

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

Marius Kaufmann<sup>1</sup>   Amir Ali Aali<sup>2</sup>   Kilian Fin Braun<sup>1</sup>

<sup>1</sup>Masters of Computer Science

<sup>2</sup>Masters of Data Science

22<sup>nd</sup> 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:

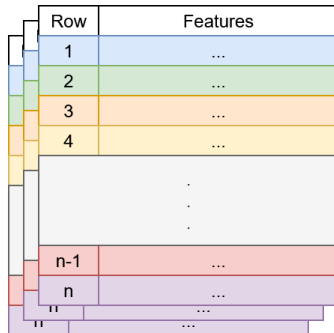
- ▶ **First dimension:** Instances
- ▶ **Second dimension:** Timesteps
- ▶ **Third dimension:** Features

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 first instance (top) has rows 1, 2, 3, 4, and n-1, n. The second instance (middle) has rows 1, 2, 3, 4, and n-1, n. The third instance (bottom) has rows 1, 2, 3, 4, and n-1, n. 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 and n. The columns are labeled 'Row' and 'Features'.

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

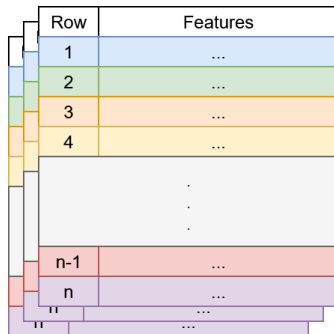
# Data Input Format

## Input 3D numpy array:

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

## 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 values 1, 2, 3, 4, followed by a large grey block representing multiple rows, and then n-1 and n. The 'Features' column contains ellipses (...) for each row. The instances are stacked vertically, with the top instance being light blue, the next light green, then light orange, and the bottom instance being light purple. The 'Row' column is highlighted in the same color as the instance it belongs to.

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

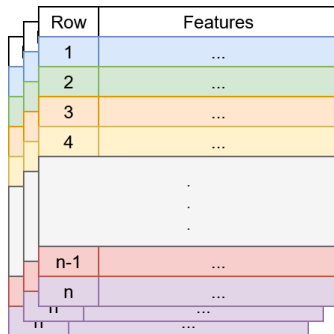
# Data Input Format

## Input 3D numpy array:

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

## In practice

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



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, the first four rows are numbered 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 rows. At the bottom of the stack, two more rows are visible, labeled 'n-1' and 'n', followed by an ellipsis (...). These bottom 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	...
...	

## 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 instances of 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 instances of 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 instances of the dataset randomly.
3. Compute the split offsets based on the proportions.
4. Split the instances into three sets.

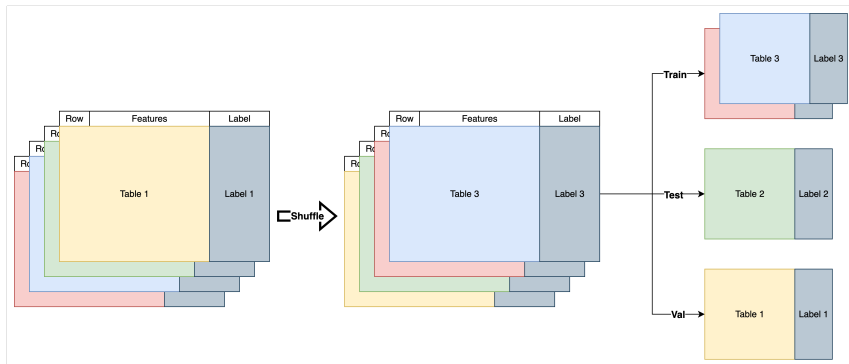
## Splitting II (Random Split - Classification Data)

### How it works:

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



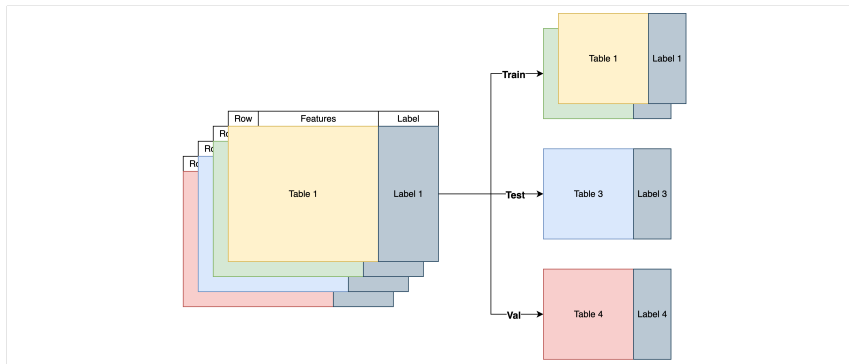
## Splitting III (Random Split - Classification Data)



Random split example

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

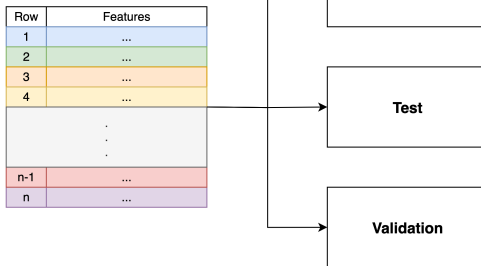
Works very similar to the random split, but it **doesn't shuffle** the dataset anymore.



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

# 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?

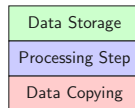
- ▶ For `ForecastingDataSet`:
  - ▶ Windowed format in final step
  - ▶ Copying unavoidable
- ▶ For `ClassificationDataSet`:
  - ▶ Random splitting strategy offered
  - ▶ Copying unavoidable

# Data-flow

Forecasting Dataset Data-Flow    Classification Dataset Data-Flow

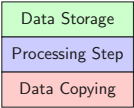
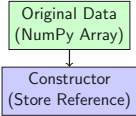
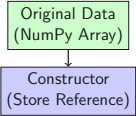
Original Data  
(NumPy Array)

Original Data  
(NumPy Array)



# Data-flow

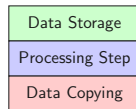
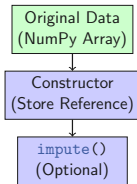
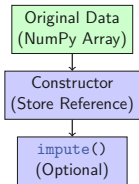
Forecasting Dataset Data-Flow    Classification Dataset Data-Flow





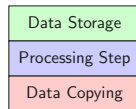
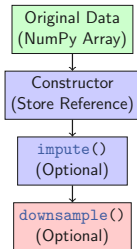
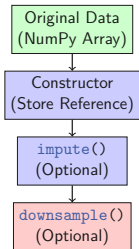
# Data-flow

## Forecasting Dataset Data-Flow    Classification Dataset Data-Flow



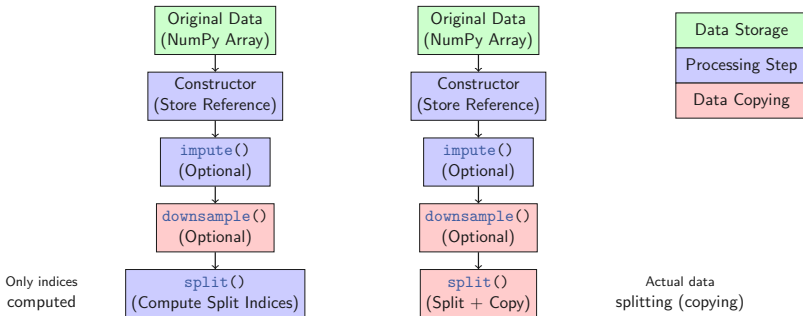
# Data-flow

## Forecasting Dataset Data-Flow    Classification Dataset Data-Flow



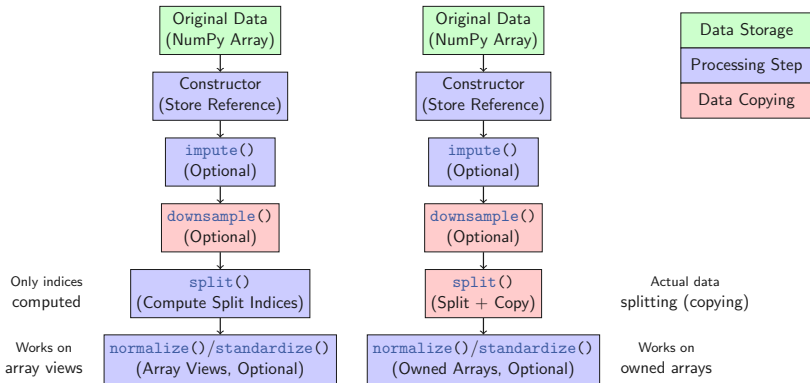
# Data-flow

## Forecasting Dataset Data-Flow    Classification Dataset Data-Flow



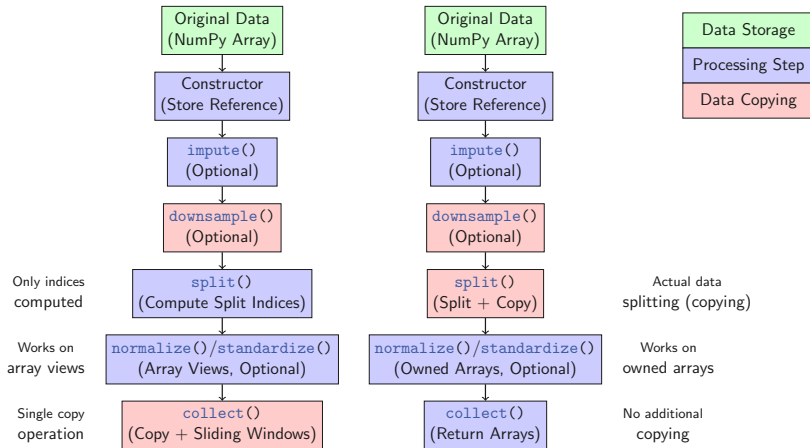
# Data-flow

## Forecasting Dataset Data-Flow    Classification Dataset Data-Flow



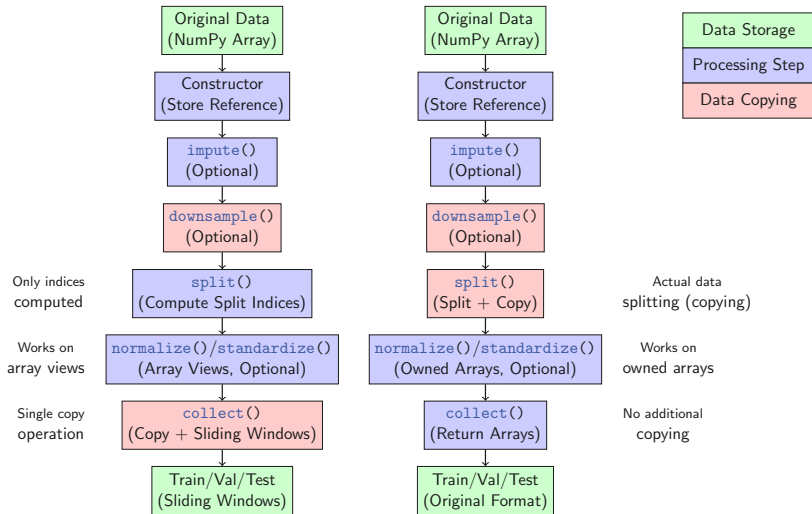
# Data-flow

## 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

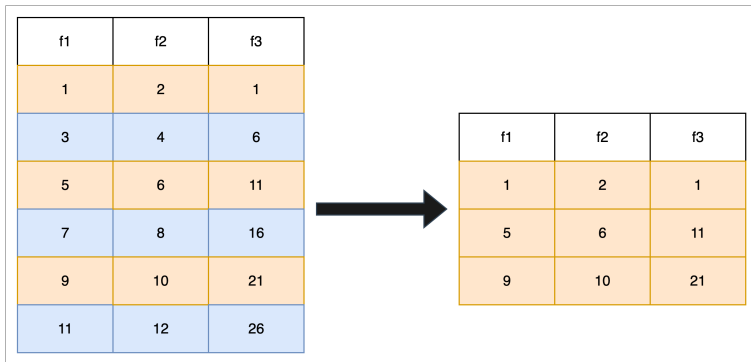
## Necessary parameter when downsampling:

- ▶ Downsampling factor: How many data points to skip

## Example:

- ▶ Downsampling factor of 2: Every second data point is kept

## 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.

## 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 **training** 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 **training** 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 **training** 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 **training** 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 **training** 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 **training** 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.

# Imputing

**Goal:** Impute missing data.

# Imputing

**Goal:** Impute missing data.

**How:**

- ▶ Median

# Imputing

**Goal:** Impute missing data.

**How:**

- ▶ Median
- ▶ Mean

# Imputing

**Goal:** Impute missing data.

**How:**

- ▶ Median
- ▶ Mean
- ▶ Forward-Fill

# Imputing

**Goal:** Impute missing data.

**How:**

- ▶ Median
- ▶ Mean
- ▶ Forward-Fill
- ▶ Backward-Fill



## Unit Tests I

**Goal:** Ensure the correctness of the implemented methods.

## Unit Tests I

**Goal:** Ensure the correctness of the implemented methods.

**Bottleneck:**

- ▶ Rust code is tightly integrated with PyO3.

## Unit Tests I

**Goal:** Ensure the correctness of the implemented methods.

**Bottleneck:**

- ▶ Rust code is tightly integrated with PyO3.
- ▶ PyO3 is not compatible with the standard Rust testing framework.

## Unit Tests I

**Goal:** Ensure the correctness of the implemented methods.

### Bottleneck:

- ▶ Rust code is tightly integrated with PyO3.
- ▶ PyO3 is not compatible with the standard Rust testing framework.
- ▶ PyO3 is a Rust crate that allows Rust code to be called from Python.

## Unit Tests I

**Goal:** Ensure the correctness of the implemented methods.

### Bottleneck:

- ▶ Rust code is tightly integrated with PyO3.
- ▶ PyO3 is not compatible with the standard Rust testing framework.
- ▶ PyO3 is a Rust crate that allows Rust code to be called from Python.
- ▶ PyO3 provides a way to write Python bindings for Rust code.

## Unit Tests I

**Goal:** Ensure the correctness of the implemented methods.

### Bottleneck:

- ▶ Rust code is tightly integrated with PyO3.
- ▶ PyO3 is not compatible with the standard Rust testing framework.
- ▶ PyO3 is a Rust crate that allows Rust code to be called from Python.
- ▶ PyO3 provides a way to write Python bindings for Rust code.

### Solution:

- ▶ Use the PyO3 testing framework.

## Unit Tests I

**Goal:** Ensure the correctness of the implemented methods.

### Bottleneck:

- ▶ Rust code is tightly integrated with PyO3.
- ▶ PyO3 is not compatible with the standard Rust testing framework.
- ▶ PyO3 is a Rust crate that allows Rust code to be called from Python.
- ▶ PyO3 provides a way to write Python bindings for Rust code.

### Solution:

- ▶ Use the PyO3 testing framework.
- ▶ Mimic the Python API in Rust.

## Unit Tests I

**Goal:** Ensure the correctness of the implemented methods.

### Bottleneck:

- ▶ Rust code is tightly integrated with PyO3.
- ▶ PyO3 is not compatible with the standard Rust testing framework.
- ▶ PyO3 is a Rust crate that allows Rust code to be called from Python.
- ▶ PyO3 provides a way to write Python bindings for Rust code.

### Solution:

- ▶ Use the PyO3 testing framework.
- ▶ Mimic the Python API in Rust.
- ▶ Write unit tests in Rust that can be called from Python.



## Unit Tests I

**Goal:** Ensure the correctness of the implemented methods.

### Bottleneck:

- ▶ Rust code is tightly integrated with PyO3.
- ▶ PyO3 is not compatible with the standard Rust testing framework.
- ▶ PyO3 is a Rust crate that allows Rust code to be called from Python.
- ▶ PyO3 provides a way to write Python bindings for Rust code.

### Solution:

- ▶ Use the PyO3 testing framework.
- ▶ Mimic the Python API in Rust.
- ▶ Write unit tests in Rust that can be called from Python.
- ▶ Use the PyO3 testing framework to run the tests.

## Unit Tests II

**Example:** Testing the `impute()` method.

**How it works:**

- ▶ Create a simple numpy array with missing values.

## Unit Tests II

**Example:** Testing the `impute()` method.

### How it works:

- ▶ Create a simple numpy array with missing values.
- ▶ Call the `impute()` method with a specific strategy.

## Unit Tests II

**Example:** Testing the `impute()` method.

### How it works:

- ▶ Create a simple numpy array with missing values.
- ▶ Call the `impute()` method with a specific strategy.
- ▶ Check if the missing values are filled correctly.

## Unit Tests II

**Example:** Testing the `impute()` method.

### How it works:

- ▶ Create a simple numpy array with missing values.
- ▶ Call the `impute()` method with a specific strategy.
- ▶ Check if the missing values are filled correctly.

### Coverage:

- ▶ The unit tests cover most of the implemented methods.

## Unit Tests II

**Example:** Testing the `impute()` method.

### How it works:

- ▶ Create a simple numpy array with missing values.
- ▶ Call the `impute()` method with a specific strategy.
- ▶ Check if the missing values are filled correctly.

### Coverage:

- ▶ The unit tests cover most of the implemented methods.
- ▶ Since tests are not native Rust tests, we couldn't use the standard Rust coverage tools.

## Unit Tests II

**Example:** Testing the `impute()` method.

### How it works:

- ▶ Create a simple numpy array with missing values.
- ▶ Call the `impute()` method with a specific strategy.
- ▶ Check if the missing values are filled correctly.

### Coverage:

- ▶ The unit tests cover most of the implemented methods.
- ▶ Since tests are not native Rust tests, we couldn't use the standard Rust coverage tools.
- ▶ We used the PyO3 testing framework to run the tests and check the coverage.

## Unit Tests II

**Example:** Testing the `impute()` method.

### How it works:

- ▶ Create a simple numpy array with missing values.
- ▶ Call the `impute()` method with a specific strategy.
- ▶ Check if the missing values are filled correctly.

### Coverage:

- ▶ The unit tests cover most of the implemented methods.
- ▶ Since tests are not native Rust tests, we couldn't use the standard Rust coverage tools.
- ▶ We used the PyO3 testing framework to run the tests and check the coverage.
- ▶ The coverage is not as detailed as with the standard Rust testing framework, but it is sufficient for our needs.



## Unit Tests III

### How we calculated the coverage:

- ▶ We used the PyO3 testing framework to run the tests.

## Unit Tests III

### How we calculated the coverage:

- ▶ We used the PyO3 testing framework to run the tests.
- ▶ Counted the number of all methods.

## Unit Tests III

### How we calculated the coverage:

- ▶ We used the PyO3 testing framework to run the tests.
- ▶ Counted the number of all methods.
- ▶ Counted the number of methods that were called during the tests.

## Unit Tests III

### How we calculated the coverage:

- ▶ We used the PyO3 testing framework to run the tests.
- ▶ Counted the number of all methods.
- ▶ Counted the number of methods that were called during the tests.
- ▶ Calculated the coverage as a percentage.

## Unit Tests III

### How we calculated the coverage:

- ▶ We used the PyO3 testing framework to run the tests.
- ▶ Counted the number of all methods.
- ▶ Counted the number of methods that were called during the tests.
- ▶ Calculated the coverage as a percentage.

### Results:

- ▶ Number of all methods: 47

## Unit Tests III

### How we calculated the coverage:

- ▶ We used the PyO3 testing framework to run the tests.
- ▶ Counted the number of all methods.
- ▶ Counted the number of methods that were called during the tests.
- ▶ Calculated the coverage as a percentage.

### Results:

- ▶ Number of all methods: 47
- ▶ Number of methods called during tests: 40

## Unit Tests III

### How we calculated the coverage:

- ▶ We used the PyO3 testing framework to run the tests.
- ▶ Counted the number of all methods.
- ▶ Counted the number of methods that were called during the tests.
- ▶ Calculated the coverage as a percentage.

### Results:

- ▶ Number of all methods: 47
- ▶ Number of methods called during tests: 40
- ▶ Coverage: 85.1%

# Benchmarking

## Goal:

- ▶ Compare vs. PyTorch TimeSeriesDataSet



# Benchmarking

## Goal:

- ▶ Compare vs. PyTorch TimeSeriesDataSet
- ▶ Additionally: Numpy and Python

# Benchmarking

## Goal:

- ▶ Compare vs. PyTorch TimeSeriesDataSet
- ▶ Additionally: Numpy and Python
- ▶ Numpy: Use of API

# Benchmarking

## Goal:

- ▶ Compare vs. PyTorch TimeSeriesDataSet
- ▶ Additionally: Numpy and Python
- ▶ Numpy: Use of API
- ▶ Python: Baseline

# Benchmarking

## Goal:

- ▶ Compare vs. PyTorch TimeSeriesDataSet
- ▶ Additionally: Numpy and Python
- ▶ Numpy: Use of API
- ▶ Python: Baseline

## How:

- ▶ Implement similar Module

# Benchmarking

## Goal:

- ▶ Compare vs. PyTorch TimeSeriesDataSet
- ▶ Additionally: Numpy and Python
- ▶ Numpy: Use of API
- ▶ Python: Baseline

## How:

- ▶ Implement similar Module
- ▶ Vary parameters

# Benchmarking

## Goal:

- ▶ Compare vs. PyTorch TimeSeriesDataSet
- ▶ Additionally: Numpy and Python
- ▶ Numpy: Use of API
- ▶ Python: Baseline

## How:

- ▶ Implement similar Module
- ▶ Vary parameters
- ▶ Test on real data

# Benchmarking

## Goal:

- ▶ Compare vs. PyTorch TimeSeriesDataSet
- ▶ Additionally: Numpy and Python
- ▶ Numpy: Use of API
- ▶ Python: Baseline

## How:

- ▶ Implement similar Module
- ▶ Vary parameters
- ▶ Test on real data
- ▶ Measure timings and memory use

# Measurements

## Timing:

- ▶ Measure timing for each method



# Measurements

## Timing:

- ▶ Measure timing for each method
- ▶ Works for Rust, Python and Numpy

# Measurements

## Timing:

- ▶ Measure timing for each method
- ▶ Works for Rust, Python and Numpy
- ▶ But Torch: Not so much

# Measurements

## Timing:

- ▶ Measure timing for each method
- ▶ Works for Rust, Python and Numpy
- ▶ But Torch: Not so much
- ▶ Also: Torch input & output formatting

# Measurements

## Timing:

- ▶ Measure timing for each method
- ▶ Works for Rust, Python and Numpy
- ▶ But Torch: Not so much
- ▶ Also: Torch input & output formatting

## Peak Memory:

- ▶ Track memory use via Python

# Measurements

## Timing:

- ▶ Measure timing for each method
- ▶ Works for Rust, Python and Numpy
- ▶ But Torch: Not so much
- ▶ Also: Torch input & output formatting

## Peak Memory:

- ▶ Track memory use via Python
- ▶ Record peak use

# Measurements

## Timing:

- ▶ Measure timing for each method
- ▶ Works for Rust, Python and Numpy
- ▶ But Torch: Not so much
- ▶ Also: Torch input & output formatting

## Peak Memory:

- ▶ Track memory use via Python
- ▶ Record peak use
- ▶ **But:** Measurements show the same value (312 MB)

# Measurements

## Timing:

- ▶ Measure timing for each method
- ▶ Works for Rust, Python and Numpy
- ▶ But Torch: Not so much
- ▶ Also: Torch input & output formatting

## Peak Memory:

- ▶ Track memory use via Python
- ▶ Record peak use
- ▶ **But:** Measurements show the same value (312 MB)
- ▶ Setup or measurement error.

## Total setup durations I

- ▶ **Goal:** Measure total setup over different parameters

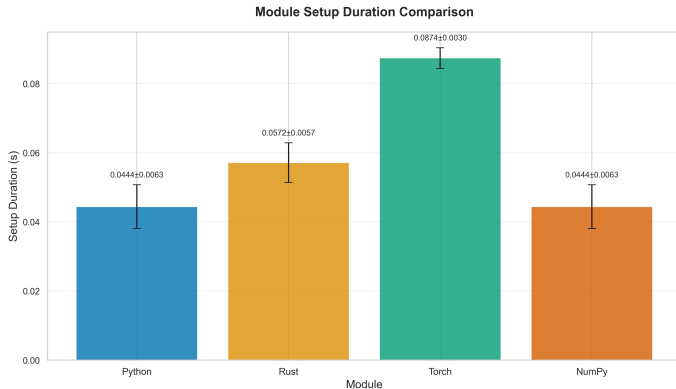


## Total setup durations I

- ▶ **Goal:** Measure total setup over different parameters
- ▶ **Here:** Fixed stride, normalization, downsampling, imputing and splitting

## Total setup durations I

- ▶ **Goal:** Measure total setup over different parameters
- ▶ **Here:** Fixed stride, normalization, downsampling, imputing and splitting



Setup durations on GunPoint

## Total setup durations I

- ▶ **Goal:** Measure total setup over different parameters
- ▶ **Here:** Fixed stride, normalization, downsampling, imputing and splitting

### Explanation:

- ▶ Numpy uses vectorized operations in C

## Total setup durations I

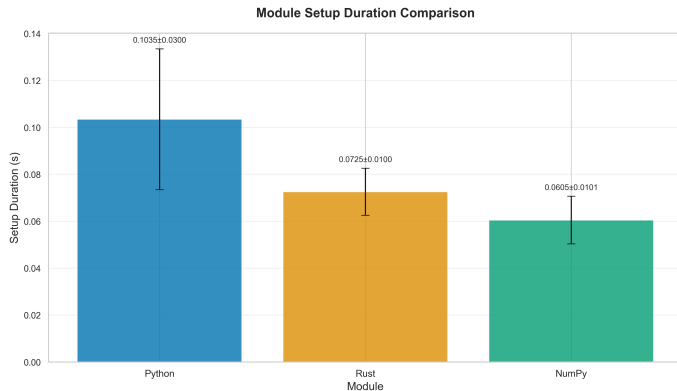
- ▶ **Goal:** Measure total setup over different parameters
- ▶ **Here:** Fixed stride, normalization, downsampling, imputing and splitting

### Explanation:

- ▶ Numpy uses vectorized operations in C
- ▶ Torch overhead from Pandas

## Total setup durations II

- **Goal:** Measure total setup over different parameters



Setup durations on GunPoint

## Total setup durations II

- ▶ **Goal:** Measure total setup over different parameters

### Explanation:

- ▶ More processing benefits Rust and Numpy

## Total iteration durations

- ▶ **Goal:** Measure total data retrieval

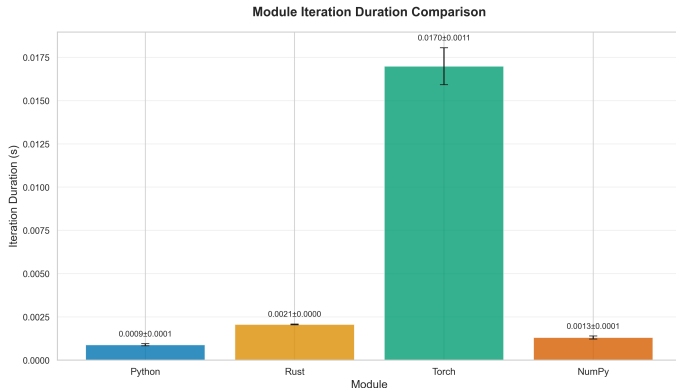
## Total iteration durations

- ▶ **Goal:** Measure total data retrieval
- ▶ **Motivation:** Pytorch uses lazy compute



## Total iteration durations

- ▶ **Goal:** Measure total data retrieval
- ▶ **Motivation:** Pytorch uses lazy compute



Iteration durations on GunPoint

## Total iteration durations

- ▶ **Goal:** Measure total data retrieval
- ▶ **Motivation:** Pytorch uses lazy compute

### Explanation:

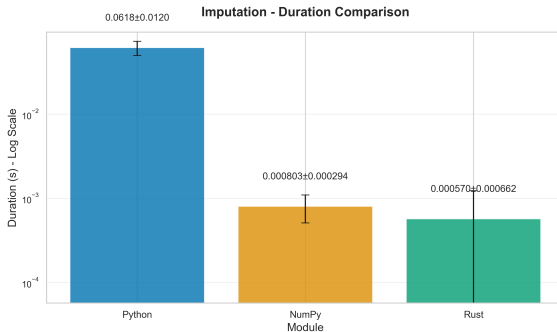
- ▶ PyTorch slowest due to deferred preprocessing during retrieval

## Imputing durations

- ▶ **Goal:** Measure imputing in isolation

# Imputing durations

- **Goal:** Measure imputing in isolation



Imputing durations on GunPoint

## Imputing durations

- ▶ **Goal:** Measure imputing in isolation

### Explanation:

- ▶ Rust benefits from compiler

## Imputing durations

- ▶ **Goal:** Measure imputing in isolation

### Explanation:

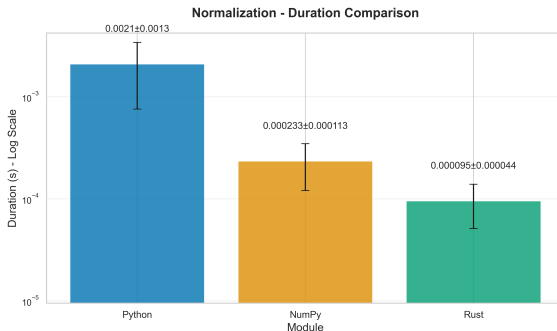
- ▶ Rust benefits from compiler
- ▶ NumPy benefits from partial vectorization

## Normalization durations

- ▶ **Goal:** Measure normalization in isolation

## Normalization durations

- **Goal:** Measure normalization in isolation



Normalization durations on GunPoint



## Normalization durations

- ▶ **Goal:** Measure normalization in isolation

### Explanation:

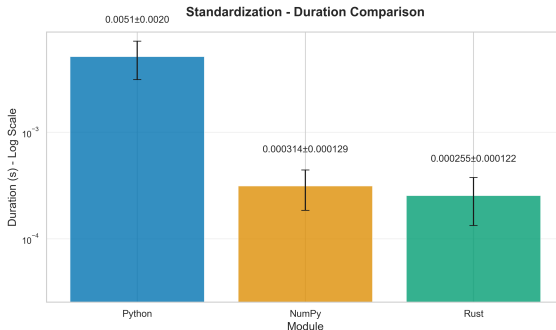
- ▶ **Again:**
  - ▶ Rust benefits from compiler
  - ▶ NumPy benefits from partial

## Standardization durations

- ▶ **Goal:** Measure standardization in isolation

# Standardization durations

- **Goal:** Measure standardization in isolation



Standardization durations on GunPoint

## Standardization durations

- ▶ **Goal:** Measure standardization in isolation

### Explanation:

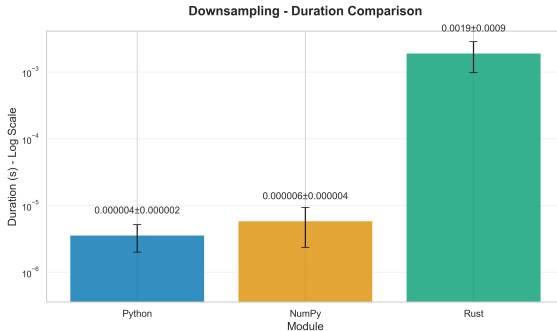
- ▶ **Again:**
  - ▶ Rust benefits from compiler
  - ▶ NumPy benefits from partial

## Downsampling durations

- ▶ **Goal:** Measure downsampling in isolation

# Downsampling durations

- **Goal:** Measure downsampling in isolation



Downsampling durations on GunPoint

## Downsampling durations

- ▶ **Goal:** Measure downsampling in isolation

### Explanation:

- ▶ Rust slowest due to costly data copying

## Data collection durations

- ▶ **Goal:** Measure data collection in isolation



# Data collection durations

- **Goal:** Measure data collection in isolation



Data collection durations on GunPoint

## Data collection durations

- ▶ **Goal:** Measure data collection in isolation

### Explanation:

- ▶ Rust slowest due to Python data transfer