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

Group 1


Marius Kaufmann[1] Amir Ali Aali[2] Kilian Fin Braun[1]

[1]Masters of Computer Science
[2]Masters of Data Science


19th Jul, 2025

RWTH AACHEN UNIVERSITY

# 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:

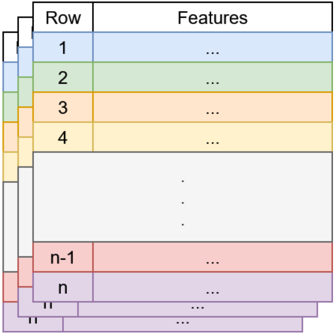
- **First dimension:** Instances
- **Second dimension:** Timesteps
- **Third dimension:** Features

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

# Data Input Format

## Input 3D numpy array:

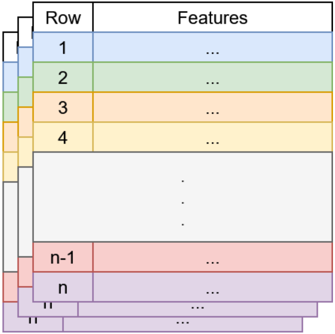
- **First dimension:** Instances
- **Second dimension:** Timesteps
- **Third dimension:** Features



# Data Input Format

## Input 3D numpy array:

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




## In practice

- Forecasting datasets:

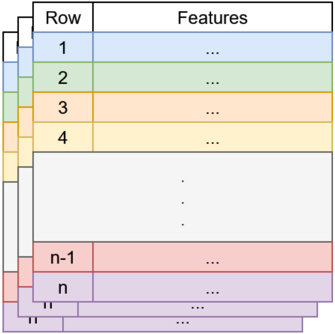
# Data Input Format

## Input 3D numpy array:

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

## In practice

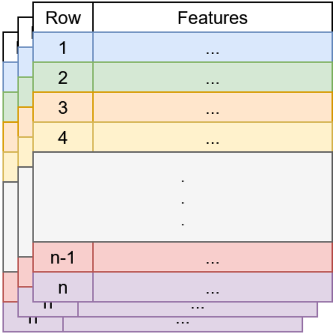
- Forecasting datasets:
    - One instance



# Data Input Format

## Input 3D numpy array:

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



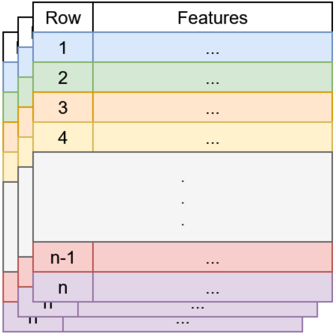

## In practice

- Forecasting datasets:
  - One instance
- Classification datasets:

# 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

# 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**

Original Data
(NumPy Array)

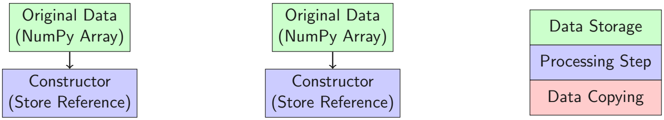
**Classification Dataset Data-Flow**

Original Data
(NumPy Array)

| |
|---|
| Data Storage |
| Processing Step |
| Data Copying |

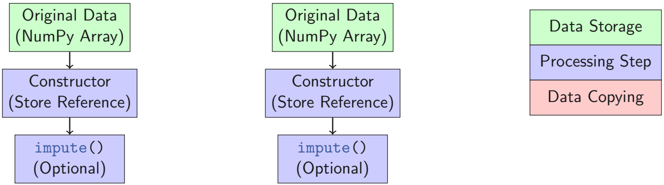
# Data-flow

**Forecasting Dataset Data-Flow** **Classification Dataset Data-Flow**

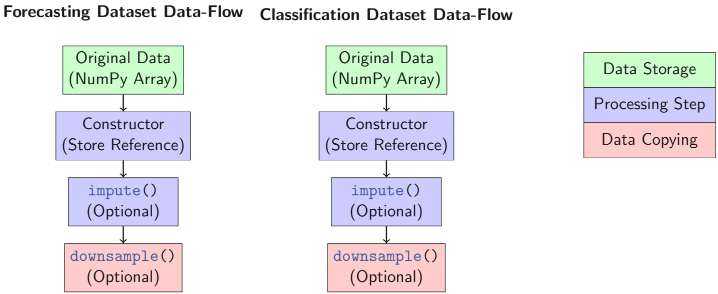


# Data-flow

**Forecasting Dataset Data-Flow** **Classification Dataset Data-Flow**

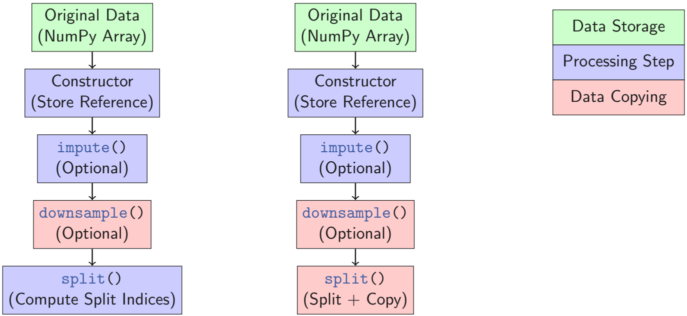


# Data-flow

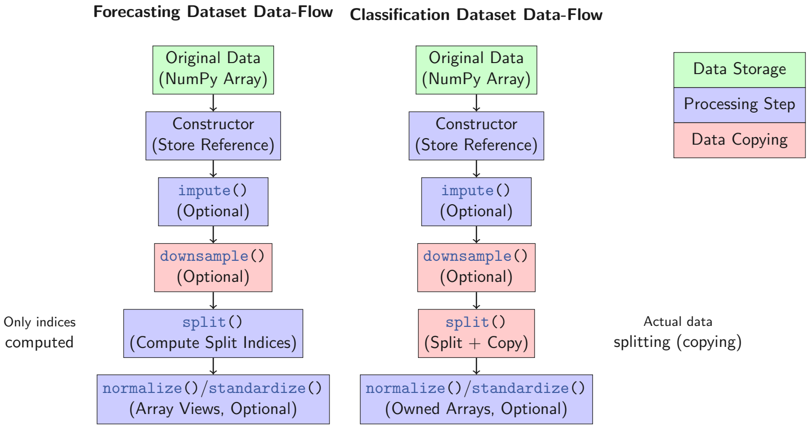

Forecasting Dataset Data-Flow
Original Data (NumPy Array)
Constructor (Store Reference)
impute() (Optional)
downsample() (Optional)
Classification Dataset Data-Flow
Original Data (NumPy Array)
Constructor (Store Reference)
impute() (Optional)
downsample() (Optional)
Data Storage
Processing Step
Data Copying

# Data-flow

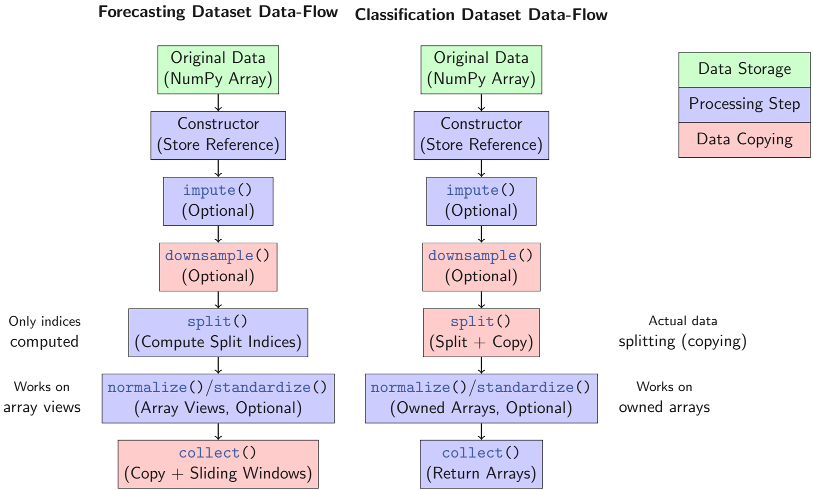
**Forecasting Dataset Data-Flow** **Classification Dataset Data-Flow**



# Data-flow

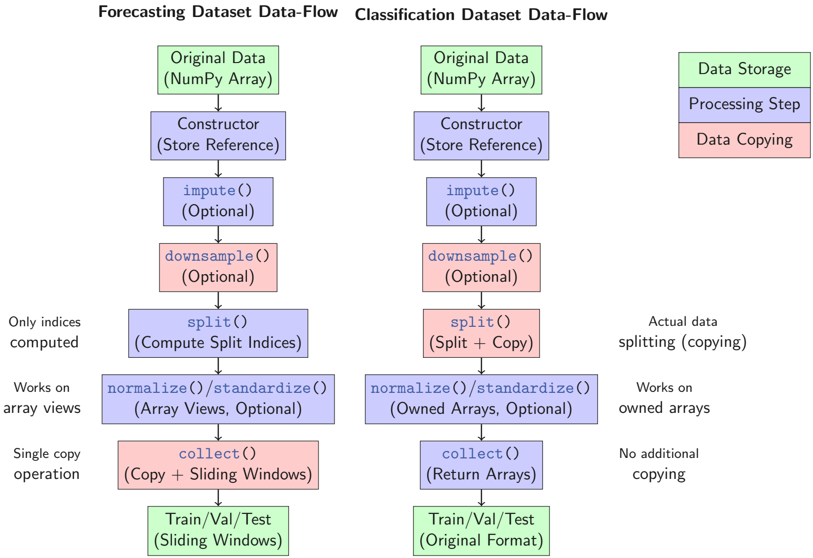

Forecasting Dataset Data-Flow
Original Data (NumPy Array)
Constructor (Store Reference)
impute() (Optional)
downsample() (Optional)
split() (Compute Split Indices)
normalize()/standardize() (Array Views, Optional)
Only indices computed
Classification Dataset Data-Flow
Original Data (NumPy Array)
Constructor (Store Reference)
impute() (Optional)
downsample() (Optional)
split() (Split + Copy)
normalize()/standardize() (Owned Arrays, Optional)
Actual data splitting (copying)
Data Storage
Processing Step
Data Copying

# Data-flow



Forecasting Dataset Data-Flow
Original Data (NumPy Array)
Constructor (Store Reference)
impute() (Optional)
downsample() (Optional)
split() (Compute Split Indices)
normalize()/standardize() (Array Views, Optional)
collect() (Copy + Sliding Windows)
Only indices computed
Works on array views
Classification Dataset Data-Flow
Original Data (NumPy Array)
Constructor (Store Reference)
impute() (Optional)
downsample() (Optional)
split() (Split + Copy)
normalize()/standardize() (Owned Arrays, Optional)
collect() (Return Arrays)
Actual data splitting (copying)
Works on owned arrays
Data Storage
Processing Step
Data Copying

# Data-flow



Forecasting Dataset Data-Flow
Original Data (NumPy Array)
Constructor (Store Reference)
impute() (Optional)
downsample() (Optional)
split() (Compute Split Indices)
normalize()/standardize() (Array Views, Optional)
collect() (Copy + Sliding Windows)
Train/Val/Test (Sliding Windows)
Only indices computed
Works on array views
Single copy operation
Classification Dataset Data-Flow
Original Data (NumPy Array)
Constructor (Store Reference)
impute() (Optional)
downsample() (Optional)
split() (Split + Copy)
normalize()/standardize() (Owned Arrays, Optional)
collect() (Return Arrays)
Train/Val/Test (Original Format)
Actual data splitting (copying)
Works on owned arrays
No additional copying
Data Storage
Processing Step
Data Copying

# 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

```python
# 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

```python
# 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

## Neccessary 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

### Neccessary 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

| f1 | f2 | f3 |
|---|---|---|
| 1 | 2 | 1 |
| 3 | 4 | 6 |
| 5 | 6 | 11 |
| 7 | 8 | 16 |
| 9 | 10 | 21 |
| 11 | 12 | 26 |

| f1 | f2 | f3 |
|---|---|---|
| 1 | 2 | 1 |
| 5 | 6 | 11 |
| 9 | 10 | 21 |

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)

## Neccessary 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)

## Neccessary 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)

## Neccessary 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:

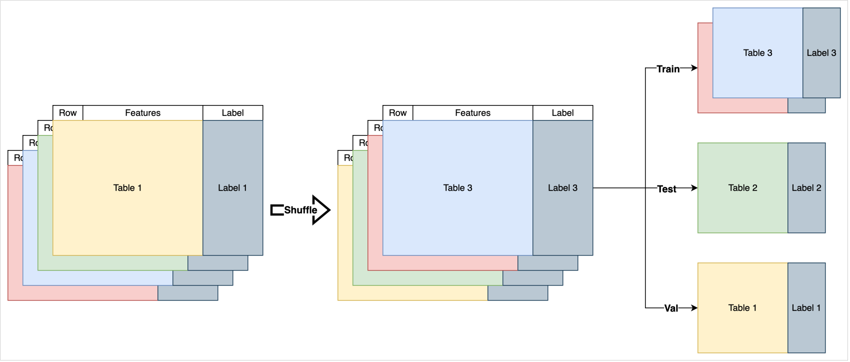
1. Validate the proportions of train, validation, and test sets.
2. Shuffle the dataset randomly.
3. Compute the split offsets based on the proportions.
4. Split the instances into three sets.

# 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.

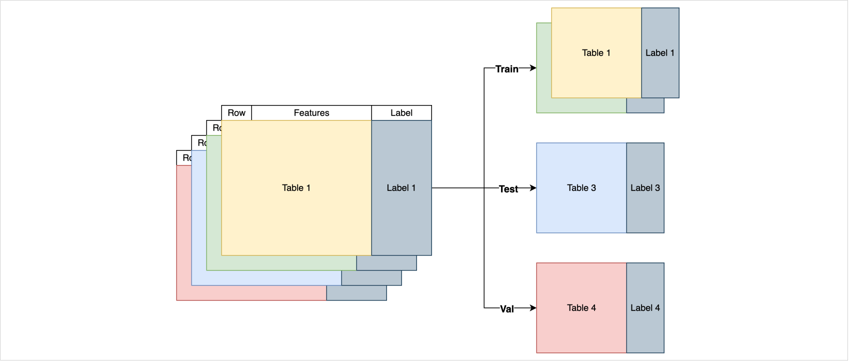
# Splitting III (Random Split - Classification Data)




Random split example

# Splitting IV (In-Order Split - Classification Data)

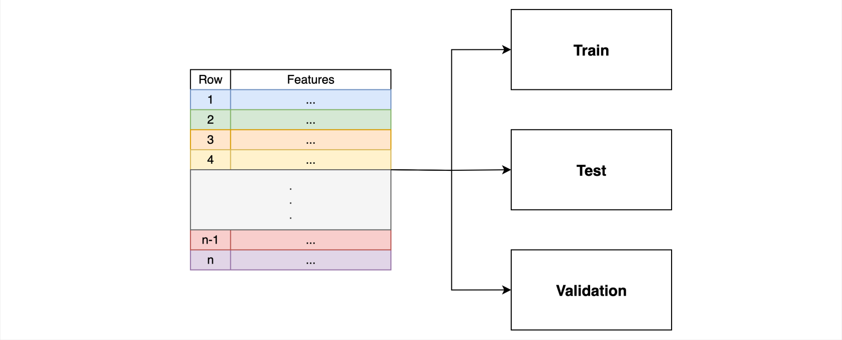
Works very similar to the random split, but it **doesn't shuffle** the dataset anymore.




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 **timestamps** and not isntances 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:

$$x' = \frac{x - \text{mean}}{\text{std}} \tag{1}$$

## 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:

$$x' = \frac{x - \text{mean}}{\text{std}} \tag{1}$$

- 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:

$$x' = \frac{x - \min}{\max - \min} \tag{2}$$

# 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:

$$x' = \frac{x - \min}{\max - \min} \tag{2}$$

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

# Kilian's Part