Dataset Generator Configuration Documentation

Overview

This document describes the JSON configuration format for generating synthetic datasets for data science instruction, including support for time series with lagged features and seasonality.

Root Structure

```
{
  "dataset_config": {
    "name": "string",
    "description": "string",
    "random_seed": integer,
    "n_rows": integer,
    "correlations": [...],
    "features": [...],
    "target": {...}
  }
}
```

Root Fields

Field	Type	Required	Description
`name`	string	Yes	Unique identifier for the dataset
`description`	string	No	Human-readable description
`random_seed`	integer	No	Seed for reproducibility (omit for random)
`n_rows`	integer	Yes	Number of observations to generate

Features Array

Each feature is defined as an object with the following structure:

```
{
  "name": "string",
  "description": "string",
  "data_type": "float|int|categorical",
  "distribution": {...},
  "missing_rate": 0.0,
  "outlier_rate": 0.0,
  "outlier_method": "string",
  "outlier_multiplier": float,
  "lags": [1, 2, 3]
}
```

Feature Fields

Field	Type	Required	Description
`name`	string	Yes	Variable name (valid Python identifier)
`description`	string	No	Human-readable description
`data_type`	string	Yes	One of: `float`, `int`, `categorical`, `datetime`
`distribution`	object	Yes	Distribution specification (see below)
`missing_rate`	float	No	Proportion of missing values (0.0-1.0), default: 0.0

Field	Type	Required	Description
`outlier_rate`	float	No	Proportion of outliers (0.0-1.0), default: 0.0 (not applicable to datetime)
`outlier_method`	string	No	One of: `extreme_high`, `extreme_low`, `extreme_both`
`outlier_multiplier`	float	No	Multiplier for outlier generation, default: 3.0
`lags`	array	No	List of lag periods (e.g., [1, 2, 3]) for time series

---

Time Series: Lagged Features

For time series datasets, you can specify lag periods to automatically generate lagged versions of features.

Example:

```json
{
  "name": "price",
  "data_type": "float",
  "distribution": {
    "type": "normal",
    "mean": 100,
    "std": 10
  },
  "lags": [1, 2, 3]
}
```

Process: 1. The base feature `price` is generated according to the distribution 2. Lagged features `price_lag1`, `price_lag2`, `price_lag3` are automatically created 3. First N rows of each lagged feature contain NaN (where N = lag period) 4. Lagged features can be used in target expressions: `"expression": "0.8*price + 0.15*price_lag1"`

Use Cases: - Stock price prediction using historical prices - Sales forecasting with previous period sales - Temperature prediction with past temperatures - Any autoregressive time series model

---

Distribution Types

1. Uniform Distribution

Generates values uniformly between min and max.

```json
{
  "type": "uniform",
  "min": 0.0,
  "max": 1.0
}
```

Example:

```json
{
  "name": "random_value",
  "data_type": "float",
  "distribution": {
    "type": "uniform",
    "min": 0,
    "max": 100
  }
}
```

2. Normal Distribution

Generates values from a normal (Gaussian) distribution.

```json
{
  "type": "normal",
  "mean": 0.0,
  "std": 1.0,
  "min_clip": null,
  "max_clip": null
}
```

Parameters: - `mean`: Center of distribution - `std`: Standard deviation - `min_clip`: Optional minimum value (clips lower values) - `max_clip`: Optional maximum value (clips upper values)

Example:

```json
{
  "name": "test_score",
  "data_type": "float",
  "distribution": {
    "type": "normal",
    "mean": 75,
    "std": 10,
    "min_clip": 0,
    "max_clip": 100
  }
}
```

3. Weibull Distribution

Generates values from a 3-parameter Weibull distribution (useful for skewed data).

```json
{
  "type": "weibull",
  "shape": 1.5,
  "scale": 1.0,
  "location": 0.0
}
```

Parameters: - `shape`: Shape parameter (k) - controls skewness - `scale`: Scale parameter ( ) - stretches/compresses - `location`: Location parameter - shifts distribution

Example:

```json
{
  "name": "customer_lifetime",
  "data_type": "int",
  "distribution": {
    "type": "weibull",
    "shape": 1.2,
    "scale": 24,
    "location": 1
  }
}
```

4. Random Walk

Generates values that evolve randomly from a starting point.

```json
{
  "type": "random_walk",
  "start": 100.0,
  "step_size": 1.0,
  "drift": 0.0
}
```

Parameters: - `start`: Initial value - `step_size`: Maximum step size per observation - `drift`: Directional bias per step (positive = upward trend)

Example:

```json
{
  "name": "stock_price",
  "data_type": "float",
  "distribution": {
    "type": "random_walk",
    "start": 100.0,
    "step_size": 2.5,
    "drift": 0.1
  }
}
```

5. Sequential

Generates sequential integers (useful for IDs or time indices).

```json
{
  "type": "sequential",
  "start": 1,
  "step": 1
}
```

Example:

```json
{
  "name": "day",
  "data_type": "int",
  "distribution": {
    "type": "sequential",
    "start": 1,
    "step": 1
  }
}
```

6. Sequential Datetime

Generates sequential datetime values for time series data. Supports hourly, daily, weekly, monthly, quarterly, and yearly intervals.

```json
{
  "type": "sequential_datetime",
  "start": "ISO datetime string",
  "interval": "hourly|daily|weekly|monthly|quarterly|yearly"
}
```

Parameters: - `start`: ISO format datetime string (e.g., "2024-01-01", "2024-01-01T00:00:00", "2024-01-01T09:30:00") - `interval`: Time interval between sequential values

Supported Intervals: - `hourly`: Increment by 1 hour - `daily`: Increment by 1 day - `weekly`: Increment by 1 week (7 days) - `monthly`: Increment by 1 month (handles variable month lengths) - `quarterly`: Increment by 3 months - `yearly`: Increment by 1 year

Important Notes: - Must use `data_type: "datetime"` with this distribution - Datetime features cannot be used in target expressions - Outlier injection is not applicable to datetime features - Missing values can be applied to datetime features

Examples:

Hourly Time Series:

```json
{
  "name": "timestamp",
  "data_type": "datetime",
  "distribution": {
    "type": "sequential_datetime",
    "start": "2024-01-01T00:00:00",
    "interval": "hourly"
```

```
  }
}
```

Daily Time Series:

```json
{
  "name": "date",
  "data_type": "datetime",
  "distribution": {
    "type": "sequential_datetime",
    "start": "2024-01-01",
    "interval": "daily"
  }
}
```

Monthly Time Series:

```json
{
  "name": "month",
  "data_type": "datetime",
  "distribution": {
    "type": "sequential_datetime",
    "start": "2020-01-01",
    "interval": "monthly"
  }
}
```

Quarterly Business Data:

```json
{
  "name": "quarter_start",
  "data_type": "datetime",
  "distribution": {
    "type": "sequential_datetime",
    "start": "2023-01-01",
    "interval": "quarterly"
  }
}
```

Yearly Data:

```json
{
  "name": "year",
  "data_type": "datetime",
  "distribution": {
    "type": "sequential_datetime",
    "start": "2010-01-01",
    "interval": "yearly"
  }
}
```

Categorical Variables

For `data_type: "categorical"`, a `categories` array must be provided with exactly 10 labels (one per decile).

```json
{
  "name": "risk_level",
  "data_type": "categorical",
  "distribution": {
    "type": "normal",
    "mean": 0.5,
    "std": 0.2
  },
  "categories": [
    "Very Low",
    "Very Low",
    "Low",
    "Low",
    "Medium",
    "Medium",
    "Medium",
    "High",
    "High",
    "Very High"
  ]
}
```

Process: 1. Generate continuous values using specified distribution 2. Rank values and divide into 10 deciles (0-9) 3. Map each decile to corresponding category label 4. Categories array position 0 = 1st decile (lowest 10%), position 9 = 10th decile (highest 10%)

Creating Imbalanced Classes: Repeat labels to create imbalanced distributions:

```
"categories": [
  "Rare Event",       // Decile 0 (10%)
  "Common",           // Decile 1 (10%)
  "Common",           // Decile 2 (10%)
  "Common",           // Decile 3 (10%)
  "Common",           // Decile 4 (10%)
  "Common",           // Decile 5 (10%)
  "Common",           // Decile 6 (10%)
  "Common",           // Decile 7 (10%)
  "Common",           // Decile 8 (10%)
  "Common"            // Decile 9 (10%)
]
// Results in 10% "Rare Event", 90% "Common"
```

Correlations

Define pairwise correlations between features.

```
{
  "correlations": [
    {
      "variables": ["var1", "var2"],
      "correlation": 0.75,
      "method": "cholesky"
    }
  ]
}
```

Fields: - `variables`: Array of exactly 2 feature names - `correlation`: Correlation coefficient (-1.0 to 1.0) - `method`: Always use `"cholesky"` (Cholesky decomposition)

Example:

```
{
  "correlations": [
    {
      "variables": ["height", "weight"],
```

```json
      "correlation": 0.80,
      "method": "cholesky"
    },
    {
      "variables": ["income", "education_years"],
      "correlation": 0.65,
      "method": "cholesky"
    }
  ]
}
```

Important Notes: - Correlations are applied to continuous values before categorical conversion - All correlated variables must exist in features array - Correlation matrix must be positive semi-definite (valid correlation structure)

Missing Data

Specify the proportion of missing values for each feature.

```json
{
  "name": "income",
  "data_type": "float",
  "distribution": {...},
  "missing_rate": 0.15
}
```

Process: 1. Generate complete data 2. Randomly select `missing_rate` × `n_rows` observations 3. Replace with `NaN` (float), `None` (int), or empty string (categorical)

Example Use Cases: - Survey data: 5-20% missing - Administrative data: 1-5% missing - Complete data: 0%

Outliers

Inject outliers into numeric features to simulate real-world anomalies.

```json
{
  "name": "transaction_amount",
  "data_type": "float",
  "distribution": {...},
  "outlier_rate": 0.02,
  "outlier_method": "extreme_high",
  "outlier_multiplier": 3.0
}
```

Outlier Methods:

`extreme_high`

Replace outliers with high extreme values. - Formula: `value = Q3 + multiplier × IQR`

`extreme_low`

Replace outliers with low extreme values. - Formula: `value = Q1 - multiplier × IQR`

`extreme_both`

Replace outliers with both high and low extremes (50/50 split). - High: `Q3 + multiplier × IQR` - Low: `Q1 - multiplier × IQR`

Where: - Q1 = 25th percentile - Q3 = 75th percentile - IQR = Q3 - Q1 (Interquartile Range)

Example:

```json
{
  "name": "response_time",
  "data_type": "float",
  "distribution": {
    "type": "normal",
    "mean": 200,
    "std": 50
  },
  "outlier_rate": 0.05,
  "outlier_method": "extreme_both",
  "outlier_multiplier": 4.0
}
```

Target Variable

Define the target variable using a Python expression based on features.

```
{
  "target": {
    "name": "string",
    "description": "string",
    "data_type": "float|int|categorical",
    "expression": "python expression",
    "noise_percent": float,
    "categories": [...],
    "missing_rate": 0.0,
    "outlier_rate": 0.0,
    "outlier_method": "string",
    "outlier_multiplier": float,
    "seasonality_multipliers": [...]
  }
}
```

Target Fields

Field	Type	Required	Description
`name`	string	Yes	Target variable name
`description`	string	No	Human-readable description
`data_type`	string	Yes	One of: `float`, `int`, `categorical`
`expression`	string	Yes	Python expression using feature names
`noise_percent`	float	No	Percentage noise (0-100), default: 0
`categories`	array	Conditional	Required if `data_type: "categorical"` (10 labels)
`missing_rate`	float	No	Proportion missing (0.0-1.0)
`outlier_rate`	float	No	Proportion outliers (0.0-1.0)
`seasonality_multipliers`	array	No	Primary seasonal multipliers for time series

Field	Type	Required	Description
secondary_seasonality_multipliers		No	Secondary seasonal multipliers for time series

---

Time Series: Feature Generation (Smooth Values)

Important: For time series datasets (identified by the presence of a `datetime` feature), numeric features are generated using a **difference-based approach** to create smooth, realistic time series with gradual changes instead of random jumps.

How it works: 1. A starting value is drawn from the specified distribution 2. Small changes/differences are generated (2-5% of the distribution scale) 3. Changes accumulate using cumulative sum to create smooth progression 4. This only applies to: `uniform`, `normal`, and `weibull` distributions 5. Excludes: `datetime` features, `sequential`, and `random_walk` (already smooth)

Cross-sectional datasets (no datetime feature) continue to use direct random sampling.

Comparison: - Old approach: Each value independently drawn from distribution → large jumps between consecutive values - **New approach**: Values evolve gradually → smooth, realistic time series

Example: Temperature in a `normal(20, 5)` distribution: - Old: Values could jump from 18°C to 27°C to 14°C (unrealistic) - New: Values change gradually: 18°C → 18.2°C → 18.5°C → 18.3°C (realistic)

Important Note on Correlations: There is a fundamental trade-off between correlation strength and temporal smoothness. The rank-based correlation transformation can disrupt smoothness by reshuffling values. To mitigate this, an exponential moving average smoothing filter is applied after correlations, which maintains approximate correlations (within 10-15% of target) while improving smoothness.

Recommendation: For time series where smoothness is critical, minimize correlations or use weaker coefficients (0.3-0.5 instead of 0.7-0.9).

Time Series: Seasonality

For seasonal time series, specify multiplicative seasonality factors that cycle through the data.

Example:

```
{
  "target": {
    "name": "sales",
    "data_type": "float",
    "expression": "1000 + 50*advertising + 30*price_lag1",
    "noise_percent": 5.0,
    "seasonality_multipliers": [0.8, 0.85, 0.9, 1.0, 1.1, 1.15, 1.2, 1.15, 1.1, 1.0, 1.3, 1.4
  }
}
```

Process: 1. Calculate target values from expression 2. Apply primary seasonality: `target_value × seasonality_multipliers[row_index % period]` 3. Apply secondary seasonality (if specified) 4. Add noise (after seasonality) 5. Apply type conversion

Seasonality Pattern: - Array length determines the period (12 = monthly, 4 = quarterly, etc.) - Values cycle through: row 0 uses multiplier[0], row 12 uses multiplier[0] again - Multipliers > 1.0 indicate high season, < 1.0 indicate low season

Example Patterns:

Retail (Holiday Season):

```
"seasonality_multipliers": [0.9, 0.85, 0.9, 0.95, 1.0, 1.0, 1.05, 1.0, 0.95, 1.05, 1.3, 1.5]
// November-December spike
```

Tourism (Summer Peak):

```
"seasonality_multipliers": [0.7, 0.75, 0.85, 0.95, 1.1, 1.3, 1.4, 1.35, 1.1, 0.95, 0.8, 0.7]
// June-August peak
```

Quarterly Business:

```
"seasonality_multipliers": [1.0, 0.95, 1.05, 1.15]
// Q4 spike
```

Time Series: Secondary Seasonality

For time series with **multiple overlapping seasonal patterns**, you can specify both `seasonality_multipliers` and `secondary_seasonality_multipliers`. Both patterns are multiplicative and can have different periodicities.

Example: Retail Sales (Monthly + Weekly Patterns)

```
{
  "target": {
    "name": "sales",
    "data_type": "float",
    "expression": "1000 + 50*advertising + 30*price_lag1",
    "noise_percent": 5.0,
    "seasonality_multipliers": [0.8, 0.85, 0.9, 0.95, 1.0, 1.05, 1.1, 1.15, 1.2, 1.15, 1.05,
    "secondary_seasonality_multipliers": [0.9, 0.95, 1.0, 1.05, 1.1, 1.05, 0.95]
  }
}
```

In this example: - **Primary** (12 values): Monthly pattern with November-December peak (holiday shopping) - **Secondary** (7 values): Weekly pattern with mid-week and weekend peaks

Process: 1. Calculate target values from expression 2. Apply primary seasonality: `value × seasonality_multipliers[i % 12]` 3. Apply secondary seasonality: `value × secondary_seasonality_multipliers[i % 7]` 4. Add noise 5. Apply type conversion

Common Use Cases:

Energy Usage:

```
"seasonality_multipliers": [1.2, 1.15, 1.0, 0.85, 0.8, 0.9, 1.1, 1.15, 0.95, 0.85, 0.95, 1.1
"secondary_seasonality_multipliers": [1.1, 1.05, 1.0, 0.95, 0.9, 0.85, 0.95]
// Primary: Yearly (heating/cooling), Secondary: Weekly (weekday vs weekend)
```

Website Traffic:

```
"seasonality_multipliers": [0.9, 0.95, 1.0, 1.05, 1.1, 1.05, 1.0, 0.95, 1.0, 1.05, 1.1, 1.15]
"secondary_seasonality_multipliers": [0.85, 0.9, 0.95, 1.0, 1.05, 1.15, 1.1]
// Primary: Monthly, Secondary: Day of week
```

Restaurant Sales:

```
"seasonality_multipliers": [0.85, 0.9, 0.95, 1.0, 1.05, 1.1, 1.1, 1.0, 0.95, 1.0, 1.15, 1.25]
"secondary_seasonality_multipliers": [0.7, 0.75, 0.8, 0.85, 0.95, 1.3, 1.4]
// Primary: Monthly, Secondary: Day of week (Fri-Sat peak)
```

Important Notes: - Both seasonality arrays are optional - Can use different periodicities (e.g., 12 and 7, or 4 and 52) - Both patterns are multiplicative (they multiply together) - Applied before noise injection - Order: primary first, then secondary

---

Expression Syntax

Available: - Feature names as variables (including lagged features like `price_lag1`) - Arithmetic operators: `+`, `-`, `*`, `/`, `**` (power), `//` (floor division), `%` (modulo) - NumPy functions: `np.exp()`, `np.log()`, `np.sqrt()`, `np.sin()`, `np.cos()`, `np.abs()`, etc. - Parentheses for grouping

Examples:

Linear Regression

```json
{
  "name": "price",
  "data_type": "float",
  "expression": "50000 + 3000*bedrooms + 2500*bathrooms + 100*sqft",
  "noise_percent": 5.0
}
```

Polynomial Regression

```json
{
  "name": "yield",
  "data_type": "float",
  "expression": "10 + 2*fertilizer - 0.1*fertilizer**2 + 0.5*rainfall",
  "noise_percent": 10.0
}
```

Time Series with Lags

```json
{
  "name": "next_price",
  "data_type": "float",
  "expression": "0.7*price + 0.2*price_lag1 + 0.1*price_lag2",
  "noise_percent": 3.0
}
```

Logistic (Binary Classification)

```json
{
  "name": "approved",
  "data_type": "categorical",
  "expression": "1 / (1 + np.exp(-(-5 + 0.1*credit_score + 2*income_k - 1.5*debt_ratio)))",
  "noise_percent": 3.0,
  "categories": [
    "Denied", "Denied", "Denied", "Denied", "Denied",
    "Approved", "Approved", "Approved", "Approved", "Approved"
  ]
}
```

Seasonal Time Series

```json
{
  "name": "monthly_sales",
  "data_type": "float",
  "expression": "5000 + 100*marketing_spend + 50*sales_lag1",
  "noise_percent": 8.0,
  "seasonality_multipliers": [0.9, 0.9, 0.95, 1.0, 1.05, 1.0, 1.0, 0.95, 0.95, 1.05, 1.2, 1.
}
```

Noise Application

Noise is applied as a percentage of the calculated value's range:

1. Calculate target values from expression
2. Apply seasonality (if specified)
3. Compute range: `max_value - min_value`
4. For each observation, add random noise: `±(noise_percent/100) × range × random()`

Example: - Expression yields values from 50 to 150 (range = 100) - `noise_percent: 10.0` - Each value gets ±10 added randomly (10% of 100)

Complete Examples

Example 1: Simple Linear Regression

```json
{
  "dataset_config": {
    "name": "simple_regression",
    "description": "Teaching simple linear regression",
    "random_seed": 123,
    "n_rows": 500,
    "features": [
      {
        "name": "study_hours",
        "description": "Hours spent studying",
        "data_type": "float",
        "distribution": {
          "type": "uniform",
          "min": 0,
          "max": 10
        },
        "missing_rate": 0.0
      }
    ],
    "target": {
      "name": "test_score",
      "description": "Test score out of 100",
      "data_type": "float",
      "expression": "50 + 5*study_hours",
      "noise_percent": 10.0
    }
  }
}
```

Example 2: Time Series with Lags and Seasonality

```
{
  "dataset_config": {
    "name": "retail_sales_forecast",
    "description": "Monthly retail sales with seasonality",
    "random_seed": 42,
    "n_rows": 60,
    "features": [
      {
        "name": "date",
        "description": "Monthly date",
        "data_type": "datetime",
        "distribution": {
          "type": "sequential_datetime",
          "start": "2020-01-01",
          "interval": "monthly"
        }
      },
      {
        "name": "advertising",
        "description": "Advertising spend in thousands",
        "data_type": "float",
        "distribution": {
          "type": "normal",
          "mean": 50,
          "std": 10,
          "min_clip": 20
        },
        "lags": [1, 2]
      },
      {
        "name": "base_demand",
        "description": "Baseline customer demand",
        "data_type": "float",
        "distribution": {
          "type": "random_walk",
          "start": 1000,
          "step_size": 50,
          "drift": 5
        },
        "lags": [1]
```

```
      }
    ],
    "target": {
      "name": "sales",
      "description": "Monthly sales",
      "data_type": "float",
      "expression": "base_demand + 8*advertising + 3*advertising_lag1 + 0.2*base_demand_lag1"
      "noise_percent": 5.0,
      "seasonality_multipliers": [0.85, 0.9, 0.95, 1.0, 1.05, 1.0, 1.0, 0.95, 0.95, 1.05, 1.2
    }
  }
}
```

Example 3: Binary Classification

```
{
  "dataset_config": {
    "name": "loan_approval",
    "description": "Binary classification for loan approval",
    "random_seed": 456,
    "n_rows": 1000,
    "correlations": [
      {
        "variables": ["income", "credit_score"],
        "correlation": 0.60,
        "method": "cholesky"
      }
    ],
    "features": [
      {
        "name": "income",
        "description": "Annual income in thousands",
        "data_type": "float",
        "distribution": {
          "type": "normal",
          "mean": 60,
          "std": 20,
          "min_clip": 20,
          "max_clip": 150
        },
```

```
      "missing_rate": 0.05
    },
    {
      "name": "credit_score",
      "description": "Credit score 300-850",
      "data_type": "int",
      "distribution": {
        "type": "normal",
        "mean": 680,
        "std": 80,
        "min_clip": 300,
        "max_clip": 850
      },
      "missing_rate": 0.02
    },
    {
      "name": "debt_ratio",
      "description": "Debt to income ratio",
      "data_type": "float",
      "distribution": {
        "type": "uniform",
        "min": 0.1,
        "max": 0.6
      },
      "missing_rate": 0.03,
      "outlier_rate": 0.02,
      "outlier_method": "extreme_high",
      "outlier_multiplier": 2.5
    }
  ],
  "target": {
    "name": "approved",
    "description": "Loan approval decision",
    "data_type": "categorical",
    "expression": "1 / (1 + np.exp(-(-8 + 0.05*credit_score + 0.08*income - 10*debt_ratio))
    "noise_percent": 5.0,
    "categories": [
      "Rejected", "Rejected", "Rejected", "Rejected",
      "Rejected", "Rejected",
      "Approved", "Approved", "Approved", "Approved"
    ]
  }
```

```
  }
}
```

Example 4: Hourly Time Series Data

```
{
  "dataset_config": {
    "name": "server_metrics",
    "description": "Hourly server performance metrics",
    "random_seed": 999,
    "n_rows": 168,
    "features": [
      {
        "name": "timestamp",
        "description": "Hourly timestamp",
        "data_type": "datetime",
        "distribution": {
          "type": "sequential_datetime",
          "start": "2024-01-01T00:00:00",
          "interval": "hourly"
        }
      },
      {
        "name": "cpu_usage",
        "description": "CPU usage percentage",
        "data_type": "float",
        "distribution": {
          "type": "normal",
          "mean": 45,
          "std": 15,
          "min_clip": 0,
          "max_clip": 100
        },
        "lags": [1, 24]
      },
      {
        "name": "requests_per_hour",
        "description": "HTTP requests per hour",
        "data_type": "int",
        "distribution": {
```

```json
          "type": "random_walk",
          "start": 5000,
          "step_size": 500,
          "drift": 10
        },
        "lags": [1]
      }
    ],
    "target": {
      "name": "response_time_ms",
      "description": "Average response time in milliseconds",
      "data_type": "float",
      "expression": "100 + 2*cpu_usage + 0.01*requests_per_hour + 0.5*cpu_usage_lag1",
      "noise_percent": 10.0
    }
  }
}
```

Validation Rules

The generator will validate:

1. **Feature names** are valid Python identifiers and unique
2. **Data types** match distribution compatibility
3. **Datetime** features must use `sequential_datetime` distribution type
4. **Sequential datetime** distributions must use `datetime` data type
5. **Correlation variables** reference existing features
6. **Expression** references only defined feature names (including lagged features, excluding datetime/categorical)
7. **Categorical** features have exactly 10 category labels
8. **Rates** (missing, outlier, noise) are between 0 and 1 (or 0-100 for noise_percent)
9. **Distribution parameters** are valid (e.g., std > 0, min < max, valid ISO datetime strings)
10. **Correlation matrix** is positive semi-definite
11. **Lags** are positive integers
12. **Seasonality multipliers** (both primary and secondary) are numeric values

---

Notes

- **Order matters**: Features are generated in order. Random walks and sequential distributions depend on order.
- **Correlations**: Applied before categorical conversion and outlier injection.
- **Missing data**: Applied after all other transformations.
- **Categorical deciles**: Always create 10 equal-sized bins (10% each).
- **NumPy**: Available in expressions as `np.*`
- **Lagged features**: First N rows contain NaN where N is the lag period
- **Seasonality**: Applied to target after expression evaluation but before noise (primary then secondary)
- **Time series**:
  - Lagged features are automatically created and available in expressions
  - Numeric features use difference-based generation for smooth, gradual changes
  - Cross-sectional datasets use direct random sampling
- **Datetime features**:
  - Stored as ISO format strings in the CSV output
  - Cannot be used in mathematical expressions
  - Outlier injection is not applicable
  - Missing values can be applied (represented as empty cells in CSV)