Rapid Time Serires Datasets Library

Efficient AI with Rust Lab

Marius Kaufmann (422046), Amir Ali Aali (463040), and Kilian Fin Braun (422030)

RWTH Aachen University, Germany
{amir.ali.aali, marius.kaufmann, kilian.braun}@rwth-aachen.de

Table of Contents

1 Introduction

Time series datasets are a common data format in many fields, such as finance and healthcare. They play an important role in many real-world machine learning applications.

They consist of sequences of data points collected over time, and are often used for tasks such as forecasting and classification.

However, these datasets usually require various preprocessing requirements, such as handling missing values, downsampling, normalization, and standardization before they can be used effectively for training models. These preprocessing steps can quickly become computationally expensive for larger datasets, especially when working in Python, where the performance is often not as good as in lower-level languages like Rust or C++.

In this seminar project, we aim to create a library that provides efficient preprocessing operations for time series datasets, implemented in Rust, but usable in Python via PyO3 bindings. The library is designed to be used in machine learning pipelines, and provides a simple interface for preprocessing time series datasets. Our solution enables smooth preprocessing of time series datasets for both forecasting and classification tasks, while maintaining high performance and efficiency.

1.1 Key Goal

A key design goal of this library was to minimize unnecessary data copying between Python and Rust, as this can lead to significant performance overhead. We achieve this by passing data by reference from Python to Rust, using the `numpy` crate, which allows us to work with `numpy` arrays directly in Rust without copying the data. This is particularly important for large time series datasets, where copying the data can be very expensive in terms of performance.

Moreover, we abstract core preprocessing operations like imputation, downsampling, normalization, and standardization into a pipeline-like interface.

1.2 Integration with PyTorch Lightning

To integrate our library into modern machine learning workflows, we also provide a wrapper in the form of a PyTorch Lightning DataModule. This allows our users to load preprocessied time series datasets directly into PyTorch models with a user friendly interface. The DataModule handles the data loading, preprocessing, and batching, making it easy to use our library in PyTorch-based machine learning projects.

1.3 Structure of the Report

In this report we explain the details of the architecture, design and usage of the library. We also give a detailed report of the performance benchmarks, testing strategies, and the integration with PyTorch Lightning.

2 Binding and Design

Since our goal is to create a time series data library that is usable in Python, but implemented in Rust, we used PyO3 to create a Python binding for our Rust library. PyO3 is a Rust crate that allows users to write native Python modules in Rust. It provides a way to make Rust methods, types and classes available in Python by annotating Rust code with special macros, and building a library that can be imported in Python.

2.1 Passing data to Rust

Since our library is used in Python, users will have loaded a time series dataset into their Python environment, and then use our library on this data to prepare it for usage e.g. in machine learning tasks. As our library is implemented in Rust, we need to pass the data from Python to Rust somehow, to be able to operate on it.

PyO3 natively supports data-APIs from Python to Rust, where e.g. a Python list of floating point numbers can be passed to a Rust Vector. This native API operates py passing the data by value, and therefore creates a copy of the data on which the Rust implementation then works. Time series datasets can become quite large. Therefore, passing the data using the native API is slow, it takes around 9 seconds on a medium size dataset (the "ElectricityLoadDiagrams20112014" dataset, which has a size of 678.1 MB) on one of our modern machines. We use Rust for its superior performance, therefore having such a high overhead simply for passing the data to Rust, without performing any operations, is unacceptable.

We solved this problem by passing the data by reference. Natively, this would not be possible, but the Rust crate `numpy` offers a Rust API to `numpy` arrays, which makes passing data by reference from Python to Rust possible. Since in that case, we only need to pass a reference to where the data is stored, the passing is instantaneous and does not slow down the library in any way.

It has to be mentioned that this approach limits the library to only work with `numpy` arrays. By design, `numpy` arrays only consist of elements of the same type, and therefore only arrays of 64-bit floating point numbers are supported. But this is not a serious limitation, as time series data is typically represented as floating point numbers, and the PyTorch `DataLoader`, which is a standard way to load data in PyTorch for machine learning tasks, also only supports floating point data.

2.2 Interface design

The implementation supports two kinds of time series datasets: Forecasting datasets and classification datasets. Since the requirements for the two types of datasets are sufficiently different, we decided to implement them as separate classes. Nevertheless, our goal was to implement the offered preprocessing operations only once, and make them usable for both kinds of datasets.

For consistency of the data layout between the different kinds of datasets, data is expected to be passed as a 3D `numpy` array. The first dimension represents the number of instances. For forecasting datasets this is typically one, for classification datasets this is higher. The second dimension represents the number of timesteps, and the third dimension represents the number of features. This is a common way to represent time series data, and it allows for efficient processing in Rust.

The overall idea is that the class is to be used as a kind of pipeline, storing and manipulating the data internally. The user only passes a reference to the data in the constructor, and retrieves the ready results in the end, and does not have to worry about its storage in the process. For both kinds of datasets, that is, both classes, the interface and is equivalent, and the expected call order is almost the same, with only two differences in the method parameters. These will be explained in the following sections.

The overall structure of the pipeline looks as shown in Figure 1 (exemplary for a `ForecastingDataSet`, but it is almost equivalent for the `ClassificationDataSet` class):

```
# create a ForecastingDataSet instance (pass data and split ratios)
forecasting_data_set = ForecastingDataSet(data, 0.7, 0.2, 0.1)

# call the pipeline methods
forecasting_data_set.impute(ImputeStrategy.ForwardFill)
forecasting_data_set.downsample(2)
forecasting_data_set.split()
forecasting_data_set.normalize()
forecasting_data_set.standardize()

# collect the results (returns the ready data)
forecasting_data_set_res = forecasting_data_set.collect(3, 1, 1)
```

Figure 1: Example usage of the `ForecastingDataSet` class

The overall pipeline workflow goes as follows:

1. The data is passed to Rust in the constructor, which instantiates the provided class. Additionally, the user specifies the proportions of the data that are to be used for training, validation, and testing.
2. If the user wants to impute the missing data, the `impute(strategy)` method can be called. Here, the user can specify the imputation strategy to be used.
3. If the user wants to downsample the data by some factor, the `downsample()` method is called.
4. Calling the `split()` method on `ForecastingDataSet`, no arguments have to be passed, since the split proportions are passed in the constructor. For classification strategies, the `split_strategy` that should be used can also be indicated.
5. If the user wants to normalize the data, the `normalize()` method is called.
6. If the user wants to standardize the data, the `standardize()` method is called.
7. To collect the results, the `collect()` method is called. For forecasting datasets, this method takes three arguments (`past_window`, `future_horizon`, `stride`), for classification datasets, this method does not take arguments.

The `split()` and `collect()` operations are mandatory, since they're essential parts of the pipelines data-flow. The preprocessing operations `impute()`, `downsample()`, `normalize()`, and `standardize()` are optional. Note that, in a realistic use case, the user would choose to call either the `normalize()` or the `standardize()` method, but not both. The call order is expected to be as shown in the example. In case an incorrect call order is used that would lead to a loss of data integrity, an error is raised preventing the user from proceeding with the pipeline.

The difference between the interface of the `split()` method is due to the fact that for forecasting datasets, the temporal splitting strategy is the only valid one, while for classification datasets, the user can choose between in-order and random splitting - requiring a parameter. Similarly, the `collect()` method for forecasting datasets takes three additional parameters (`past_window`, `future_horizon`, `stride`) that are used to construct sliding windows from the data, while for classification datasets, no such parameters are needed, since the data is not converted into sliding windows.

A more detailed presentation of the pipeline will be given in the upcoming subsection 2.4

2.3 Internal data handling

In the constructor, a reference to the data that is to be operated on is passed as a reference to a `numpy` array. This reference is then stored to the class. Since the data that is referenced is stored in the Python memory, this reference needs to be stored using a `Py<...>` smart pointer, which is a "GIL-independent reference to an object allocated on the Python heap" [Dev]. In subsequent method calls, where access to the data is needed, this reference is used to "bind" the data in Rust, which aquires the GIL (Global Interpreter Lock) to ensure that the data is not modified while it is being accessed.

As a general principle, we designed the library to copy data only when it is absolutely necessary. Apart from the `downsample()` operation, on which we'll elaborate in a later section, this is exactly once in the librarys data-flow. It is not possible to implement our functionality without copying the data at least once, since we offer splitting capabilities, which split the data into multiple independent arrays. Where the split and therefore the actual copying is performed is different for the two types of datasets.

Forecasting datasets For forecasting datasets, the `collect(past_window, future_horizon, stride)` method returns the data split into the three aforementioned parts (train, validation, test) and additionally converts them into sliding windows, using the specified parameters. This will be elaborated on in a later section. For now, it is only important to understand that in addition to splitting the data, it is also converted to a different format. This conversion must happen at the final step, just before returning the data. Otherwise, e.g. normalizing the data would cause a huge overhead, since in the process of constructing the sliding windows, data is possibly duplicated - and therefore all copies would have to be changed instead of just the original data.

But if we now actually split the data during the `split()` method, which requires a full copy of the data, and then construct the sliding windows during the `collect()` method which also requires copying the data, we would have to copy the data twice. To avoid this, for forecasting datasets, the `split()` method only computes the indices of the original full data array, where the split would be performed. No actual splitting - and therefore no copying of the data - is done yet. Since for forecasting data, a temporal split is the only valid splitting strategy, this index suffices to store an unambiguous division of the original array into the three parts. The actual split is performed in the `collect()` method, together with the construction of the sliding windows. Therefore, the data is only copied once in the implementation for forecasting datasets.

Classification datasets For classification datasets, the requirements look slightly different.

On the one hand, there are two valid splitting strategies: in-order and random splitting. While for in-order splitting, the instances are kept in the original order, which is the order in which they were passed, in random splitting they are randomly shuffled before being divided into three parts. As a consequence, for classification data using the random splitting strategy, simply storing the indices on which to split the data into three parts does not suffice anymore, since the re-ordering of the datapoints due to the shuffle would then be lost.

On the other hand, the data does not have to be converted into sliding windows, the format of returned values looks like the original data. In sum, this allows for an implementation of the data-flow that is different from the one of the forecasting data, but also only requires to copy the data once: The data can be split and copied into three separate arrays in the `split()` method. The `normalize()` and `standardize()` method then work on the copies of the data, and not on the original array. In the `collect()` method, the previously copied arrays are then simply returned directly, without having to be copied again. Hence, the data is only copied once in the implementation for classification datasets, too.

Generic interfaces for normalize and standardize At first glance, this now poses a problem to our goal to implement preprocessing operations only once, and use them for both kinds of datasets, since we have to call the `normalize()` and `standardize()` methods in very different scenarios: For forecasting datasets, the data remains in the original array, and only the split indices were computed. For classification data, the data is already split into three separate arrays.

But we found a way to use one single generic implementation for both cases: In the Rust `numpy` implementation, there are two kinds of arrays. The struct `Array<...>` represents an actual owned array. The struct `ArrayView<...>` on the other hands represents a view on an array, or possibly also on a part of it. Both of them are inheritants of the `ArrayBase<...>` class, which is one of the fundamental classes of the Rust `numpy` implementation. It offers an interface that allows to read and manipulate the underlying array, be it an actual owned array, or a view on another array.

Creating a view on a part of an array is highly efficient, since no data has to be copied. Hence, given the split indices of the original array, it is possible to create views on the three parts of the array (train, validate and test) very easily for forecasting datasets. Using the generic parent class `ArrayBase<...>` as a parameter type, it is possible to make the `normalize()` and `standardize()` methods callable using both actual owned arrays and array views - mitigating the overhead of having to implement the functionality twice.

The method signature then looks as shown in Figure 2 (exemplary for `normalize()`, but it is the same for `standardize()`):

```
pub fn normalize<S>(
    train_view: &mut ArrayBase<S, Dim<[usize; 3]>>,
    val_view: &mut ArrayBase<S, Dim<[usize; 3]>>,
    test_view: &mut ArrayBase<S, Dim<[usize; 3]>>
) -> PyResult<()>
    where S: DataMut<Elem = f64>
{ ... }
```

Figure 2: Signature of the `normalize()` method

As mentioned before, it can be called using both owned arrays and array views, as shown in Figure 3 and Figure 4.

```
fn normalize(&mut self, _py: Python) -> PyResult<()> {
    check_arrays_set(&self.train_data, &self.val_data, &self.test_data)?;

    normalize(
        &mut self.train_data.as_mut().unwrap(),
        &mut self.val_data.as_mut().unwrap(),
        &mut self.test_data.as_mut().unwrap()
    )?;
    Ok(())
}
```

Figure 3: Calling the `normalize()` method with owned arrays in the `ClassificationDataSet` class

```
fn normalize(&mut self, _py: Python) -> PyResult<()> {
    let (mut train_view, mut val_view, mut test_view) = get_split_views_mut(
        _py,
        &self.data,
        self.train_split_index,
        self.val_split_index
    )?;

    normalize(&mut train_view, &mut val_view, &mut test_view)?;
    Ok(())
}
```

Figure 4: Calling the `normalize()` method with array views in the `ForecastingDataSet` class

2.4 Data-flow Visualization

The different data handling strategies for forecasting and classification datasets result in distinct data-flows, as visualized in Figure 5. The key difference lies in when the actual data copying occurs: forecasting datasets defer copying until the final `collect()` step to avoid double-copying (once for splitting, once for sliding windows), while classification datasets perform the split immediately to accommodate random shuffling strategies.

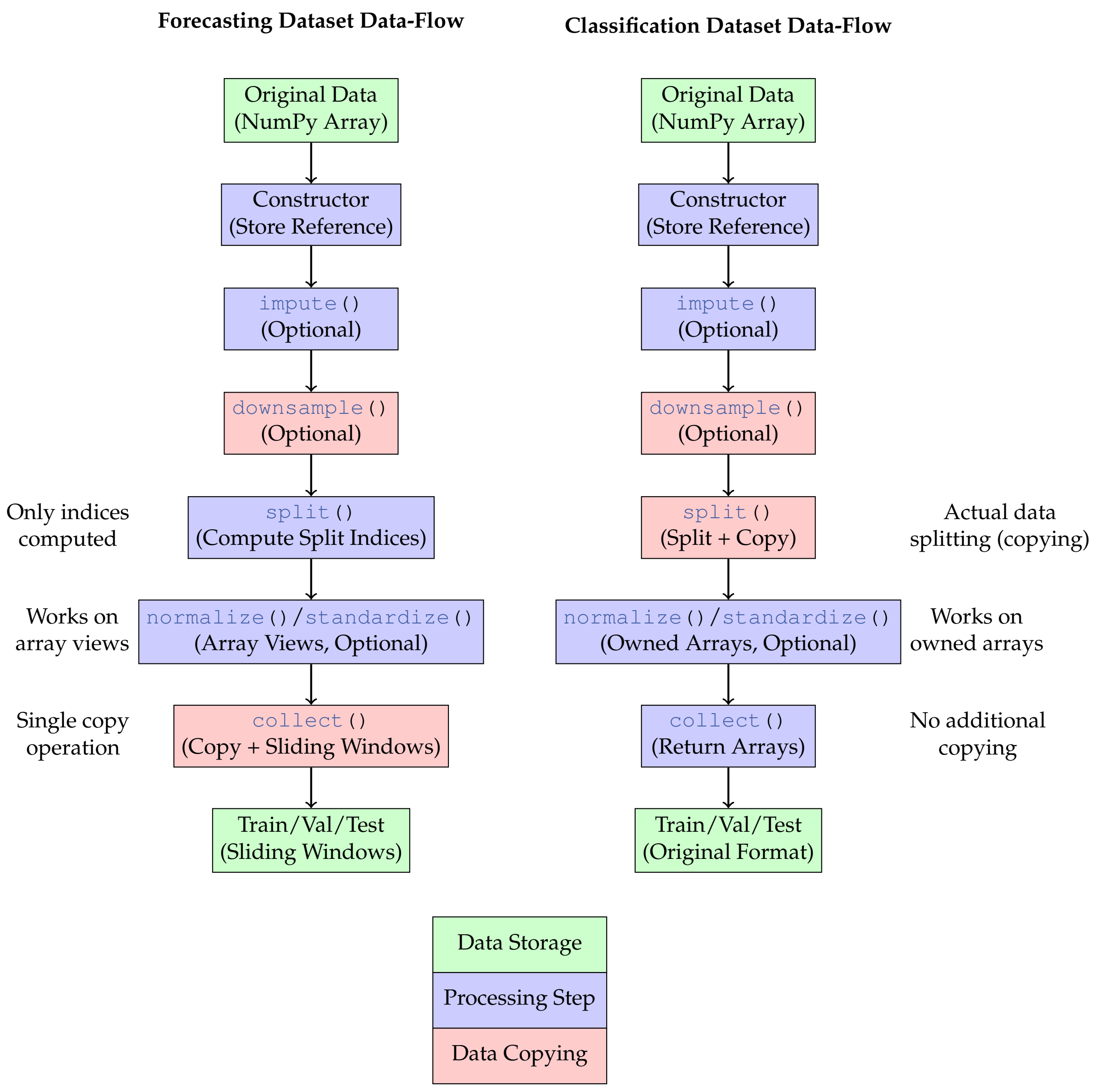

Forecasting Dataset Data-Flow
Classification Dataset Data-Flow
Original Data (NumPy Array)
Constructor (Store Reference)
impute() (Optional)
downsample() (Optional)
split() (Compute Split Indices)
Only indices computed
normalize()/standardize() (Array Views, Optional)
Works on array views
collect() (Copy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

Figure 5: Data-flow comparison between forecasting and classification datasets. Red boxes indicate where data copying occurs, blue boxes indicate where data is processed without copying. Green boxes represent Python-side data storage.

Why it is necessary to copy data in the `downsample()` step is explained in section 4.

2.5 Integration with LightningDataModule

To make the library usable in a machine learning context, we integrated it with the PyTorch Lightning framework. This allows users to easily use our library in their machine learning pipelines, with the user not even having to understand our libraries interface. The integration is done by implementing a class that inherits from the `LightningDataModule` class, which is the base class for all Lightning data modules. Such a `LightningDataModule` can then be used in the `LightningModule` as a data source. The `LightningModule` offers a unified interface for simplifying machine learning workflows. Our class is called `RustDataModule`, and it provides a simple interface to use our library in a Lightning context.

The user only ever needs to interact with the `RustDataModule` class by passing the data as a `numpy` array to the constructor, along with all parameters that let him choose which optional preprocessing features should be used, how the data should be split, and so on. No knowledge about the internal workings of the Rust implmentation is required. The user can then use the `RustDataModule` class in the `LightningModule` as a data source, and the data will be automatically prepared for usage in the machine learning pipeline. The signature of the `RustDataModule` constructor looks as follows:

```python
def __init__(
    self,
    dataset: np.ndarray,
    dataset_type: DatasetType,
    past_window: int = 1,
    future_horizon: int = 1,
    stride: int = 1,
    labels: np.ndarray | None = None,
    batch_size: int = 32,
    num_workers: int = 0,
    downsampling_rate: int = 0,
    normalize: bool = False,
    standardize: bool = False,
    impute_strategy: ImputeStrategy = ImputeStrategy.LeaveNaN,
    splitting_strategy: SplittingStrategy = SplittingStrategy.InOrder,
    splitting_ratios: tuple = (0.7, 0.2, 0.1),  # Train, validation, test ratios
):
...
```

Figure 6: Signature of the `RustDataModule` constructor

The reference to the `numpy` array, along with all options, are saved to the class. At the appropriate time, when the data modules `setup()` method is called, an instance of the appropriate class (`ForecastingDataSet` or `ClassificationDataSet`) is created, and the data is passed to it. All preprocessing methods are called according to the chosen options. The resulting split data is then again stored to the `RustDataModule` class. It can be retrieved in the form of a PyTorch `DataLoader`, which is a standard way to load data in PyTorch. The `DataLoader` is used automatically by the `LightningModule` to load the different parts of the data.

2.6 Enum options

For different options, such as the desired splitting and imputation strategies, we use Rust enums to represent these different options. These enums are then made available in Python via PyO3, so that the user can choose the desired options when calling the methods. The enums are defined in the Rust code, and then annotated with the `#pyo3::pyclass` macro to make them available in Python.

The enums are defined as shown in Figure 7.

```rust
macro_rules! py_enum {
    (
        $(#[$meta:meta])*
        $vis:vis $name:ident { $($variant:ident),* $(,)? }
    ) => {
        #[pyclass]
        #[derive(PartialEq, Clone, Debug)]
        $(#[$meta])*
        $vis enum $name {
            $($variant),*
        }
    };
}

py_enum! {
    pub SplittingStrategy {
        Temporal,
        Random,
    }
}

py_enum! {
    pub ImputeStrategy {
        LeaveNaN,
        Mean,
        Median,
        ForwardFill,
        BackwardFill,
    }
}
```

Figure 7: Definition of the option enums

2.7 File structure

The Rust implementation is divided into the following files:

- `lib.rs`: The main entry point for the binding, where the exposed classes are registered.
- `classification_data_set.rs`: Implements the `ClassificationDataSet` class, which is used for classification datasets.
- `forecasting_data_set.rs`: Implements the `ForecastingDataSet` class, which is used for forecasting datasets.
- `splitting.rs`: Implements the splitting strategies for the datasets.
- `collecting.rs`: Implements the collecting methods for the datasets.
- `preprocessing.rs`: Implements the preprocessing methods for the datasets, such as normalization, standardization, downsampling and imputation.
- `data_abstract.rs`: Contains the previously mentioned exposed enums, which are used to represent the different options for splitting and imputation strategies.
- `utils.rs`: Contains utility functions that are used in the implementation, such as checking if the arrays are set, or getting the split views.
- `abrev_types.rs`: Contains type abbreviations for commonly used types in the implementation.

The tests are implemented in the `tests` directory.

3 Imputing

Imputation is an important step in data pre-processing. The goal is to replace missing data with values. It is typically one of the initial steps in a pre-processing pipeline. In our implementation, we do it as the first step right after calling the constructor.

There are several methods for estimating missing values. Ideally, the data that is imputed is given by an informed model of the data source to preserve the characteristics of the underlying distribution.

There are also imputation methods that work with a wide variety of time series datasets. We decided to implement four of these methods: Median imputation, mean imputation, and forward- and backward-fill imputation. Users also have the option to leave missing data in the data set.

When using imputing methods like mean-imputation it is important to avoid data leakage, just like in normalization or standardization. This is why we implemented imputation on a per-dataset basis, as shown in Figure 8. This is possible, as the proportions of the data that should be used for training, validation and testing are already specified during the construction. Using the specified proportions, three views on the respective parts of the data are created, and the imputation function is called on them.

```
pub fn impute(
    _py: Python,
    train_view: &mut ArrayViewMut3<f64>,
    val_view: &mut ArrayViewMut3<f64>,
    test_view: &mut ArrayViewMut3<f64>,
    strategy: ImputeStrategy
) -> PyResult<()> {
    if strategy == ImputeStrategy::LeaveNaN {
        return Ok(());
    }

    impute_view(_py, &strategy, train_view);
    impute_view(_py, &strategy, val_view);
    impute_view(_py, &strategy, test_view);
    Ok(())
}
```

Figure 8: The `impute()` method handles each part of the dataset individually.

The imputation methods are implemented in separate functions. Imputation is done on a per-instance and per-feature basis. For median and mean imputation, a replacement value is first calculated that will be used to insert missing data. Then, the method iterates over the slice of data and replaces missing values.

For forward-fill and backward-fill, the method iterates over the data while keeping track of the last seen value. When a valid data point is encountered, the last seen value is updated. If a missing data point is found, it is replaced with the last seen value. For these methods, there is an important edge case: When the first or last value is missing, these methods wont replace those. This can be fixed by doing the reverse method instead. So if forward-fill encounters a missing value in the first slot, the method could backward-fill starting from the first good data point. We decided against this, however and implemented the pure version of these imputation methods. An illustration can be seen in Figure 9.

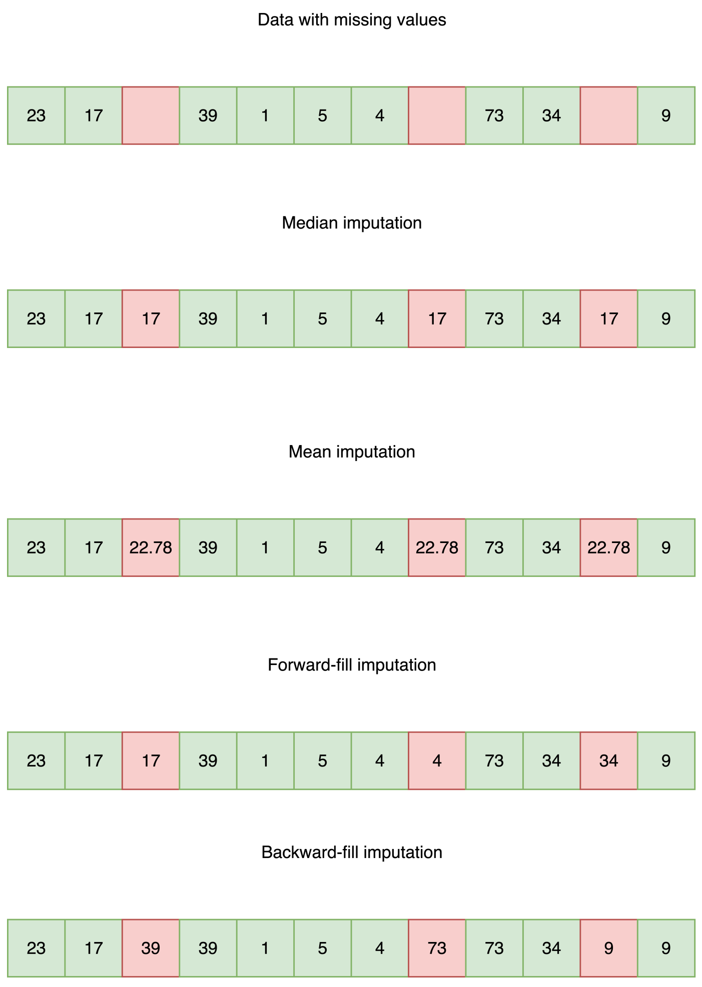


Figure 9: Imputing

4 Downsampling

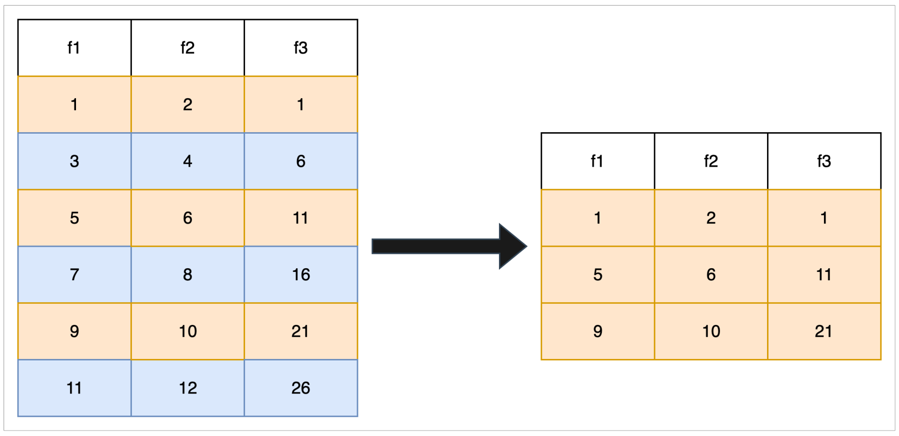
Downsampling is the process of reducing the number of data points in a time series dataset. This is useful for reducing the size of the dataset and speeding up the training process, especially when dealing with large datasets.

When calling the function `downsample()`, we need to pass one parameter:

– `factor`: the factor by which to downsample the dataset, which is an integer greater than 1.

The downsampling process works by taking every `factor`-th data point in the dataset. For example, if the factor is 2, we take every second data point, effectively halving the size of the dataset.

An illustration of how downsampling with `factor` 2 works is shown in Figure 10.



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

Figure 10: Downsampling

In the Rust side, we loop through the dataset and create a new dataset with only the data points that are at indices that are multiples of the downsampling factor. This is done efficiently using the `ndarray` library.

A code snippet of how one may use the downsampling method is shown in Figure 11.

```
classification_data_set.downsample(2)
```

Figure 11: Usage of the downsampling method

As elaborated on before, the Rust `numpy` crate allows to use views on arrays, which are highly efficient and do not require copying the data. Therefore, our first idea was to use a view on the downsampled part the original array, instead of creating a downsampled copy. But this is not possible, since it is only possible to create views on contiguous parts of the original array, and downsampling does not yield a contiguous part of the original array. Therefore, we have to copy the data in the downsampling step.

5 Splitting

With this library, we provide a simple and efficient way to split time series datasets into the three training, validation, and test sets. We support three splitting methods: random, in-order and temporal splitting.

Random splitting is useful for datasets where the order of the instances does not matter like in classification data, while temporal splitting is essential for time series data where the order timesteps is crucial like in the forecasting data. It is important to mention that in the case of classification data, so for in-order and random splitting, the data is split by instance (so on the first dimension), while in the case of forecasting data, so for temporal splitting, the data is split by timestep (so on the second dimension).

List of supported splitting methods:

– classification data : random splitting AND in-order splitting
– forecasting data : temporal splitting

For classification data, when calling the function `split()` we would need to pass the following parameter:

– `split_strategy`: the splitting strategy to use.

The desired proportions of the training, validation, and test sets are passed in the constructor of the class, so they do not have to be passed again when calling the `split()` method. This is due to the fact that they are required in an earlier step of the pipeline, so they are already stored in the class.

As we can see the pipeline of **random splitting** in Figure 12, the steps are as follows:

1. Validate the proportions of the training, validation, and test sets.
2. Compute the number of instances in the dataset.
3. Compute the split offsets for the training, validation, and test sets.
4. Shuffle the instances.
5. Split the instances into the three sets.
6. Return the training, validation, and test sets.

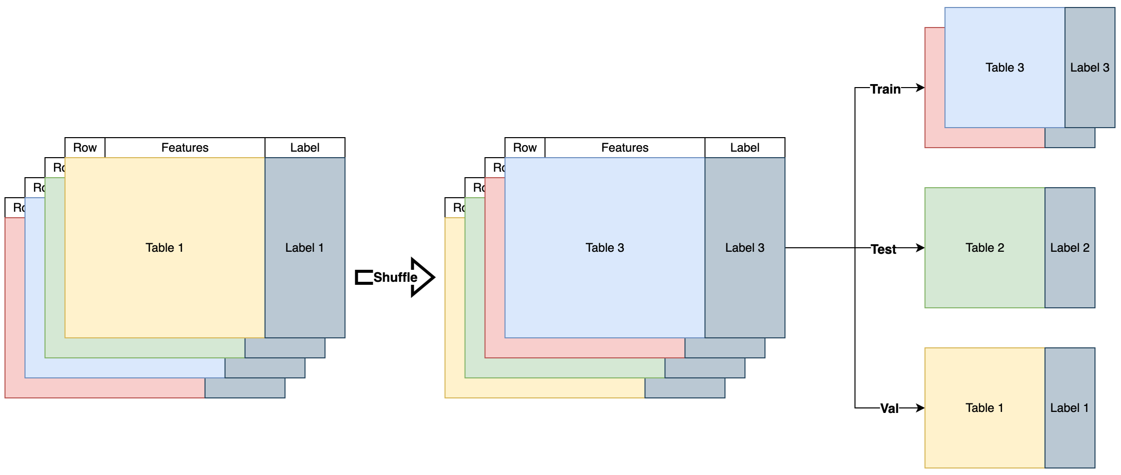


Figure 12: Random Splitting

In case of **In-Order splitting**, as shown in Figure 13, we follow a similar approach, but we do not shuffle the instances any more. The steps are as follows:

1. Validate the proportions of the training, validation, and test sets.
2. Compute the number of timesteps in the dataset.
3. Compute the split offsets for the training, validation, and test sets.
4. Split the instances into the three sets.
5. Return the training, validation, and test sets.

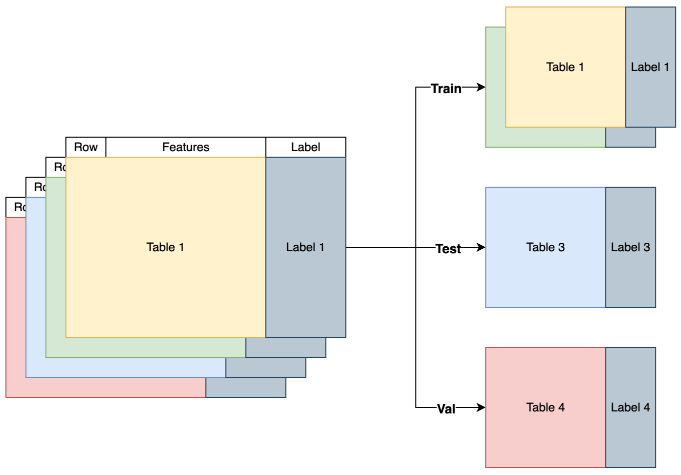


Figure 13: In-Order Splitting

And finally, for **Temporal splitting**, as shown in Figure 14, we follow a similar approach, but this time we are dealing with forecasting data, where in most cases we only have one instance in the dataset and we split based on the timesteps.

The steps are as follows:

1. Validate the proportions of the training, validation, and test sets.
2. Compute the number of timesteps in the dataset.
3. Compute the split offsets for the training, validation, and test sets.
4. Split the dataset into the three sets based on the computed offsets.
5. Return the training, validation, and test sets.

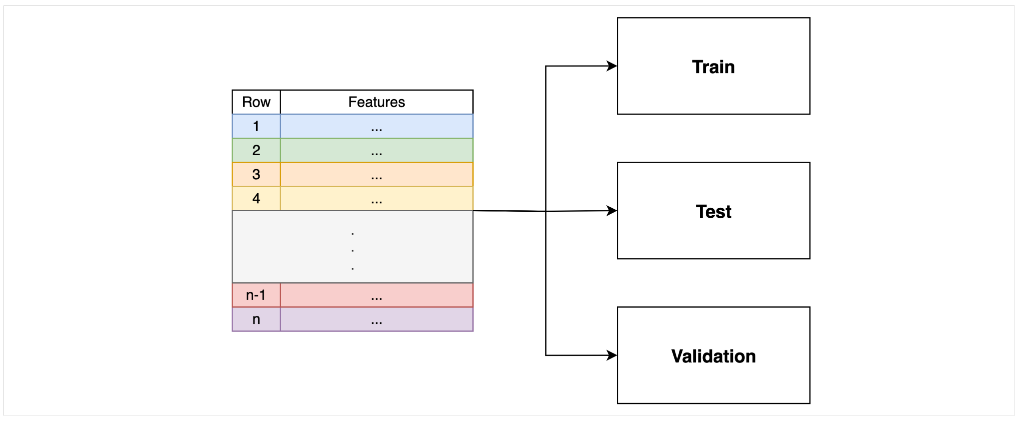


Figure 14: Temporal Splitting

A code snippet of how one may use the splitting method is shown in Figure 15.

```
classification_data_set.split(SplittingStrategy.InOrder)
```

Figure 15: Usage of the `split()` method for `ClassificationDataSet`

6 Standardization and Normalization

Another important step in preprocessing time series data is standardization and normalization.

Standardization is the process of scaling the data to have a mean of 0 and a standard deviation of 1, while min-max normalization scales the data to a range between 0 and 1.

We perform these two operations distinctly on each column of the dataset, which is essential for time series data where each column represents a different feature.

It is also worth mentioning that key features such as mean, standard deviation, min, and max are computed only once for the training set and then applied to the training, validation and test sets. This is crucial to avoid data leakage, which can lead to overfitting and unrealistic performance metrics.

6.1 Standardization

Standardization is performed by subtracting the mean and dividing by the standard deviation for each feature. This ensures that the data has a mean of 0 and a standard deviation of 1.

Here are the main steps of the standardization process:

1. Compute the mean and standard deviation for each feature in the training set.
2. Through a for loop iterate over each feature and apply the standardization formula:

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

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

6.2 Min-Max Normalization

Min-max normalization scales the data to a range between 0 and 1. This is particularly useful for algorithms that are sensitive to the scale of the data, such as neural networks.

Here are the main steps of the min-max normalization process:

1. Compute the minimum and maximum values for each feature in the training set.
2. Through a for loop iterate over each feature and apply the min-max normalization formula:

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

3. Apply the same min and max values to the validation and test sets.

In the exceptional scenario, where the difference between the minimum and maximum values is zero, we set their values to 1, which avoids division by zero errors.

A code snippet of how one may use the standardization and normalization methods is shown in Figure 16.

```
classification_data_set.normalize()
classification_data_set.standardize()
```

Figure 16: Usage of the standardization and normalization methods

7 Testing

We have implemented unit tests for nearly all the methods in the library, as well as integration tests that directly test the compiled Python module. These tests cover various scenarios and edge cases to ensure the correctness of the implementation.

Since our rust code is deeply integrated with bindings and the the Pyo3 library, we were not able to write pure Rust tests. Instead, we have mimicked python environment in the Rust tests, which allows us to test the methods of the Python module as if they were called from Python. We did this by using the PyO3 library, which allows us to write Python code in Rust and test it as if it were called from Python.

One might wonder why we did not use the `pytest` library to test the Python code directly. The reason is that we wanted to ensure that the Rust code is working correctly and that the bindings are working as expected. This way, we can catch any issues in the Rust code before they propagate to the Python side. Also, there were some functions that were not directly callable from Python, as they were private functions in the Rust code. Since these should not be exposed to Python by design, this was not an option. We wanted to test these functions as well, so we had to write tests in Rust and compile them differently from how we compiled for the Python bindings.

7.1 Running the Tests

To run the tests, we would need to follow these steps:

1. One may want to clean up all the build artifacts from previous builds. This can be done by running the following command:

   ```
   cargo clean
   ```

2. Ensure that the `maturin` library is installed. This can be done by running the following command:

   ```
   pip install maturin
   ```

3. Now build the Rust code with the `test_expose` feature enabled. This can be done by running the following command:

   ```
   maturin develop --features test_expose
   ```

4. (Only on Mac) Find the location of the `libpython3.12.dylib` file. This can be done by running the following command:

   ```
   find $(python3 -c "import sys; print(sys.prefix)") -name "libpython3.12.dylib"
   ```

5. (Only on Mac) Set the `DYLD_LIBRARY_PATH` environment variable to the path of the `libpython3.12.dylib` file. This can be done by running the following command:

   ```
   export DYLD_LIBRARY_PATH=given_path_from_above
   ```

6. Finally, run the tests with the following command:

   ```
   cargo test --features test_expose
   ```

7.2 Test Coverage

As also mentioned above, we have implemented the unit- and integration tests through mimicking the Python environment in Rust. This came with some down sides, which doesn't allow us to use the
`cargo-tarpaulin` library to measure the test coverage. However, we have manually checked the coverage of the tests and ensured that all the methods are covered by the tests. We have also ensured that the tests cover various scenarios and edge cases to ensure the correctness of the implementation.

In order to have a rough estimate of the test coverage, we have counted the number of all the functions in the library and the number of functions that are covered by the tests.

The results are as follows:

- Total number of functions(excluding bindings): 47
- Number of test cases: 40
- Test coverage: **85%**

Please note that this is a rough estimate and the actual test coverage may vary. We have also ensured that the tests cover various scenarios and edge cases to ensure the correctness of the implementation.

8 Benchmarking

In order to benchmark our implementation, we implemented four Python modules with built in timing and memory profiling. One of these modules calls our Python wrapper. The second module is a pure Python implementation of the same methods we implemented in Rust. This serves as a baseline comparison, which we aim to outperform. The third module is an implementation using numpy methods where possible. We expect this implementation to be on par or faster then ours, since we rely on the same API in our Rust implementation. The fourth module is a implementation focussing on the TimeSeriesDataSet class from Torch.

The TimeSeriesDataSet class and its methods were specifically requested for benchmarking because they are commonly used. However, the benchmarking module does not offer the same features as the Rust implementation. To keep the comparison fair, we focused our benchmarks on the features provided by Torch. It is also important to note that the class expects the data to be in a Pandas DataFrame. We thus must apply transformations to the data before we can create the dataset. Additionally, there is no method to handle missing data in the form of NaN values in individual columns. Finally, scaling occurs while retrieving samples from the data loader, so timing this process is not straightforward. To get the data in the same shape from the data loader as our implementation, we must transform the samples returned from the data loader on the fly.

Benchmarking was performed on a classification dataset from the UCR/UEA archive. We selected the multivariate JapaneseVowels subset and the univariate GunPoint subset. For each subset, we tested all four modules, recording the timing and peak memory usage, in different configurations. The configurations varied in terms of window size, stride, downsampling rate, normalization, standardization, imputation, and splitting strategy. As mentioned earlier, the Torch module was only benchmarked on a subset of these configurations by setting downsampling to 1, turning imputing off, deactivating normalization and by only splitting in order. Each module was measured ten times.

8.1 Total setup duration

The first important metric is the total setup duration. The first analysis will be comparing all four modules in the total setup time it takes. This measures how long the call to the setup() function takes. On both the JapaneseVowels and GunPoint subset, it is clearly visible that Torch takes the longest, seen in Figure 17 and Figure 18. This probably due to the data conversion that needs to be done due to TimeSeriesDataSet expected format being a pandas Dataframe. The fact that Python tends to outperform our implementation can be attributed to the comparatively little preprocessing happening. Using Rust in these cases probably only adds overhead.

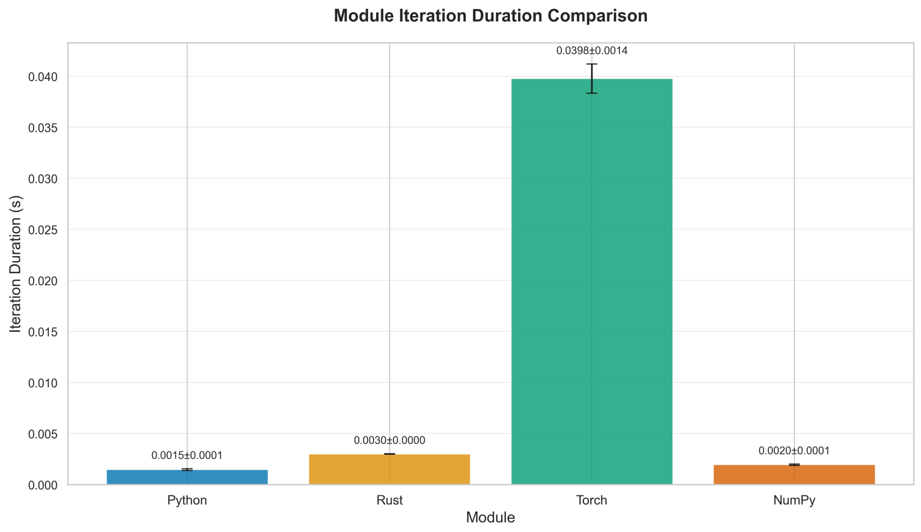


Figure 17: Total setup time on JapaneseVowels

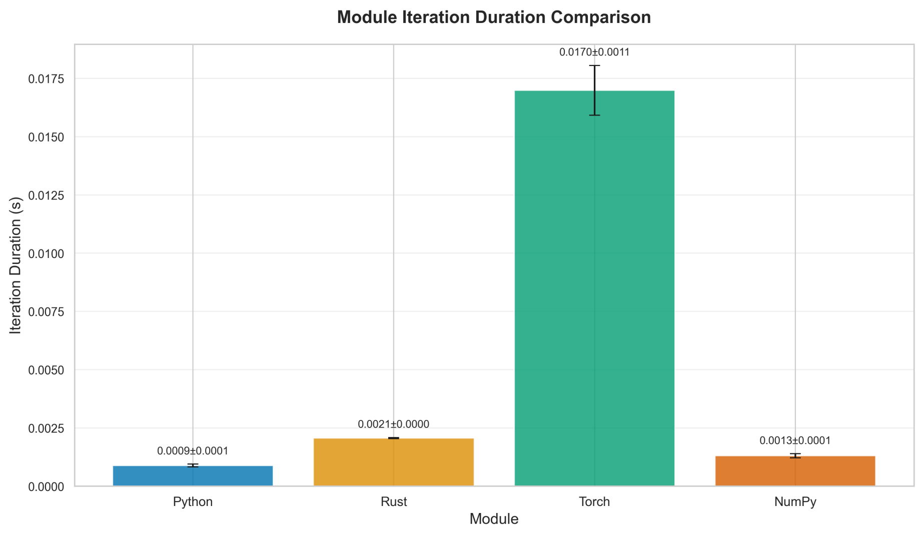


Figure 18: Total setup time on GunPoint

In order to gauge how the total setup time varies when the parameters we fixed due to Torch's limitations are also varied, we can compare the total setup time of the remaining three modules, seen in Figure 20 and Figure 19. Here it is Python that is the slowest overall while numpy is the fastest. This is more in line with our expectations. It is also expected that numpy performs the best.

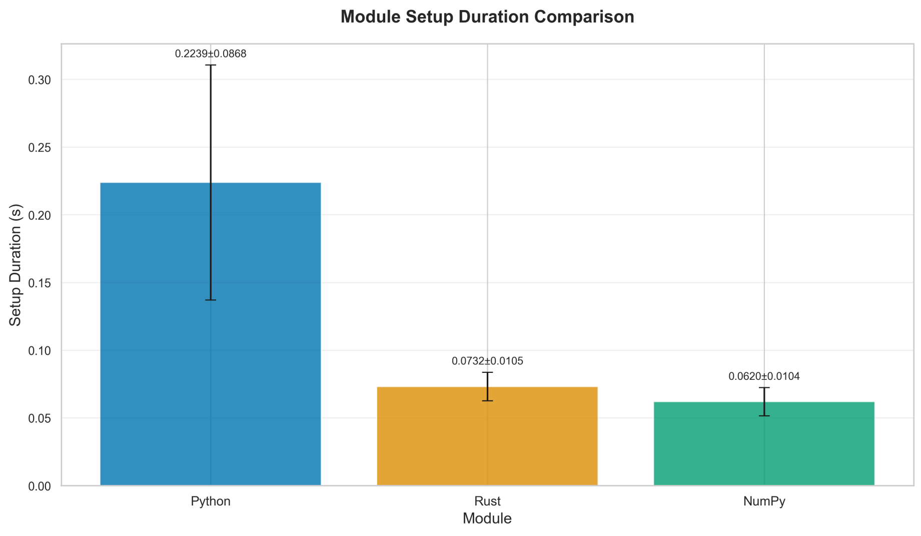


Figure 19: Total setup time on JapaneseVowels

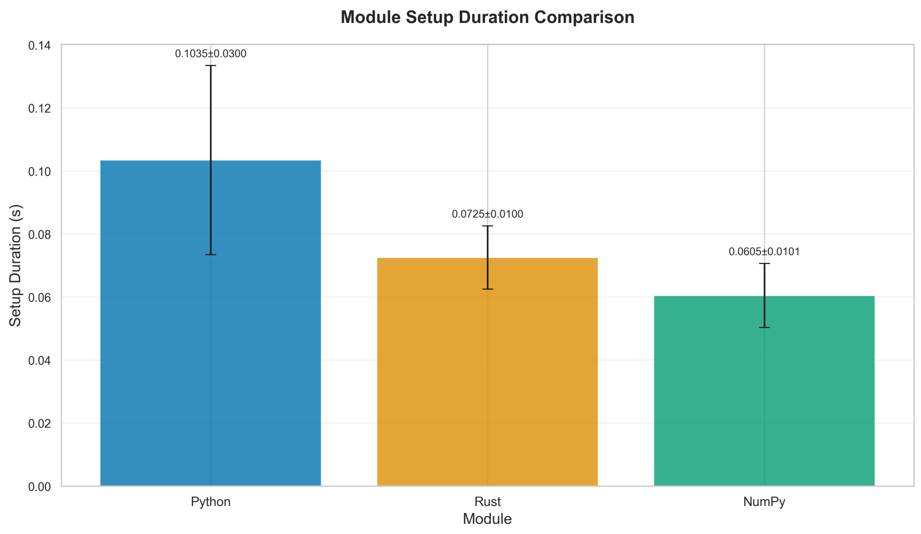


Figure 20: Total setup time on GunPoint

8.2 Total iteration duration

This measure reflects how fast the data can be retrieved from the dataloaders. This is relevant during the training cycle of a model. It also symbolizes a tradeoff, as computations not done upfront can be deferred to this stage, seen in Figure 21 and Figure 22. As we expected, pytorch performed the worst here overall. This is due to the fact that operations like scaling are deferred to the point when the data is actually retrieved and thus means a big overhead during the training cycle. The other three modules vary only minimally. This is expected since all only retrieve data from the DataLoader without doing any computations.

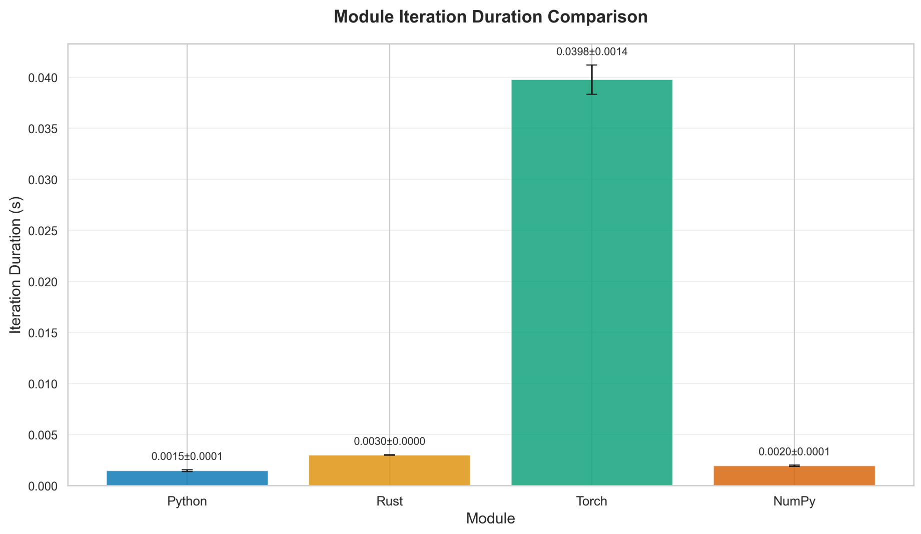


Figure 21: Total iteration time on JapaneseVowels

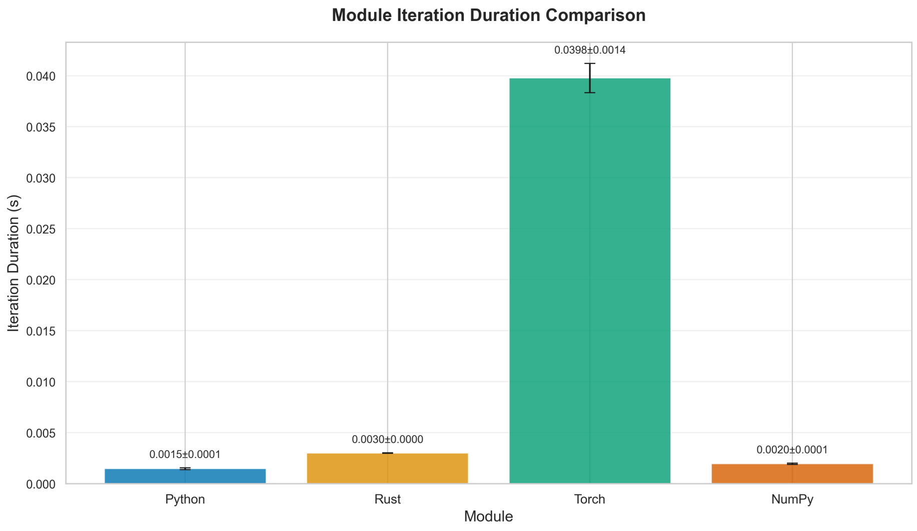


Figure 22: Total iteration time on GunPoint

8.3 Preprocessing durations

Next, we will examine the performance measurements of each preprocessing step. These are only captured for the NumPy, Rust, and PyTorch modules. First, the imputing measurements. This measures the time it takes to impute missing values, shown in Figure 23 and Figure 24. As expected, Python takes the longest, followed by NumPy, and then Rust. This is expected since both Rust and NumPy are generally faster. However, NumPy uses features like vectorized assignments, which speed up operations significantly, but it also uses the slow Python for-loop.

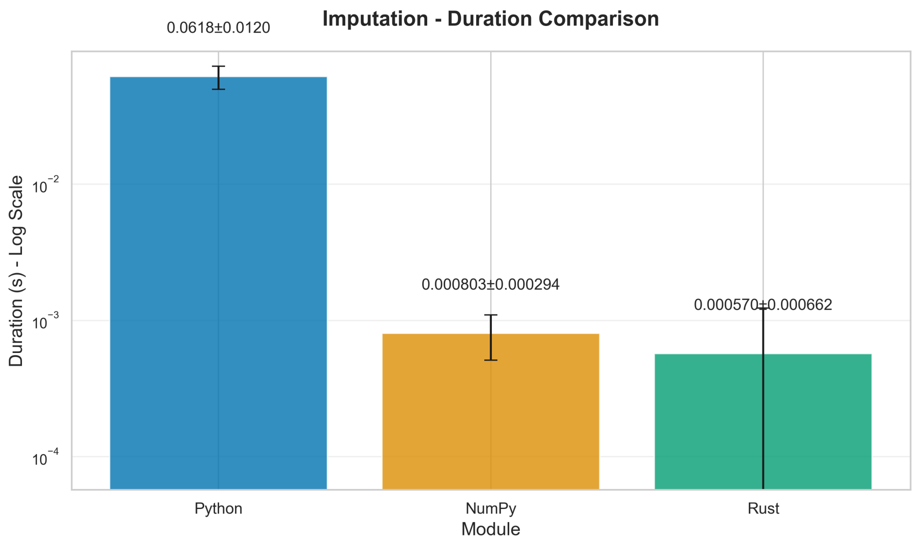


Figure 23: Total duration of imputing on GunPoint

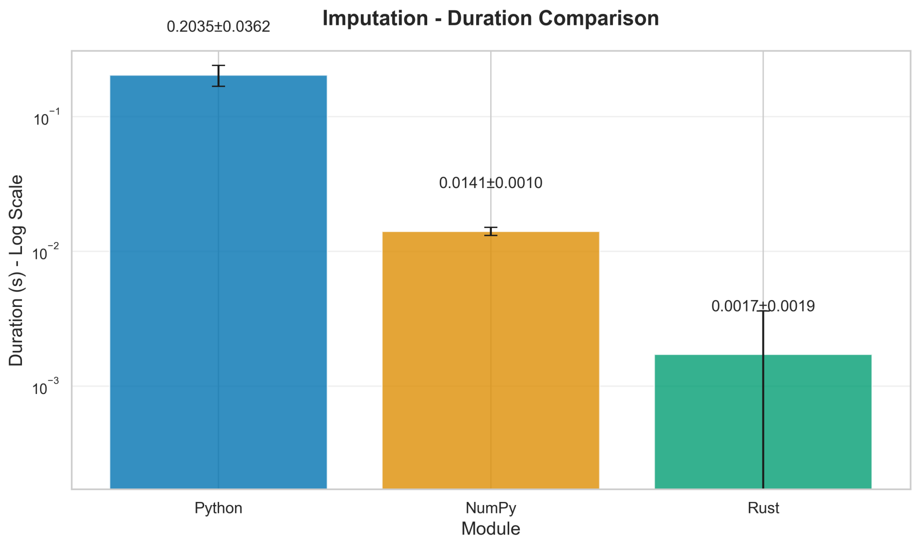


Figure 24: Total duration of imputing on JapaneseVowels

Again, for normalization, Python is the slowest, followed by Rust, and then Numpy. Numpy is the fastest because it avoids Python for-loops and uses fully vectorized operations. See Figure 25 and Figure 26. The same observations apply to standardization, shown in Figure 27 and Figure 28.

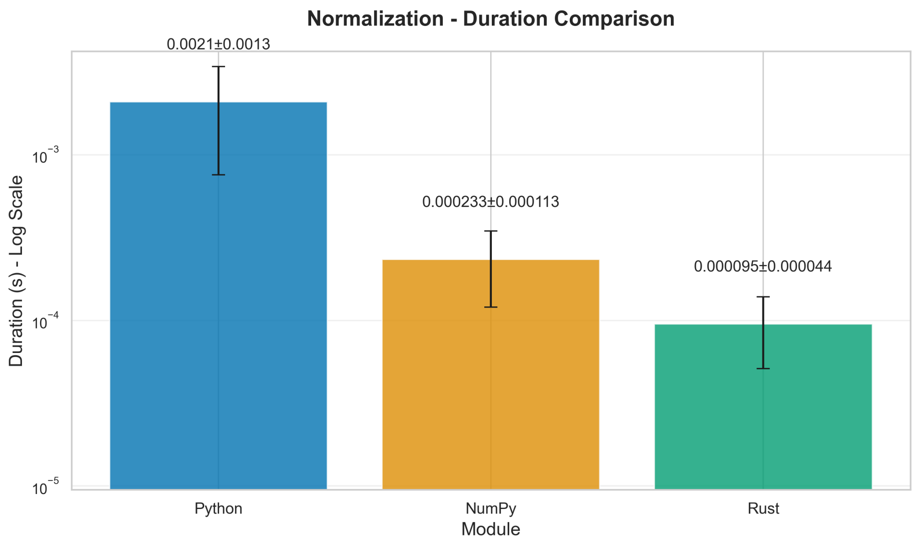


Figure 25: Total duration of normalizing on GunPoint

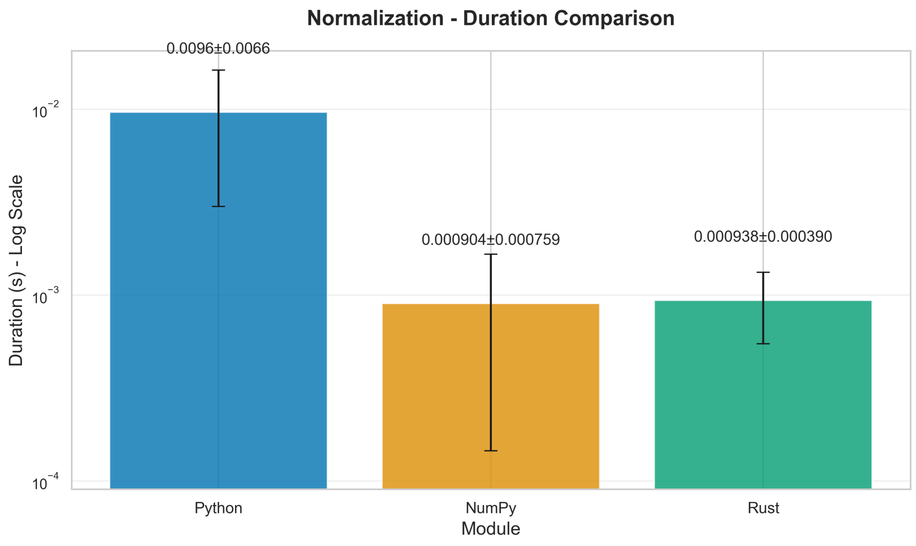


Figure 26: Total duration of normalizing on JapaneseVowels

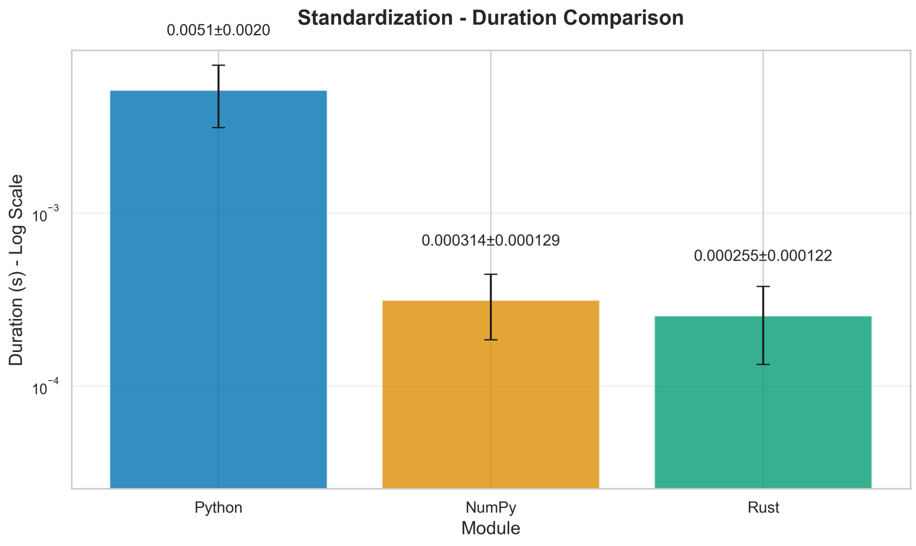


Figure 27: Total duration of standardizing on GunPoint

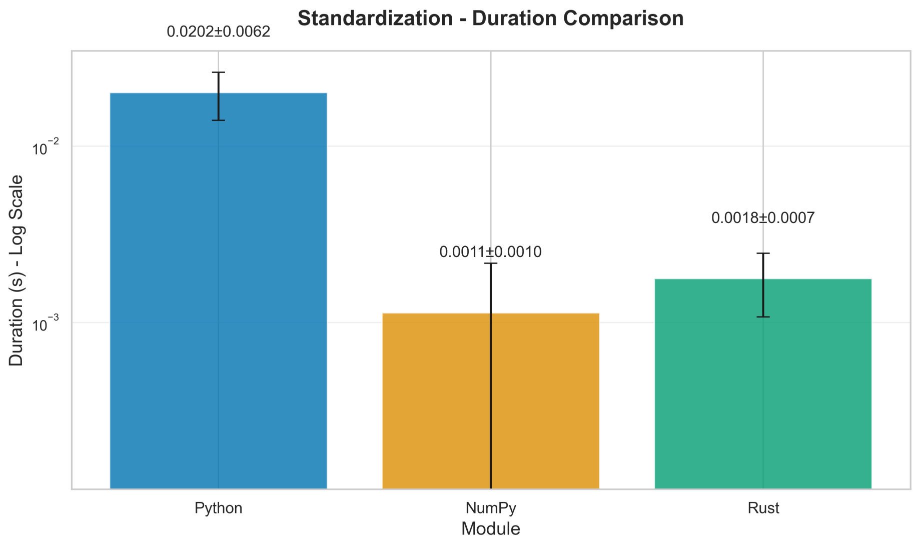


Figure 28: Total duration of standardizing on JapaneseVowels

The Rust implementation takes the longest by far for the downsampling operation, while Python and NumPy are comparable. This is because in our Rust implementation, we copy the data, which induces many heap operations and slows the code. See Figure 29 and Figure 30.

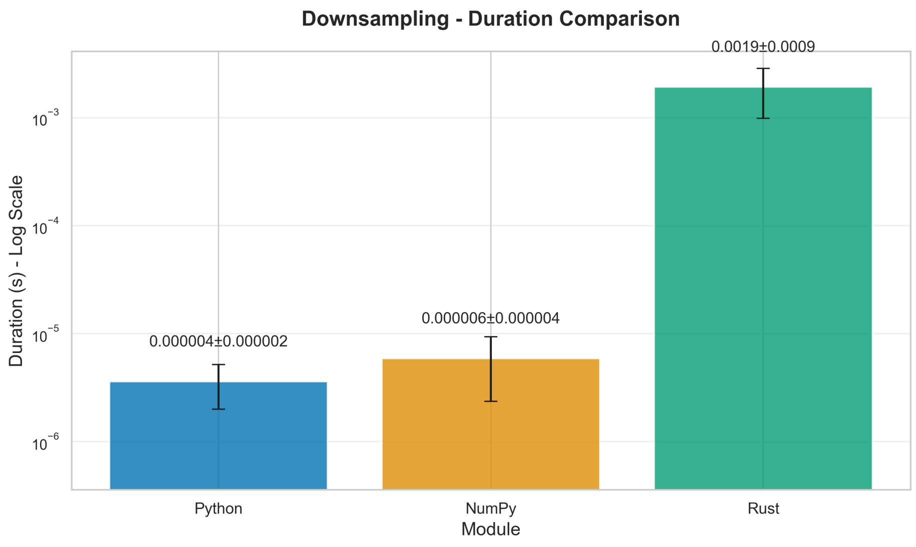


Figure 29: Total duration of downsampling on GunPoint

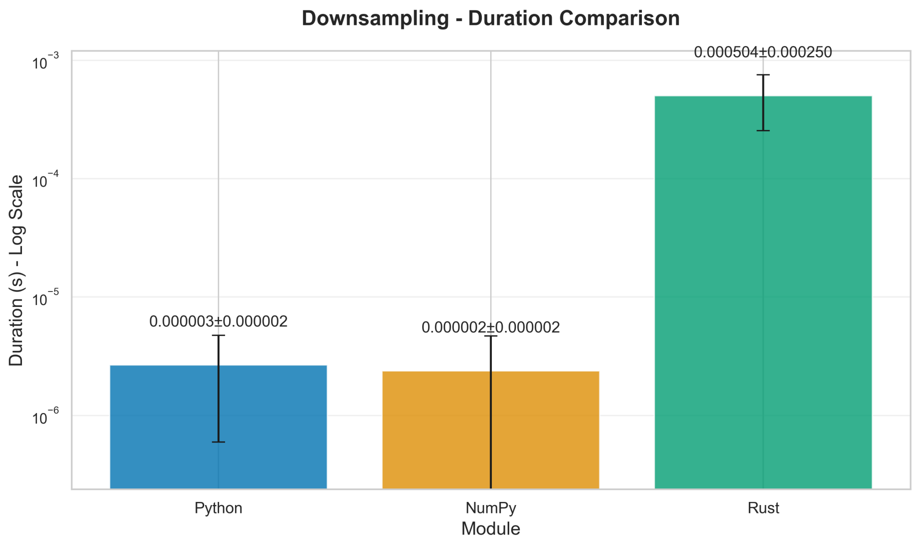


Figure 30: Total duration of downsampling on JapaneseVowels

The collection of data is the most time-consuming process in Rust. This is due to the fact that the Python and NumPy modules do not require any further action. However, the Rust module is required to transfer the data back to the Python module. However, the discrepancy is negligible. See Figure 31 and Figure 32.

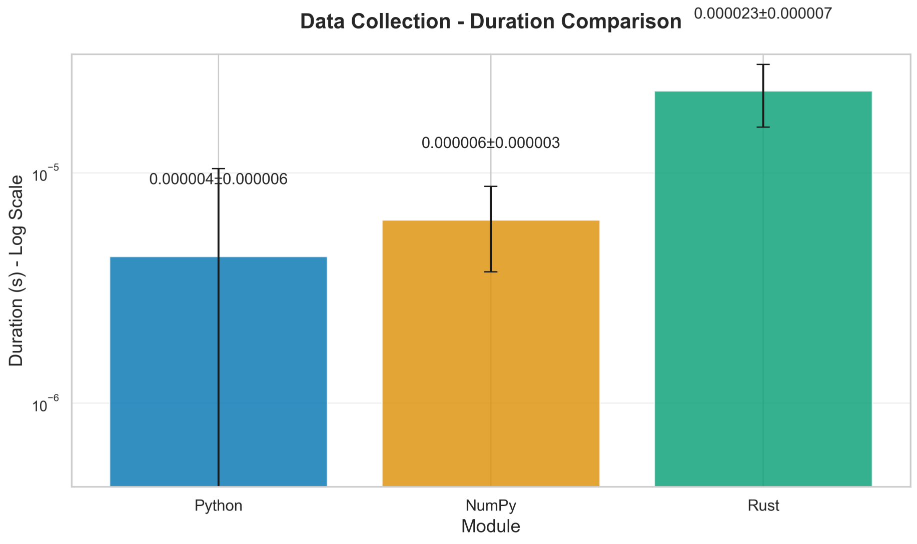


Figure 31: Total duration of collecting the data on GunPoint

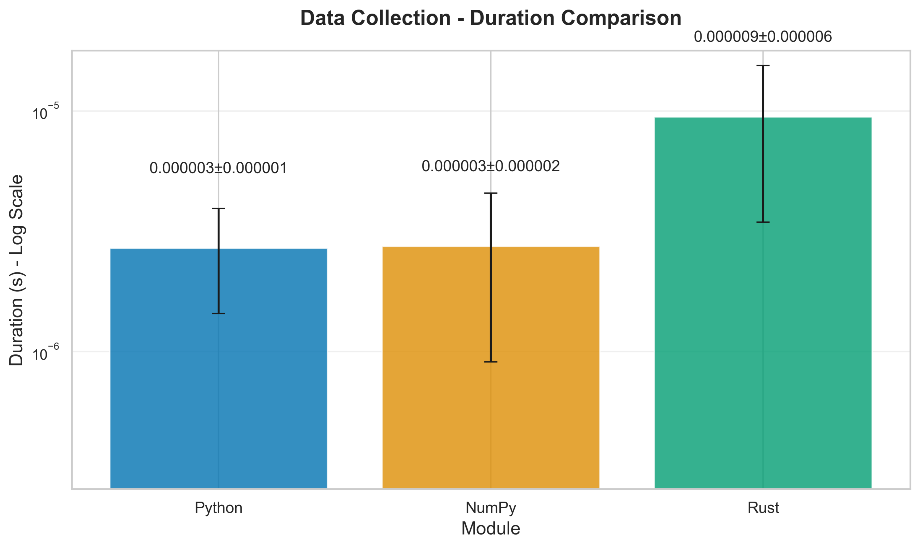


Figure 32: Total duration of collecting the data on JapaneseVowels

8.4 Memory footprint

Benchmarking also included a routine to measure peak memory usage while executing various methods. However, the reported peak memory usage is always 312 MB. This likely indicates a measurement error. Unfortunately, there wasn't enough time to fix it after noticing this.

9 AI usage acknowledgements

AI tools were used to get a better understanding of the project requirements. No code was directly generated by AI tools, but they were used to clarify concepts and provide explanations. They were also used to suggest design patterns and best practices. Also, AI tools were used to get suggestions on how to improve the code, which were then implemented, but not directly copied.

10 Authors' contributions

Marius Kaufmann

- Overall module and pipeline design
- Implementation of the Rust binding (`ForecastingDataSet` and `ClassificationDataSet` classes)
- Integration of passing data from Python to Rust by reference using the `numpy` crate
- Implementation of data splitting capabilities
- Implementation of the collection of data from Rust to Python
- Implementation of the `RustDataModule` class (PyTorch Lightning integration)
- Several code reviews and refactorings for teammates
- section 2 of this report

Amir Ali Aali

- Implementation of the downsampling capabilities
- Implementation of the normalization and standardization capabilities
- Implementation of the unit- and integration tests
- Several code reviews for teammates
- section 1, section 5, section 6, section 4, section 7 of this report

Kilian Fin Braun

- Implementation of the imputation capabilities
- Designing the return format for forecasting data (sliding windows)
- Naive Python implementation, NumPy implementation and PyTorch implementation to benchmark against
- Benchmarking of the module
- section 3, section 8 of this report

References

[Dev] PyO3 Developers. *pyo3::Py - Rust*. URL: https://docs.rs/pyo3/latest/pyo3/struct.Py.html (visited on 07/07/2025).

Zentrales Prüfungsamt/Central Examination Office

RWTH AACHEN UNIVERSITY

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Aali, Amir Ali	463040
Name, Vorname/Last Name, First Name	Matrikelnummer (freiwillige Angabe) Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

Rapid Time Series Datasets Library — Efficient AI with Rust Lab

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, 18.07.2025	[illegible]
Ort, Datum/City, Date	Unterschrift/Signature

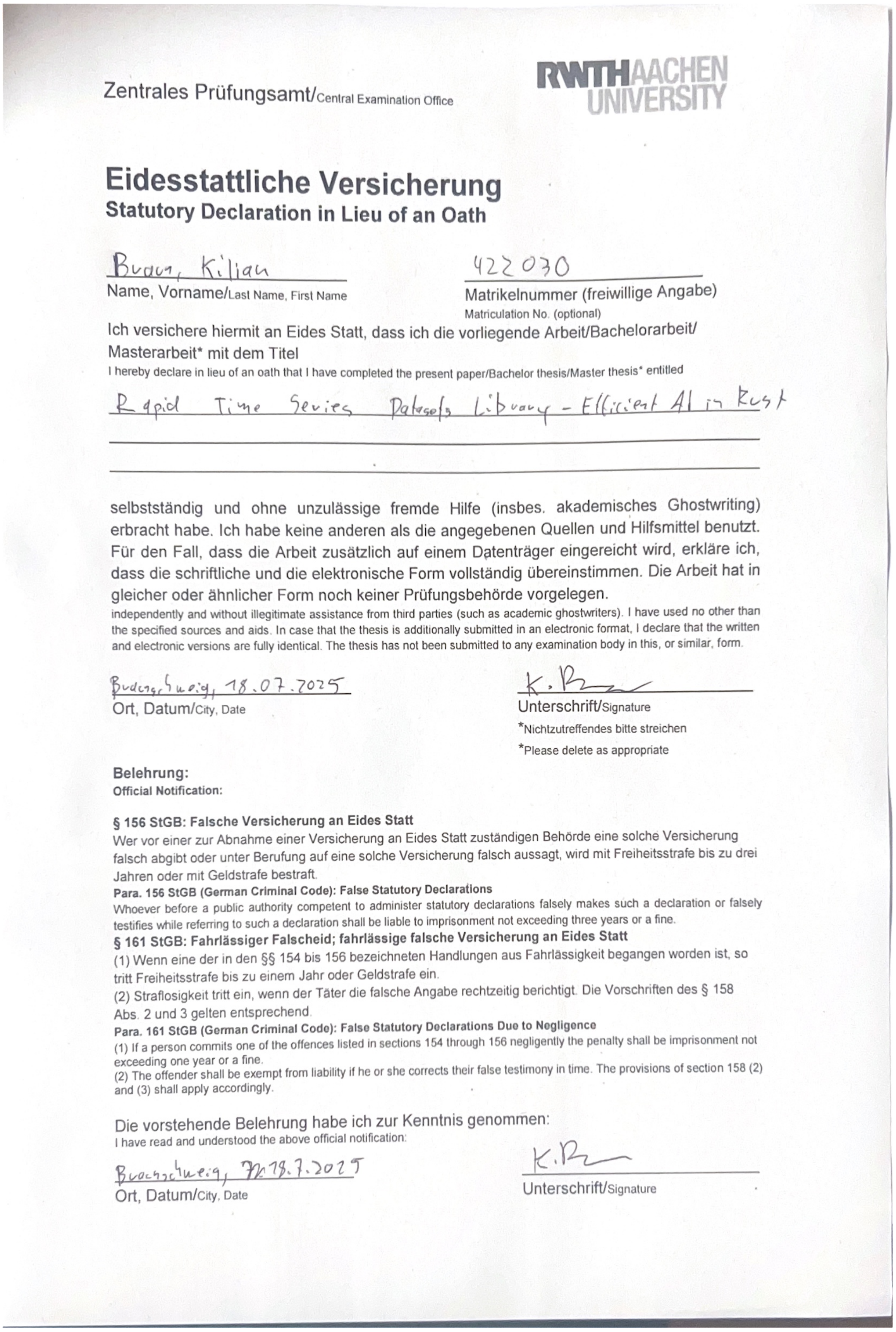
*Nichtzutreffendes bitte streichen
*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, 18.07.2025	[illegible]
Ort, Datum/City, Date	Unterschrift/Signature



Zentrales Prüfungsamt/Central Examination Office

RWTH AACHEN UNIVERSITY

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Braun, Kilian	422030
Name, Vorname/Last Name, First Name	Matrikelnummer (freiwillige Angabe) Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

Rapid Time Series Datasets Library - Efficient AI in Rust

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Braunschweig, 18.07.2025	[illegible]
Ort, Datum/City, Date	Unterschrift/Signature

*Nichtzutreffendes bitte streichen
*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Braunschweig, ~~7~~ 18.7.2025	[illegible]
Ort, Datum/City, Date	Unterschrift/Signature

Zentrales Prüfungsamt/Central Examination Office

RWTHAACHEN UNIVERSITY

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Kaufmann, Marius

Name, Vorname/Last Name, First Name

422046

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/~~Bachelorarbeit/~~ ~~Masterarbeit~~* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

Rapid Time Series Datasets Library —
Efficient AI in Rust

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, 18.07.2025

Ort, Datum/City, Date

[illegible]

Unterschrift/Signature

*Nichtzutreffendes bitte streichen
*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, 18.07.2025

Ort, Datum/City, Date

M. Kaufmann

Unterschrift/Signature