSession
from rtdip_sdk.pipelines.monitoring.spark.data_quality.moving_average import MovingAverage
```

```
spark = SparkSession.builder.master("local[1]").appName("MovingAverageExample").getOrCreate()

data = [
    ("A2PS64V0J.:ZUX09R", "2024-01-02 03:49:45.000", "Good", 1.0),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 07:53:11.000", "Good", 2.0),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 11:56:42.000", "Good", 3.0),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 16:00:12.000", "Good", 4.0),
    ("A2PS64V0J.:ZUX09R", "2024-01-02 20:03:46.000", "Good", 5.0),
]

columns = ["TagName", "EventTime", "Status", "Value"]

df = spark.createDataFrame(data, columns)

moving_avg = MovingAverage(
    df=df,
    window_size=3,
)

moving_avg.check()
```

1.19.1 `check()`

Computes and logs the moving average using a specified window size.

1.19.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.20 Forecasts

1.21 `ArimaPrediction`

Bases: `DataManipulationBaseInterface`, `InputValidator`

Extends the timeseries data in given DataFrame with forecasted values from an ARIMA model. It forecasts a value column of the given time series dataframe based on the historical data points and constructs full entries based on the preceding timestamps. It is advised to place this step after the missing value imputation to prevent learning on dirty data.

It supports dataframes in a source-based format (where each row is an event by a single sensor) and column-based format (where each row is a point in time).

The similar component AutoArimaPrediction wraps around this component and needs less manual parameters set.

ARIMA-Specific parameters can be viewed at the following statsmodels documentation page: [ARIMA Documentation]

Example

```
import numpy as np
import matplotlib.pyplot as plt
import numpy.random
import pandas
```

```
from pyspark.sql import SparkSession

from rtdip_sdk.pipelines.data_quality.forecasting.spark.arima import ArimaPrediction

import rtdip_sdk.pipelines._pipeline_utils.spark as spark_utils

spark_session = SparkSession.builder.master("local[2]").appName("test").getOrCreate()
df = pandas.DataFrame()

numpy.random.seed(0)
arr_len = 250
h_a_l = int(arr_len / 2)
df['Value'] = np.random.rand(arr_len) + np.sin(np.linspace(0, arr_len / 10, num=arr_len))
df['Value2'] = np.random.rand(arr_len) + np.cos(np.linspace(0, arr_len / 2, num=arr_len)) + 5
df['index'] = np.asarray(pandas.date_range(start='1/1/2024', end='2/1/2024', periods=arr_len))
df = df.set_index(pandas.DatetimeIndex(df['index']))

learn_df = df.head(h_a_l)

# plt.plot(df['Value'])
# plt.show()

input_df = spark_session.createDataFrame(
        learn_df,
        ['Value', 'Value2', 'index'],
)
arima_comp = ArimaPrediction(input_df, to_extend_name='Value', number_of_data_points_to_analyze=h_a_l, number_of_data_poin
                     order=(3,0,0), seasonal_order=(3,0,0,62))
forecasted_df = arima_comp.filter().toPandas()
print('Done')
```

Parameters:

Name	Type	Description	Default
`past_data`	`DataFrame`	PySpark DataFrame which contains training data	*required*
`to_extend_name`	`str`	Column or source to forecast on	*required*
`past_data_style`	`InputStyle`	In which format is past_data formatted	`None`
`value_name`	`str`	Name of column in source-based format, where values are stored	`None`
`timestamp_name`	`str`	Name of column, where event timestamps are stored	`None`
`source_name`	`str`	Name of column in source-based format, where source of events are stored	`None`
`status_name`	`str`	Name of column in source-based format, where status of events are stored	`None`
`external_regressor_names`	`List[str]`	Currently not working. Names of the columns with data to use for prediction, but not extend	`None`
`number_of_data_points_to_predict`	`int`	Amount of points to forecast	`50`
`number_of_data_points_to_analyze`	`int`	Amount of most recent points to train on	`None`
`order`	`tuple`	ARIMA-Specific setting	`(0, 0, 0)`
`seasonal_order`	`tuple`	ARIMA-Specific setting	`(0, 0, 0, 0)`
`trend`	`str`	ARIMA-Specific setting	`None`
`enforce_stationarity`	`bool`	ARIMA-Specific setting	`True`
`enforce_invertibility`	`bool`	ARIMA-Specific setting	`True`
`concentrate_scale`	`bool`	ARIMA-Specific setting	`False`
`trend_offset`	`int`	ARIMA-Specific setting	`1`
`missing`	`str`	ARIMA-Specific setting	`'None'`

1.21.1 `InputStyle`

Bases: `Enum`

Used to describe style of a dataframe

1.21.2 `filter()`

Forecasts a value column of a given time series dataframe based on the historical data points using ARIMA.

Constructs full entries based on the preceding timestamps. It is advised to place this step after the missing value imputation to prevent learning on dirty data.

Returns:

Name	Type	Description
`DataFrame`	`DataFrame`	A PySpark DataFrame with forecasted value entries depending on constructor parameters.

1.21.3 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.22 `ArimaAutoPrediction`

Bases: `ArimaPrediction`

A wrapper for ArimaPrediction which uses pmdarima auto_arima for data prediction. It selectively tries various sets of p and q (also P and Q for seasonal models) parameters and selects the model with the minimal AIC.

Example

```
import numpy as np
import matplotlib.pyplot as plt
import numpy.random
import pandas
from pyspark.sql import SparkSession

from rtdip_sdk.pipelines.data_quality.forecasting.spark.arima import ArimaPrediction

import rtdip_sdk.pipelines._pipeline_utils.spark as spark_utils
from rtdip_sdk.pipelines.data_quality.forecasting.spark.auto_arima import ArimaAutoPrediction

spark_session = SparkSession.builder.master("local[2]").appName("test").getOrCreate()
df = pandas.DataFrame()

numpy.random.seed(0)
arr_len = 250
h_a_l = int(arr_len / 2)
df['Value'] = np.random.rand(arr_len) + np.sin(np.linspace(0, arr_len / 10, num=arr_len))
df['Value2'] = np.random.rand(arr_len) + np.cos(np.linspace(0, arr_len / 2, num=arr_len)) + 5
df['index'] = np.asarray(pandas.date_range(start='1/1/2024', end='2/1/2024', periods=arr_len))
df = df.set_index(pandas.DatetimeIndex(df['index']))

learn_df = df.head(h_a_l)

# plt.plot(df['Value'])
# plt.show()

input_df = spark_session.createDataFrame(
        learn_df,
        ['Value', 'Value2', 'index'],
)
arima_comp = ArimaAutoPrediction(input_df, to_extend_name='Value', number_of_data_points_to_analyze=h_a_l, number_of_data_
                     seasonal=True)
forecasted_df = arima_comp.filter().toPandas()
print('Done')
```

Parameters:

Name	Type	Description	Default
`past_data`	`DataFrame`	PySpark DataFrame which contains training data	*required*
`to_extend_name`	`str`	Column or source to forecast on	`None`
`past_data_style`	`InputStyle`	In which format is past_data formatted	`None`
`value_name`	`str`	Name of column in source-based format, where values are stored	`None`
`timestamp_name`	`str`	Name of column, where event timestamps are stored	`None`
`source_name`	`str`	Name of column in source-based format, where source of events are stored	`None`
`status_name`	`str`	Name of column in source-based format, where status of events are stored	`None`
`external_regressor_names`	`List[str]`	Currently not working. Names of the columns with data to use for prediction, but not extend	`None`
`number_of_data_points_to_predict`	`int`	Amount of points to forecast	`50`
`number_of_data_points_to_analyze`	`int`	Amount of most recent points to train on	`None`
`seasonal`	`bool`	Setting for AutoArima, is past_data seasonal?	`False`
`enforce_stationarity`	`bool`	ARIMA-Specific setting	`True`
`enforce_invertibility`	`bool`	ARIMA-Specific setting	`True`
`concentrate_scale`	`bool`	ARIMA-Specific setting	`False`
`trend_offset`	`int`	ARIMA-Specific setting	`1`
`missing`	`str`	ARIMA-Specific setting	`'None'`

1.23 `LinearRegression`

Bases: `MachineLearningInterface`

This function uses pyspark.ml.LinearRegression to train a linear regression model on time data. And the uses the model to predict next values in the time series.

Parameters:

Name	Type	Description	Default
`df`	`Dataframe`	DataFrame containing the features and labels.	*required*
`features_col`	`str`	Name of the column containing the features (the input). Default is 'features'.	`'features'`
`label_col`	`str`	Name of the column containing the label (the input). Default is 'label'.	`'label'`
`prediction_col`	`str`	Name of the column to which the prediction will be written. Default is 'prediction'.	`'prediction'`

Returns: PySparkDataFrame: Returns the original PySpark DataFrame without changes.

1.23.1 `evaluate(test_df)`

Evaluates the trained model using RMSE.

Parameters:

Name	Type	Description	Default
`test_df`	`DataFrame`	The testing dataset to evaluate the model.	*required*

Returns:

Type	Description
`Optional[float]`	Optional[float]: The Root Mean Squared Error (RMSE) of the model or None if the prediction columnd doesn't exist.

1.23.2 `predict(prediction_df)`

Predicts the next values in the time series.

1.23.3 `split_data(train_ratio=0.8)`

Splits the dataset into training and testing sets.

Parameters:

Name	Type	Description	Default
`train_ratio`	`float`	The ratio of the data to be used for training. Default is 0.8 (80% for training).	`0.8`

Returns:

Type	Description
`tuple[DataFrame, DataFrame]`	tuple[DataFrame, DataFrame]: Returns the training and testing datasets.

1.23.4 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.23.5 `train(train_df)`

Trains a linear regression model on the provided data.

1.24 `KNearestNeighbors`

Bases: `MachineLearningInterface`

Implements the K-Nearest Neighbors (KNN) algorithm to predict missing values in a dataset. This component is compatible with time series data and supports customizable weighted or unweighted averaging for predictions.

Example:

```
from src.sdk.python.rtdip_sdk.pipelines.machine_learning.spark.k_nearest_neighbors import KNearestNeighbors
from pyspark.ml.feature import StandardScaler, VectorAssembler
from pyspark.sql import SparkSession
spark = ... # SparkSession
raw_df = ... # Get a PySpark DataFrame
assembler = VectorAssembler(inputCols=["feature1", "feature2"], outputCol="assembled_features")
df = assembler.transform(raw_df)
scaler = StandardScaler(inputCol="assembled_features", outputCol="features", withStd=True, withMean=True)
scaled_df = scaler.fit(df).transform(df)
knn = KNearestNeighbors(
    df=scaled_df,
    features_col="features",
    label_col="label",
    timestamp_col="timestamp",
    k=3,
    weighted=True,
```

```
    distance_metric="combined",  # Options: "euclidean", "temporal", "combined"
    temporal_weight=0.3  # Weight for temporal distance when using combined metric
)
train_df, test_df = knn.randomSplit([0.8, 0.2], seed=42)
knn.train(train_df)
predictions = knn.predict(test_df)
```

Parameters:

```
df (pyspark.sql.Dataframe): DataFrame containing the features and labels
features_col (str): Name of the column containing the features (the input). Default is 'features'
label_col (str): Name of the column containing the label (the input). Default is 'label'
timestamp_col (str, optional): Name of the column containing timestamps
k (int): The number of neighbors to consider in the KNN algorithm. Default is 3
weighted (bool): Whether to use weighted averaging based on distance. Default is False (unweighted averaging)
distance_metric (str): Type of distance calculation ("euclidean", "temporal", or "combined")
temporal_weight (float): Weight for temporal distance in combined metric (0 to 1)
```

1.24.1 `predict(test_df)`

Predicts labels using the specified distance metric.

1.24.2 `train(train_df)`

Sets up the training DataFrame including temporal information if specified.

1.25 `DataBinning`

Bases: `MachineLearningInterface`

Data binning using clustering methods. This method partitions the data points into a specified number of clusters (bins) based on the specified column. Each data point is assigned to the nearest cluster center.

Example

```
from src.sdk.python.rtdip_sdk.pipelines.machine_learning.spark.data_binning import DataBinning

df = ... # Get a PySpark DataFrame with features column

binning = DataBinning(
    df=df,
    column_name="features",
    bins=3,
    output_column_name="bin",
    method="kmeans"
)
binned_df = binning.train().predict()
binned_df.show()
```

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	Dataframe containing the input data.	*required*
`column_name`	`str`	The name of the input column to be binned (default: "features").	`'features'`
`bins`	`int`	The number of bins/clusters to create (default: 2).	`2`
`output_column_name`	`str`	The name of the output column containing bin assignments (default: "bin").	`'bin'`
`method`	`str`	The binning method to use. Currently only supports "kmeans".	`'kmeans'`

1.25.1 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

1.25.2 `train()`

Filter anomalies based on the k-sigma rule