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

16th Jun, 2025

Python/Rust Bindings I

Goal: Passing data from Python to Rust.

Python/Rust Bindings I

Goal: Passing data from Python to Rust.

Simple approach:

- Pass data as Python list to Rust Vector

Python/Rust Bindings I

Goal: Passing data from Python to Rust.

Simple approach:

- Pass data as Python list to Rust Vector
- Simple, works out of the box

Python/Rust Bindings I

Goal: Passing data from Python to Rust.

Simple approach:

- Pass data as Python list to Rust Vector
- Simple, works out of the box

Problem:

- Pass by value

Python/Rust Bindings I

Goal: Passing data from Python to Rust.

Simple approach:

- Pass data as Python list to Rust Vector
- Simple, works out of the box

Problem:

- Pass by value
- Data has to be copied

Python/Rust Bindings I

Goal: Passing data from Python to Rust.

Simple approach:

- Pass data as Python list to Rust Vector
- Simple, works out of the box

Problem:

- Pass by value
- Data has to be copied
- For electricity data set (∼700 MB):
  - Takes time

Python/Rust Bindings II

Solution:

- Pass data by reference
- No overhead

Python/Rust Bindings II

Solution:

- Pass data by reference
- No overhead

How?

Python/Rust Bindings II

Solution:

- Pass data by reference
- No overhead

How?

- Use `numpy.ndarray`!

Python/Rust Bindings II

Solution:

- Pass data by reference
- No overhead

How?

- Use `numpy.ndarray`!
- Rust can access data using `numpy` crate

Python/Rust Bindings II

Solution:

- Pass data by reference
- No overhead

How?

- Use `numpy.ndarray`!
- Rust can access data using `numpy` crate
- **Con:** Requires a little more manual handling

Python/Rust Bindings II

Solution:

- Pass data by reference
- No overhead

How?

- Use `numpy.ndarray`!
- Rust can access data using `numpy` crate
- **Con:** Requires a little more manual handling
- **Pro:** Includes handy built-in functions

Python/Rust Bindings III

Is it worth it?

Python/Rust Bindings III

Is it worth it?

- Passing to Rust:

```python
# pass as list
rust_lib.store_vec(data)
```

✓ 8.8s Python

```python
# pass as numpy array
rust_lib.store_num(data)
```

✓ 0.0s Python

Python/Rust Bindings III

Is it worth it?

- Passing to Rust:

```python
# pass as list
rust_lib.store_vec(data)
```

✓ 8.8s Python

```python
# pass as numpy array
rust_lib.store_num(data)
```

✓ 0.0s Python

- Returning to Python:

```python
# split from list
(first, second) = rust_lib.split_vec()
```

✓ 5.9s Python

```python
# split from numpy array
(first, second) = rust_lib.split_num()
```

✓ 1.6s Python

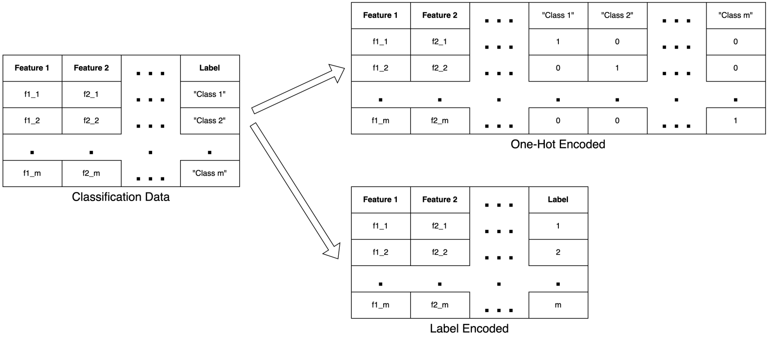
Data Abstraction I

In time series datasets, we often have to deal with mainly two types of data:

- **Forecasting Data:**
  - Contains only floating point values
  - Used for predicting future values
  - Example: Stock prices, weather data
- **Classification Data:**
  - Contains a mix of floating point and categorical values
  - Used for classifying time series data into categories
  - Example: Medical data, sensor data

Data Abstraction II

We require categorical columns to be provided as either one-hot or label-encoded values.
This enables us to save both datasets in a unified way, which is a table of floating point values.



Data Abstraction III

Each dataset type has its own specific parameters for the constructor.

- **Forecasting Dataset:**
  - **data:** The whole dataset as a numpy array
  - **past_length:** Number of past observations to consider for each data point
  - **future_horizon:** Number of future observations to consider for each data point
  - **stride:** The step size for sliding window
- **Classification Dataset:**
  - **data:** The whole dataset as a numpy array
  - **labels:** The labels for the whole dataset as a numpy array

Data Point Representation

For our current implementation, we defined a function *.get(index)* that returns a data point at the given index.
In each of the two dataset types, we have a different representation of the data point.

- **Forecasting Data Point:**
  - **ID:** A unique identifier for the data point
  - **Past:** A vector of floating point values representing past observations
  - **Future:** A vector of floating point values representing future observations
- **Classification Data Point:**
  - **ID:** A unique identifier for the data point
  - **Features:** A vector of floating point values representing the features of the data point
  - **Label:** A vector of floating point values representing the label of the data point

Splitting Strategies

As one of the main features of our library, we provide different splitting strategies for the datasets.

- **Random Split:**
  - Randomly splits the dataset into training and test sets
  - Can be used only for classification data
- **Temporal Split:**
  - Splits the dataset ordered by time
  - Can be used for both forecasting and classification data

Kilian's Part