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
```

2. Machine Learning

2.1 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'`

2.1.1 system_type() staticmethod

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

2.1.2 train()

Filter anomalies based on the k-sigma rule

2.2 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.

2.2.1 `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:

Name	Type	Description
`float`		The Root Mean Squared Error (RMSE) of the model.

2.2.2 `predict(prediction_df)`

Predicts the next values in the time series.

2.2.3 `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:

Name	Type	Description
`DataFrame`		Returns the training and testing datasets.

2.2.4 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

2.2.5 `train(train_df)`

Trains a linear regression model on the provided data.

3. Monitoring

3.1 `FlatlineDetection`

Bases: `MonitoringBaseInterface`

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_quality.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()
```

3.1.1 `check()`

Detects flatlining in the specified columns and logs warnings if detected.

Returns:

Type	Description
`DataFrame`	pyspark.sql.DataFrame: The original PySpark DataFrame unchanged.

3.1.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

3.2 `CheckValueRanges`

Bases: `MonitoringBaseInterface`

Monitors data in a DataFrame by checking specified columns against expected value ranges. Logs events when values exceed the specified ranges.

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	The DataFrame to monitor.	*required*
`columns_ranges`	`dict`	A dictionary where keys are column names and values are dictionaries specifying 'min' and/or 'max', and optionally 'inclusive' values. Example: { 'temperature': {'min': 0, 'max': 100, 'inclusive': 'both'}, 'pressure': {'min': 10, 'max': 200, 'inclusive': 'left'}, 'humidity': {'min': 30} # Uses default inclusive }	*required*
`default_inclusive`	`str`	Default inclusivity setting if not specified per column. Can be 'both', 'neither', 'left', or 'right'. Default is 'both'. - 'both': min <= value <= max - 'neither': min < value < max - 'left': min <= value < max - 'right': min < value <= max	`'both'`

- **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 = [
    (1, 25, 100),
    (2, -5, 150),
    (3, 50, 250),
    (4, 80, 300),
    (5, 100, 50),
]

columns = ["ID", "temperature", "pressure"]

df = spark.createDataFrame(data, columns)

columns_ranges = {
    "temperature": {"min": 0, "max": 100, "inclusive": "both"},
    "pressure": {"min": 50, "max": 200, "inclusive": "left"},
}

check_value_ranges = CheckValueRanges(
    df=df,
```

```
    columns_ranges=columns_ranges,
    default_inclusive="both",
)

result_df = check_value_ranges.check()
```

3.2.1 `check()`

Executes the value range checking logic. Identifies and logs any rows where specified columns exceed their defined value ranges.

Returns:

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

3.2.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

3.3 `IdentifyMissingDataInterval`

Bases: `MonitoringBaseInterface`

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'`

- **Example**

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

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

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

3.3.1 `check()`

Executes the identify missing data logic.

Returns:

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

3.3.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

3.4 `IdentifyMissingDataPattern`

Bases: `MonitoringBaseInterface`

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()
```

3.4.1 `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.

3.4.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

4. Data Wranglers

4.1 `MissingValueImputation`

Bases: `WranglerBaseInterface`

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_quality.missing_value_imputation import (MissingValueImputation,)

@pytest.fixture(scope="session") def spark_session(): return SparkSession.builder.master("local[2]").appName("test").getOrCreate()

spark = spark_session()

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

data = [# Setup controlled Test ("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"), ("A2PS64V0J.:ZUX09R", "2024-01-02 11:56:42.000", "Good", "9.0"), ("A2PS64V0J.:ZUX09R", "2024-01-02 16:00:12.000", "Good", "10.0"), ("A2PS64V0J.:ZUX09R", "2024-01-02 20:13:46.000", "Good", "11.0"), # Tolerance Test ("A2PS64V0J.:ZUX09R", "2024-01-03 00:07:20.000", "Good", "10.0"), #("A2PS64V0J.:ZUX09R", "2024-01-03 04:10:54.000", "Good", "9.0"), ("A2PS64V0J.:ZUX09R", "2024-01-03 08:14:28.000", "Good", "8.0"), ("-4O7LSSAM_3EA02:2GT7E02I_R_MP", "31.12.2023 00:01:43", "Good", "4686.259766"), ("-4O7LSSAM_3EA02:2GT7E02I_R_MP", "31.12.2023 00:02:44", "Good", "4691.161621"), ("-4O7LSSAM_3EA02:2GT7E02I_R_MP", "31.12.2023 00:04:44", "Good", "4686.259766"), ("-4O7LSSAM_3EA02:2GT7E02I_R_MP", "31.12.2023 00:05:44", "Good", "4691.161621"), ("-4O7LSSAM_3EA02:2GT7E02I_R_MP", "31.12.2023 00:11:46", "Good", "4686.259766"), ("-4O7LSSAM_3EA02:2GT7E02I_R_MP", "31.12.2023 00:13:46", "Good", "4691.161621"), ("-4O7LSSAM_3EA02:2GT7E02I_R_MP", "31.12.2023 00:16:47", "Good", "4691.161621"), ("-4O7LSSAM_3EA02:2GT7E02I_R_MP", "31.12.2023 00:19:48", "Good", "4696.063477"), ("-4O7LSSAM_3EA02:2GT7E02I_R_MP", "31.12.2023 00:20:48", "Good", "4691.161621"),]

df = spark.createDataFrame(data, schema=schema)

missing_value_imputation = MissingValueImputation(spark, df) imputed_df = missing_value_imputation.filter()

print(imputed_df.show(imputed_df.count(), False))

```

Parameters: df (DataFrame): Dataframe containing the raw data. tolerance_percentage (int): Percentage value that indicates how much the time series data points may vary in each interval

### 4.1.1 `filter()`

Imputate missing values based on [Spline Interpolation, ]

### 4.1.2 `system_type()` `staticmethod`

**Attributes:**

| Name | Type | Description |
|---|---|---|
| `SystemType` | `Environment` | Requires PYSPARK |

## 4.2 `IntervalFiltering`

Bases: `WranglerBaseInterface`

Cleanses a DataFrame by removing rows outside a specified interval window. Example:

**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. interval_unit (str): 'hours', 'minutes', 'seconds' or 'milliseconds' to specify the unit of the interval. | *required* |

### 4.2.1 `filter()`

Filters the DataFrame based on the interval
```

4.2.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

4.3 `ArimaPrediction`

Bases: `WranglerBaseInterface`

Extends a column in given DataFrame with a ARIMA model.

- **ARIMA-Specific parameters can be viewed at the following statsmodels documentation page**
  https://www.statsmodels.org/dev/generated/statsmodels.tsa.arima.model.ARIMA.html

Example

```
    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(pd.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, column_name='Value', number_of_data_points_to_analyze=h_a_l, number_of_data_points_
                     order=(3,0,0), seasonal_order=(3,0,0,62))
forecasted_df = arima_comp.filter()
```

Parameters:

Name	Type	Description	Default
past_data	DataFrame	PySpark DataFrame to extend	*required*
column_name	str	Name of the column to be extended	*required*
timestamp_column_name	str	Name of the column containing timestamps external_regressor_column_names (List[str]): Names of the columns with data to use for prediction, but not extend	None
number_of_data_points_to_predict	int	Amount of most recent rows used to create the model	50
number_of_data_points_to_analyze	int	Amount of rows to predict with the model	None
order	tuple	ARIMA-Specific setting	(0, 0, 0)
seasonal_order	tuple	ARIMA-Specific setting	(0, 0, 0, 0)
trend	str	ARIMA-Specific setting	'c'
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'

4.3.1 filter()

Predicts value to predict and extends the in the constructor inputted Dataframe.

Other columns will be filled with NaN other similar None values.

Returns:

Name	Type	Description
DataFrame	DataFrame	A PySpark DataFrame with extended index and filled column_to_predict.

4.3.2 system_type() staticmethod

Attributes:

Name	Type	Description
SystemType	Environment	Requires PYSPARK

4.4 DuplicateDetection

Bases: WranglerBaseInterface

Cleanses a PySpark DataFrame from duplicates.

Example

```
from rtdip_sdk.pipelines.monitoring.spark.data_quality.duplicate_detection import DuplicateDetection
from pyspark.sql import SparkSession
from pyspark.sql.dataframe import DataFrame
from pyspark.sql.functions import desc

duplicate_detection_monitor = DuplicateDetection(df)

result = duplicate_detection_monitor.filter()
```

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	PySpark DataFrame to be converted	*required*

4.4.1 `filter()`

Returns:

Name	Type	Description
`DataFrame`	`DataFrame`	A cleansed PySpark DataFrame from all the duplicates.

4.4.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

4.5 `KSigmaAnomalyDetection`

Bases: `WranglerBaseInterface`

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_quality.k_sigma_anomaly_detection import KSigmaAnomalyDe

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`

4.5.1 `filter()`

Filter anomalies based on the k-sigma rule

4.5.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

4.6 Normalization

4.7 `Denormalization`

Bases: `WranglerBaseInterface`

4.7.1 TODO

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*

4.7.2 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

4.8 `NormalizationBaseClass`

Bases: `WranglerBaseInterface`

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.

4.8.1 `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*

4.8.2 `normalize()`

Applies the specified normalization to each column in column_names.

Returns:

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

4.8.3 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

4.9 `NormalizationMean`

Bases: `NormalizationBaseClass`

4.10 `NormalizationMinMax`

Bases: `NormalizationBaseClass`

4.11 `NormalizationZScore`

Bases: `NormalizationBaseClass`

5. Transformers

5.1 `ColsToVector`

Bases: `TransformerInterface`

Converts columns containing numbers to a column containing a vector.

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	PySpark DataFrame	*required*
`input_cols`	`list[str]`	List of columns to convert to a vector.	*required*
`output_col`	`str`	Name of the output column where the vector will be stored.	*required*
`override_col`	`bool`	If True, the output column can override an existing column.	`False`

5.1.1 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

5.2 `PolynomialFeatures`

Bases: `TransformerInterface`

This transformer takes a vector column and generates polynomial combinations of the input features up to the specified degree. For example, if the input vector is [a, b] and degree=2, the output features will be [a, b, a^2, ab, b^2].

Parameters:

Name	Type	Description	Default
`df`	`DataFrame`	PySpark DataFrame	*required*
`input_col`	`str`	Name of the input column in the DataFrame that contains the feature vectors	*required*
`output_col`	`str`		*required*
`poly_degree`	`int`	The degree of the polynomial features to generate	*required*
`override_col`	`bool`	If True, the output column can override an existing column.	`False`

5.2.1 `system_type()` `staticmethod`

Attributes:

Name	Type	Description
`SystemType`	`Environment`	Requires PYSPARK

6. Utilities

6.1 `parse_time_string_to_ms(time_str)`

Parses a time string and returns the total time in milliseconds.

Parameters:

Name	Type	Description	Default
`time_str`	`str`	Time string (e.g., '10ms', '1s', '2m', '1h').	*required*

Returns:

Name	Type	Description
`float`	`float`	Total time in milliseconds.

Raises:

Type	Description
`ValueError`	If the format is invalid.