

Efficient AI with Rust Lab
Rapid Time Series Datasets Library
RWTH Aachen University
Group 1

Marius Kaufmann¹ Amir Ali Aali² Kilian Fin Braun¹

¹Masters of Computer Science

²Masters of Data Science

19th Jul, 2025



Overview

Goal

- ▶ Preprocessing of time series datasets

Overview

Goal

- ▶ Preprocessing of time series datasets
- ▶ Python package implemented in Rust

Overview

Goal

- ▶ Preprocessing of time series datasets
- ▶ Python package implemented in Rust
- ▶ Passing data by reference

Overview

Goal

- ▶ Preprocessing of time series datasets
- ▶ Python package implemented in Rust
- ▶ Passing data by reference
 - ▶ Using `numpy` crate

Overview

Goal

- ▶ Preprocessing of time series datasets
- ▶ Python package implemented in Rust
- ▶ Passing data by reference
 - ▶ Using `numpy` crate

Scope

- ▶ Two types of datasets

Overview

Goal

- ▶ Preprocessing of time series datasets
- ▶ Python package implemented in Rust
- ▶ Passing data by reference
 - ▶ Using `numpy` crate

Scope

- ▶ Two types of datasets
 - ▶ `ForecastingDataSet`

Overview

Goal

- ▶ Preprocessing of time series datasets
- ▶ Python package implemented in Rust
- ▶ Passing data by reference
 - ▶ Using `numpy` crate

Scope

- ▶ Two types of datasets
 - ▶ `ForecastingDataSet`
 - ▶ `ClassificationDataSet`

Overview

Goal

- ▶ Preprocessing of time series datasets
- ▶ Python package implemented in Rust
- ▶ Passing data by reference
 - ▶ Using `numpy` crate

Scope

- ▶ Two types of datasets
 - ▶ `ForecastingDataSet`
 - ▶ `ClassificationDataSet`
- ▶ Functionality
 - ▶ `impute()`

Overview

Goal

- ▶ Preprocessing of time series datasets
- ▶ Python package implemented in Rust
- ▶ Passing data by reference
 - ▶ Using `numpy` crate

Scope

- ▶ Two types of datasets
 - ▶ `ForecastingDataSet`
 - ▶ `ClassificationDataSet`
- ▶ Functionality
 - ▶ `impute()`
 - ▶ `downsample()`

Overview

Goal

- ▶ Preprocessing of time series datasets
- ▶ Python package implemented in Rust
- ▶ Passing data by reference
 - ▶ Using `numpy` crate

Scope

- ▶ Two types of datasets
 - ▶ `ForecastingDataSet`
 - ▶ `ClassificationDataSet`
- ▶ Functionality
 - ▶ `impute()`
 - ▶ `downsample()`
 - ▶ `split()`

Overview

Goal

- ▶ Preprocessing of time series datasets
- ▶ Python package implemented in Rust
- ▶ Passing data by reference
 - ▶ Using `numpy` crate

Scope

- ▶ Two types of datasets
 - ▶ `ForecastingDataSet`
 - ▶ `ClassificationDataSet`
- ▶ Functionality
 - ▶ `impute()`
 - ▶ `downsample()`
 - ▶ `split()`
 - ▶ `normalize()` / `standardize()`

Data Input Format

Input 3D numpy array:

Data Input Format

Input 3D numpy array:

- ▶ **First dimension:** Instances

Data Input Format

Input 3D numpy array:

- ▶ **First dimension:** Instances
- ▶ **Second dimension:** Timesteps

Data Input Format

Input 3D numpy array:

- ▶ **First dimension:** Instances
- ▶ **Second dimension:** Timesteps
- ▶ **Third dimension:** Features

Data Input Format

Input 3D numpy array:

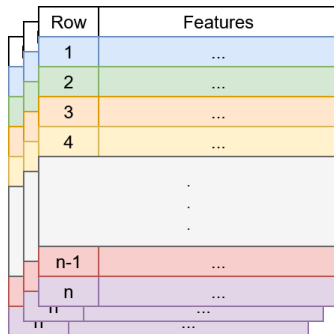
- ▶ **First dimension:** Instances
- ▶ **Second dimension:** Timesteps
- ▶ **Third dimension:** Features

Row	Features
1	...
2	...
3	...
4	...
.	
.	
.	
n-1	...
n	...

Data Input Format

Input 3D numpy array:

- ▶ **First dimension:** Instances
- ▶ **Second dimension:** Timesteps
- ▶ **Third dimension:** Features

The diagram illustrates a 3D numpy array structure. It shows a stack of multiple 2D arrays, each representing an instance. The top-most 2D array is shown in detail with two columns: 'Row' and 'Features'. The 'Row' column contains indices 1, 2, 3, 4, followed by a vertical ellipsis, then n-1, n, and another vertical ellipsis. The 'Features' column contains three dots (...) for each row. The rows are color-coded: blue for row 1, green for row 2, orange for rows 3 and 4, light grey for the ellipsis, red for row n-1, and purple for row n. Below the top array, several other identical-looking 2D arrays are shown, slightly offset to the left and bottom, indicating a stack of instances. The entire stack is represented by a vertical ellipsis on the left side.

Row	Features
1	...
2	...
3	...
4	...
⋮	
n-1	...
n	...
⋮	

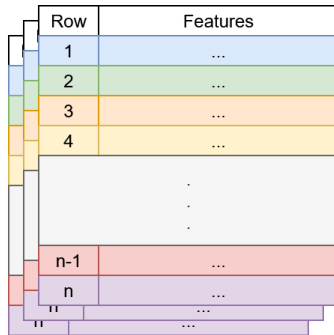
Data Input Format

Input 3D numpy array:

- ▶ **First dimension:** Instances
- ▶ **Second dimension:** Timesteps
- ▶ **Third dimension:** Features

In practice

- ▶ Forecasting datasets:



The diagram illustrates a 3D numpy array structure for forecasting datasets. It shows a stack of 3D arrays, each representing a different instance. The top-most array is a 2D table with two columns: 'Row' and 'Features'. The 'Row' column contains the values 1, 2, 3, 4, followed by a large grey block representing rows 5 to n-2, and then n-1 and n. The 'Features' column contains ellipses (...) for each row. The rows are color-coded: blue for row 1, green for row 2, orange for row 3, yellow for row 4, and red for row n-1. The bottom-most array in the stack is purple and also shows rows n-1 and n with ellipses in the 'Features' column. The 'Row' column for the bottom array shows 'n-1' and 'n'.

Row	Features
1	...
2	...
3	...
4	...
...	...
n-1	...
n	...

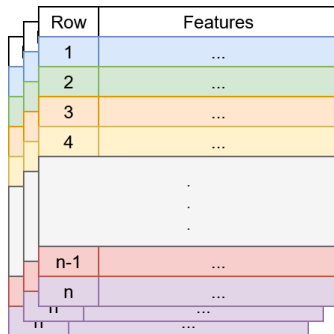
Data Input Format

Input 3D numpy array:

- ▶ **First dimension:** Instances
- ▶ **Second dimension:** Timesteps
- ▶ **Third dimension:** Features

In practice

- ▶ Forecasting datasets:
 - ▶ One instance



The diagram illustrates a 3D numpy array structure for forecasting datasets. It shows a stack of 2D arrays, each representing an instance. The top instance is a 2D array with two columns: 'Row' and 'Features'. The 'Row' column contains the values 1, 2, 3, 4, followed by a vertical ellipsis, then n-1, n, and another vertical ellipsis. The 'Features' column contains three dots (representing multiple features) for each row. The rows are color-coded: row 1 is light blue, row 2 is light green, row 3 is light orange, row 4 is light yellow, the middle rows are light grey, row n-1 is light red, and row n is light purple. The bottom instance is also shown in light purple, with its 'Row' column containing a vertical ellipsis and its 'Features' column containing three dots.

Row	Features
1	...
2	...
3	...
4	...
...	
n-1	...
n	...
...	

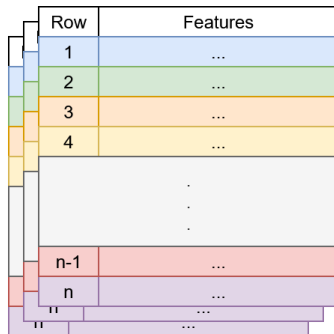
Data Input Format

Input 3D numpy array:

- ▶ **First dimension:** Instances
- ▶ **Second dimension:** Timesteps
- ▶ **Third dimension:** Features

In practice

- ▶ Forecasting datasets:
 - ▶ One instance
- ▶ Classification datasets:



The diagram illustrates a 3D numpy array structure. It shows a stack of multiple 2D tables. The top-most table is clearly visible, with a header row containing 'Row' and 'Features'. Below the header, there are four rows with indices 1, 2, 3, and 4, each followed by an ellipsis (...). These rows are color-coded: row 1 is light blue, row 2 is light green, row 3 is light orange, and row 4 is light yellow. Below these, there is a large grey rectangular block representing a continuation of the data. At the bottom of the stack, two more rows are visible, labeled 'n-1' and 'n', with ellipses in the 'Features' column. These rows are color-coded in shades of red and purple. The entire stack is shown with a 3D perspective, with the top table slightly offset to reveal the ones underneath.

Row	Features
1	...
2	...
3	...
4	...
...	
n-1	...
n	...

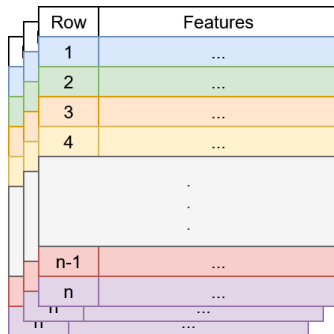
Data Input Format

Input 3D numpy array:

- ▶ **First dimension:** Instances
- ▶ **Second dimension:** Timesteps
- ▶ **Third dimension:** Features

In practice

- ▶ Forecasting datasets:
 - ▶ One instance
- ▶ Classification datasets:
 - ▶ Multiple instances

A diagram illustrating a 3D numpy array structure. It shows a stack of four 2D tables. The first table has a 'Row' column and a 'Features' column. The rows are numbered 1, 2, 3, and 4, each with a different background color (light blue, light green, light orange, and light yellow respectively). The 'Features' column contains three dots '...' for each row. Below these four tables is a larger, light grey table with three rows, each containing a single dot '.' in the 'Features' column. Below this is a table with two rows, labeled 'n-1' and 'n' in the 'Row' column, with light red and light purple backgrounds respectively. The 'Features' column contains three dots '...' for each row. Below these are two more tables, each with a single row labeled 'n' in the 'Row' column, with light purple backgrounds. The 'Features' column contains three dots '...' for each row. The tables are stacked such that the 'n-1' and 'n' rows of the lower tables are offset to the right, showing they represent different instances or time steps.

Row	Features
1	...
2	...
3	...
4	...
	.
	.
	.
n-1	...
n	...
n	...
n	...

Performance considerations

Copying

- ▶ Copying data is expensive

Performance considerations

Copying

- ▶ Copying data is expensive
- ▶ Avoid unnecessary copies

Performance considerations

Copying

- ▶ Copying data is expensive
- ▶ Avoid unnecessary copies
- ▶ Copy only when absolutely necessary

Performance considerations

Copying

- ▶ Copying data is expensive
- ▶ Avoid unnecessary copies
- ▶ Copy only when absolutely necessary
 - ▶ Only once

Performance considerations

Copying

- ▶ Copying data is expensive
- ▶ Avoid unnecessary copies
- ▶ Copy only when absolutely necessary
 - ▶ Only once

When to copy?

Performance considerations

Copying

- ▶ Copying data is expensive
- ▶ Avoid unnecessary copies
- ▶ Copy only when absolutely necessary
 - ▶ Only once

When to copy?

- ▶ For `ForecastingDataSet`:

Performance considerations

Copying

- ▶ Copying data is expensive
- ▶ Avoid unnecessary copies
- ▶ Copy only when absolutely necessary
 - ▶ Only once

When to copy?

- ▶ For `ForecastingDataSet`:
 - ▶ Windowed format in final step

Performance considerations

Copying

- ▶ Copying data is expensive
- ▶ Avoid unnecessary copies
- ▶ Copy only when absolutely necessary
 - ▶ Only once

When to copy?

- ▶ For `ForecastingDataSet`:
 - ▶ Windowed format in final step
 - ▶ Copying unavoidable

Performance considerations

Copying

- ▶ Copying data is expensive
- ▶ Avoid unnecessary copies
- ▶ Copy only when absolutely necessary
 - ▶ Only once

When to copy?

- ▶ For `ForecastingDataSet`:
 - ▶ Windowed format in final step
 - ▶ Copying unavoidable
- ▶ For `ClassificationDataSet`:

Performance considerations

Copying

- ▶ Copying data is expensive
- ▶ Avoid unnecessary copies
- ▶ Copy only when absolutely necessary
 - ▶ Only once

When to copy?

- ▶ For `ForecastingDataSet`:
 - ▶ Windowed format in final step
 - ▶ Copying unavoidable
- ▶ For `ClassificationDataSet`:
 - ▶ Random splitting strategy offered

Performance considerations

Copying

- ▶ Copying data is expensive
- ▶ Avoid unnecessary copies
- ▶ Copy only when absolutely necessary
 - ▶ Only once

When to copy?

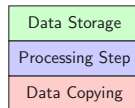
- ▶ For `ForecastingDataSet`:
 - ▶ Windowed format in final step
 - ▶ Copying unavoidable
- ▶ For `ClassificationDataSet`:
 - ▶ Random splitting strategy offered
 - ▶ Copying unavoidable

Data-flow

Forecasting Dataset Data-Flow Classification Dataset Data-Flow

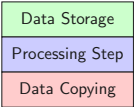
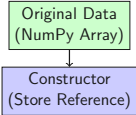
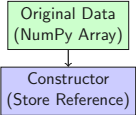
Original Data
(NumPy Array)

Original Data
(NumPy Array)



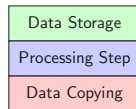
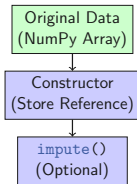
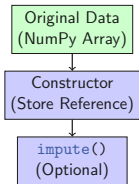
Data-flow

Forecasting Dataset Data-Flow Classification Dataset Data-Flow



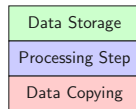
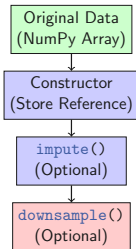
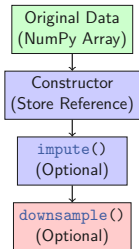
Data-flow

Forecasting Dataset Data-Flow Classification Dataset Data-Flow



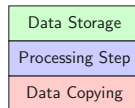
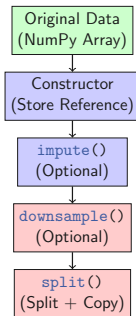
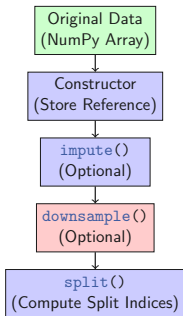
Data-flow

Forecasting Dataset Data-Flow Classification Dataset Data-Flow



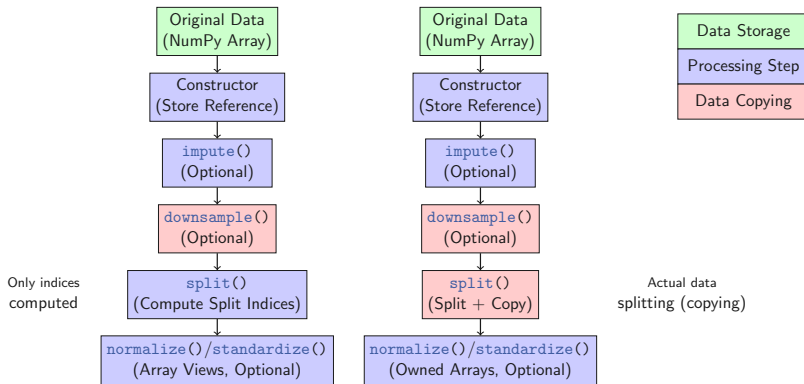
Data-flow

Forecasting Dataset Data-Flow Classification Dataset Data-Flow



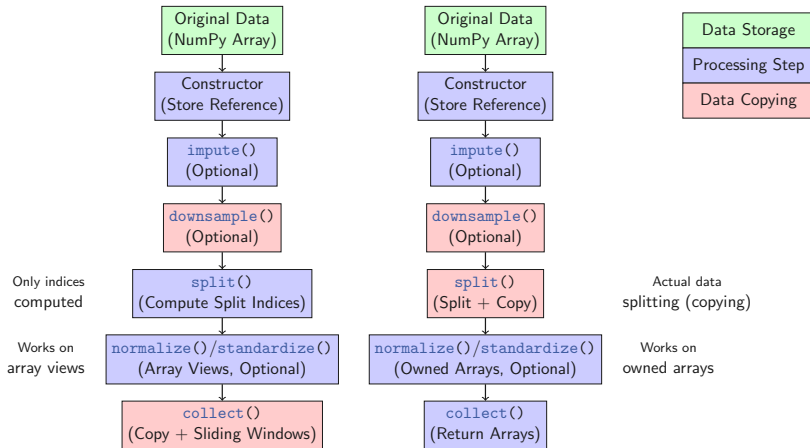
Data-flow

Forecasting Dataset Data-Flow Classification Dataset Data-Flow



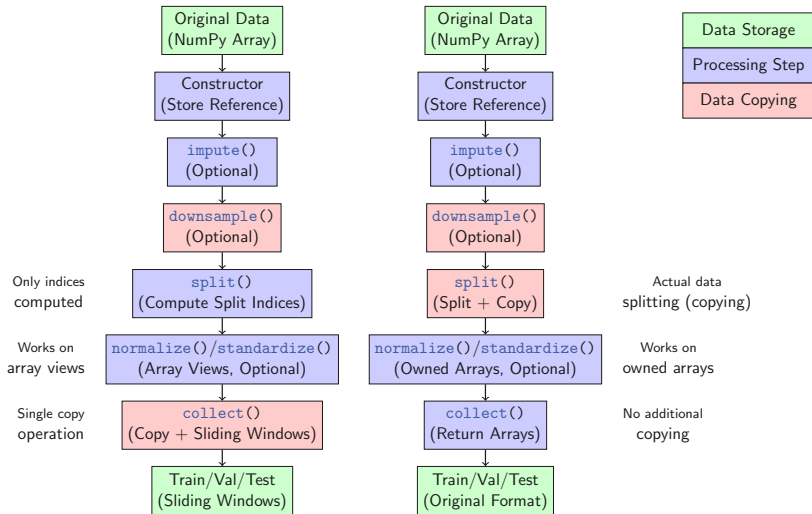
Data-flow

Forecasting Dataset Data-Flow Classification Dataset Data-Flow



Data-flow

Forecasting Dataset Data-Flow Classification Dataset Data-Flow



Pipeline Design

ForecastingDataSet

```
# Create instance
fore = ForecastingDataSet(
    data, 0.7, 0.2, 0.1
)

# call the pipeline methods
fore.impute(
    ImputeStrategy.Median
)
fore.downsample(2)
fore.split()

fore.normalize()
fore.standardize()

# collect the results
fore_res = fore.collect(3, 1, 1)
```

Pipeline Design

ForecastingDataSet

```
# Create instance
fore = ForecastingDataSet(
    data, 0.7, 0.2, 0.1
)

# call the pipeline methods
fore.impute(
    ImputeStrategy.Median
)
fore.downsample(2)
fore.split()

fore.normalize()
fore.standardize()

# collect the results
fore_res = fore.collect(3, 1, 1)
```

ClassificationDataSet

```
# create instance
clas = ClassificationDataSet(
    data, labels, 0.7, 0.2, 0.1
)

# call the pipeline methods
clas.impute(
    ImputeStrategy.Median
)
clas.downsample(2)
clas.split(
    SplittingStrategy.Random
)
clas.normalize()
clas.standardize()

# collect the results
clas_res = clas.collect()
```

Downsampling I

Goal: Reduce the number of data points in a time series dataset.

Downsampling I

Goal: Reduce the number of data points in a time series dataset.

Benefits:

- ▶ Reduces memory usage

Downsampling I

Goal: Reduce the number of data points in a time series dataset.

Benefits:

- ▶ Reduces memory usage
- ▶ Speeds up processing time

Downsampling I

Goal: Reduce the number of data points in a time series dataset.

Benefits:

- ▶ Reduces memory usage
- ▶ Speeds up processing time

Necessary parameter when downsampling:

- ▶ Downsampling factor: How many data points to skip

Downsampling I

Goal: Reduce the number of data points in a time series dataset.

Benefits:

- ▶ Reduces memory usage
- ▶ Speeds up processing time

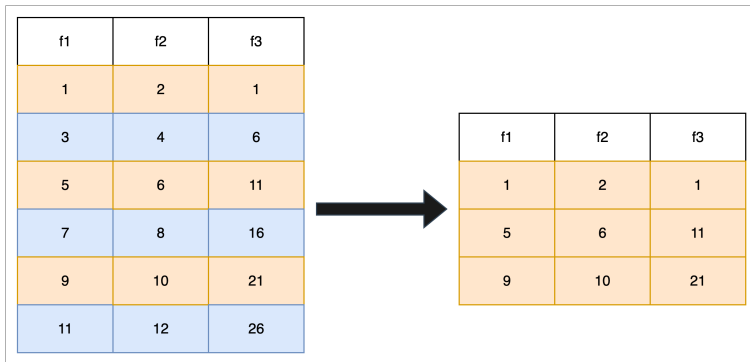
Necessary parameter when downsampling:

- ▶ Downsampling factor: How many data points to skip

Example:

- ▶ Downsampling factor of 2: Every second data point is kept as shown in Figure 1

Downsampling II



Downsampling example with a factor of 2

Downsampling III

How it works:

- ▶ The downsampling function takes a time series dataset and a downsampling factor as input.

Downsampling III

How it works:

- ▶ The downsampling function takes a time series dataset and a downsampling factor as input.
- ▶ It iterates over the dataset and keeps every n -th data point, where n is the downsampling factor.

Downsampling III

How it works:

- ▶ The downsampling function takes a time series dataset and a downsampling factor as input.
- ▶ It iterates over the dataset and keeps every n -th data point, where n is the downsampling factor.

Bottleneck of passing the data by reference:

- ▶ Not possible. A copy is needed.

Downsampling III

How it works:

- ▶ The downsampling function takes a time series dataset and a downsampling factor as input.
- ▶ It iterates over the dataset and keeps every n -th data point, where n is the downsampling factor.

Bottleneck of passing the data by reference:

- ▶ Not possible. A copy is needed.
- ▶ Creating view only possible on contiguous data.

Downsampling III

How it works:

- ▶ The downsampling function takes a time series dataset and a downsampling factor as input.
- ▶ It iterates over the dataset and keeps every n -th data point, where n is the downsampling factor.

Bottleneck of passing the data by reference:

- ▶ Not possible. A copy is needed.
- ▶ Creating view only possible on contiguous data.
- ▶ Downsampling does not yield a contiguous data structure.

Splitting I

Goal: Split a time series dataset into three parts: training, validation, and test.

Splitting I

Goal: Split a time series dataset into three parts: training, validation, and test.

Different splitting strategies:

- ▶ Random split (Classification Data)

Splitting I

Goal: Split a time series dataset into three parts: training, validation, and test.

Different splitting strategies:

- ▶ Random split (Classification Data)
- ▶ In-Order split (Classification Data)

Splitting I

Goal: Split a time series dataset into three parts: training, validation, and test.

Different splitting strategies:

- ▶ Random split (Classification Data)
- ▶ In-Order split (Classification Data)
- ▶ Temporal split (Forecasting Data)

Splitting I

Goal: Split a time series dataset into three parts: training, validation, and test.

Different splitting strategies:

- ▶ Random split (Classification Data)
- ▶ In-Order split (Classification Data)
- ▶ Temporal split (Forecasting Data)

Necessary parameter when splitting:

- ▶ Training set ratio

Splitting I

Goal: Split a time series dataset into three parts: training, validation, and test.

Different splitting strategies:

- ▶ Random split (Classification Data)
- ▶ In-Order split (Classification Data)
- ▶ Temporal split (Forecasting Data)

Necessary parameter when splitting:

- ▶ Training set ratio
- ▶ Validation set ratio

Splitting I

Goal: Split a time series dataset into three parts: training, validation, and test.

Different splitting strategies:

- ▶ Random split (Classification Data)
- ▶ In-Order split (Classification Data)
- ▶ Temporal split (Forecasting Data)

Necessary parameter when splitting:

- ▶ Training set ratio
- ▶ Validation set ratio
- ▶ Test set ratio

Splitting II (Random Split - Classification Data)

How it works:

1. Validate the proportions of train, validation, and test sets.

Splitting II (Random Split - Classification Data)

How it works:

1. Validate the proportions of train, validation, and test sets.
2. Shuffle the dataset randomly.

Splitting II (Random Split - Classification Data)

How it works:

1. Validate the proportions of train, validation, and test sets.
2. Shuffle the dataset randomly.
3. Compute the split offsets based on the proportions.

Splitting II (Random Split - Classification Data)

How it works:

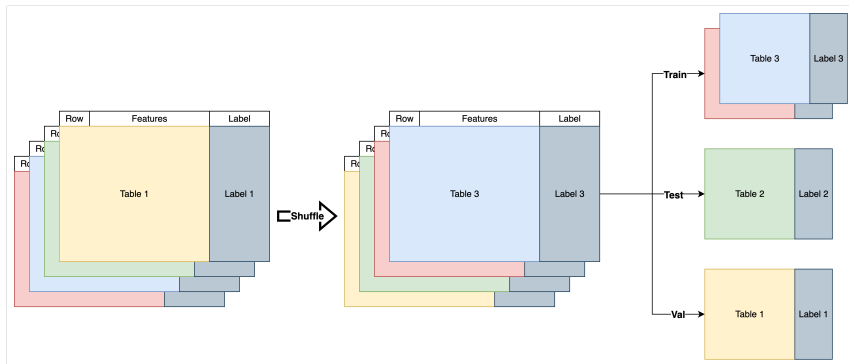
1. Validate the proportions of train, validation, and test sets.
2. Shuffle the dataset randomly.
3. Compute the split offsets based on the proportions.
4. Split the instances into three sets.

Splitting II (Random Split - Classification Data)

How it works:

1. Validate the proportions of train, validation, and test sets.
2. Shuffle the dataset randomly.
3. Compute the split offsets based on the proportions.
4. Split the instances into three sets.
5. Return the three sets as separate datasets.

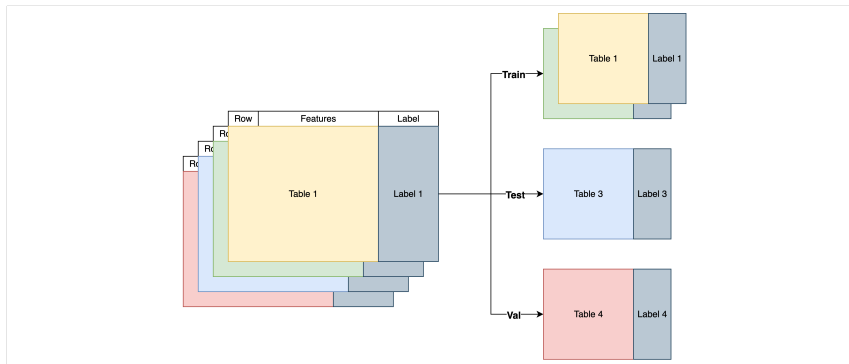
Splitting III (Random Split - Classification Data)



Random split example

Splitting IV (In-Order Split - Classification Data)

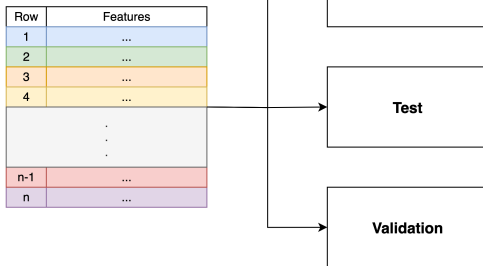
Works very similar to the random split, but it **doesn't shuffle** the dataset anymore.



In-Order split example

Splitting V (Temporal Split - Forecasting Data)

Similar to the in-order split, but this time we are dealing with forecasting data, which in most cases is only one instance and we split over **timesteps** and not instances anymore.



Temporal split example

Standardization

Goal: Transform each feature in a time series dataset to have a mean of 0 and a standard deviation of 1.

Standardization

Goal: Transform each feature in a time series dataset to have a mean of 0 and a standard deviation of 1.

How it works

- ▶ Compute the mean and standard deviation for each feature column in the dataset.

Standardization

Goal: Transform each feature in a time series dataset to have a mean of 0 and a standard deviation of 1.

How it works

- ▶ Compute the mean and standard deviation for each feature column in the dataset.
- ▶ Through a for-loop iterate over each feature and apply the standardization formula:

$$x' = \frac{x - \text{mean}}{\text{std}} \quad (1)$$

Standardization

Goal: Transform each feature in a time series dataset to have a mean of 0 and a standard deviation of 1.

How it works

- ▶ Compute the mean and standard deviation for each feature column in the dataset.
- ▶ Through a for-loop iterate over each feature and apply the standardization formula:

$$x' = \frac{x - \text{mean}}{\text{std}} \quad (1)$$

- ▶ Apply the same mean and standard deviation to the validation and test sets.

Min-Max Normalization

Goal: Transform each feature in a time series dataset to a range between 0 and 1.

Min-Max Normalization

Goal: Transform each feature in a time series dataset to a range between 0 and 1.

How it works

- ▶ Compute the minimum and maximum for each feature in the dataset.

Min-Max Normalization

Goal: Transform each feature in a time series dataset to a range between 0 and 1.

How it works

- ▶ Compute the minimum and maximum for each feature in the dataset.
- ▶ Through a for-loop iterate over each feature and apply the min-max normalization formula:

$$x' = \frac{x - \min}{\max - \min} \quad (2)$$

Min-Max Normalization

Goal: Transform each feature in a time series dataset to a range between 0 and 1.

How it works

- ▶ Compute the minimum and maximum for each feature in the dataset.
- ▶ Through a for-loop iterate over each feature and apply the min-max normalization formula:

$$x' = \frac{x - \min}{\max - \min} \quad (2)$$

- ▶ Apply the same min and max to the validation and test sets.

Kilian's Part