Synthetic Data Generator Module For Capping Devices

Ahmadreza F. Farahani
Davide Aiello

Contents

1 Introduction **3**

2 Technological Stack **3**
2.1 Front-end . . . 4
2.2 Back-end . . . 4
2.3 Database . . . 4
2.4 Additional Tools and Services . . . 4

3 MVC Pattern **5**
3.1 Application of MVC in the project . . . 5

4 Front End **6**
4.1 Introduction . . . 6
4.1.1 Configuration Context . . . 6
4.1.2 Configuration Class . . . 7
4.2 Generate Data page . . . 7
4.3 Search bar and sort button . . . 8
4.4 New Configuration and Load Configuration button . . . 9
4.5 Train Model Button . . . 9
4.6 Machineries Selection . . . 9
4.7 Sensors Selection . . . 10
4.8 Save Configuration . . . 10
4.9 Run and Stop Simulation buttons . . . 11

5 Back End **11**
5.1 API Documentation . . . 11
5.2 Business Logic . . . 12
5.3 Validation Process . . . 12
5.4 Threads syncronization . . . 13

6 Pandas **15**
6.1 Data Structures . . . 15

7 PyTorch **16**
7.1 Dynamic Computational Graph . . . 16
7.2 Tensors . . . 16
7.3 Neural Network Module . . . 16
7.4 Autograd . . . 16
7.5 Community Support . . . 16

8 PyTorch Forecasting **16**

9 Preprocessing **17**
9.1 Dataset . . . 17
9.2 Extract Class . . . 18
9.2.1 Raw data . . . 19
9.2.2 Reformatted raw data . . . 19
9.2.3 Preprocess reformatted data . . . 19
9.3 Sensor heads . . . 20
9.4 Correlation . . . 21

10 Temporal Fusion Transformer **24**
10.1 Background . . . 24
10.2 Transformers . . . 25
10.2.1 Self-Attention Mechanism . . . 25
10.2.2 Positional Encoding . . . 25
10.2.3 Sequential Processing . . . 25
10.2.4 Time Embeddings . . . 25
10.2.5 Conclusion . . . 26
10.3 Multi-horizon forecasting . . . 26
10.3.1 Architecture . . . 26
10.3.2 Quantile Loss . . . 27

11 Training **28**
11.1 Arguments and Hyperparameters . . . 28
11.2 Trainer Wrapper . . . 30
11.3 Results . . . 33

12 Evaluation **35**
12.1 Prediction by Validation . . . 35
12.2 Prediction by Variable . . . 35

13 Generation **36**
13.1 Prediction . . . 36
13.2 Database . . . 39

14 Conclusion **40**

15 Suggestions **41**

1 Introduction

This project extends Mario Deda's proof of concept for the bottling company Arol but serving as an independent entity as well. The main augmentation being the introduction of a distinctive feature named "Data Generation." This innovative feature empowers users to create a template, serving as a blueprint from which synthetic data for selected sensors and machineries can be systematically generated. This capability provides a powerful tool for users to simulate and manipulate data in a controlled environment, fostering enhanced flexibility and utility within the application.

AROL S.p.A. manufactures complex types of machinery that fill and cap bottles and jars and produces capsules. 95% of components and applications are designed in-house. World leader in its sector, it has offices in Turin and other Italian cities and the United States, Mexico, Brazil, China, and India.

In this project we develop a fully configurable and dynamic synthetic data generation tool that can generate artificial machinery sensor data. The tool's operational parameters must be utterly configurable from an interactive user interface. Sequential and parallel solutions are possible depending on the efficiency of the simulation and its accuracy (e.g., how many devices are simulated at the same time).

For this purpose, we use state-of-the-art machine learning models. All the models have been developed under PyTorch framework which we discuss briefly in this report.

2 Technological Stack

The project is a web application with both front-end and back-end components, connected to a database. The technological stack includes:

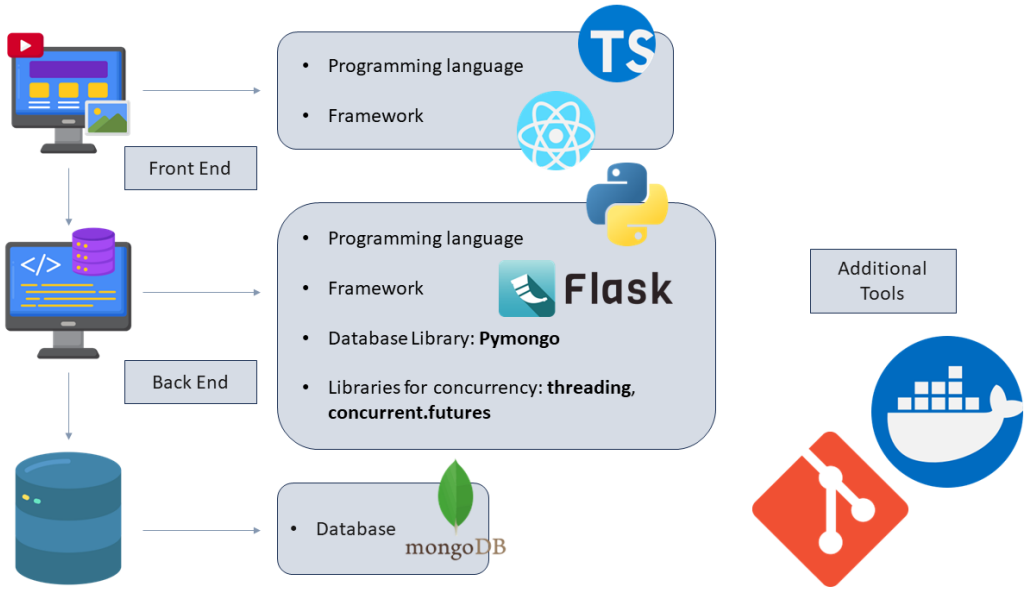


Figure 1: Technological Stack

2.1 Front-end

1. **Programming Language:** TypeScript

   A superset of JavaScript that adds static typing to the language, providing better tooling, autocompletion, and catching potential errors during development.

2. **Framework:** React

   A JavaScript library for building user interfaces. It enables the creation of reusable UI components, making it easier to develop dynamic and interactive web applications. The UI components used are provided by `@chakra-ui/react`

2.2 Back-end

1. **Programming Language:** Python

   A versatile programming language known for its readability and ease of use. It is widely used for web development, among other applications.

2. **Framework:** Flask

   A lightweight and flexible web micro-framework for Python, it offers essential tools and features for building web applications, making it suitable for small to medium-sized projects, much like our own.

3. **Database Library:** PyMongo

   The official Python library for MongoDB. PyMongo facilitates interaction with the MongoDB database from the back-end server written in Python. It allows for operations such as data insertion, updating, and retrieval in a MongoDB environment.

4. **Libraries for concurrent development:** threading, concurrent.futures

   Python's standard libraries used to support concurrent and multi-threaded code development, enabling parallel execution and managing concurrent access to shared resources.

2.3 Database

1. **Database:** MongoDB

   NoSQL document-oriented database providing scalability and flexibility for handling diverse data types.

2.4 Additional Tools and Services

1. **Dependency Management:**
   - `npm`: Package manager for the front-end.
   - `pip`: Package manager for the back-end.
2. **Code Versioning:**
   - `Git`: Distributed version control system used to track changes to the source code.
3. **Containerization:**

- `Docker:` Used for containerizing the application and ensuring consistency of the environment between development and testing.

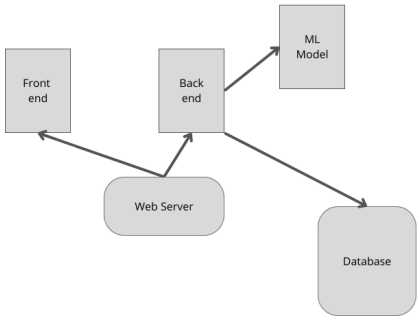


Figure 2: System design diagram. In this diagram, the front-end and back-end are hosted on the same web server

3 MVC Pattern

The Model-View-Controller (MVC) architectural pattern is a widely adopted design paradigm that divides an application into three interconnected components, each with a distinct role. The **Model** manages the data and business logic, the **View** handles the presentation and user interface, and the **Controller** orchestrates the flow of information between the Model and View, interpreting user inputs and triggering appropriate responses. This separation of concerns enhances modularity, maintainability, and scalability in software development.

3.1 Application of MVC in the project

In the application, the MVC pattern has been employed strategically to ensure a structured and efficient development approach. The **Model** is represented by the backend, particularly the interaction with the MongoDB database. It encapsulates the logic for managing configuration data, handling operations like insertion, retrieval, updating, and deletion of configurations.

On the **View** side, the frontend component provides a user-friendly interface, allowing users to seamlessly interact with the application. It encompasses features such as configuration creation, modification, retrieval, and deletion. The View component communicates with the backend, triggering actions related to configurations based on user interactions.

The **Controller** acts as an intermediary between the View and Model. It interprets user requests from the View, communicates with the Model to perform necessary operations, and updates the View accordingly. Additionally, the Controller manages the logic for training machine learning models, running simulations, and enforcing data validation rules.

4 Front End

4.1 Introduction

In order to ensure continuity with the original project, a decision was made to adopt the layout of the original application (sidebar and top bar) and introduced a new section (immediately visible from the sidebar and the first and only page of the entire application), called **Generate Data**. The UI components used are mostly provided by `@chakra-ui/react`.

Since this project is designed as an extension of the original application but capable of operating independently, a decision has been made to render other areas of the application inaccessible while remaining visible. This approach enables a future definitive integration with the previous project or keeping the two entities separate, possibly by completely removing sidebar and top bar.

A similar approach was taken for the top bar, where, since authorizations and permissions were beyond the scope of the project, generic names were used, and the ability to return to the login page or log out is denied.

In addition, given that access to machinery and sensor data in the original application required authentication for the Postgres database due to security reasons, an alternative approach was chosen. Rather than authenticating for database access, machinery and sensor data were retrieved from pre-saved JSON files stored in the working folders. This decision aligns with the original choice of not implementing any authentication mechanism.

Furthermore, this decision was also influenced by the fact that the sensor data exclusively pertained to the `JF891` machinery. In order to enhance flexibility for the extension to other machineries, the disentanglement from the Postgres database is exploited, and the availability of all sensors is extended for all machineries. However, since only the categories drive, eqtq, and plc were present, the ns section is omitted. Consequently, sensors related to that section will not be considered.

4.1.1 Configuration Context

Of notable importance is also referencing the application's context. In React, "context" is a mechanism that allows for data sharing between components in a component tree without the need to explicitly pass properties through each level. Context becomes invaluable when you have data that needs to be accessible by many components at various nesting levels, eliminating the need for "prop drilling" (passing props through many layers of components). In the case of this application, the context holds this interface:

```
interface ConfigurationContextProps {
    configuration: Configuration;
    setConfiguration: Dispatch<SetStateAction<Configuration>>;
    isSaved: boolean;
    setIsSaved: Dispatch<SetStateAction<boolean>>;
    toast : any
}
```

In particular:

- `configuration`: Object of type `Configuration`, the class used for configurations within the application.
- `setConfiguration`: Dispatch function to set the state of `configuration`.
- `isSaved`: Boolean indicating whether the configuration has been saved.

- `setIsSaved`: Dispatch function to set the state of `isSaved`.
- `toast`: Represents the toast object, a UI element commonly used to display brief and non-intrusive notifications or messages to users. These messages typically appear on the screen for a short duration, providing information such as the success of an action, receipt of a message, or the status of an operation

4.1.2 Configuration Class

These classes form the core of the software architecture, serving as the foundation for efficiently organizing and manipulating machinery configurations.

```
class Configuration {
    name: string
    machineriesSelected: MachinerySelected[]
}
```

At the forefront is the `Configuration class`, which encapsulates the entire configuration, holding a name and an array of selected machineries. Beyond instantiation, this class offers methods for streamlined configuration management. It provides efficient mechanisms to sort machineries, seamlessly add new machinery, and rename configurations.

```
class MachinerySelected {
    uid: string
    modelName: string
    sensorsSelected: SensorSelected[]
    faultFrequency: number
    faultProbability: number
}
```

Moving on, the `MachinerySelected class` represents individual machinery within a configuration, with attributes such as a unique identifier (`uid`), a model name, and details about fault frequency and probability. This class is equipped with methods to modify and retrieve information regarding the machinery's sensors and fault parameters.

```
class SensorSelected {
    name: string
    category: string
    heads: number[]
    dataFrequency: number
}
```

Within the `MachinerySelected class` is responsible for encapsulating information about sensors, including name, category, and data frequency. Methods within this class facilitate the addition, removal, and retrieval of sensor-related information, contributing to the dynamic nature of the configuration system.

4.2 Generate Data page

The page is designed with a set of components to ensure an optimal user experience. At the top, there is a search bar allowing users to search for machinery. Next to it, there is a dropdown menu for sorting machinery by identifier or model name.

Immediately below, you find the name of the configuration, labeled as `New Configuration` in the case of a new configuration. Three buttons follow: the first one to reset the current configuration, the second one to open the list of previously saved configurations and load one as

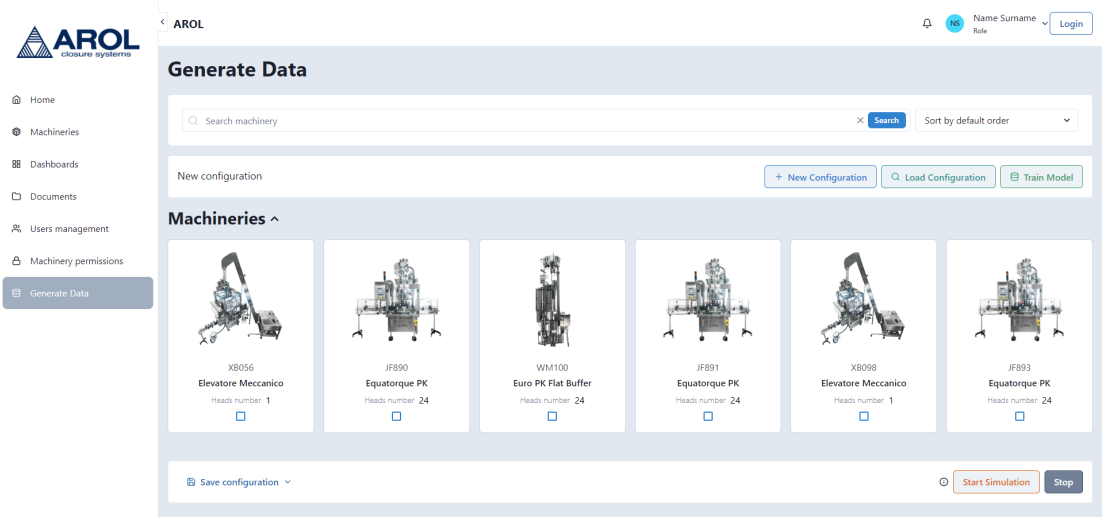


Figure 3: Generate Data page

the current configuration, and the third one to initiate the training of the Deep Learning model for data generation.

In the central section of the page, there is a toggle with transition that shows or hides the available machines that can be selected to populate the current configuration. If at least one machine is selected, another toggle with transition appears, providing the option to show the card for selecting specific sensors for that particular machine.

At the bottom, you'll find the menu related to saving. Pressing on it will open a menu that offers the option to save the current configuration with a name or to update a previously saved configuration. To the right, there is the button to start the simulation, which will generate data based on the selected machines and sensors. Next to it, there is the button to stop the current simulation.

To enhance the user experience, each action is followed by a notification message in the top-right corner, indicating information, warnings, successes, or errors, implemented through toast messages.

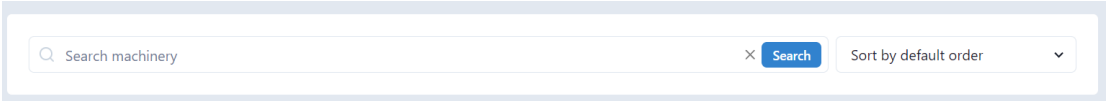


Figure 4: Search bar and sort button

4.3 Search bar and sort button

The integration of a search bar and sorting button as continuity elements, consistent with other sections of the original application, serves as tools for filtering or searching specific machinery. The decision is motivated by the fact that all machinery is accessible from a single page. Consequently, in the event of an increase in the number of machines, such tools have the potential to enhance the user experience. Notably, these tools also impact the sensor cards. In the scenario of selecting multiple machines, each machine would have a corresponding card in the **Sensors** section. Through interaction with the search bar and sorting button, it is possible to filter not

only the machines but also their respective sensor cards, thereby eliminating the need to search for a specific card, particularly in cases where the page is densely populated. For what the sort button is concerned, users have the flexibility to sort results by default order, Machinery ID, or Machinery Model.

4.4 New Configuration and Load Configuration button

As previously mentioned, by pressing the **New Configuration** button, the current configuration will be reset without any form of saving. For this reason, as soon as the current configuration undergoes a change, its name will become `Unsaved Configuration` (unless the configuration already has a name, in which case it will remain unchanged). Additionally, a tooltip will appear next to the **New Configuration** button, reminding the user that the current configuration is unsaved. When pressing the **Load Configuration** button, a scrollable modal will appear, presenting a list of previously saved configurations. Within this list, users can search for a specific configuration using the search bar, delete a saved configuration, or load one from the available options. If no configurations are saved, the message *No saved configuration* will be displayed.

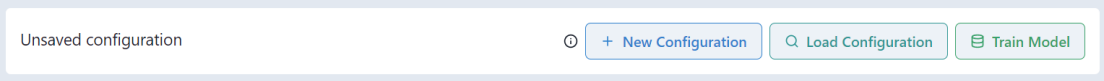
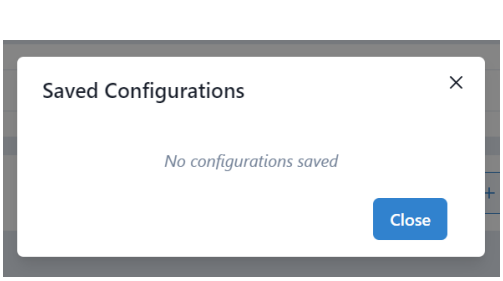


Figure 5: Top buttons



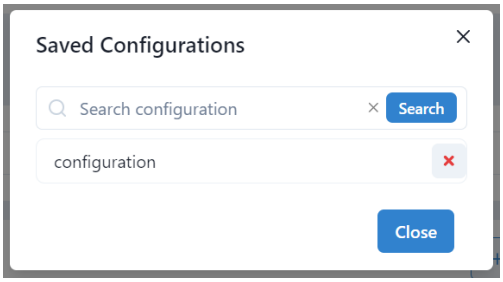


Figure 6: Load Configuration menu

4.5 Train Model Button

Through the user interface, the user has the option to train the model by clicking on the dedicated button. Since the practice of training the model is not within the expertise of the average user, it has been decided to keep the training-related parameters within a configuration file in the backend directory. In the event that a pre-trained model already exists on the server, the user will be prompted to decide whether to retrain the model (following any modification to the configuration parameters file) or to proceed and use the current version of the model available on the server.

4.6 Machineries Selection

Through this section, it is possible to select, via the respective checkbox, the machinery that you want to add to the current configuration. Upon selecting one of these, the **Sensors** section will appear.

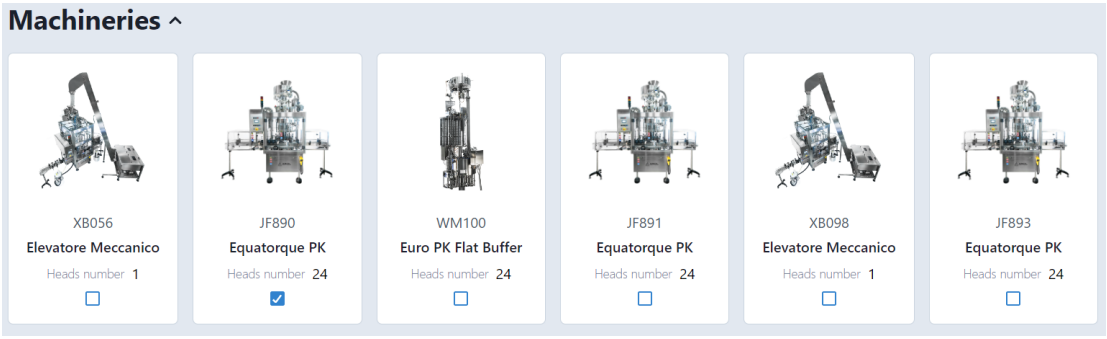


Figure 7: Machieries selection

4.7 Sensors Selection

With the aim of enhancing the user experience, this section is visible only if at least one machine is selected, minimizing on-screen elements to only the necessary components. By pressing the toggle, similar to the machinery section, the sensor cards, each related to a selected machine, will smoothly transition onto the screen. Through this, it is possible to choose the sensor category, and for each category, the specific sensor. For each sensor, you can also specify the number of heads, accessible through an additional toggle. Sensor selection is only possible if the data frequency for the specific sensor has been previously entered.

At the bottom, it is also possible to enter both the frequency and probability of a fault, and this choice can be made for each machine. The time unit for data and fault generation is the second.

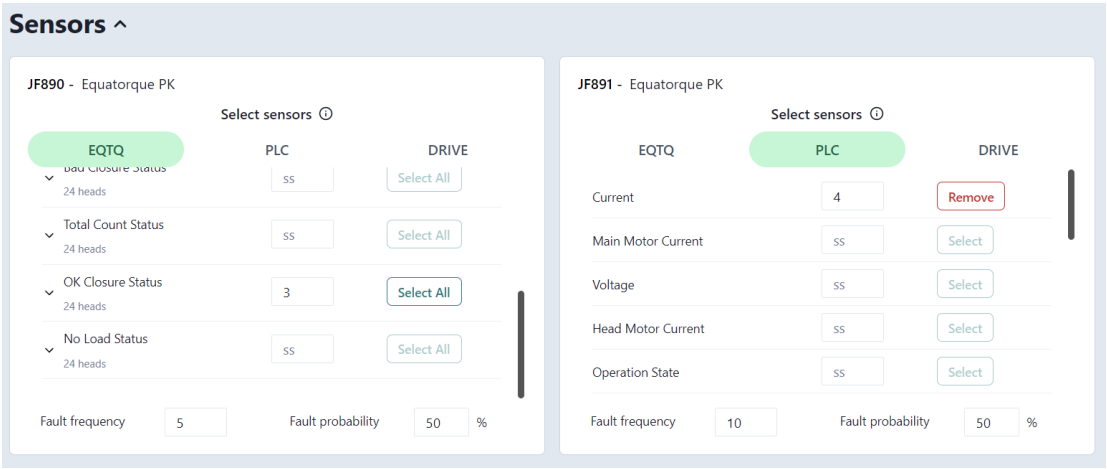


Figure 8: Sensors selection

4.8 Save Configuration

The **Save Configuration** menu at the bottom of the page offers two options: Save and Save As. Since the concept behind loading configurations is that users can immediately utilize a saved configuration, the decision has been made to allow saving only a configuration that is fully filled out: at least one machine with its respective fault frequency and fault probability, and at least one sensor with its corresponding data frequency.

The button will not be clickable until at least one machine and one sensor have been selected. After that, for any missing fields, a warning message will appear, guiding the user through the configuration settings. If a configuration has never been saved, it will be possible to save it by providing a name through a modal. If a configuration with the same name already exists, it will be possible to overwrite the existing saved configuration.

Once the configuration is saved, future modifications can be saved using **Save** or a copy can be saved with a different name using **Save As**.

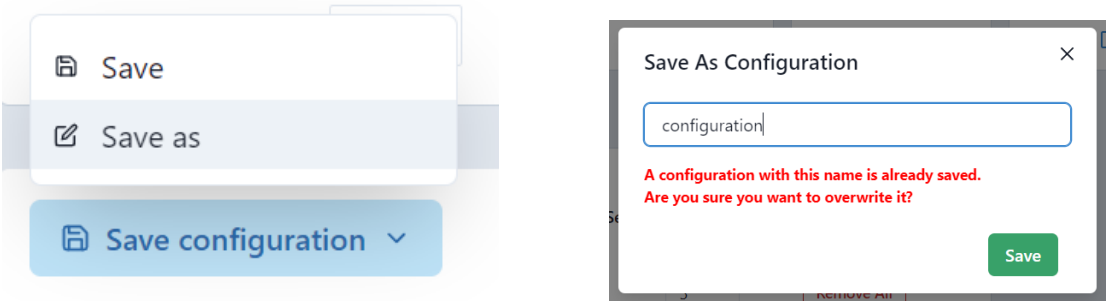


Figure 9: Save Configuration menu

4.9 Run and Stop Simulation buttons

The **Run Simulation** button will be orange upon loading the page. In this case, clicking on the button will display notification messages guiding the user on how to complete a configuration to run a simulation. The required conditions include: having a trained model, selecting at least one machine with fault frequency and probability, and choosing at least one sensor with data frequency.

Once all the requirements are met, the button's color will change from orange to green, indicating that it is possible to launch a simulation. During the simulation, you can observe the passage of seconds and the generation of data related to machinery and sensors in the terminal.

While a simulation is running, the **Stop** button will change from gray to red and become enabled, but almost every other interaction in the user interface is disabled. Clicking it will stop the current simulation. After stopping the simulation, it will be possible to view statistics related to the current simulation: configuration name, generated samples, generated faults, and tue number of involved machinery.

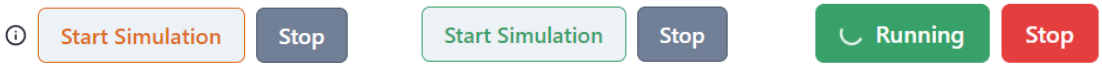


Figure 10: Aspects of the Run Simulation button

5 Back End

5.1 API Documentation

The Api documentation is available on the README.md file on the GitHhub repository.

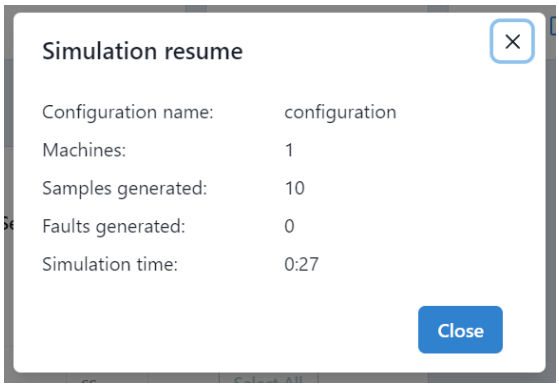


Figure 11: Simluation statistics

5.2 Business Logic

1. **Insert Configuration:** Receive configuration data from the user interface. Validate the input data, ensuring it includes a valid name and a list of selected machinery with associated sensors and parameters.
2. **Update Configuration:** Accept configuration data and a specific configuration name from the user interface. Verify that the name is a valid string. Validate the input machinery data and update the corresponding configuration in the backend by modifying the selected machineries.
3. **Retrieve Configuration:** Receive a configuration name from the user interface. Verify the name's validity, then query the MongoDB database to retrieve the configuration information. Return the configuration details to the user interface.
4. **Retrieve Configuration Names:** Fetch all configurations from the database and extract their names. Provide the list of configuration names to the user interface.
5. **Delete Configuration:** Accept a configuration name from the user interface. Validate the name, then delete the corresponding configuration from the MongoDB database. Return a success message or an error if the configuration is not found.
6. **Check for Trained Model Existence:** Verify whether a machine learning model has already been trained by checking for the presence of the "checkpoints" directory. Communicate the status (trained or not trained) to the user interface.
7. **Train Machine Learning Model:** Trigger the training process for the machine learning model. Provide a success message upon completion or an error message if any issues arise during training.
8. **Run Simulation:** Receive configuration data for machinery and sensors. Validate the input data, initialize a thread pool for concurrent data generation, and execute the simulation. Return the simulation results, including the number of samples generated, faults, and simulation time, to the user interface.
9. **Stop Simulation:** Check if a simulation is in progress. If so, stop the ongoing simulation. Communicate the status of the interruption to the user interface.

5.3 Validation Process

- **Check for Data Existence:**

Ensure the presence of received data before proceeding with the validation process. If no data is received, return an error response indicating the absence of data.

- **Configuration Validation:**
  - Verify the existence and correct data type of essential fields for configuration data, including `'name'` (`str`) and `'machineriesSelected'` (`list`).
  - Ensure that `'machineriesSelected'` contains at least one machinery item, and the data type is a `list`.
- **Machinery Validation:**

  For each machinery item in `'machineriesSelected'`:
  - Validate the presence and data type of mandatory fields, such as `'sensorsSelected'` (`list`), `'uid'` (`str`), `'modelName'` (`str`), `'faultFrequency'` (`int`), and `'faultProbability'` (`int`).
  - Confirm that `'sensorsSelected'` is not empty for each machinery.
  - Check that `'faultProbability'` is within the valid range of 0 to 100.
  - Ensure `'faultFrequency'` is greater than zero.
- **Sensor Validation:**

  For each machinery item in `'machineriesSelected'`:
  - Validate mandatory fields: `'name'` (`str`), `'category'` (`str`), `'dataFrequency'` (`int`), and `'heads'` (`list`).
  - Confirm that `'heads'` is not empty.
  - Ensure `'dataFrequency'` is greater than zero.
  - For each element in `'heads'`, validate that it is of type `int`.
- **Validation Result:**

  Return a boolean indicating the overall validation result, along with potential error details. This process ensures that configuration data adheres to specified criteria, promoting data integrity and preventing issues during subsequent operations.

5.4 Threads syncronization

For the data generation and loading into the database, a multithreading approach has been chosen. This section will solely address the management of concurrency among threads, omitting details about data generation and how it is inserted into the database.

To fulfill the requirements of concurrently generating data for multiple machines, we opted for a thread pool architecture. This pool comprises threads operating at the sensor level, complemented by an additional main thread responsible for tracking simulation time and dispatching tasks to the other threads. In other words, one thread was generated for each sensor, unless the number of sensors exceeded the number of logical threads supported by the CPU. In such instances, the number of threads in the pool would be capped at the maximum value of logic threads supported by the CPU, minus one to account for the main thread.

As the requirements took fault generation into consideration, the chosen approach revolves around the fault frequency and probability. Since faults are defined at the machinery level, the idea was to trigger a fault for the sensor data immediately following a fault occurrence. A fault happens when the elapsed time is a multiple of the fault frequency, and a randomly generated value is less than the user-provided probability. In other words, when a fault occurs, all data for the next sensor associated with that machinery will be out of scale. This applies to all selected heads values. In cases where multiple sensors share the same data frequency, the sensor producing out-of-scale values will be randomly chosen among them.

To achieve the goal, a Python data structure (defined as a class) was used as if it were the classic shared data structure employed by threads in C.

```
class Thread_struct:
    stop_generation = threading.Event()
    thread_pool = ThreadPoolExecutor()
    num_samples_generated = 0
    num_faults_generated = 0
    lock = Lock()
    mongo = PyMongo()
```

This structure consists of:

- `threading.Event()` object that will be used as a communication mechanism to signal the main thread to interrupt the simulation.
- `thread_pool` is an object of the `ThreadPoolExecutor` class representing the thread pool.
- `num_samples_generated` represents the number of generated samples (excluding faults).
- `num_faults_generated` is the number of generated faults.
- `lock` is a class from the `threading` module in Python. It provides a way to synchronize access to shared resources by multiple threads.
- `mongo` is an instance of the PyMongo class, typically used in Flask applications for interacting with MongoDB databases.

The main thread, the one that received the command to launch the simulation from the user through the endpoint, will be responsible for tracking the elapsed time since the start of the simulation. Before that, it initializes the dictionary `machineryFault` with machine identifiers as keys and 0 as values. This dictionary will be used variable for fault generation and management.

When the elapsed time becomes a multiple of the fault frequency, and the randomly generated number is less than the user-provided probability, a fault occurs. The main thread will then update the value associated with the key of the machine that triggered the fault to 1 in the `machineryFault` dictionary.

For each sensor, when the current simulation time is a multiple of the sensor's data frequency, a task is enqueued in the thread pool's execution queue. This task includes the function to be executed, with a Python dictionary as a parameter containing the machine's name, the internal sensor data structure, and a value indicating whether a fault should be generated or not.

Each thread in the thread pool checks for fault generation, generates data, loads it into the appropriate format and collection based on machinery and sensor. Subsequently, it acquires the lock to update either the `num_samples_generated` or `num_faults_generated` value, then releases the lock. After completing the task, the thread returns to the pool

It's important to note that the **Pymongo** library inherently handles thread safety, eliminating the need for additional locks during database operations. This built-in thread safety allows

multiple threads to interact with MongoDB concurrently without risking data inconsistencies. Consequently, the thread efficiently loads data into the correct format and collection within MongoDB, leveraging Pymongo's internal mechanisms for managing concurrent access without relying on the lock used for the other shared data.

Throughout the simulation, this process continues, hinging on the check `Thread_struct.stop_generation.is_set()`, until the `Thread_struct.stop_generation` object is set. This object, an instance of the Event class from the threading library, functions as a communication mechanism between threads and is set by a thread upon pressing the stop button on the graphical interface. Upon exiting the main loop, `Thread_struct.thread_pool_executor.shutdown()` will be called to gracefully terminate the `ThreadPoolExecutor`. The thread pool will not accept additional tasks but will finish processing those still in the queue. The event flag will be cleared using `Thread_struct.stop_ generation.clear()`, and simulation statistics will be sent back to the front end in JSON format.

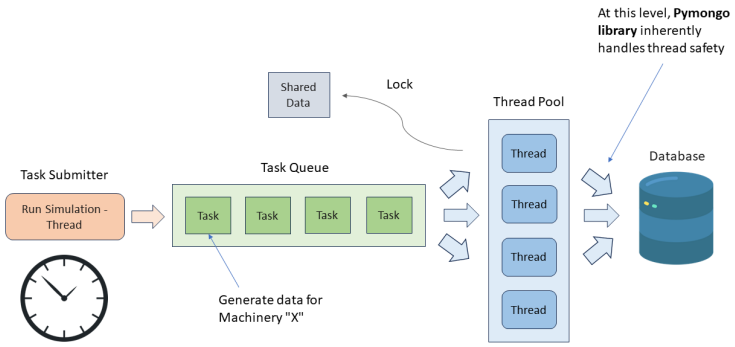


Figure 12: Threads synchronization schema

6 Pandas

Pandas [1] is a powerful data manipulation and analysis library for Python. It provides data structures for efficiently storing and manipulating large datasets. Here, we provide a brief overview of some key aspects of the pandas framework.

6.1 Data Structures

Pandas introduces two primary data structures: Series and DataFrame.

- **DataFrames**: A DataFrame is a two-dimensional, tabular data structure with labeled axes (rows and columns). It is the most commonly used pandas object.
- **Series**: In pandas, a Series is a one-dimensional labeled array holding data of the same type. It has an index for each element. You can create a Series from a list or dictionary, and it supports various operations like indexing, slicing, and element-wise operations. It is a fundamental data structure often used to represent a column in a table or a single vector of data.

This is just a brief introduction to the pandas framework. For more details and advanced usage, refer to the official documentation [2].

7 PyTorch

PyTorch [3] is an an open-source machine learning framework. PyTorch is widely used for developing and training deep learning models. It is known for its flexibility, dynamic computation graph, and a user-friendly interface, making it popular among researchers and practitioners in the machine learning community. We briefly explain key features of PyTorch.

7.1 Dynamic Computational Graph

PyTorch uses a dynamic computation graph, which allows for more flexibility during model training and development. This is in contrast to static computation graphs used by some other frameworks.

7.2 Tensors

PyTorch uses tensors as the fundamental building blocks for creating and manipulating data. Tensors are similar to NumPy arrays but with additional capabilities, and they can be easily moved between CPU and GPU for efficient computation.

7.3 Neural Network Module

PyTorch provides a *torch.nn* module that offers a collection of tools and modules for building and training neural networks. It includes various pre-defined layers, loss functions, and optimization algorithms.

7.4 Autograd

PyTorch includes automatic differentiation through its Autograd system. This enables automatic computation of gradients during the backward pass, simplifying the process of training neural networks.

7.5 Community Support

PyTorch has a vibrant and active community, which contributes to its growth and improvement. This community support includes documentation, tutorials, and a wealth of pre-trained models that can be leveraged for various tasks. PyTorch is used in a wide range of applications, including natural language processing, computer vision, and reinforcement learning. It is developed and maintained by Facebook's AI Research lab (FAIR), but it is an open-source project with contributions from researchers and developers worldwide [4].

8 PyTorch Forecasting

PyTorch Forecasting aims to ease state-of-the-art timeseries forecasting with neural networks for both real-world cases and research alike. The goal is to provide a high-level API with maximum flexibility for professionals and reasonable defaults for beginners. Specifically, the package provides

- A timeseries dataset class which abstracts handling variable transformations, missing values, randomized subsampling, multiple history lengths, etc.
- A base model class which provides basic training of timeseries models along with logging in tensorboard and generic visualizations such actual vs predictions and dependency plots

- Multiple neural network architectures for timeseries forecasting that have been enhanced for real-world deployment and come with in-built interpretation capabilities

For a large number of additional machine learning applications time is of the essence: predictive maintenance, risk scoring, fraud detection, etc. The order of events and time between them is crucial to create a reliable forecast.

9 Preprocessing

The goal of preprocessing is to prepare the data in a format that is suitable for the chosen algorithm, enhance the quality of the input data, and improve the overall performance of the model. The specific preprocessing steps depend on the nature of the data and the requirements of the task at hand.

9.1 Dataset

We have three categories for each machinery in the dataset. The original data have been stored to the MongoDB [5, 6].

```
@dataclass
class EQTQ(BaseSensorData):
    all_sensors: list = field(default_factory=lambda: ['LockDegree', 'Index',
                    'stsClosureOK', 'stsNoLoad',
                    'MaxLockPosition','stsBadClosure',
                    'AverageTorque', 'stsTotalCount',
                    'MinLockPosition', 'stsNoClosure', 'AverageFriction'])
```

First, we extracted the data from original recorded dataset and in the future steps we try to pre-process the data for the training purpose.

One key goal in this process, is to capture modularity in developing the codes. All the written python scripts are well documented and commented.

Moreover, we wrote classes in python which it can be used further in the main application.

```
@dataclass
class DRIVE(BaseSensorData):
    all_sensors: list = field(default_factory=lambda: ['Tcpu','Twindings'
                                                ,'Tboard','Tplate'])
```

Furthermore, we have designed a dataclass [7] object for each category that will be used in further processing.

```
@dataclass
class PLC(BaseSensorData):
    all_sensors: list = field(default_factory=lambda: ['OperationMode', 'Alarm',
                    'OperationState', 'MainMotorSpeed',
                    'MainMotorCurrent','HeadMotorSpeed',
                    'HeadMotorCurrent', 'ProdSpeed',
                    'PowerVoltage', 'PowerCurrent', 'AirConsumption',
                    'LubeLevel', 'test1', 'test2', 'test3', 'TotalProduct'])

    def set_unk_variables(self):
        self.unk_variables = ['value']
```

All these classes inherited from *BaseSensorData*() which sets different methods and characteristics of the behaviour of each category. All these classes will be used to extract data for training and generating new data.

9.2 Extract Class

We have developed a class for extracting raw data from MongoDB. In this section, we categorize each method in this object and explain how it can be utilized to transform raw data into a format suitable for training in the model.

```
class Extract:
    """
    This class have been designed to extract and transform raw data
    into a new space for future analysis.


    Methods:
        get_raw_data():
            This method extracts raw data from MongoD json files available in
            ./data folder.
        extract_raw_data():
            This is the first step to get the data based on heads and different
            sensors.
        preprocess_data(extr_df):
            Preprocesses the data to from original timestamps.
        fill_data(df, det_sensors):
            Fill the NaN values in the DataFrame.
    """
    def __init__(
            self,
            sensors: list,
            category: str,
            machinery: str,
            show_fig: bool =False,
            save_fig: bool = True,
            figs_dir: str ='./figs',
            plt_style: str = 'Solarize_Light2',
            data_path: str = './data',
            verbose: bool = True
            ):

        """
        Initialize a new instance of Extract.

        Args:
            sensors (list): list of all sensors we want to analyze
            category (str): name of category ['eqtq', 'drive']
            machinery (str): name of machinery
            show_fig (bool): whether to show the figs while extracting
            save_fig (bool): whether to save the figs while extracting
            figs_dir (str): figs directory.
            plt_style (str): plot styles.
```

```
    data_path (str): original data path of the sensors.
    verbose (bool): whether to print any information regarding extraction.
"""
```

9.2.1 Raw data

The initial method in the *Extract*() class is *get_raw_data*(). This method is responsible for extracting raw sensor data from various sensor categories.

In Table 1, a sample from the *EQTQ* category is presented, showcasing unstructured data in MongoDB format. Subsequent steps involve reformatting the table using the methods provided within the *Extract*() class.

	_id	folder	samples
0	{'$oid': '62c2f77c56ebca966f9f27c2'}	Head_01	[{'name': 'H01_Index', 'value': {'$numberInt':...
1	{'$oid': '62c2f77c56ebca966f9f27c6'}	Head_02	[{'name': 'H02_MaxLockPosition', 'value': {'$n...
2	{'$oid': '62c2f77c56ebca966f9f27c9'}	Head_03	[{'name': 'H03_MinLockPosition', 'value': {'$n...
3	{'$oid': '62c2f77c56ebca966f9f27d5'}	Head_04	[{'name': 'H04_LockDegree', 'value': {'$number...
4	{'$oid': '62c2f77c56ebca966f9f27df'}	Head_05	[{'name': 'H05_Index', 'value': {'$numberInt':...

Table 1: Raw data

Within the Pandas *DataFrame* framework, the entire dataset is currently in a unified format, incorporating all sensors and headers together.

However, our objective is to segregate different headers and sensors, enabling individual training for each.

9.2.2 Reformatted raw data

Now we use second method which is *extract_raw_data*(). Now we get a better structured *DataFrame* that can be used for further processing.

In the table. 2 we have *AvergaeFriction* sensor in *EQTQ* category for four different heads. Moreover, we changed the timstamps to date time for better visualization.

	Head_09	Head_10	Head_11	Head_12
2022-07-04 14:21:42.474	NaN	NaN	NaN	NaN
2022-07-04 14:21:42.600	-294.80	NaN	NaN	NaN
2022-07-04 14:21:42.664	NaN	-295.75	NaN	NaN
2022-07-04 14:21:42.788	NaN	NaN	-354.62	NaN
2022-07-04 14:21:42.875	NaN	NaN	NaN	-291.31

Table 2: Reforming raw data

The current format is satisfactory for preprocessing; however, it contains NaN values and lacks a comprehensive representation suitable for future data analysis. Consequently, we employ alternative methods to preprocess the existing format.

9.2.3 Preprocess reformatted data

Using the *preprocess_data*() and *fill_data*() methods, we conduct preprocessing on the reformatted data for utilization in the model training phase. Table 3 provides a glimpse of the final preprocessed data. Additionally, we have isolated deterministic sensors within this dataset.

Deterministic sensors are those with either a constant value or periodic constants (e.g., the *Index* sensor denoting the index of each head in each machinery) that do not require further training.

	Head_09	Head_10	Head_11	Head_12
2022-07-04 14:21:44.690	-285.29	-297.51	-353.96	-289.95
2022-07-04 14:22:39.404	-284.86	-297.59	-353.93	-289.88
2022-07-04 14:22:39.698	-284.42	-297.68	-353.90	-289.81
2022-07-04 14:22:40.288	-283.99	-297.76	-353.86	-289.74
2022-07-04 14:22:40.459	-283.56	-297.85	-353.83	-289.67

Table 3: Preprocessed data

As observed, all NaN values have been successfully filled. To accomplish this, we employed interpolation, a technique used to estimate or fill in missing values in a dataset by approximating them based on the known values in the dataset.

In the context of filling NaN (Not a Number) values, interpolation entails estimating the missing values using the present values in the dataset.

For this process, we specifically utilized linear interpolation to fill the dataset.

The linear interpolation formula is given by:

$$y = y_1 + \frac{(x - x_1) \cdot (y_2 - y_1)}{x_2 - x_1}$$

where:

x is the target point,
x_1, y_1 are the coordinates of the first known point,
x_2, y_2 are the coordinates of the second known point.

9.3 Sensor heads

We have designed a method which can visualize the original data distribution over different heads in different sensors with different categories.

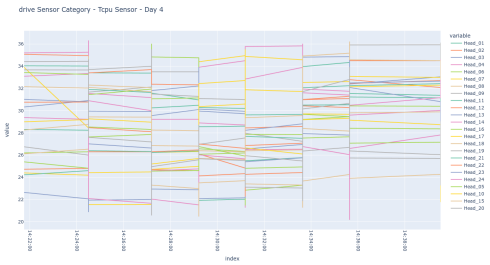
```
def plot_heads(df, sensor, category, colors, show_fig,
               save_folder='./figs/data') -> None:
    """
    Plot all heads sensor data based on the sensor in each category. All the
    plots should be based on the output of filled_data() method in Extract class.

    Args:
        df (pd.DataFrame): extracted dataframe for a specific sensor.
        sensor (str): sensor name.
        category (str): sensor category.
        colors (list): colors for different heads.
        show_fig (bool): show figures while running the function.
        save_folder (str): folder to save the plots.

    Returns:
```

None
"""

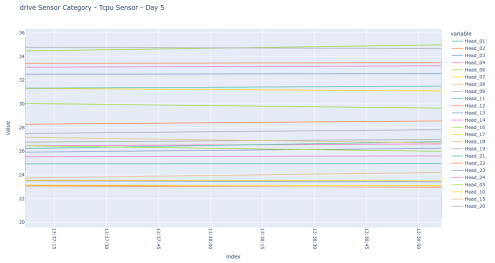
The original data exhibits varying distributions on different recording days.

Our objective is to categorize the data for each day, taking into account the fluctuating frequency of captured data on different days, a factor crucial for consideration during training.

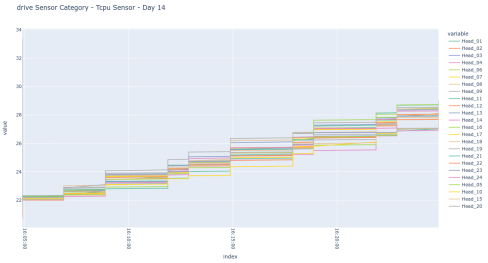
Conversely, categories like *PLC* lack distinct heads. Nevertheless, with the implementation of this function, we can visualize single-value columns within this category or in other potential categories that may contain a single head in the future.



(a) Cpu Temperature day 4



(b) Cpu Temperature day 5



(c) Cpu Temperature day 14

Figure 13: Data distribution of different heads in *Drive* category and *Tcpu* sensor

In Fig. 13, the figures depict the distribution within the *Drive* category. As previously discussed, the distribution varies for each day. To further explore the presence of this pattern, we now move on to examining other categories.

We see that the different distribution pattern still exists in *EQTQ* category. We can also see the correlation of the different heads which we try to discover in the future section.

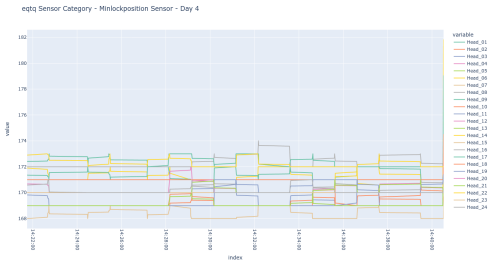
Additionally, we observe that even single-value sensors in the *PLC* category exhibit distinct distributions.

During the training phase, we take into account the days as one of the categorical features in the generated data.

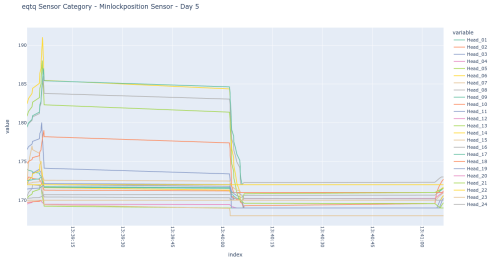
This consideration aims to enhance the performance and accuracy of predicting future samples by incorporating the variations observed across different recording days.

9.4 Correlation

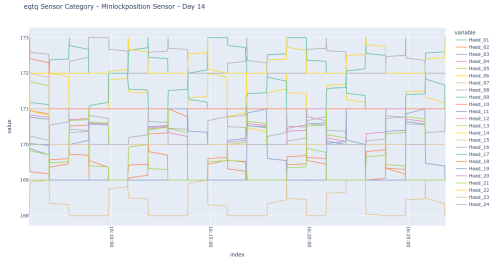
In statistics, a correlation matrix is a table showing correlation coefficients between variables. Each cell in the table represents the correlation between two variables, often denoted as ρ or r.



(a) MinLockPosition day 4

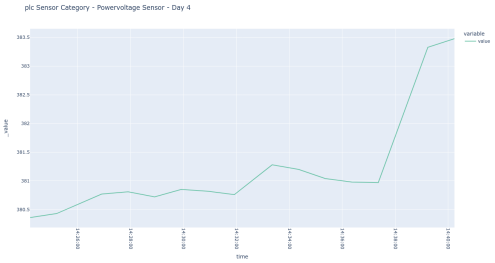


(b) MinLockPosition day 5

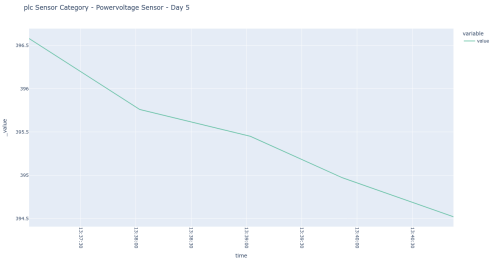


(c) MinLockPosition day 14

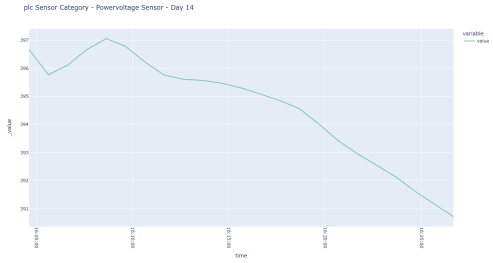
Figure 14: Data distribution of different heads in $EQTQ$ category and $MinLockPosition$ sensor



(a) PowerVoltage day 4



(b) PowerVoltage day 5



(c) PowerVoltage day 14

Figure 15: Data distribution of different heads in PLC category and $PowerVoltage$ sensor

Let's consider a correlation matrix R for n variables:

$$R = \begin{bmatrix} 1 & r_{12} & r_{13} & \dots & r_{1n} \\ r_{21} & 1 & r_{23} & \dots & r_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & r_{n3} & \dots & 1 \end{bmatrix}$$

In this matrix:

- The diagonal elements (e.g., r_{11}, r_{22}, r_{33}, ...) represent the correlation of each variable with itself, which is always 1.
- Off-diagonal elements (e.g., r_{12}, r_{23}, r_{34}, ...) represent the correlation between different pairs of variables.

The correlation coefficient r ranges from -1 to 1:

- $r = 1$: Perfect positive correlation.
- $r = -1$: Perfect negative correlation.
- $r = 0$: No correlation.

The correlation matrix is symmetric because the correlation between variable i and j is the same as the correlation between variable j and i.

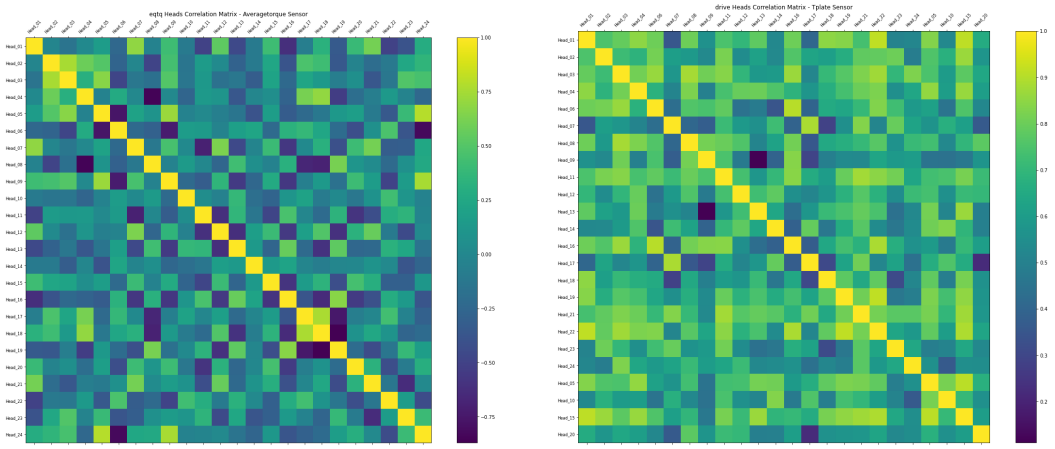
We have developed a function in listing below to capture the correlation matrix between different heads of each sensor belonging to a specific category.

```
def plot_correlation(df, sensor, category, show_fig,
                     save_folder='./figs/correlation') -> None:
    """
    Plot correlations of different heads in each category. All the
    plots should be based on the output of filled_data() method in Extract
    class.

    Args:
        df (pd.DataFrame): extracted dataframe for a specific sensor.
        sensor (str): sensor name.
        category (str): sensor category.
        show_fig (bool): show figures while running the function.
        save_folder (str): folder to save the plots.

    Returns:
        None
    """
```

We can now visualize whether different heads of any sensor are correlated with each other. This step is crucial in the model training process.

Failing to consider the correlation may lead to biased results and affect the model's performance. Accounting for these correlations ensures a more accurate and reliable outcome in the training process.

As depicted in fig. 16, there is a notable high correlation between different heads in two distinct categories.



(a) Average Torque in EQTQ category (b) Board Temperature in Drive category

Figure 16: Correlation matrix of different sensors

These results strongly suggest the necessity to account for the influence of other heads when predicting each individual head. Incorporating such correlations into the model is crucial for achieving more accurate and meaningful predictions.

10 Temporal Fusion Transformer

To develop the desired model, we have chosen to utilize deep neural networks. These networks are known for their scalability and robustness, contributing to enhanced performance in data generation.

10.1 Background

There have been lots of models available for predicting time-sequence data. Two of most famous neural networks are RNN [8] and LSTM [9]. But there exists problems with these two models which we explain briefly about it:

1. **Difficulty with Long-Term Dependencies:**
   - **RNNs:** Vanishing gradient problem.
   - **LSTMs:** Limited ability with very long sequences.
2. **Computational Complexity:**
   - Both RNNs and LSTMs can be computationally expensive.
3. **Limited Parallelization:**
   - Sequential processing limits parallelization.
4. **Sensitivity to Hyperparameters:**
   - Tuning hyperparameters (e.g., learning rates) is crucial and challenging.
5. **Difficulty in Capturing Irregular Patterns:**
   - May struggle with abrupt shifts or irregular patterns.

6. **Prone to Overfitting:**
    - Large parameter space makes them prone to overfitting, requiring regularization.

While RNNs and LSTMs are powerful, understanding these drawbacks is crucial for choosing suitable architectures based on the characteristics of the temporal data.

10.2 Transformers

Transformers [10], originally designed for natural language processing, have proven to be versatile in capturing temporal features in sequential data.

The self-attention mechanism, positional encoding, and sequential processing are key elements that contribute to their effectiveness in handling temporal dependencies.

10.2.1 Self-Attention Mechanism

The self-attention mechanism allows each position in the input sequence to attend to all other positions.

This mechanism captures dependencies between distant elements, enabling the model to assign higher weights to positions that are temporally closer. The attention scores are computed based on the relationships between positions, facilitating the capture of temporal patterns.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q, K, and V represent query, key, and value matrices, respectively, and d_k is the dimension of the key vectors.

10.2.2 Positional Encoding

To address the lack of inherent understanding of sequence order, positional encodings are added to the input embeddings.

These encodings provide information about the position of each element in the sequence, aiding the model in understanding the temporal order of the data.

10.2.3 Sequential Processing

Transformers process input sequences sequentially through layers of self-attention and feedforward neural networks in both the encoder and decoder.

The layer-wise processing captures temporal dependencies as each layer refines the representation based on the information gathered from the previous layer.

10.2.4 Time Embeddings

In certain applications, time-related information can be explicitly incorporated into the input embeddings.

This may include features such as timestamps or time intervals. By providing the model with information about the temporal aspects of the data, transformers can better capture time-dependent patterns.

10.2.5 Conclusion

While transformers are not specifically designed for time series forecasting, their ability to capture temporal features has been demonstrated across various domains.

The combination of self-attention, positional encoding, and sequential processing makes transformers a powerful tool for handling sequential data with inherent temporal dependencies.

10.3 Multi-horizon forecasting

Multi-horizon forecasting, i.e. the prediction of variables-of-interest at multiple future time steps, is a crucial problem within time series machine learning.

In contrast to one-step-ahead predictions, multi-horizon forecasts provide users with access to estimates across the entire path, allowing them to optimize their actions at multiple steps in future.

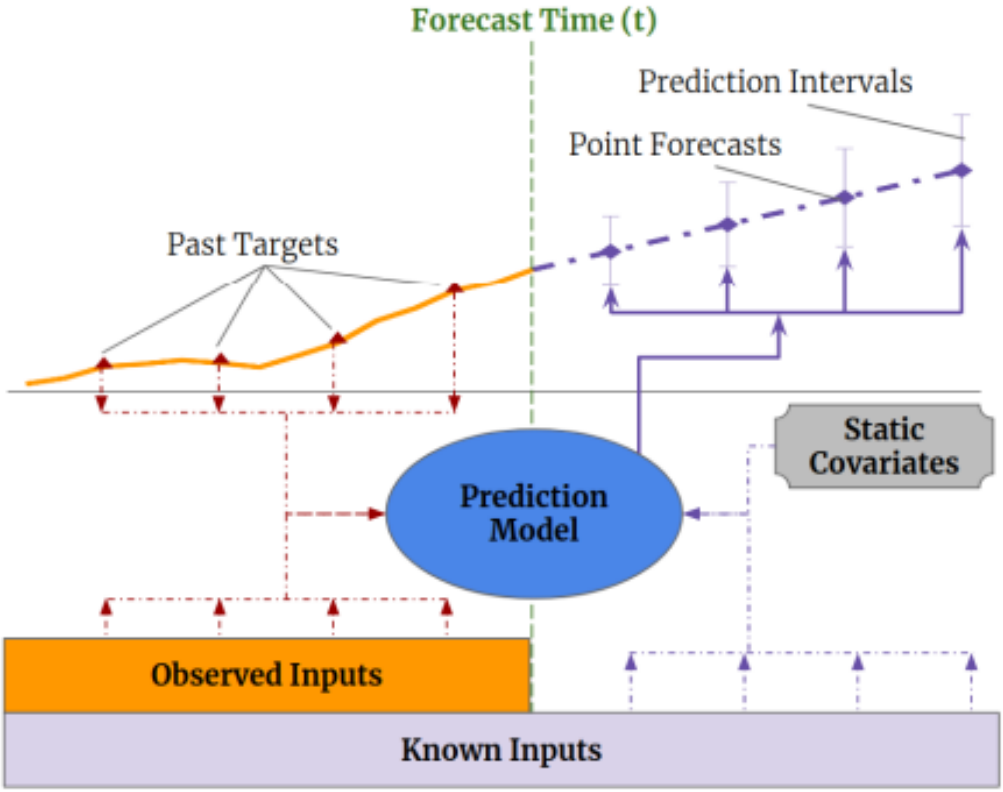


Figure 17: Illustration of multi-horizon forecasting with static covariates, past-observed and a priori-known future time-dependent inputs.

10.3.1 Architecture

In this paper the authors have developed TFT (Temporal Fusion Transform) [11] to use canonical components to efficiently build feature representations for each input type (i.e. static, known, observed inputs) for high forecasting performance on a wide range of problems.

The major constituents of TFT are:

1. Gating mechanisms to skip over any unused components of the architecture, providing adaptive depth and network complexity to accommodate a wide range of datasets and scenarios.
2. Variable selection networks to select relevant input variables at each time step.
3. Static covariate encoders to integrate static features into the network, through encoding of context vectors to condition temporal dynamics.
4. Temporal processing to learn both long- and short-term temporal relationships from both observed and known time-varying inputs. A sequenceto-sequence layer is employed for local processing, whereas long-term dependencies are captured using a novel interpretable multi-head attention block.
5. Prediction intervals via quantile forecasts to determine the range of likely target values at each prediction horizon. Fig. 18 shows the high level architecture of Temporal Fusion Transformer (TFT), with individual components described in detail in the subsequent sections.

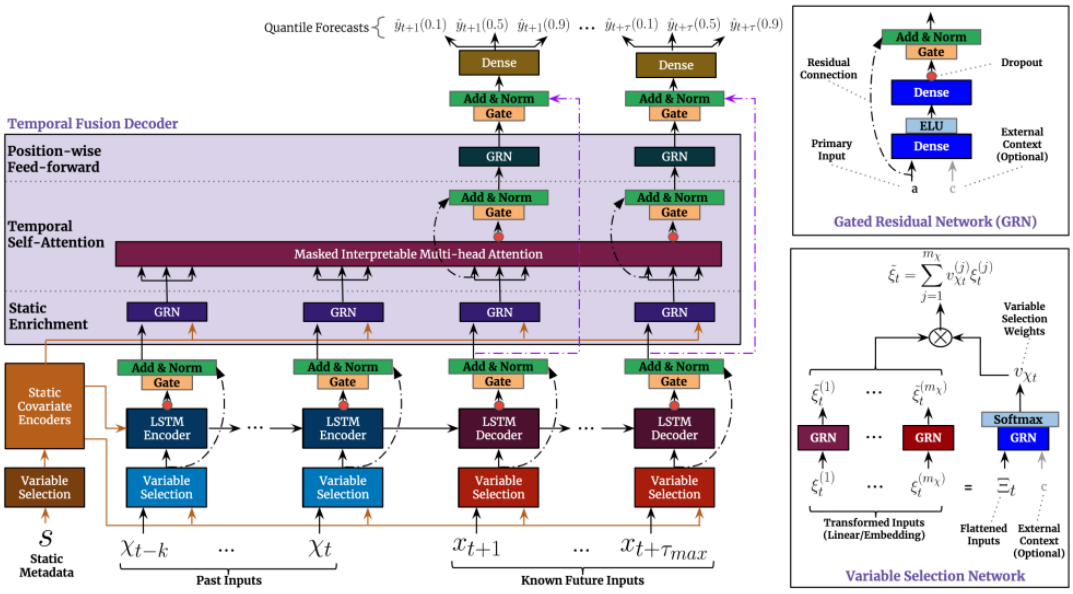


Figure 18: TFT architecture.

10.3.2 Quantile Loss

In this paper they have used quantile loss [12]. The quantile loss $L_\tau(y, \hat{y})$ for a specific quantile τ is defined as:

$$L_\tau(y, \hat{y}) = \begin{cases} \tau(y - \hat{y}) & \text{if } y \geq \hat{y} \\ (1 - \tau)(\hat{y} - y) & \text{if } y < \hat{y} \end{cases}$$

In this formula:

- L_τ is the quantile loss,

- y is the true target value,
- $\hat{y}$ is the predicted value,
- τ is the quantile level, where $0 < \tau < 1$.

This piecewise-defined function penalizes the model based on whether the prediction is above or below the true target, with the degree of penalty determined by the quantile level τ.

11 Training

During training, we leverage multiprocessing within the PyTorch framework. Specifically, we run four processes to build models for generating artificial data.

To facilitate this, we have created a function capable of training all available categories.

Our primary framework for this task is pytorch-forecasting, which relies on torch-lightning and torch. Within this framework, we utilize the TFT model for training. However, users also have the flexibility to develop their own models by referring to the documentation provided by this framework.

The preprocessed data from earlier stages serves as input for training other custom-developed models as well.

11.1 Arguments and Hyperparameters

Before delving into the training function, let's examine the hyperparameters for training the model. There are two sets of hyperparameters that we need to consider.

TrainArgs; are arguments are used to train the base model designed with TFT architecture. We have developed a trainer wrapper for training the models. For future developments, users should design their own training wrapper and training pipeline.

```
@dataclass
class TrainArgs:
    max_prediction_length: int = 5
    n_process: int = 3
    max_encoder_length: int = 10
    learning_rate: float = 0.09
    epochs: int = 150
    batch_size: int = 32
    clipping: float = 0.1
    accelerator: str = "cuda"
    devices: str = "auto"
    model_summary: bool = False
    min_delta: float = 1e-4
    patience: int = 10
    verbose: bool = False
    early_mode: str = "min"
    hidden_size: int = 8
    hidden_continuous_size: int = 8
    attention_head_size: int = 8
    dropout: float = 0.1
    target: list = field(default_factory=lambda: ['Head_01'])
    group_ids: list = field(default_factory=lambda: ["day"])
```

```
    path: str = None
    lr_tuning: str = True
    logs_dir: str = "./logs"
    checkpoints_dir: str = "./checkpoints"


    def set_train_cutoff(self, train_data):
        self.training_cutoff = train_data['time_idx'].max() —
                               self.max_prediction_length
```

Within the *TrainArgs* class, there are various sets of hyperparameters, as elucidated in the listing above. Notably, we have set *max_prediction_length* and *max_encoder_length* to 5 and 10, respectively.

While these values may seem small for capturing extensive temporal connections in time sequence data, they are chosen pragmatically due to the constraints of the available dataset.

The modularity inherent in these parameters allows users to tailor their own models, taking into account the size of their dataset and specific requirements.

Regarding the *learning_rate*, there are two options available. One can either utilize a value specified in the configuration class or opt for the learning rate tuner module in torch-lightning.

The learning rate tuner module iteratively explores and determines the largest batch size for a given model that does not result in an out-of-memory (OOM) error.

This provides flexibility in choosing an appropriate learning rate based on the specific requirements and constraints of the training process.

The *n_process* parameter, referring to the number of processors for multiprocessing. Other parameters are related to the model hyperparameters.

```
@dataclass
class InferArgs:
    heads: int = 24
    lr_tuning: bool = True
    sensor: str = 'LockDegree'
    category: str = 'eqtq'
    machinery: str = 'ejda1'
```

InferArgs; represent the arguments used in inferencing and data generation after training.

Interestingly, they can also be employed as a class passed to the train function for training each sub-model. Each object from this class signifies a sensor for which we intend to generate data.

To streamline configurations, we leverage the dataclass in Python. By decorating a class with the dataclass decorator, common special methods are automatically generated based on the class attributes.

This approach helps reduce boilerplate code, enhancing the conciseness and readability of the classes.

Before proceeding with the training of models, it is imperative for the user to create pickles for all deterministic sensors during the training phase.

Deterministic sensors are those that do not require any additional training. To create these pickles in Python, the user should execute the following command in the main directory:

```
$ python3 det.py
```

11.2 Trainer Wrapper

To expedite work and enhance modularity, we have designed a wrapper specifically for the training phase.

This wrapper is not only applicable for training but can also be utilized in the data generation process, tailored to our specific task. With this class, users gain the capability to seamlessly train, evaluate, and validate input data.

```
class Trainer:
    """
    This class have been designed to train, evaluate and validate the output
    data of the Extract() class. We accelerate the work and modularity using
    this wrapper.
    """
    def __init__(
            self,
            learning_rate: float = 0.09,
            epochs: int = 50,
            batch_size: int = 128,
            device: str = 'cpu'
            ):
        """
        Initialize a new instance of Extract.

        Args:
            learning_rate (list): learning rate value in the optimization
            epochs (str): how many times models should be trained
            batch_size (str): number of batches
            device (bool): which device we want to do the training
        """


        self.device = device
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.epochs = epochs
```

All methods in the Trainer class have been defined as shown in listing above. For each training step, the user is required to define a *DataLoader* in PyTorch.

In PyTorch, a *DataLoader* is a utility that facilitates the efficient loading and iteration over a dataset during the training or evaluation of a machine learning model. Situated within the *torch.utils.data* module, a *DataLoader* handles tasks such as data loading, shuffling, batching, and other data-related operations seamlessly.

```
    def create_dataloaders(
            self,
            data: pd.DataFrame,
            training_cutoff: int,
            max_prediction_length: int,
```

```
        max_encoder_length: int,
        target: list = field(default_factory=list),
        unk_vars: list = field(default_factory=list),
        group_ids: list = field(default_factory=list),
        ):

    """
    This method creates dataloaders to be passed to the trainer module.

    Args:
        data (pd.DataFrame): the preprocessed data from Extract class
        training_cutoff (int): cut of for the trianing data to avoid errors
        max_prediction_length (int): prediction length of the output
        max_encoder_length (int): input data length
        target (list): which head or column we want to predict
        unk_vars (list): which variables are important for predicting the
                         output
        group_ids (list): grouping based on a certain categorical column
    """
```

The *create_dataloaders*() method in Trainer wrapper helps the user to create his own custom *DataLoader* module. Data argument is the *DataFrame* created with the Extract() class in previous steps.

After creating the *DataLoader*, user should set the trainer object that can be used for the training. The *set_trainer*() provides the side characteristics of the training procedure. We use three callbacks for the training.

```
def set_trainer(
        self,
        machinery: str,
        category: str,
        sensor: str,
        head: str,
        devices: str,
        logs_dir: str,
        checkpoints_dir: str,
        clipping: float = 0.1,
        model_summary: bool = False,
        min_delta: float = 1e—4,
        patience: int = 10,
        verbose: bool = False,
        early_mode: str = "min",
        ):

    """
    This method creates dataloaders to be passed to the trainer module.

    Args:
        machinery (str): machinery name
        category (str): category of the machine
        sensor (str): name of the sensor available in the specific category
        head (str): which is the main head of the sensor to predict based on it
```

```
        devices (str): which devices should be used for training
        checkpoints_dir (str): checkpoints directory to save the model
        logs_dir (str): logs directory to save the model losses for
                        tensorboard
        clipping (float): model parameter
        model_summary (bool): whether to have model summary
        min_delta (float): parameter of the model
        patience (int): patience for the validation loss to decrease
        verbose (bool): whether to print logs in the training
        early_mode (str): which algorithm we use for stopping the training

    Generate()
    Returns:
        pl.Trainer: the trainer object for training the model
    """
```

These callbacks are used from *lightning.pytorch.callbacks* to accelerate the training and increase the performance of trained models.

- *EarlyStopping*: This callback can be used to monitor a metric and stop the training when no improvement is observed.
- *ModelCheckpoint*: Save the model periodically by monitoring a quantity. In every validation step, if the loss decrease from the previous, it will be considered as current checkpoint of the model.
- *LearningRateMonitor*: Automatically monitor and logs learning rate for learning rate schedulers during training.

Additional methods, such as *set_model*(), are responsible for creating a TFT model object, which serves as the main model used during training.

The *find_lr*() method within this class aids the trainer in determining the optimal learning rate, contributing to an acceleration of the training speed.

Furthermore, the *load_model*() method serves the dual purpose of loading models post-training or setting the model for data generation in subsequent steps.

```python
def fit(
    self,
    ):
    """
    Trains the model and saves best models based on the validation loss.
    """
    self.trainer.fit(
        self.tft,
        train_dataloaders=self.train_dataloader,
        val_dataloaders=self.val_dataloader,
    )
    best_model_path = self.trainer.checkpoint_callback.best_model_path
    self.model = TemporalFusionTransformer.load_from_checkpoint(
                                        best_model_path,
                                        map_location=torch.device(self.device))
```

Once all the training components are set, the $fit()$ method is employed to train the model. It's important to note that for each sensor and head, a separate Trainer wrapper is required.

To streamline this process, we have developed the $train()$ function within *train.py* to handle the training for each sensor in each category and machinery.

Within the $train()$ function, a dictionary is created for all the machines, which may need to be modified by the user.

Alternatively, this information can also be included in the $TrainArgs$ object discussed earlier for greater configurability.

```
machines = {
    'XB056' : 1,
    'JF890' : 24,
    'WM100' : 24,
    'JF891' : 24,
    'XB098' : 1,
    'JF893' : 24,
}
```

To implement multiprocessing, we utilize the *torch.multiprocessing* module within the Torch framework.

Different numbers of processors are assigned to run the model for each machine and its sensor individually. Post-training, all checkpoints are saved in the directory specified by the user in the $TrainArgs$ object.

Additionally, we have $TensorBoard$ [13] logs. These logs enable users to visualize the training progress across different epochs.

One can observe how the val_loss changes over time, providing a comprehensive view of the training steps.

For the generation process, it's imperative that all checkpoints are available and have been trained.

If the checkpoints are not present, the system automatically attempts to run the training method. Users can also initiate the training of models using:

```
$ python3 train.py
```

11.3 Results

After training is done we can visualize the results obtained during training phase. First, let's take a look at final validation losses for one sensor with different heads.

	Head_09	Head_10	Head_11	Head_12
EQTQ - LockDegree (val_loss)	0.0625	0.0002	0.0004	0.0132
EQTQ - stsClosureOk (val_loss)	1.5655	0.2713	0.2946	0.0203
Drive - Tplate (val_loss)	0.433	1.256	0.527	0.279
Drive - Tboard (val_loss)	2.784	2.861	2.496	2.956

Table 4: Validation losses for different sensors in *EQTQ* and *Drive* category

In Table 4, we observe validation losses for four different sensors and their corresponding heads. Notably, the *Tboard* sensor does not perform as well as the others, likely due to the insufficient data available for this sensor.

To improve the loss value, one could consider increasing the dataset size for this sensor and potentially augmenting the complexity of the model.

	ProdSpeed	PowerVoltage	PowerCurrent	test1
val_loss	362.523	0.837	0.023	0.802

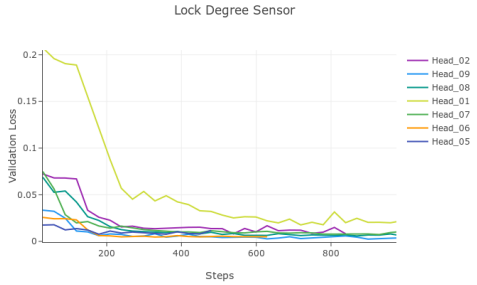
Table 5: Validation losses for different sensors in *PLC* category

In Table 5, losses for the PLC sensors are displayed. Notably, the *ProdSpeed* sensor exhibits a high loss value.

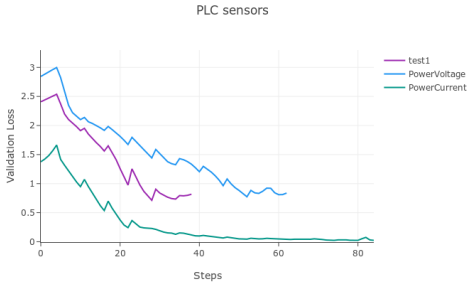
It's worth mentioning that the initial loss value was 4000, and though it has been reduced, further improvements could be achieved with more organized and expansive data for this particular sensor.

Furthermore, exceptional results are evident in the *LockDegree* sensor across all heads.

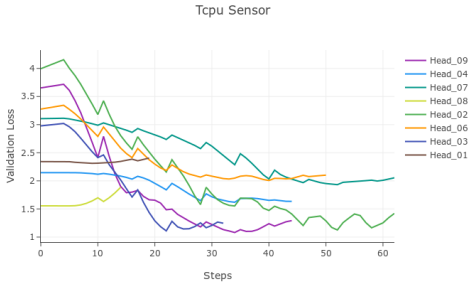
The model demonstrates an ability to accurately predict future values based on the encoder data provided to the model. *TensorBoard* logs offer a valuable resource for tracking and visualizing the training progress.



(a) *EQTQ*



(b) *PLC*



(c) *Drive*

Figure 19: Progress of the validation loss in different categories

In Fig. 19, various figures depicting chosen heads and sensors are presented.

Notably, the best results are observed for the *EQTQ* category, owing to the substantial amount

of data available. The model also performs well in the Drive category and some of the *PLC* sensors.

In the subsequent steps, the focus will be on evaluating the generation and prediction quality of the trained models. Before proceeding, let's examine the training log in the terminal.

```
MODEL: XB056,  eqtq,   Index,           Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: TRUE,   PID: 20247
MODEL: XB056,  eqtq,   MaxLockPosition, Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: FALSE,  PID: 20247
MODEL: XB056,  eqtq,   stsClosureOK,    Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: FALSE,  PID: 20248
MODEL: XB056,  eqtq,   LockDegree,      Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: FALSE,  PID: 20246
MODEL: XB056,  eqtq,   stsBadClosure,   Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: TRUE,   PID: 20248
MODEL: XB056,  eqtq,   stsNoLoad,       Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: TRUE,   PID: 20246
MODEL: XB056,  eqtq,   stsTotalCount,   Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: FALSE,  PID: 20247
MODEL: XB056,  eqtq,   AverageTorque,   Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: FALSE,  PID: 20246
MODEL: XB056,  eqtq,   MinLockPosition, Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: FALSE,  PID: 20248
MODEL: XB056,  drive,  Tcpu,            Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: FALSE,  PID: 20248
MODEL: XB056,  eqtq,   stsNoClosure,    Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: TRUE,   PID: 20246
MODEL: XB056,  drive,  Tplate,          Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: FALSE,  PID: 20248
MODEL: XB056,  drive,  Twindings,       Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: FALSE,  PID: 20246
MODEL: XB056,  eqtq,   AverageFriction, Head_01,   TRAINING: SUCCESSFULL,  DETERMINISTIC: FALSE,  PID: 20247
```

Figure 20: Train output logs in terminal

In Fig. 20, the training logs provide insights into successfully trained sensors. Furthermore, it indicates sensors that do not require training and can be predicted algorithmically.

The *PID* information is also available, illustrating which processor each model is running on. This comprehensive view aids in assessing the training progress and the status of individual sensors.

12 Evaluation

Once the models are trained, the next step involves evaluating their performance in predicting new data.

Beginning with the validation dataset allows for a comparison between the predicted values and the ground truth.

This evaluation is crucial in assessing the model's ability to generalize and make accurate predictions on unseen data.

12.1 Prediction by Validation

In the Trainer wrapper from the previous stage, a dedicated method, $val_pred()$, has been designed for predicting validations and comparing them with the actual future values.

This method encapsulates the process of generating predictions on the validation data and facilitates a comparison with the ground truth.

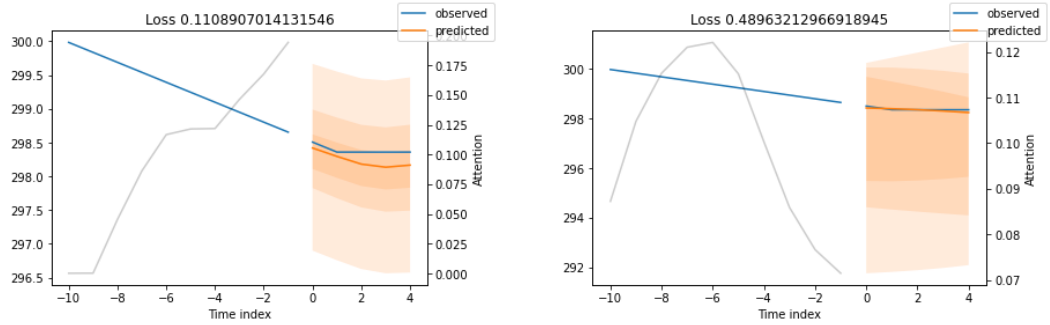
The results of predictions for $Head_01$ are visualized in Fig. 21. It's evident that the new data has been successfully predicted with acceptable accuracy.

To further enhance accuracy and capture periodic content in each sensor, consider increasing the number of entries in the dataset and adjusting the encoder input length accordingly.

Fine-tuning these parameters can lead to improved performance and more accurate predictions.

12.2 Prediction by Variable

The original framework for the TFT model (pytorch-forecasting) provides methods to predict data based on calculated variables in the training.



(a) *AverageFriction* in *EQTQ* (b) *Tcpu* in *Drive*

Figure 21: Plotting the validation set

It enables the visualization of the distribution of the generated data based on different variables in the validation dataset.

This insight into the distribution helps in understanding how well the model captures the underlying patterns and relationships in the data.

Fig. 22 displays figures pertaining to averages of different metrics within the original distribution. The predictions for various variables appear satisfactory.

The $pred_by_variable()$ method within the Trainer wrapper has been utilized to generate these results, showcasing the model's ability to predict data based on calculated variables in the training.

13 Generation

After running the simulation in the backend, we create a *Generate*() object for each sensor, which can have different numbers of heads.

This object is utilized to generate samples, whether they are true samples or fault samples.

Additionally, an algorithm has been developed to transform a sample into a fault. Although all faults are stored in the database, they will not be used in future generations.

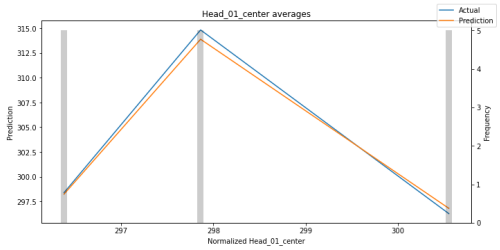
This process contributes to creating a diverse dataset that includes both normal and fault samples for training and evaluation purposes.

13.1 Prediction

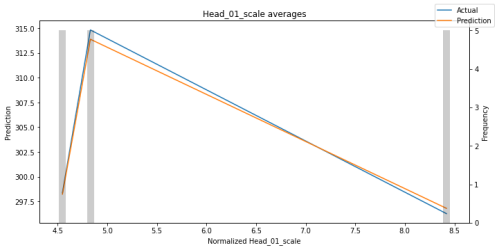
The *Generate*() object relies on the *Trainer*() wrapper class to load the models and employ them for new predictions.

For each head of each sensor, the desired model is loaded. Once the predicted values reach $max_encoder_length$, the object automatically generates a new set of samples.

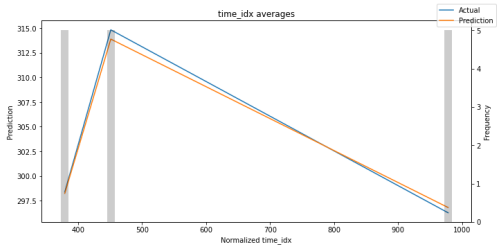
```
class Generate:
    """
    This class have been designed to for generating artificial samples. In each
    method we generate a fault or a real sample. For the generation we need to
    use available dataset.
```



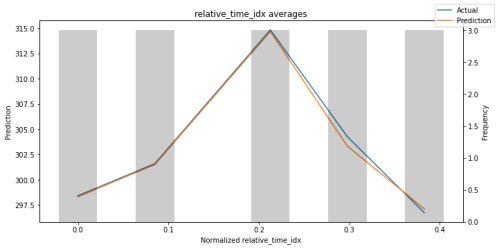
(a) Center averages



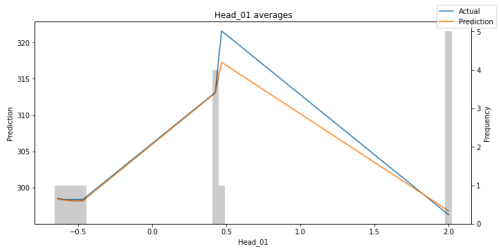
(b) Scale averages



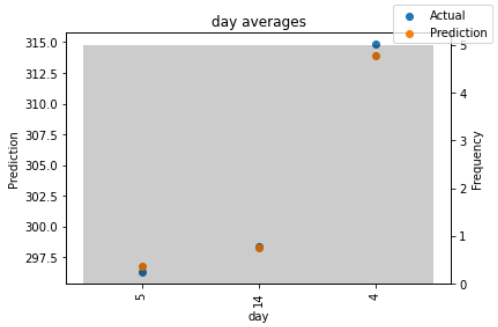
(c) Time_idx average



(d) Relative time_idx average



(e) Head_01 averages



(f) Day averages

Figure 22: Plotting validation by variable for *AverageFriction* sensor in *EQTQ* category

Methods:
predict():
Predicting the future correct samples.
get_type():
Type of original values (int pr float)
get_values():
get predicted values.
gen_fault():
get fault samples with a specific algorithm.
"""

```
def __init__(
    self,
    metadata,
    type_pickle: str = './pickles/type_sensors.pickle',
    det_pickle: str = './pickles/det_sensors.pickle',
    device: str = 'cpu'
):
```

```
    """
    Initialize a new instance of Generate.

    Args:
        metadata (dict): dictionary of the sensors metadata
        type_pickle (str): type of sensors generated from det.py
        det_pickle (str): deterministic sensors generated from det.py
        device (str): which device we should use for prediction
    """
```

```
def gen_fault(
    self,
    alpha=3,
    beta=0.1
):
    """
    Args:
        alpha (int): Coefficient for generating fault.
        bias_percent (float): bias for calculating the fault sample.

    Returns:
        dict: dictionary of predicted fault values.
    """
```

This process ensures continuous and dynamic sample generation based on the models loaded for each sensor head.

The $Generate()$ class differs when it comes to the PLC category. To handle this specific category, a separate class, $GeneratePlc()$, has been developed.

If you have a new category and wish to include it in the simulation, you would need to develop a class similar to $Generate()$ to manage the model for that particular category.

This modular approach allows for flexibility in accommodating new categories within the simulation framework.

One of the methods in this class is $gen\_fault()$, designed specifically for generating fault samples.

The formula used for generating fault samples involves two coefficients: α, a coefficient for the standard deviation when not equal to zero, and β, used for integer values. In the case of integer values, the formula is as follows:

$$h_t = h_t + \beta \dot{h}_t$$

And for the float values:

$$h_t = h_t + \sigma(\alpha + \mathcal{U}(0, 1))$$

Where standard deviation define as:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}}$$

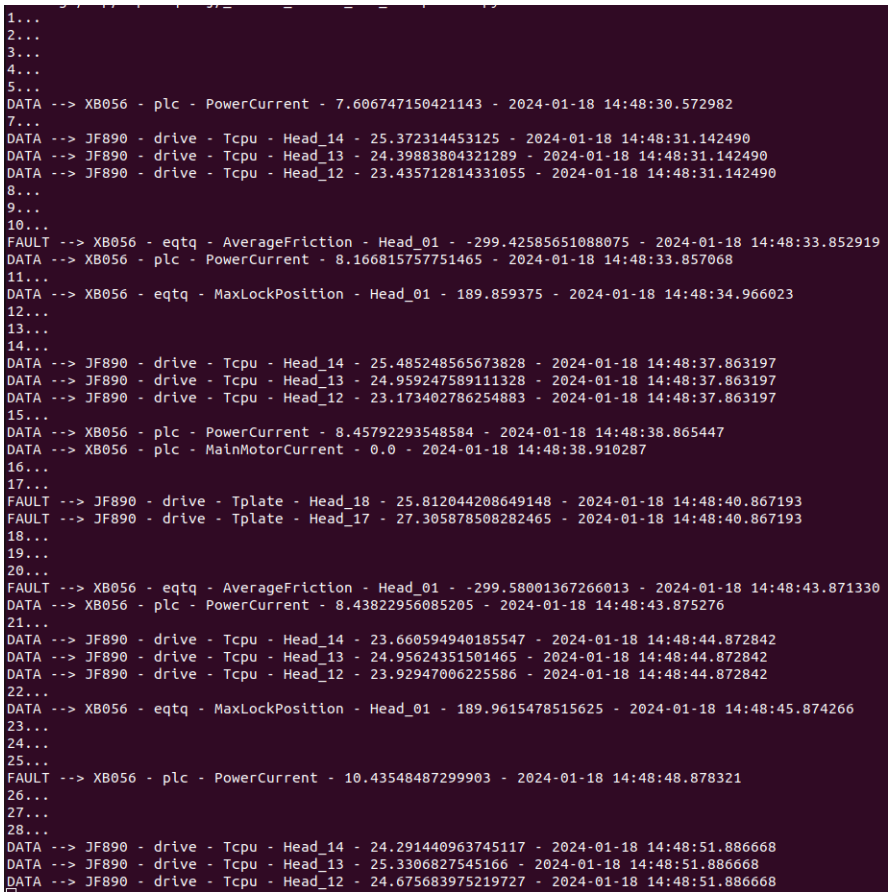


Figure 23: Generation log in the terminal

In the above formulation x_i represents each individual data point, $\bar{x}$ is the mean (average) of the data points and n is the number of samples per each head.

Moreover, in the fig. 23 you can see a sample generation log in *server.py*. All the data will be stored into the MongoDB database in real time.

13.2 Database

```
def put_mongo(sample, metadata, col):
    """
    This function is used to put the data in the db collection. For each category
    and each available heads we have tu put it in the category collection.

    Args:
        sample (dict): dictionary of the generated samples (one sample
                       for each head).
        metadata (dict): metadata in each action_pool in the simulation.
        col (flask_pymongo.wrappers.Collection): db collection to be used to
                                                 put the data

    Returns:
```

```
    None
"""
heads = [f'Head_{i:>02}' for i in metadata['sensor']['heads']]
db_heads = [f'H{i:>02}_' for i in metadata['sensor']['heads']]
for c, head in enumerate(heads):
    x = metadata['sensor']['head_ids'][head]
    date = int(sample[head].name.timestamp() * 1000)
    name = db_heads[c] + metadata['sensor']['name']
    col[metadata['sensor']['category']].update_one(
        filter = {'_id': x},
        update = {
            '$push': {'samples': {'name': name,
                      'value': round(sample[head][head].item(), 3),
                      'time': {'$numberLong':date}}},
            "$set": { "last_time": {"$numberLong": date}},
            '$inc': {'n_samples': 1}
        })

    col[metadata['sensor']['category']].update_one(
        filter = {'$and': [{'_id': x}, {'first_time': {"$eq":""}}]},
        update = {"$set" : {"first_time" : {"$numberLong": date}}},
    )
```

For storing the generated data in the database, the pymongo [14] library is utilized. This framework allows the execution of queries to the database directly in Python.

To facilitate the posting of data to the database, a function has been created in the *utils.py* file.

This function streamlines the process of inserting data into the database, providing a convenient and efficient way to manage the generated information.

The pymongo framework is a Python driver for MongoDB, a popular NoSQL database. MongoDB is a document-oriented database that stores data in JSON-like BSON (Binary JSON) documents.

The pymongo provides a Python interface for interacting with MongoDB, allowing you to connect to a MongoDB database, perform CRUD (Create, Read, Update, Delete) operations, and more.

In listing. **??**, efficient queries have been written for posting the generated samples, whether they are real or fault samples, to the database using pymongo.

After the generation process, a dump of all instances in the database will be saved in the specified folder.

This dump serves as a record of the generated data and can be presented to the user in the original repository for further analysis or reference.

14 Conclusion

After training numerous models and designing various algorithms to predict future data, TFT performed exceptionally well in every aspect of data generation. We encountered several challenges during our work, but we successfully found solutions for each problem.

One of the primary objectives of this project was to develop a scalable and modular model that could be easily used, fine-tuned, and further processed by company workers. We achieved success in fulfilling this goal.

1. Small dataset; one of the main challenges we encountered was the limited size of the dataset. The original data at our disposal had a very low number of instances (e.g., in the Drive category, we only had 20 samples per sensor). One solution involved developing an algorithm that mimicked this scarcity of data, but it proved to be neither robust nor scalable.

These algorithms fail to capture temporal connections between different timestamps when the dataset size is increased. The reason is that these temporal connections follow a non-linear pattern, necessitating processing in non-linear models.

While the models perform well with the small dataset, they can achieve even better optimization and performance by increasing the volume of data.

2. Different recording days; the original data was recorded on different days, with each day exhibiting a distinct pattern, as discussed earlier. To address this issue, we treated days as one of the categorical features during training.

Furthermore, within the $Trainer()$ class, we implemented methods capable of predicting future data based on a specific day. The generated data aligns with the pattern observed on that particular day.

3. Different frequency; the available dataset presented varying frequencies across the heads and sensors. This discrepancy posed a challenge, preventing us from accurately obtaining values for all sensors in a given timestamp. Consequently, the correlations were not properly identified, and our model functioned as a prototype.

To address this issue, we attempted to utilize interpolation to fill in the NaN values and uncover the desired pattern for data analysis.

4. Deterministic sensors; we identified the presence of sensors that exhibited either constant, absolute periodic, or index values, which did not require additional training. To address this issue, we developed an algorithm capable of automatically extracting these sensors in an unsupervised manner and capturing their patterns.

5. Scalability; we aimed to create models capable of accommodating varying amounts of data. To achieve this, we delved into state-of-the-art deep learning models that could be configured for large datasets in the future.

In conclusion, we successfully developed a modular, scalable, and highly accurate model based on the limited dataset available to us.

15 Suggestions

After analyzing the work and proposed systems in this project, I have a few suggestions that can enhance the performance of the current work:

- Dataset: As mentioned earlier, the dataset plays a pivotal role in this project. Increasing the volume of data can lead to better models with improved accuracy and performance.
- Recording Days: Expanding the recorded data from the original machineries across different days provides a more comprehensive understanding of how the machine operates over time.
- Stable Frequency: Maintaining a stable frequency is crucial for generating a more reliable dataset. Consistent timestamps across different sensors establish a perfect representation

for both the sensors and various machine heads. This facilitates the discovery of correlations between different sensors and more intricate patterns.

References

[1] W. McKinney, "pandas: a foundational python library for data analysis and statistics," *Python for High Performance & Scientific Computing*, vol. 14, pp. 18–25, 2011.

[2] pandas Development Team, *pandas Documentation*, 2022, https://pandas.pydata.org/pandas-docs/stable/.

[3] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[4] P. Contributors, *PyTorch Documentation*, 2022, https://pytorch.org/docs/stable/.

[5] *MongoDB Documentation*, MongoDB, Inc., 2022, https://docs.mongodb.com/.

[6] K. Chodorow and M. Dirolf, "Mongodb: The definitive guide," 2013.

[7] Python Software Foundation, "PEP 557 - Data Classes," https://www.python.org/dev/peps/pep-0557/, accessed: March 17, 2024.

[8] R. M. Schmidt, "Recurrent neural networks (rnns): A gentle introduction and overview," 2019.

[9] R. C. Staudemeyer and E. R. Morris, "Understanding lstm – a tutorial into long short-term memory recurrent neural networks," 2019.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[11] B. Lim, S. O. Arik, N. Loeff, and T. Pfister, "Temporal fusion transformers for interpretable multi-horizon time series forecasting," 2020.

[12] D. B. Or, M. Kolomenkin, and G. Shabat, "Generalized quantile loss for deep neural networks," 2020.

[13] TensorBoard. (2015) Tensorboard: Visualizing learning. https://www.tensorflow.org/tensorboard. [Online]. Available: https://www.tensorflow.org/tensorboard

[14] "pymongo: python driver for mongodb," https://pypi.org/project/pymongo/, accessed: March 17, 2024.