Dataset Generator - User Guide

Introduction

The Dataset Generator is a Streamlit application that helps you create synthetic datasets for data science instruction and practice. It provides two ways to create datasets:

1. **Chat Assistant**: Answer questions in a conversational interface to automatically generate a dataset configuration
2. **JSON Editor**: Manually write or edit JSON configurations for complete control

Getting Started

Prerequisites

- Python 3.8 or higher
- Streamlit installed
- Either Ollama running locally OR Claude API key configured

Running the Application

```
streamlit run app.py
```

The application will open in your web browser at `http://localhost:8501`.

User Interface Overview

Sidebar

The sidebar contains:

- **LLM Provider Selection**: Choose between Ollama (local) or Claude (API)
  - If Claude is selected, you can optionally override the API key from the `.env` file
- **Dataset Management**: Load, save, and delete dataset configurations
- **Generate CSV Button**: Create the actual CSV file from your configuration
- **Documentation Downloads**: Access user and API documentation

Main Tabs

1. **Chat Assistant**: Interactive Q&A to generate datasets
2. **JSON Editor**: Manual editing of dataset configurations
3. **Dataset Description**: Auto-generated markdown description of your dataset

Using the Chat Assistant

The Chat Assistant walks you through creating a dataset by asking you questions:

Example: Creating a Customer Churn Dataset

Question 1: What is the dataset you want to generate?

Answer: "I want to create a customer churn prediction dataset for a telecommunications company"

Question 2: Is the dataset time series or cross-sectional?

Answer: "Cross-sectional"

Question 3: How many rows do you want to generate?

Answer: "5000"

Question 4: Is the target variable categorical, int, or float?

Answer: "Categorical - I want to predict whether a customer will churn (Yes/No)"

Question 5: About how many categorical features do you want?

Answer: "3 categorical features - contract type, payment method, and internet service"

Question 6: About how many numeric features do you want?

Answer: "5 numeric features - tenure, monthly charges, total charges, customer service calls, and data usage"

Question 7: What percentage of the feature values are missing?

Answer: "About 5% missing values"

Question 8: What percentage of the feature values are outliers?

Answer: "About 2% outliers"

Question 9: Should there be appropriate correlations between features?

Answer: "Yes - tenure should be positively correlated with total charges, and customer service calls should be positively correlated with churn"

Question 10: If the data is time series, what is the periodicity?

Answer: "None - this is cross-sectional data"

Question 11: How much noise should be added as a percentage?

Answer: "5% noise"

Question 12: Any other general directions?

Answer: "Make the churn rate about 25% (more 'No' than 'Yes' in the categories)"

After answering all questions, the LLM will generate a JSON configuration automatically. You can then:

- View and edit it in the **JSON Editor** tab
- Generate a CSV file using the **Generate CSV** button
- View a description in the **Dataset Description** tab

Chat Assistant Controls

- **Start Over**: Clear all answers and restart the conversation
- **Regenerate**: Re-run the LLM to generate a new configuration from the same answers

Using the JSON Editor

Editing Configurations

The JSON Editor tab allows you to:

1. Directly edit the JSON configuration
2. Validate your configuration in real-time
3. See a summary of your dataset settings

Validation

The editor will show:

- Green success message if configuration is valid
- Red error messages for any validation issues
- Dataset summary with key information

Example Configuration

Here's a simple example for a house price prediction dataset:

```json
{
  "dataset_config": {
    "name": "house_prices",
    "description": "House price prediction dataset",
    "random_seed": 42,
    "n_rows": 1000,
    "correlations": [
      {
        "variables": ["square_feet", "bedrooms"],
        "correlation": 0.7,
        "method": "cholesky"
      }
    ],
    "features": [
      {
        "name": "square_feet",
        "description": "Square footage of the house",
        "data_type": "int",
        "distribution": {
```

```
        "type": "normal",
        "mean": 2000,
        "std": 500,
        "min_clip": 500
      },
      "missing_rate": 0.0,
      "outlier_rate": 0.02
    },
    {
      "name": "bedrooms",
      "description": "Number of bedrooms",
      "data_type": "int",
      "distribution": {
        "type": "normal",
        "mean": 3,
        "std": 1,
        "min_clip": 1,
        "max_clip": 8
      },
      "missing_rate": 0.0,
      "outlier_rate": 0.0
    },
    {
      "name": "age_years",
      "description": "Age of the house in years",
      "data_type": "int",
      "distribution": {
        "type": "uniform",
        "min": 0,
        "max": 100
      },
      "missing_rate": 0.05,
      "outlier_rate": 0.0
    },
    {
      "name": "location_quality",
      "description": "Quality of the location",
      "data_type": "categorical",
      "distribution": {
        "type": "normal",
        "mean": 5,
        "std": 2
```

```
        },
        "categories": [
          "Poor", "Poor", "Fair", "Fair", "Fair",
          "Good", "Good", "Good", "Excellent", "Excellent"
        ],
        "missing_rate": 0.0,
        "outlier_rate": 0.0
      }
    ],
    "target": {
      "name": "price",
      "description": "House sale price",
      "data_type": "float",
      "expression": "square_feet * 150 + bedrooms * 20000 - age_years * 1000 + 50000",
      "noise_percent": 10.0
    }
  }
}
```

LLM Provider Settings

Ollama (Default)

- Runs locally on your machine
- Requires Ollama to be installed and running
- Default endpoint: `http://localhost:11434`
- Default model: Set in `.env` file (`OLLAMA_MODEL`)
- No API key required

Claude

- Uses Anthropic's Claude API
- Requires API key (set in `.env` or override in UI)
- Default model: Set in `.env` file (`ANTHROPIC_MODEL`)
- API key can be overridden in the sidebar for each session

Overriding Claude API Key

1. Select "Claude" from the LLM Provider dropdown

2. Enter your API key in the "Claude API Key (optional override)" field
3. Leave blank to use the key from `.env` file

Generating the CSV File

Once you have a valid configuration:

1. Click the **Generate CSV** button in the sidebar
2. The CSV file will be created in the `./datasets/` directory
3. A download button will appear to save the CSV to your computer

Generated Files

- **CSV file**: `./datasets/{dataset_name}.csv`
- **JSON config**: `./datasets/{dataset_name}.json` (when saved)

Key Concepts

Percentage Input Formats

Important: Different fields use different percentage formats:

- **missing_rate**: Use 0.1 for 10% (range: 0.0 to 1.0)
- **outlier_rate**: Use 0.1 for 10% (range: 0.0 to 1.0)
- **noise_percent**: Use 10 for 10% (range: 0 to 100)

Example:

```
{
  "missing_rate": 0.05,    // 5% missing values
  "outlier_rate": 0.02,    // 2% outliers
  "noise_percent": 10.0    // 10% noise
}
```

Lagged Features

Features can have lagged versions automatically generated for time series data:

```json
{
  "name": "price",
  "lags": [1, 2, 3]
}
```

This creates `price_lag1`, `price_lag2`, `price_lag3` which can be used in target expressions.

Note: Lagged features are available for use in expressions but are **not included in the final CSV output** - only original features are saved.

Categorical Features

Categorical features require: - A distribution (like `normal` or `uniform`) to generate underlying numeric values - Exactly 10 category labels in the `categories` array - The numeric values are binned into deciles and mapped to the labels

To create imbalanced categories, repeat labels:

```json
{
  "categories": [
    "No", "No", "No", "No", "No",
    "No", "No", "Yes", "Yes", "Yes"
  ]
}
```

This creates approximately 70% "No" and 30% "Yes".

Target Expressions

Target expressions can only reference **numeric features** (int or float), not categorical or datetime features.

Valid:

```
"expression": "feature1 * 2 + feature2 - 100"
```

Invalid (references categorical or datetime feature):

```
"expression": "categorical_feature * 2"  // ERROR!
"expression": "timestamp * 100"          // ERROR!
```

For time series with lagged features:

```
"expression": "price_lag1 * 0.9 + price_lag2 * 0.1"
```

Datetime Features

Datetime features are used for time series data and support various intervals:

Supported Intervals: - `hourly` - Increment by 1 hour - `daily` - Increment by 1 day - `weekly` - Increment by 1 week - `monthly` - Increment by 1 month - `quarterly` - Increment by 3 months - `yearly` - Increment by 1 year

Example:

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

Important: - Datetime features must use `data_type: "datetime"` - The distribution type must be `sequential_datetime` - Start date must be in ISO format (e.g., "2024-01-01" or "2024-01-01T09:30:00") - Datetime features cannot be used in mathematical expressions - Outliers are not applicable to datetime features

Time Series: Smooth Feature Values

New Feature: For time series datasets (those with a datetime feature), numeric features are automatically generated with smooth, gradual changes instead of random jumps.

How it works: - Time series features use a difference-based approach - Values change gradually from one time point to the next - Creates more realistic time series data - Cross-sectional datasets (no datetime feature) use the traditional random sampling

Example: - **Old approach**: Temperature values might jump from 18°C to 27°C to 14°C (unrealistic) - **New approach**: Temperature changes gradually: 18°C → 18.2°C → 18.5°C → 18.3°C (realistic)

This automatic smoothing applies to `normal`, `uniform`, and `weibull` distributions. The `random_walk` and `sequential` distributions are already smooth.

Important Note about Correlations and Smoothness:

If you add correlations between features in a time series dataset, there's a trade-off: strong correlations can reduce smoothness because they require reshuffling values. The system applies smoothing after correlations to help, but the result won't be as smooth as without correlations.

Best Practice: For very smooth time series, either avoid correlations or use weaker values (0.3-0.5 instead of 0.7-0.9).

Seasonality in Time Series

For time series targets, you can add seasonal patterns using `seasonality_multipliers`:

```
{
  "target": {
    "name": "sales",
    "expression": "base_demand + advertising * 50",
    "seasonality_multipliers": [0.9, 0.9, 0.95, 1.0, 1.05, 1.0, 1.0, 0.95, 0.95, 1.05, 1.2, 
  }
}
```

How it works: - The array length determines the period (12 for monthly, 4 for quarterly, etc.) - Values cycle through the multipliers - Values > 1.0 indicate high season, < 1.0 indicate low season - Applied after expression calculation but before noise

Secondary Seasonality

New Feature: For time series with multiple seasonal patterns (e.g., both monthly and weekly), you can now add `secondary_seasonality_multipliers`:

```
{
  "target": {
    "name": "sales",
    "expression": "base_demand + advertising * 50",
    "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 holiday season peak - **Secondary** (7 values): Weekly pattern with weekend peaks

Common use cases: - Retail sales: Monthly (holiday shopping) + Weekly (weekend patterns) - Energy usage: Yearly (heating/cooling) + Weekly (weekday vs weekend) - Website traffic: Monthly trends + Day-of-week patterns - Restaurant sales: Monthly trends + Day-of-week patterns (weekend peaks)

How it works: 1. Calculate target from expression 2. Apply primary seasonality 3. Apply secondary seasonality (multiplies with primary) 4. Add noise

Tips and Best Practices

Starting with Chat Assistant

1. Use the Chat Assistant first to get a basic configuration
2. Review and refine the configuration in the JSON Editor
3. Generate the CSV and examine the output
4. Iterate as needed

Creating Realistic Datasets

1. **Use appropriate distributions**:

   - `normal` for natural measurements (height, weight, etc.)
   - `uniform` for evenly distributed values (random IDs, ratings)
   - `random_walk` for time series with drift
   - `sequential` for numeric time indexes or IDs
   - `sequential_datetime` for datetime time series (hourly, daily, weekly, monthly, quarterly, yearly)

2. **Add realistic correlations**: Related variables should be correlated

3. **Include some imperfection**: Real data has missing values and outliers

4. **Use meaningful noise**: Add 5-15% noise to targets for realistic predictions

Validation Checklist

Before generating CSV, ensure:

- All feature names are unique and valid Python identifiers
- Categorical features have exactly 10 category labels
- Datetime features use `data_type: "datetime"` and `distribution.type: "sequential_datetime"`
- Datetime start values are in valid ISO format
- Target expression only uses numeric features (not categorical or datetime)
- Correlations reference existing features
- Rates are in correct ranges (0-1 for rates, 0-100 for noise_percent)
- If using lags, they're defined in the feature before being used in expressions

Troubleshooting

Common Errors

"Invalid JSON syntax" - Check for missing commas, brackets, or quotes - Use a JSON validator to check syntax

"Feature X not found in expression" - Ensure all features used in expressions are defined - For lagged features, add `"lags": [1, 2, 3]` to the feature definition

"Categories array must have exactly 10 labels" - Categorical features and targets must have exactly 10 category labels

"Expression references undefined features" - Target expressions can only use numeric features - Categorical and datetime features cannot be used in expressions

"datetime data_type requires distribution type 'sequential_datetime'" - If using `data_type: "datetime"`, you must use `sequential_datetime` distribution - Specify interval (hourly, daily, weekly, monthly, quarterly, yearly)

"Invalid datetime format" - Start date must be in ISO format - Valid examples: "2024-01-01", "2024-01-01T00:00:00", "2024-01-01T09:30:00"

"Configuration has errors" - Check the validation section for specific error messages - Fix errors one at a time and re-validate

LLM Issues

Ollama not responding - Ensure Ollama is running: `ollama serve` - Check endpoint URL in `.env` file - Verify model is downloaded: `ollama pull {model_name}`

Claude API errors - Verify API key is correct - Check API key has sufficient credits - Ensure model name in `.env` is valid

Appendix: Complete Example Workflow

Step 1: Open the Application

```
streamlit run app.py
```

Step 2: Choose LLM Provider

Select "Ollama" or "Claude" in the sidebar.

Step 3: Create Configuration via Chat

Go to the "Chat Assistant" tab and answer all questions about your desired dataset.

Step 4: Review in JSON Editor

Switch to "JSON Editor" tab and review the generated configuration. Make any necessary adjustments.

Step 5: Validate

Ensure the validation section shows " Configuration is valid!"

Step 6: Save Configuration

Click " Save" in the sidebar to save the JSON configuration.

Step 7: Generate CSV

Click “ Generate CSV” to create the dataset.

Step 8: Download

Click “ Download CSV” to save the file to your computer.

Step 9: View Description

Go to “Dataset Description” tab to see an auto-generated description of your dataset.

For technical details about the JSON schema and advanced features, see the API Documentation.