PROJECT INTERNSHIP
(PROJEKTPRAKTIKUM)

ENHANCEMENT OF ADAS HARDWARE-DEVELOPMENT PROCESS THROUGH MACHINE LEARNING, CLOUD DEPLOYMENT AND VEHICLE SENSOR DATA ANALYSIS

Verbesserung des ADAS-Hardware-Entwicklungsprozesses durch maschinelles Lernen, Cloud-Einsatz und Analyse von Fahrzeugsensordaten

Student:	Sarvesh Bhalchandra Telang
Matriculation number:	422444
Supervisors (Company):	M.Eng. Massimiliano Botticelli, Dr.-Ing. David Molnar
Company:	Robert Bosch GmbH
Period of internship:	15.10.2022 – 14.04-2023

May 2023

Confirmation of the Company

We hereby confirm that we have checked and approved the contents of this internship report.

Name and Signature of an authorized representative of the company and stamp of the company[1]

[1] The signatory's affiliation with the company must be confirmed by a company stamp. This can be omitted if the signature matches the signature on the internship certificate.

Task Description

Task description of the project internship (Projektpraktikum) for Sarvesh Bhalchandra Telang, Matr. No. 422444

Enhancement of ADAS Hardware-Development Process through Machine Learning, Cloud Deployment and Vehicle Sensor Data Analysis

Verbesserung des ADAS-Hardware-Entwicklungsprozesses durch maschinelles Lernen, Cloud-Einsatz und Analyse von Fahrzeugsensordaten

---

Objectives and Work description:

PART I: Supporting AI-driven development with pyMICE

- Running pyMICE on DASK cluster with Python 3.6 for both Windows and HPC: To configure the package in a way that ensures its compatibility across various operating systems, while also ensuring that the DASK client is closed properly.
- Migrating the pyMICE virtual environment from Python 3.6 to Python 3.9: The package has been developed to work with Python 3.6, but it is important to keep up with the latest version of Python to take advantage of new features and improvements.
- Porting pyMICE to Azure ML cloud: To port pyMICE to the Azure ML cloud while setting up a DASK cluster on both single and multi-nodes of azure compute cluster.

PART II: ADAS Evaluation: Analysis of Vehicle Sensor Measurements

This part of project aims to analyze sensor data from vehicle measurements that contains 20+ sensor positions per vehicle. The analysis will provide insights into various aspects such as environmental loads within a vehicle, vehicle driving behaviour and correlation between the environmental measurements of different mounting positions within a vehicle with respect to the driving phase. This project has undertaken several key tasks. Firstly, it involved data transformation to prepare the raw data by converting it from a Matlab compatible file format to Python. Subsequently, data pre-processing and data cleaning was performed. Next, the task was to perform time series, statistical and correlation analysis on the data of multiple physical quantities such as temperature, humidity, and dew point. At last, the task was to implement an automatized pdf report generation process and save the results.

Additional tasks:

- To create a threshold function: The objective was to create a function that calculates the number of times the threshold value of temperature, humidity or dew point is exceeded either above or below the specified limit and calculate the exact duration for which the value remained above or below the threshold value.
- To develop a driver visualization tool using MATLAB Graphical User Interface to visualize the driver groups with different drive profiles and road conditions and to provide a more readable and user-friendly graphical representation of driver groups.

Declaration of authorship

I, Sarvesh Bhalchandra Telang Matr. No. 422444

declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material, which has been cited either literally or indirect from the used sources.

04.05.2023

(Sarvesh Bhalchandra Telang)

Declaration of agreement to plagiarism assessment

The reports to be reviewed are transmitted to the service URKUND, where they are checked for consistency with external sources and permanently stored in the database maintained by URKUND exclusively for the University of Kaiserslautern-Landau (RPTU) for the purpose of comparison with future reports to be reviewed.

I, Sarvesh Bhalchandra Telang Matr. No. 422444

declare that I agree that the report I have submitted and written, including my personal data, may be permanently stored in the database maintained by URKUND for the University of Kaiserslautern-Landau (RPTU) for the purpose of checking internship reports for plagiarism.

The agreement to the permanent storage of the text and the disclosure of personal data is voluntary. The agreement to the storage and use of personal data can be revoked at any time by declaration for the future.

04.05.2023

(Sarvesh Bhalchandra Telang)

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ADAS	Advanced Driver Assisting Systems
pyMICE	Python Mining Intelligent Cameras and ECUs
NSGA II	Non-dominated Sorting Genetic Algorithm II
KDF	Knowledge Discovery Framework
ML	Machine Learning
HPC	High-performance computing
VM	Virtual Machines
NSG	Network Security Group
DAG	Directed Acyclic Graphs
GUI	Graphical User Interface
HPO	Hyper-parameter Optimization
EA	Evolutionary Algorithms
API	Application Programming Interface

1 Introduction

With the rapid advancement of technology and the increasing complexity of the data generated, traditional data analysis techniques are becoming insufficient in providing actionable insights for continuous product development. This challenge is particularly pressing in today's increasingly competitive business landscape, where companies are challenged to bring innovative products to market faster and more efficiently than ever before. Moreover, traditional product development methods can be expensive and time-consuming, with experiments requiring prototypes and test benches and simulations often taking up a significant amount of time and resources. This is where AI-driven product development comes into play, leveraging machine learning applications to handle the massive amounts of data produced during the product development process. Utilization of an AI-driven strategy has brought about a significant transformation in product development, particularly in the realm of autonomous driving and Advanced Driver Assistance Systems (ADAS). These systems rely heavily on a multitude of sensors including Cameras, Lidar, Radar, IMUs, and Ultrasonic sensors that generate a substantial amount of complex data. This data is utilized for safety critical applications and features of ADAS like “adaptive cruise control (ACC)”, “blind spot detection (BSD)”, “forward collision warning (FCW)”, lane assistance, signs or traffic light recognition, emergency braking, surround sensing, and obstacle detection. The complexity and high dimensionality of this data can be daunting, but with AI, the analysis of this data can be automatized, making it possible to gain a deeper understanding of these complex physical phenomena. This, in turn, allows for the development of better designs, leading to increased efficiency and accuracy in product development, and ultimately contributing to the improved safety and performance of ADAS. [BHS21a]

The primary objective of this project internship is to contribute towards the enhancement of the ADAS mechanical or hardware development process. This will be achieved through the utilization of cutting-edge techniques used in AI-driven product development and computer science such as machine learning, cloud deployment, and the analysis of vehicle sensor measurements. One of the key aspects of the internship will be to investigate the impact of environmental loads on various mounting locations within a vehicle by thoroughly analyzing the vehicle sensor measurements. This will allow for a more comprehensive understanding of the various factors that can affect the performance of ADAS hardware and lead to the development of more efficient and accurate mounting location models for ADAS sensors. The implementation phase of the project is split into two distinct parts. The first part, known as 'PART I: Supporting Artificial Intelligence (AI) Driven Development with pyMICE', concentrates on AI-driven product development using an AI framework. The second part, known as 'PART II: ADAS Evaluation: Analysis of Vehicle Sensor Measurements', focuses on analyzing the impact of environmental loads on the vehicle.

The second part entails analyzing the dataset from Bosch's internal vehicle measurement database, which assists engineers and researchers in gaining a better understanding of the environmental loads found in various mounting locations within a vehicle. This allows to

make informed decisions about the final mounting positions for ADAS sensors. The motivation for this project stemmed from the need to address common challenges faced by engineers, such as uncertainty around environmental loads, feasibility of customer requirements, and identifying mounting locations that allow for higher service times. Unlike other measurement techniques, which are limited to one mounting location or product, this project focuses on long-term measurements of multiple vehicles to record temperature, humidity, and dew point in various mounting locations within each vehicle. This enables engineers and researchers to identify trends and patterns that may not be readily observable through measurements taken over a short period of time. For instance, long-term measurements help in identifying seasonal environmental factors that impact vehicle performance and enable more accurate forecasting of future trends and more accurate models. The measurements are continuously uploaded to a centralized database, enriched with driver logs, and can be visualized and analyzed using a MATLAB-based Graphical User Interface (GUI).

2 Theoretical Background

This section sets the groundwork for the implementation of 'PART I: Supporting Artificial Intelligence (AI) Driven Development with pyMICE', by providing a solid theoretical foundation and essential terminology.

To begin, the first subsection offers an introduction to both the Knowledge Discovery Framework and pyMICE, setting the stage for subsequent sections. Building on this foundation, the second section provides detailed information about the use case of pyMICE, specifically its application in the Automotive Camera Joining Process.

As pyMICE involves working with distributed computing libraries like DASK, the following two subsections dive deeper into DASK and its significance in machine learning. It aims to provide a comprehensive understanding of parallel computing using DASK and its application.

The subsequent section will explain the significance of Migration process of Python virtual environment, which will be crucial for understanding the implementation of the tasks associated with migration process of pyMICE in the implementation phase.

Finally, the last section provides a high-level overview of cloud deployment and Microsoft Azure ML fundamentals. This will enable to understand the final task of deploying code over the Azure ML cloud.

2.1 Knowledge Discovery Framework and pyMICE

The Knowledge Discovery Framework (KDF) is an enhancement of classic data analysis that utilizes AI and advanced machine learning & data pre-processing techniques to discover knowledge from large and complex data sets. The KDF approach involves transferring data from a system where measurements and simulations are performed to an AI framework for analysis. This AI framework employs a knowledge discovery process that involves several steps such as Data pre-processing, Data exploration, Model selection, Model validation, Model exploration, and Model exploitation. The newly discovered knowledge is then validated by domain expertise and fed back into the raw data, creating a continuous cycle of improvement. This framework enables organizations to gain valuable-insights from their data and make data-driven decisions that lead to continuous product development. [BHS21a] [Bot23]

pyMICE is a versatile python package that was developed in the contest of the PhD Research "Knowledge Discovery in Combustion Concepts Development for Gasoline DI Engines" by M. Eng. Massimiliano Botticelli [Bot23, p. 8- p. 76]. It is a Knowledge Discovery Framework that is entirely designed in Python, which has many libraries suited for AI/Machine Learning. The acronym "pyMICE" originally stood for "python Mining Internal Combustion Engines," but due to the package's wide applicability, the author suggests an alternative interpretation as "python Mining Intelligent Cameras and ECUs." The framework aims for autonomous knowledge extraction from raw data. The following are some of pyMICE's features and applications: Knowledge Discovery, Basic Data Transformations, Correlation

Analysis, Data Storage (HDF5 format), Multi-processing Data Analysis, Multi-objective Optimization, Jupyter GUI. [BHS21a] [Bot23, p. 8- p. 76]

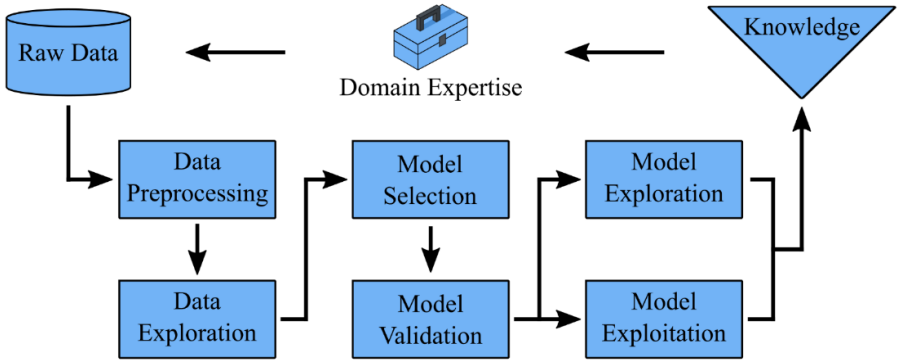


Figure 2-1: Knowledge Discovery Framework [BHS21a]

2.2 Use case: Application of pyMICE to the Automotive Camera Joining Process

Raw data containing design and process parameters such as displacement, material properties, and dimensions of camera components is analyzed to determine how these parameters influence the automotive camera's printed circuit board (PCB) warpage. The primary objective of this analysis is to optimize the design and process parameters to achieve the desired values of outputs. This refers to a multi-objective optimization problem that is solvable by pyMICE. The model selection process in pyMICE uses an XGB-NSGAII algorithm [BHS21b] [DPA02] for optimizing the hyperparameters of machine learning models. It employs the Pareto front, which represents a set of optimal solutions where no other hyperparameter can be improved without negatively affecting one or more desired outputs [BHS21b] . This approach ensures that the resulting hyperparameters yield the best possible performance across all relevant metrics. Once the hyperparameters have been optimized, the resulting models are then used to optimize the design and process parameters of the joining process. This is a straightforward optimization task that benefits from the fast predictions made by the machine learning models. By leveraging the optimized models, the process can be fine-tuned to achieve the optimum values of design and process parameters that minimize the PCB warpage while simultaneously maximizing other desired outputs. [BHS21b] [Bot23, p. 8- p. 76]

2.3 DASK

Dask is a parallel computing library in Python that enables users to analyze large datasets that cannot fit into memory. It is an excellent tool for handling big data and executing parallel computations [Roc18a] . The computations are represented as task graphs, which are primarily the directed acyclic graphs (DAGs) and defined using common Python structures such as dictionaries, tuples, and functions [Roc15] . This approach makes it easy for Python programmers to work with Dask. Additionally, Dask offers a range of schedulers that can be used to run the DAGs, including a scheduler for a single machine that uses threads, and a distributed scheduler that can scale across a cluster [Cri16] . By separating the algorithm

description from the scheduler, Dask enables easy switching between a single machine and a cluster as the size of the computation increases [Cri16] . Dask is made up of two components. The first one is Dynamic task scheduling system, which handles the actual computation work. The second one is "Big Data" collections, which includes parallel arrays, dataframes, and lists. [Roc15] [Roc14]

Dask collections create task graphs that depict the execution of the computation in parallel. Each node represents a regular Python function, while the lines that connect the nodes represent regular objects. Task graph can be seen by using the "visualize()" function on any collection object. It requires the graphviz library to be installed beforehand. [Sig22] [Roc14]

High-Level collections include standard collections like numPy arrays, Bags or lists, DataFrames but all in parallel and Machine Learning libraries such as parallel Scikit-Learn. Low-Level collections include .delayed functions executing python functions in parallel, and Futures executing function in parallel but in real-time. [Sig22]

It should be noted that Dask predominantly uses the concept of Lazy Evaluation which means that the output is generated only when specifically requested through the compute() function [Roc15] .

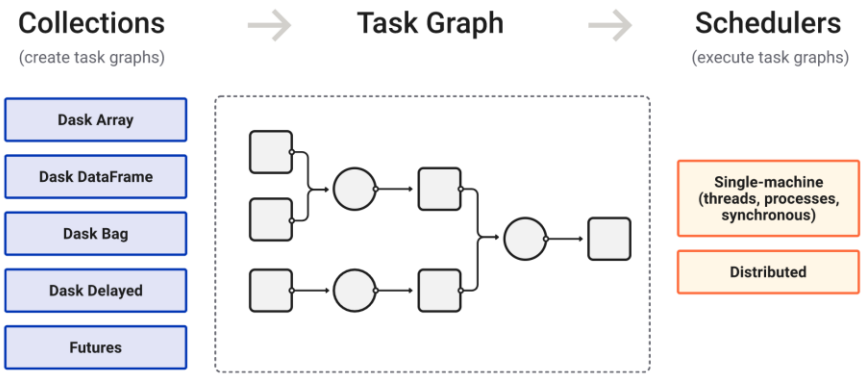


Figure 2-2: Dask collections [Roc14]

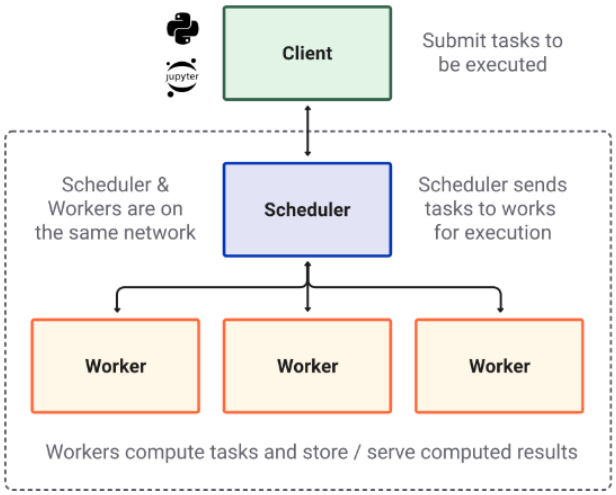


Figure 2-3: Dask architecture [Sig22]

Once the task graphs have been created by the collections, Dask must run them on hardware capable of parallel processing. The Dask architecture is shown in the figure 2-3. The parallel computing architecture of Dask consists of three main components: Dask scheduler, Dask workers, and Dask client [Roc15] .

The Dask scheduler is responsible for scheduling tasks across multiple workers. Workers perform the computations assigned to them by the scheduler. Each worker runs on a separate process or machine and can execute tasks in parallel [Sig22] . The Dask client is the interface used by the user to interact with the Dask scheduler and initiate computations. It serves as the entry point for the user and is located wherever the Python code is written. [Sig22]

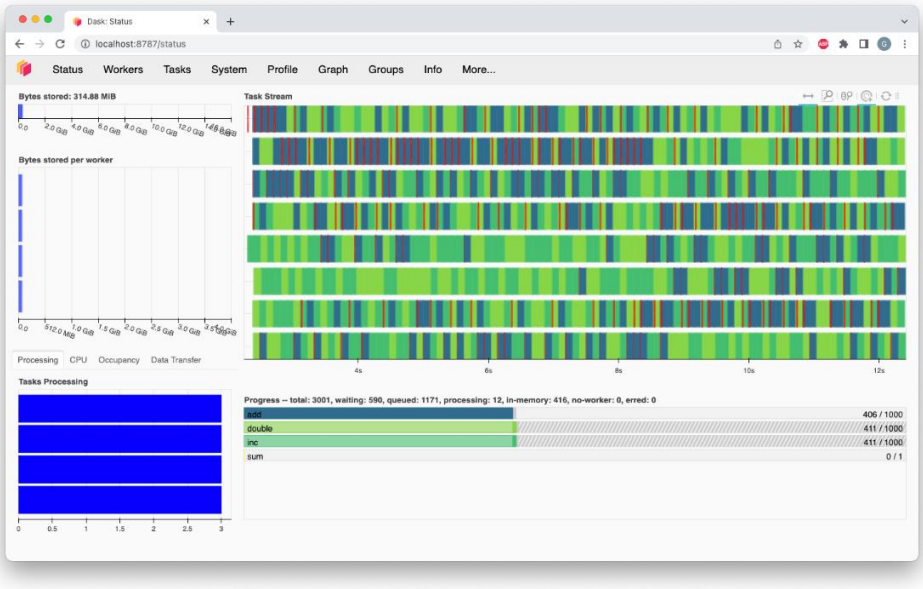


Figure 2-4: Dask dashboard [Roc18b]

Dask also offers an interactive dashboard that aids in diagnosing the cluster's status. The typical layout of a dask dashboard is shown in the figure 2-4. The dashboard has numerous plots and tables with real-time data that can be used to examine the operations and improve the code. The dashboards can assist in relearning what is quick and slow as well as how to handle it. Bokeh plots are used in the diagnostics dashboard so that users can interact with the plot elements using features like zoom, hover, pan, tap, etc. Dashboard normally has "localhost:8787" as its default address. [Roc18b] [Sig22]

2.4 DASK for Machine Learning

Scalable machine learning in Python can be achieved by utilizing Dask-ML, which merges Dask with widely used ML libraries such as Scikit-Learn, XGBoost, etc. The following paragraph seeks to delve deeper into the significance of scaling in machine learning explaining reasons why it is necessary. When working with large or complex datasets, two primary challenges arise while training the ML model.

The first one is Compute bound. This occurs when the ML models become too large or complex, resulting in slower performance during model training, prediction, or evaluation

steps and longer computation time. This challenge is also known as compute bound. To overcome this challenge, one can use Dask collections or XGBoost DMatrix, in conjunction with Dask Cluster to distribute the workload across multiple machines. Parallelization can also be achieved using joblib backend of dask to execute Scikit-Learn library in parallel processes [Das17]

The second challenge is Memory bound. This occurs when the size of datasets exceeds the available RAM, making it impossible to load the data into NumPy or pandas. This challenge is referred to as scaling data size. To address this challenge, one can use any one of high-level collections like arrays, dataframes, or bags in dask, along with one of estimators in dask ML. [Das17]

Please note that Joblib is intended for parallel computing on a single machine. It can parallelize simple Python functions across multiple cores, making it suitable for Compute-bound tasks. However, this is only a good solution if the training and test data fit comfortably in memory. In case of both bounds it is recommended to use Dask and Joblib together. In such a setup, Dask can be used to distribute computations across a cluster of machines, & then Joblib can be used to parallelize computations on each machine's local cores. [Pel22]

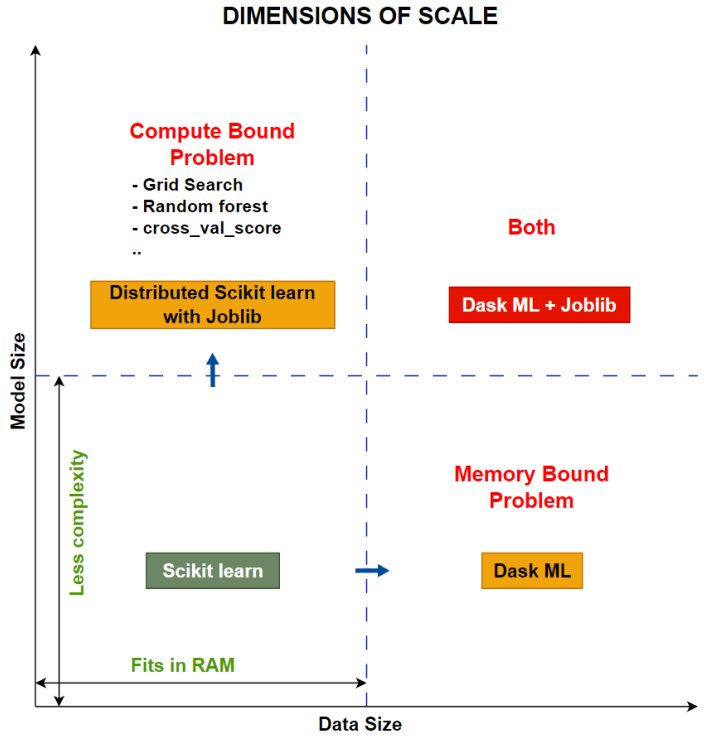


Figure 2-5: Dask-ML Scaling challenges, Source: Adapted from [Das17]

2.5 Migration of Python virtual environment from version 3.6 to 3.9

Python 3.6 was released in December 2016, and since then, several newer versions of Python have been introduced. One of the more recent stable versions is Python 3.9, which was launched in October 2020. End-of-life (EOL) of Python 3.6 was officially announced on December 23, 2021. This means that Python 3.6 will no longer receive security updates or bug fixes, and users were advised to upgrade to a newer version of Python to minimize

risks and potential security vulnerabilities. Upgrading to Python 3.9, can provide many benefits, including improved performance, new features, and better security. However, it presents several difficulties and requires extensive testing and debugging. [Wal21]

During the migration process of the pyMICE package, various challenges were encountered. One of the challenges was the introduction of new syntax features in Python 3.9 that were not present in Python 3.6, which required the code to be updated to use these features or avoid using deprecated syntax. Deprecation warnings were also generated for certain syntax from Python 3.6, which made it difficult to identify real issues in the code. [Lan20]

Python 3.9 included updates to several built-in libraries, resulting in changes to the way certain packages or modules worked. Hence, the code needed to be updated to account for these changes. Additionally, some third-party packages were not compatible with Python 3.9 or required a different version of the package, necessitating the verification of package compatibility. [Lan20]

After migrating the code to Python 3.9, it was necessary to thoroughly test it to ensure that it worked as expected. This involved creating new test cases or modifying existing ones. The performance of the code could have also been affected by several performance enhancements in Python 3.9, and thus it was necessary to understand the functions and mechanism of the pyMICE package to test its performance and make any necessary adjustments to ensure it ran efficiently.

Proper documentation of the changes made during the migration process was essential for user reference. Furthermore, reviewing the output or results of the package was imperative, and in the case of pyMICE, the results could be viewed via MTGUI, a web interface built upon Flask API. Debugging was required due to the upgrade of the Flask library to ensure proper functioning. More details regarding the debugging of package dependencies will be explained in the Implementation section.

2.6 Azure ML Fundamentals

Azure Machine Learning is a cloud service offered by Microsoft, which empowers machine learning engineers and data scientists to construct, train, implement, and manage machine learning models on a large scale. It provides an end-to-end machine learning platform that offers the ability to perform a wide range of tasks such as data preparation, model development, and deployment. It supports popular programming languages like Python and R, and enables connectivity with various Azure services including Azure Databricks, Kubernetes Service, and DevOps. [Mic23d]

Azure ML provides several key benefits that make it a popular choice for data scientists and machine learning engineers. One such benefit is scalability, as users can easily scale up or down their compute resources based on their specific needs. This makes it easier to handle large datasets or complex models. Additionally, Azure ML offers flexibility in building and deploying models, with options ranging from automated machine learning to custom code development. Multiple deployment options are also available, including on-premises, edge, and cloud. [Mic23d]

Another important benefit of Azure ML is collaboration, as it provides a centralized workspace for data scientists and machine learning engineers to work together on machine learning projects. This allows for easier code sharing and access to data. Security is also a key concern for many users, and Azure ML offers a range of security features, such as identity and access management, data encryption, and compliance certifications, that help ensure the security and privacy of user data. [Mic23d]

Finally, Azure Ml also enables connectivity with various services, such as Azure Cognitive Services, Data Factory, and Power BI. This integration allows users to build more complex solutions and leverage additional functionality. In summary, the benefits of using Azure ML include scalability, flexibility, collaboration, security, and integration with other Azure services. [Mic23d]

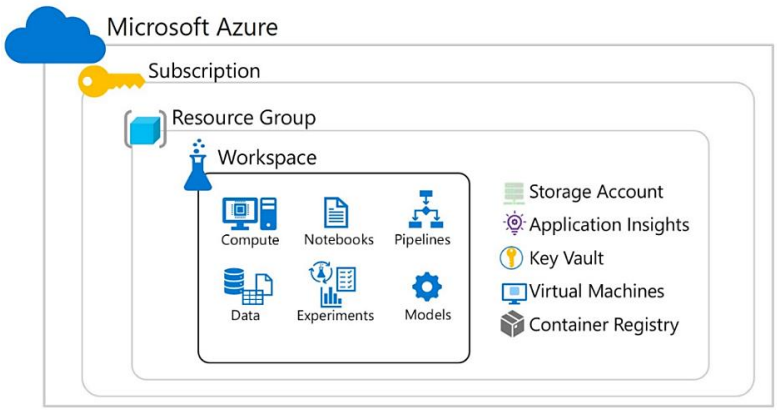


Figure 2-6: Azure ML Workspace [Mic23d]

To effectively use Azure ML, it is essential to understand the different terminologies used in the platform. The first term is a subscription, which serves as a logical container utilized for allocating resources in the Azure ML cloud. Azure ML is one of the many Azure services available in a subscription. [Mic23d]

Another important term is a resource group, which is a logical container used to hold related Azure resources. These resources share the same lifecycle, permissions, and polic-

ies. Resource groups can contain a wide range of resources, including virtual machines, databases, storage accounts, and Azure ML workspaces. [Mic23d]

Azure ML workspaces are containers that hold all Azure ML resources, including experiments, models, data, and compute resources. This centralized location provides a collaborative platform for data scientists and machine learning engineers to work together effectively. [Mic23d]

Compute refers to the processing resources used to run experiments and train models in Azure ML. Azure ML provides a range of compute options, including virtual machines, Kubernetes clusters, and Databricks. Compute can be provisioned on-demand, and users are billed only for the compute resources they use. [Mic23d]

Pipelines are a powerful feature of Azure ML that enables automation of machine learning workflows. Pipelines are composed of interconnected steps that can be executed in parallel

or sequentially. Pipelines can be scheduled to run at specific intervals, triggered by specific events, or run on-demand. [Mic23d]

Data storage is also an essential part of Azure ML, and the platform offers several options for storing and accessing data, including Azure Blob Storage, Azure Data Lake Storage, and Azure SQL Database. Data can be ingested from a variety of sources, including on-premises data stores, cloud-based data sources, and streaming data sources. [Mic23d]

Experiments are sets of steps that train and test a machine learning model. It includes the data, code, and configuration settings. Experiments can be run on different compute targets, and the results can be tracked and compared. The output of an experiment is a machine learning model. [Mic23d]

Models are the output of a machine learning experiment, and they are a mathematical representation of the relationships between the input data and the predicted outcome. Models can be registered, versioned, and deployed to production. [Mic23d]

Jobs are used to execute experiments, pipelines, and other tasks in Azure ML. Jobs can be run on-demand or scheduled to run at specific intervals, and the results of jobs can be tracked and analyzed using Azure ML tools. [Mic23d]

Finally, Azure ML provides support for running experiments and deploying models on virtual machines, which provide a flexible, scalable way to run applications in the cloud. “A container registry refers to a service used for organizing and preserving Docker container images”. Azure Container Registry provides a secure, private registry for storing and sharing container images, while Docker provides a way to package an application and its dependencies into a single container, which can then be run consistently across different environments for the purpose of development, testing, and production. Docker containers are isolated from the host system, which makes them more secure and reduces the risk of conflicts with other applications on the same machine. Docker images are the building blocks of containers and are created from a Docker file, which specifies the application, dependencies, and configurations required for the container. [Mic23d]

3 Implementation (Part I: Supporting AI-driven development with pyMICE)

The first objective involves running pyMICE on a DASK cluster using Python 3.6 on both operating systems Windows and Linux or HPCs. The implementation process for this objective began with getting familiar with the pyMICE package template, named pyTemplate. This included analyzing the package's source code and test modules to gain a better understanding of the existing codebase and its functionality. This step was important before making any modifications to the package. Figure 3-1 illustrates the basic structure of the python project template, which provides an overview of each directory and file in the example.

The "build/" directory is used to build a package and to store the built package after running the python setup.py bdist_wheel command. The "docs/" directory contains documentation for the project, including user guides, API references, and tutorials. The "notebooks/" directory is used to store Jupyter notebooks used for development, exploration, and demonstration. The "recipe/" directory contains recipe files for conda packaging, usually including the ‘meta.yaml’ file that contains information about the package name and version, source code location, build instructions, dependencies, testing instructions, and package metadata such as the package's home page, license, and description.

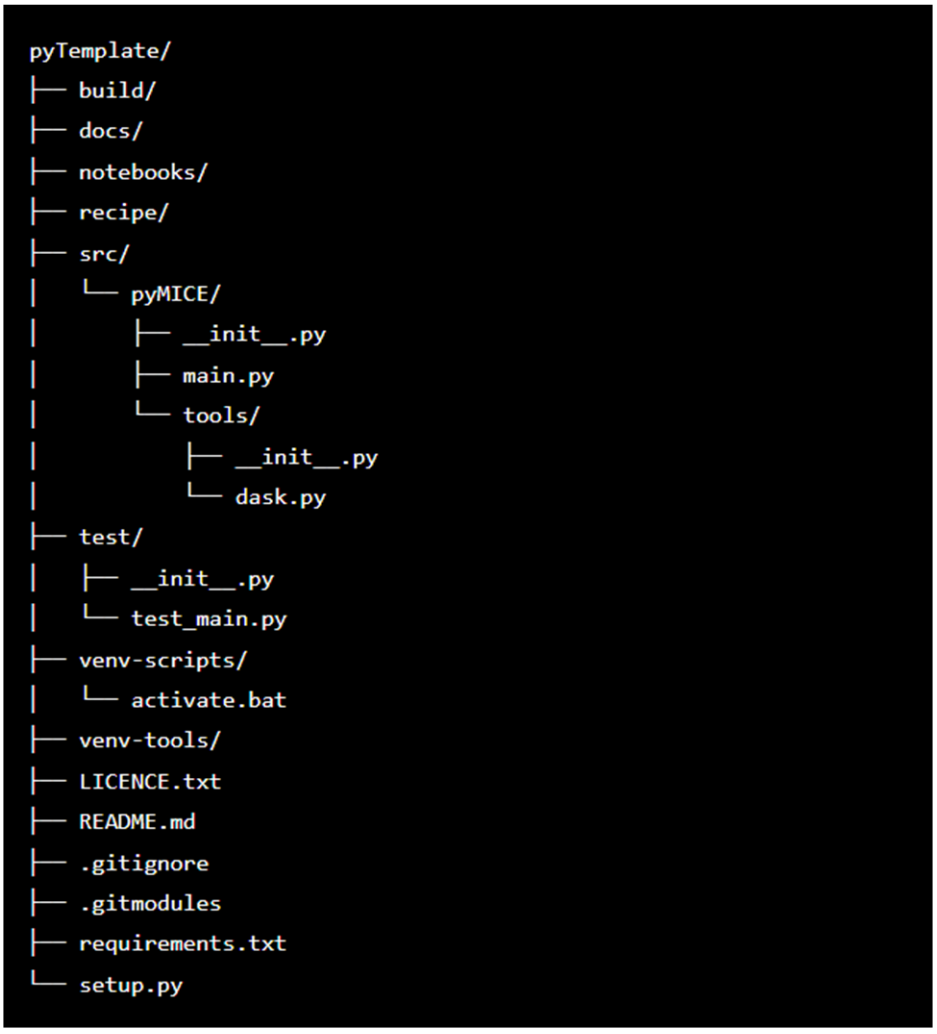


Figure 3-1: pyMICE package template

The "src/" directory contains the main source code for the project, including main.py. It has a folder named "pyMICE" that indicates the name of the package which can be used with the "import" command after installation. The "__init__.py" file inside the folder tells Python that it is a package, which can contain submodules or packages like in this case pyMICE.tools. The "test/" folder is used to store the test code for the project. In pyMICE, unit testing is done using the Pytest library. Each test module in the "test/" folder is designed to test a complete source module in the project. Test methods, which are the test cases to cover the functionality, have the prefix "test_" or suffix "_test" in their names to enable automatic discovery of test cases by Pytest. The "venv-scripts/" directory contains basic scripts to interact with the Anaconda Environment, while "venv-tools/" is a submodule used to create Anaconda Environments automatically. The ".gitignore" file is used to specify which files and directories should be ignored or not committed by Git. The ".gitmodules" file is used to specify the submodule used to create Anaconda Environments. The "LICENCE.txt" file contains the license under which the project is released, and the "README.md" file contains information about the project, including how to install and use it. Finally, the "requirements.txt" file contains the packages required to initialize a new Anaconda Environment, while the "setup.py" file is required to install the package via 'python setup.py install' and contains information about the project, including its name, version, and dependencies.

The second step in the implementation process is understanding package dependencies using an Unified Modelling Language (UML) diagram of pyMICE. Before upgrading a Python package to a new version, it's important to understand the package dependencies. A package dependency in Python indicates the versions of different packages that are compatible with a specific Python package. Dependencies are specified in a package's metadata and typically include the package name, version, and other requirements such as minimum or maximum versions of a package that can be used. It's important to ensure that

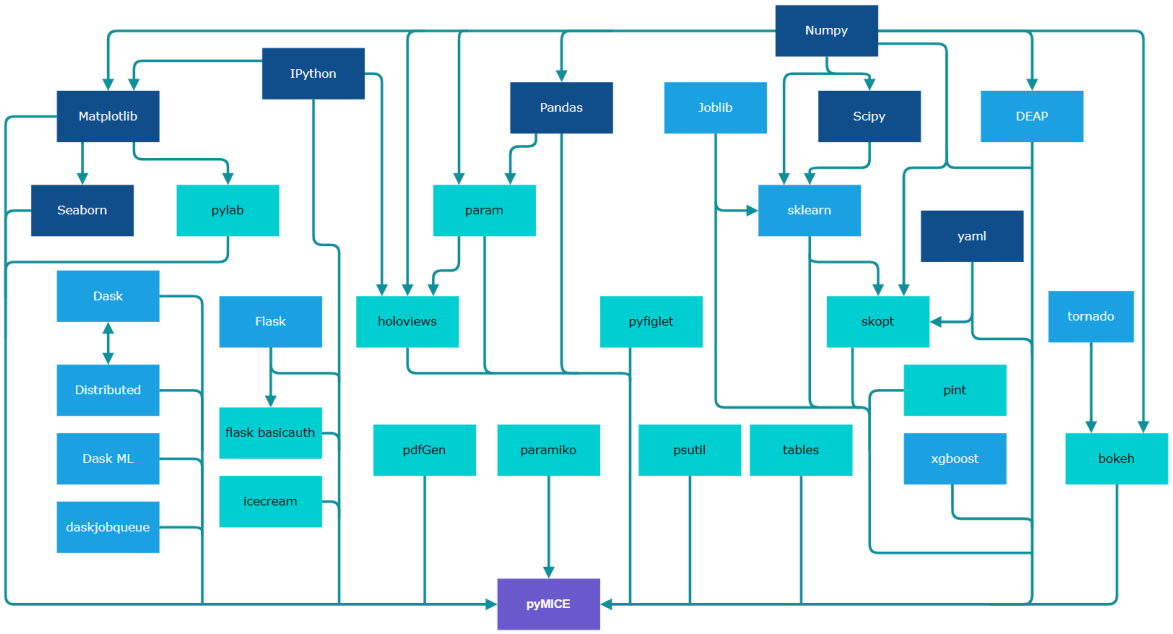


Figure 3-2: pyMICE package dependencies [BHS21a]

all package dependencies are compatible with the new version of Python before upgrading the main package. [Act22]

pyMICE is built upon a lot of packages, ranging from standard libraries such as NumPy, Pandas, and Matplotlib, to machine learning libraries like Scikit-learn, XGBoost, and DEAP (Distributed Evolutionary Algorithms in Python). These libraries are used to build custom pyMICE classes, such as pyMICE.tools for DASK, pyMICE.storagehdf and pyMICE.etl for data extraction and transformation, and pyMICE.analysis.modelling for generating models etc. In Figure 3-2, a flowchart illustrates the hierarchical structure of the libraries that constitute the foundation of pyMICE. The chart lists these libraries in order from top to bottom, and each library has designated minimum and maximum versions that are compatible with Python 3.9.15 and also with every individual library.

During the migration of pyMICE, which has a complex structure with a lot of dependencies, the UML diagram proved to be an extremely useful tool for understanding the package structure and the relationships between its various components. The UML diagram provided a visual representation of the package's classes, attributes, methods, and their interdependencies. It has been useful in identifying potential issues and conflicts that could arise during the migration process, such as dependencies on outdated or unsupported libraries.

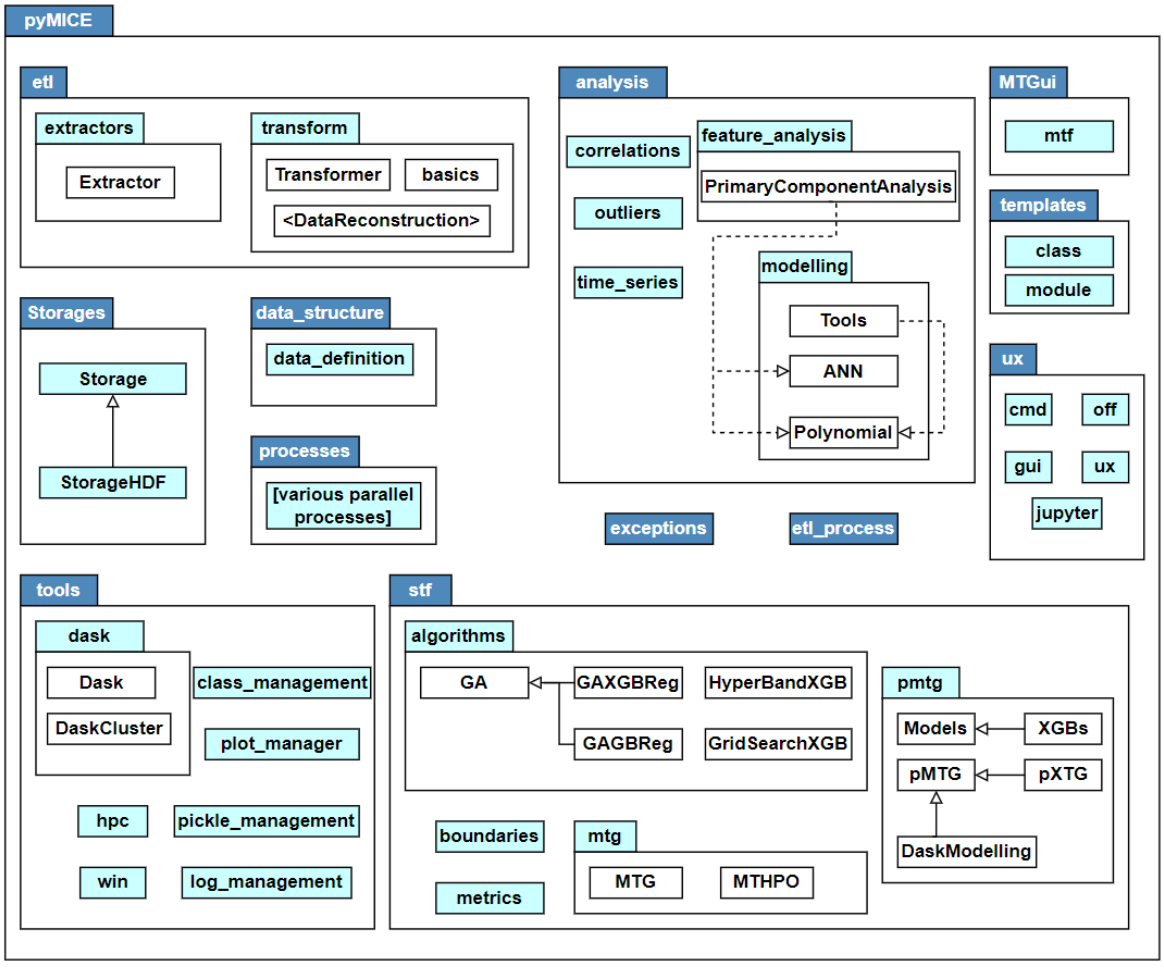


Figure 3-3: pyMICE UML Diagram (Attributes and Methods are hidden) [BHS21a]

The UML diagram in Figure 3-3 depicts the class names of the pyMICE package. During the migration process, particular attention was paid to the 'tools' and 'stf' classes, as they play a vital role in utilizing dask and distributed computing, and handling the libraries associated with Models-to-go (MTG) script, respectively. These two classes are integral to launch pyMICE and perform the model selection process in the knowledge discovery framework. The MTG process will receive a more detailed explanation in subsequent sections.

The next task in the implementation was to set up pyMICE repository on GitHub using Git. Using version control tools such as Git is essential during the migration process of the package for tracking changes and managing collaboration. To set up the package on GitHub, a new repository was created and cloned onto the local machine using the command ‘git clone <repository URL>’ through Gitbash. Git Bash is a software that provides an emulation layer for the Git terminal, with BASH being an acronym for Bourne Again Shell. A shell is a program with command line terminal that allows users to interact with an operating system by entering text commands. Git Bash installs Bash, Git utilities and Git on a Windows operating system. [Kin16]

After every modification, the package files were added, committed, and pushed to the repository using git commands: “git add <filename>”, “git commit -m "comment for reference"’, and “git push origin main”. Pull requests are used to merge modifications back into the main codebase, enabling developers to submit changes and have them reviewed. Once the pull request was reviewed and approved, the changes were merged into the initial branch using the git commands “git fetch upstream”, “git checkout main”, and “git merge upstream/main”. [Git08]

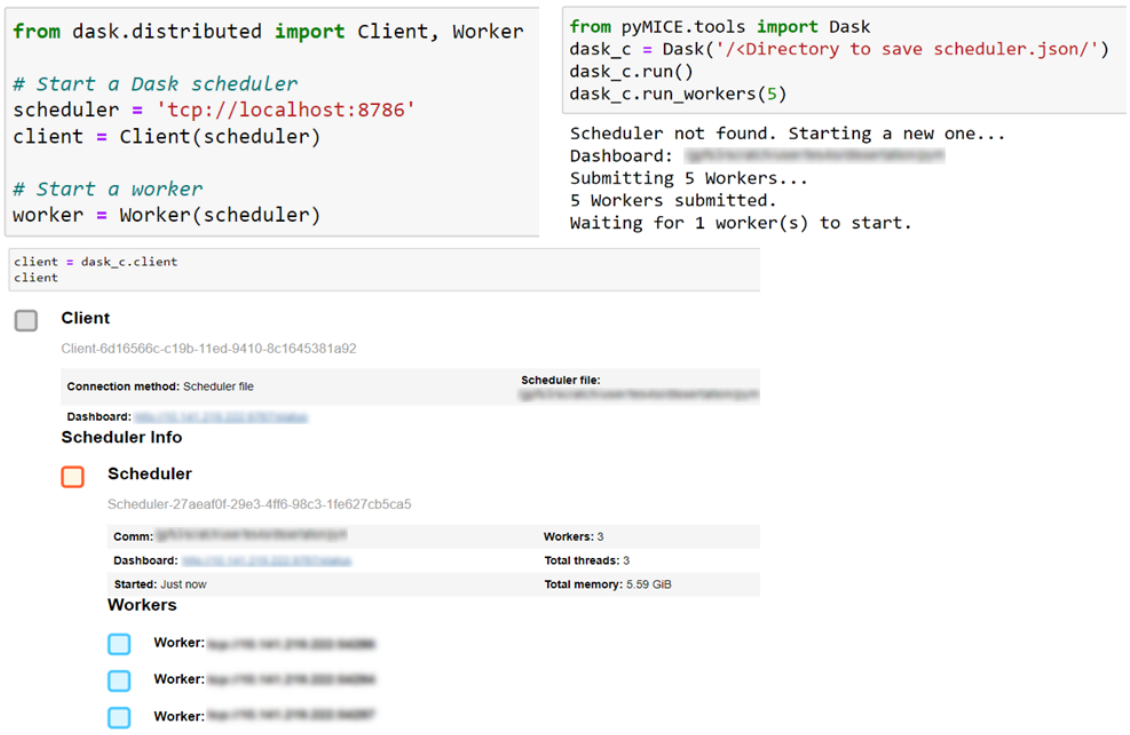


Figure 3-4: Running Dask on Local cluster

```python
#Numpy version
import numpy as np
x = np.random.normal(size=(1000 , 10000))
y = np.random.normal(size=(10000 , 1000))
result= x.dot(y)
print(result)
```

```
[[ -23.6690301   -48.69610283  160.95095097 ...   30.02553071
   -48.0558932    30.30677283]
 [  -2.38413416  -90.48267855  -16.08831868 ... -164.35312876
   -74.3963775    88.06162012]
 [ 113.32730745   61.47539111   53.85287635 ... -199.44161747
    -3.89435334  125.61076438]
 ...
 [ -26.77196921   -9.0247514   138.27057555 ...  157.40326753
   -48.16094656  -69.62026714]
 [ -27.82167037 -102.86817944  135.88904648 ...  -91.52968234
   113.8936915  -218.88119993]
 [ -16.97796848   96.84840646 -142.07020205 ...  101.97395302
    -1.08490851  -80.51782454]]
```

```python
#Dask version
import dask.array as da
x = da.random.normal(size=(1000 , 10000), chunks=1000)
y = da.random.normal(size=(10000 , 1000), chunks=1000)
result= x.dot(y)
```

```
result
```

	Array	Chunk
Bytes	7.63 MiB	7.63 MiB
Shape	(1000, 1000)	(1000, 1000)
Dask graph	1 chunks in 6 graph layers	
Data type	float64 numpy.ndarray	

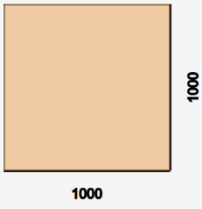


Figure 3-5: Example 1: Numpy vs Dask Array

The subsequent step in the implementation process entailed working with Dask, where basic Dask examples were executed and machine learning tasks were performed using dask. Dask can be imported by using the command 'from dask.distributed import Client, Worker'. Alternatively, Dask can also be launched through the command prompt by running the 'dask-scheduler' command with arguments like --host, --port, --dashboard-address. The pyMICE employs a class called pyMICE.tools to launch Dask through the command prompt.

```python
import joblib
from sklearn.datasets import load_digits
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC
import numpy as np
digits = load_digits()

param_space = {
    'C': np.logspace(-6, 6, 13),
    'gamma': np.logspace(-8, 8, 17),
    'tol': np.logspace(-4, -1, 4),
    'class_weight': [None, 'balanced'],
}

model = SVC(kernel='rbf')
search = RandomizedSearchCV(model, param_space, cv=3, n_iter=50, verbose=10)
```

```python
with joblib.parallel_backend('dask'):
    search.fit(digits.data, digits.target)
```

```
Fitting 3 folds for each of 50 candidates, totalling 150 fits
```

```python
print("Best parameters: ", search.best_params_)
print("Best validation score: ", search.best_score_)
```

```
Best parameters:  {'tol': 0.01, 'gamma': 0.001, 'class_weight': None, 'C': 10.0}
Best validation score:  0.9760712298274902
```

Figure 3-6: Example 2: Machine learning with Dask

In the first example (Figure 3-5), a matrix multiplication of two random arrays is computed using NumPy's standard method first, and then using Dask. The figure below (Figure 3-5) illustrates that, unlike NumPy arrays, Dask arrays do not immediately compute results. Instead, they first partition the array into smaller chunks and construct a DAG. The final result can be computed using the "compute()" method. NumPy and Dask code share a similar structure. However, Dask code has the advantage of being able to run in parallel and out-

of-core, enabling the user to perform computations on datasets larger than the memory capacity.

The second example (Figure 3-6) shows a machine learning application using Dask. Here, Joblib is used to parallelize the process of hyperparameter tuning on an SVM classifier using the digits dataset. By using Dask, the code can run the hyperparameter search more efficiently by using multiple workers to evaluate different sets of hyperparameters in parallel.

The next step was to generate .h5 file from raw data using Storage HDF class of pyMICE. This was the first step towards applying pyMICE's knowledge discovery process on the use case- Automotive Camera's joining process. It involved leveraging the data transformation and storage features of pyMICE [BHS21a] . This process is based on an Extraction, Transform, and Load (ETL) principle [Inm05] . ETL is a process of converting raw data into a format that is suitable for analysis. This process involves extracting the data from various sources, transforming it to match the required format, and then loading it into a data warehouse or destination system [Inm05] . Hierarchical Data Format (HDF5) is a file format which provides a flexible and efficient way of storing and managing large datasets [PFH11] . In this case, the Storage HDF class of pyMICE was utilized to apply the ETL process on raw data and saving the output file in .h5 format. The raw data in this project consists of two CSV files: camera_joining_input.csv and camera_joining_output.csv.

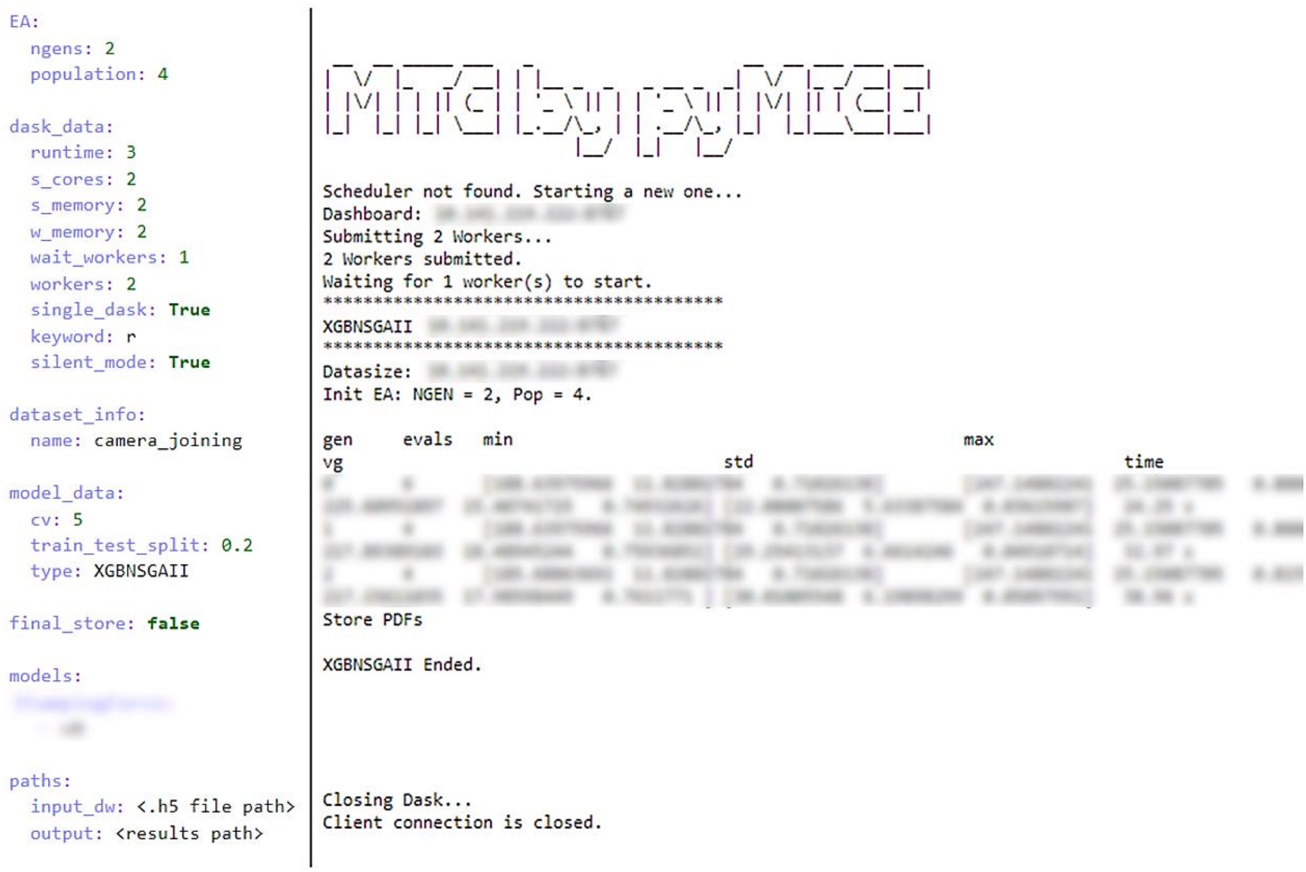


Figure 3-7: Running MTG through .yaml config (Left: .yaml config, Right: MTG Results Overview)

The former file contains input parameters, including design and process variables such as displacement, material properties, and dimensions of automotive camera components. The latter file contains desired output parameters, such as force and deflection of PCB etc. As mentioned in the introduction, the objective of this application is to optimize the process parameters i.e., minimizing PCB warpage while maximizing other desired outputs.

The concluding task of the first objective entailed running MTG on a DASK local cluster using a .yaml configuration. MTG stands for "Models-To-Go". This step corresponds to the hyperparameter optimization (HPO) of the ML models, which involves the Model Selection process using Evolutionary Algorithms (EA) - a family of optimization algorithms inspired by natural evolution principles [BHS21a] . EA generates and evaluates a population of candidate solutions that undergo selection, crossover, and mutation operators to create new generations of potentially better solutions. With time, the population evolves towards better solutions that optimize a given objective function. To run MTG, Yaml configuration requires several inputs, including the Generated .h5 file, Number of generations, Population size, Dask configuration, and Machine learning data like cross-validation and train-test split. [BHS21a]

The second objective involves migrating the pyMICE virtual environment from Python 3.6 to Python 3.9. The first task in this objective was to ensure that the Dask client, scheduler, and all workers were closed properly after each computation. However, the standard methods of closing the client and shutting down the cluster using client.close() and cluster.shutdown() were found to be ineffective when using a distributed.Client instance. As there was no specific client function available for this, a manual approach was required to kill all workers and the scheduler after every computation. Moreover, the challenge was to make sure that the client starts in the same port without switching to a random available port on re-running the process. Improper closure was causing dangling workers to remain active even after the client had been shut down, which was undesirable as these unused workers continued to consume resources. To address these issues, the close() method of the scheduler was used by calling it using run_on_scheduler. By running sys.exit(0), the workers were informed to disconnect and shutdown forcibly, closing all the connections before terminating the process. However, this approach led to multiple errors such as asyncio.cancelled.error, CommClosedError or tornado ‘stream is closed error’ in the client, as the connection was broken without a reply. Therefore, a significant amount of debugging was required to address these errors, which were not catchable using try and except. Additionally, after every interruption, it was necessary to delete the scheduler.json file and manually kill the PIDs. The objective was achieved by creating a separate close function for closing the dask client considering both operating systems windows and Linux or HPC cluster. The process is represented as a flowchart in Figure 3-8. Here “HPC” is a .py script developed as a part of pyMICE.tools which provides a set of functions to interact with a High-Performance Computing (HPC) system [BHS21a] . The script is designed to interact with an HPC system using the command-line interface and SSH connections, allowing users to submit jobs and monitor their status. [BHS21a]

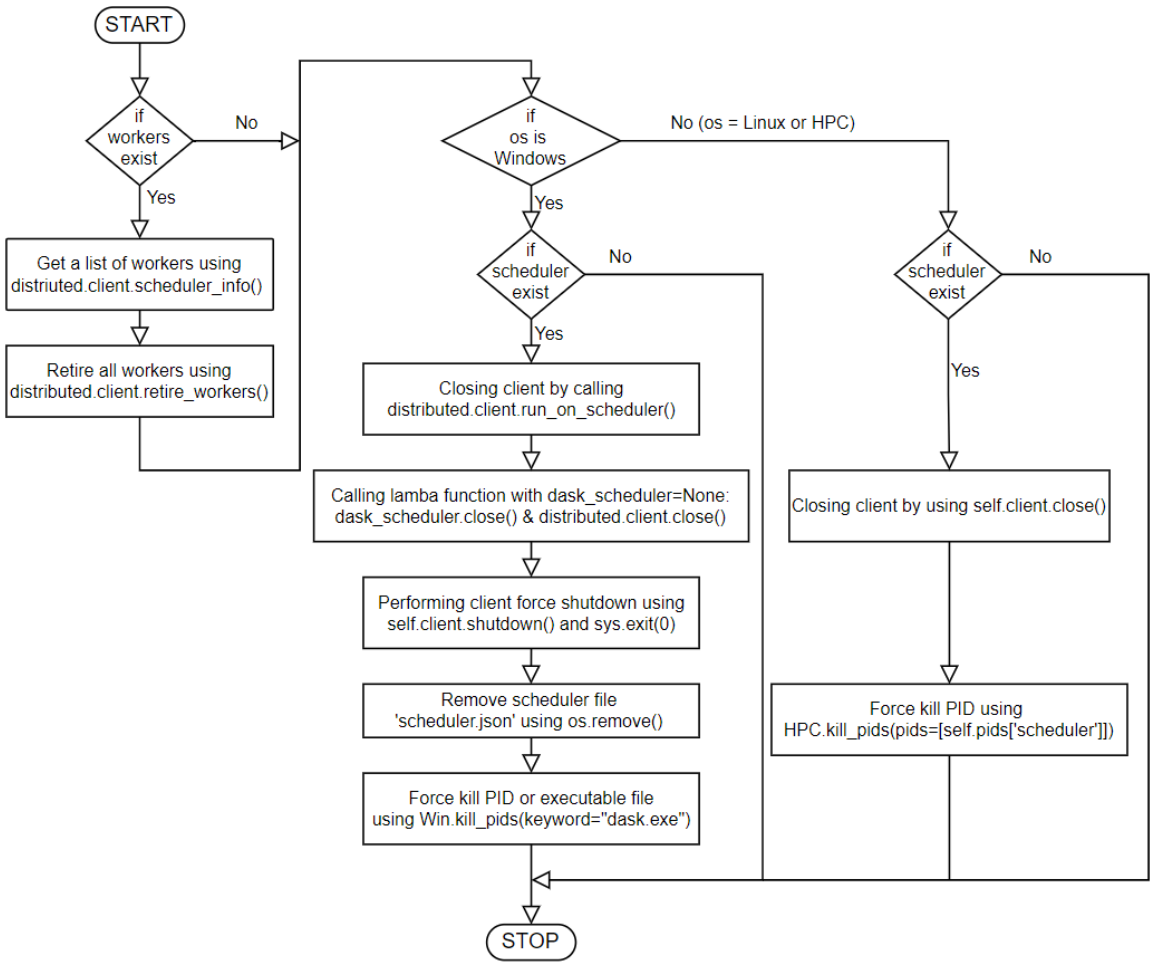


Figure 3-8: Flowchart of function for closing Dask client

Having successfully tested pyMICE on a Python 3.6 virtual environment, the next step is to save the results for comparison with the results obtained after migrating to Python 3.9. To ensure a fair performance comparison between the two python versions, it was necessary to set a standard yaml configuration for both versions. In this regard, it was decided to consider 4 number of generations and 4 population size of evolutionary algorithms as the standard configuration. To ensure that no data was lost during the migration process, a backup of the results was created and then merged on Github.

The next step in the process of migration is to upgrade virtual environment to Python 3.9 and upgrade all the dependencies to their latest version. Python virtual environments enable the installation of Python packages in a separate location that is isolated from the local system. Virtual environments simplify the process of defining and installing packages that are tailored to specific project's needs. To ensure that the project functions seamlessly with the specified package versions, exact version numbers can be defined for the required packages using a requirements.txt file. [LJ16]

This step of the project involved upgrading the virtual environment from Python 3.6 to Python 3.9, along with updating all dependencies to their latest versions. Debugging and testing were performed, which revealed several errors. The first one is syntax changes. For

example, Seaborn underwent a syntax change from version 0.10.0 to 0.12.1, which involved converting positional arguments of x and y in line and scatter plots to keyword arguments. The second one is deprecation warnings, for e.g The parameter 'eval_metric' in XGBoost underwent deprecation from version 1.0.1 to 1.7.2. This parameter was an attribute of the xgboost.sklearn.XGBModel.set_params() method and required modification to switch to a non-deprecated metric 'rmse'. Another example can be seen in the case of a distributed library, where a time-out error occurred due to a race condition after the client.retire_workers was performed, and the worker had been removed from the scheduler's perspective, but communications with the worker were still open. This error was due to the deprecation of the "reconnect" attribute from distributed.Worker, and it was resolved by upgrading Dask and Distributed libraries. Moreover, to avoid confusion and cluttering of logs, it was necessary to suppress the warnings and tracebacks. The table A-1 in appendix shows the list of few major version upgrades of pyMICE dependencies.

Once the dependencies were upgraded, the next task was to restrict Dask logging to specific events by changing the default logging level from 'info' to 'critical'. For that, the .config

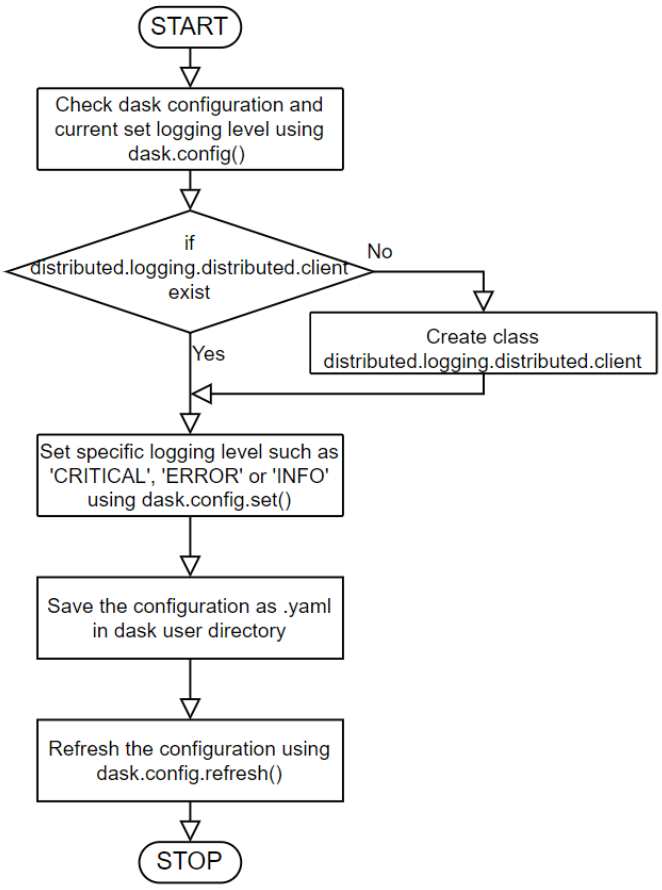


Figure 3-9: Managing Dask logging level

class of Dask was used. Figure 3-9 shows the flowchart for the code created to manage the logging level of Dask. By setting the logging level to 'critical', only critical errors can be recorded. The Dask.config.set() method was used to accomplish this task. Additionally, the logging configuration is saved to a YAML file located at '.config/dask/pyMICE_logging.yaml'

in the user's home directory. Finally, the configuration was refreshed to ensure that any changes made to the configuration will take effect while loading the Dask client. Overall, this code provides a useful approach for setting up and personalizing logging for Dask.

Having successfully performed debugging and testing, the final step was to run MTG with the standard yaml configuration with 4 generations and 4 population size that was set in the step 3.2.2 and then analyzing the performance of optimization results from both virtual environments. The table 3-1 shows the percentage increase in accuracy values due to the migration from Python 3.6 to 3.9. 'gen' refers to the generation number of the genetic algorithm. 'Min%' refers to the percentage increase in minimum accuracy found in that generation. 'Max%' refers to the percentage increase in maximum accuracy found in that generation. 'std' refers to the standard deviation values. 'Time' refers to the amount of time taken to run that generation of the genetic algorithm.

Table 3-1: Percentage increase in accuracy values due to migration of pyMICE

gen	Min%	Max%	Avg%	Std%	Time%
0th	0	0	0	0	27.28
1st	10.98	0	4.25	-68.41	15.11
2nd	12.03	0	5.70	-74.66	13.68
3rd	12.03	0.045	6.02	-71.14	-1.89
4th	12.07	**0.045**	**5.97**	-71.00	0.57
					7.67

The results from table 3-1 reveals a marginal performance improvement due to package migration. Specifically, there is a 5-6% increase in the average accuracy and a slight improvement in maximum accuracy. However, the total computation time to execute the algorithm is also increased by 7.67%.

The third objective in the implementation process involves porting pyMICE to Azure ML cloud. The first task in this objective was to set up Azure by creating resource group, workspace, and compute cluster. To set up Azure, the first step involved creating a resource group, which served as a logical container for Azure resources. This required a subscription for Azure ML service and a location. The resource group, workspace, or compute cluster could be created through the Azure ML portal using the GUI or through Azure Python SDK using Python code in a notebook. Once the resource group was created, the next step was to create a workspace, which required necessary information such as subscription, resource group, workspace name, and region. A workspace was essential for creating a compute cluster and the Azure ML environment, managing, and deploying machine learning models. The next task was to create a compute cluster. As the project involved working on DASK, a virtual network and its subnet needed to be set up first. The virtual network was created using the Azure portal by specifying the private IP. Once the virtual network was created, the compute cluster could be created using the necessary information such as compute name, virtual machine size, and cost options. Additionally, the previously created virtual network and subnet, minimum and maximum nodes, and scaling settings could be specified

in the advanced settings of the compute cluster. Initially, the minimum nodes were set to 0 to work with DASK on a local cluster only.

The next step in the porting process is to execute dask on single node or single machine in Azure ML local environment. Azure Machine Learning provides encapsulated environments for machine learning training, which are broadly categorized into curated, user-managed or local environments, and system-managed. The curated environments are pre-built in Azure ML storage and can be accessed in the workspace, containing collections of Python packages and settings that facilitate the setup process for various machine learning frameworks, enabling faster deployment time. On the other hand, in user-managed or local environments, the environment setup needs to be done manually, including installation of every package that the training script needs on the compute target along with any dependencies required for model deployment. Local environments, similar to virtual environments installed on the local disk, offer full control over development environment and dependencies and can be run through a terminal or any integrated development environment (IDE), but are limited to testing purposes only and cannot be shared with other users. In system-managed environments, the Python environment is managed by conda, and a new conda environment is created based on the given conda specification on top of a base docker image. [Mic23a]

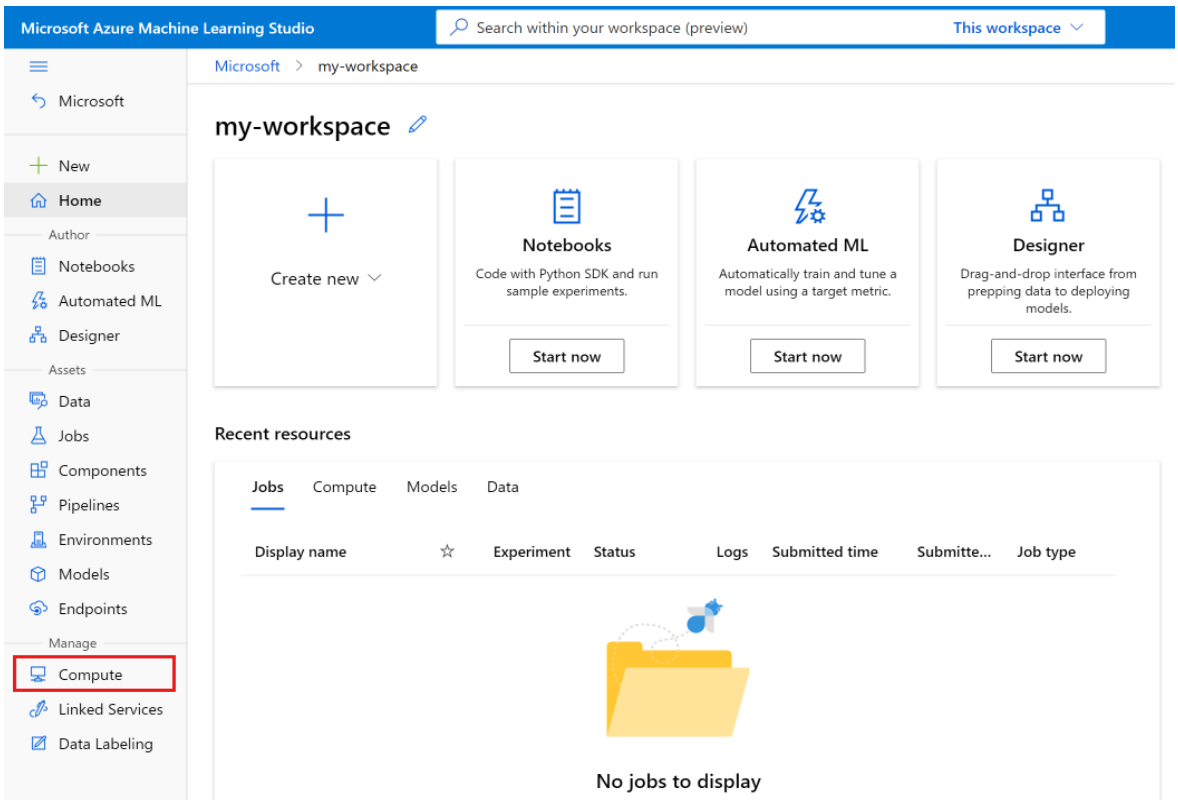


Figure 3-10: Microsoft Azure ML Studio UI

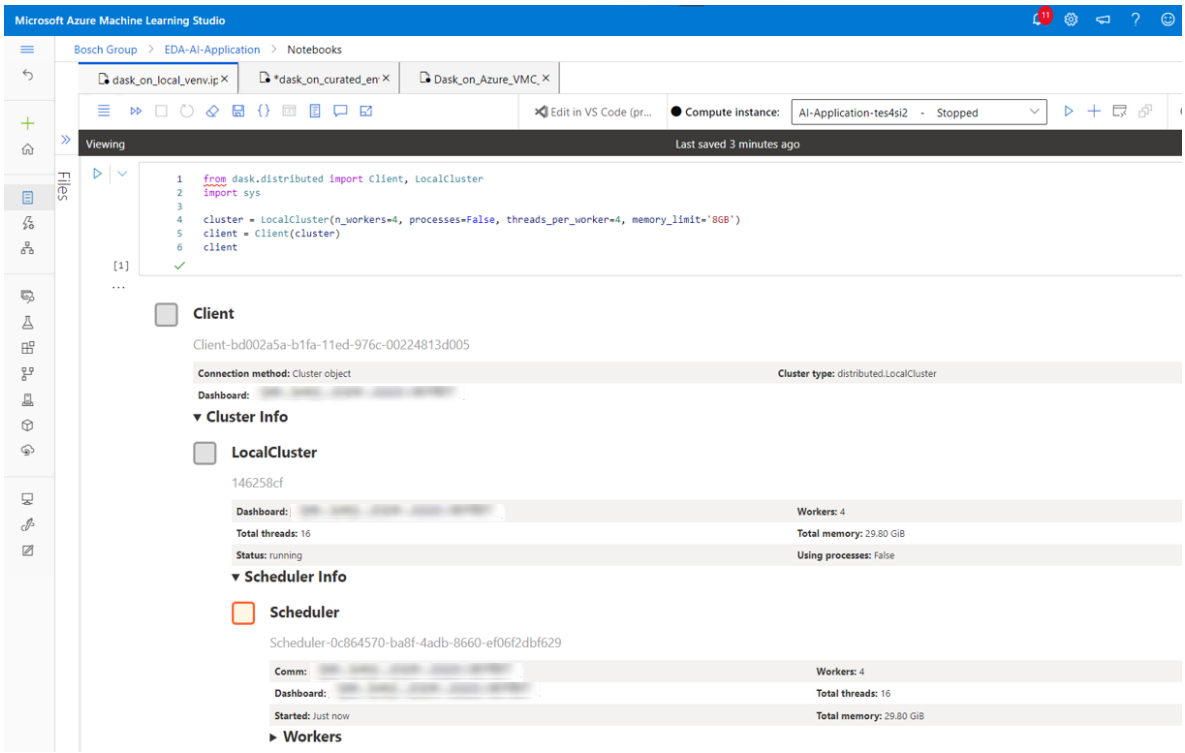


Figure 3-11: Running DASK on single node on Azure ML Local environment

To begin with, the initial task was to conduct a test of DASK on Azure ML Local environment. This involved setting up the necessary dependencies and packages for DASK, and configuring the Azure ML local environment accordingly. The environment was set up manually, including the installation of every package that the DASK script required such as dask, distributed, dask-ml, dask-job-queue, joblib and scikit learn on the compute target. Once the environment was set up, DASK was run on a single node with multiple workers using Local cluster, allowing for distributed computing on a local machine.

The next step in the porting process is to create a custom curated Azure ML environment using Docker Image and Azure container registry. Azure ML's curated and custom curated environments offer a consistent and reproducible environment across multiple machines, essential for collaboration with teams and reproducibility [Mic23a] . Moreover, they enable scaling machine learning workflows with Azure ML's distributed computing capabilities [Mic23a] . The second step in porting involved creating a custom curated environment using Docker image and Azure Container Registry in Azure ML. It required defining a set of instructions in a Dockerfile, which specified the base image, the required packages and dependencies of pyMICE, and specifying Azure ML container registry to push the Docker image to the registry [Mic23b] . The custom curated environment was then created in Azure ML by referencing the Docker image from the container registry.

The next task involved running a Dask script on the Azure Machine Learning platform through ScriptRunConfig. The code in Figure 3-12 depicts the process. Initially, the AzureML workspace was loaded using the from_config() method, which read the configuration

file and loaded the associated workspace. Subsequently, the curated environment "AzureML-lightgbm-3.2-ubuntu18.04-py37-cpu" with version 49 was loaded. To execute the script in this environment, it was necessary to create a compute target called "mycompute" using the ComputeTarget class. This target is where the script will run. The code then created an experiment named "myexp" using the Experiment class. [Mic23d]

Next, a ScriptRunConfig object was set up to specify the script to run, the compute target to use, and the environment to use. The tested script 'dask1.py' is shown in the appendix section. The script contains an example of machine learning code that performs a GridSearchCV (Cross validation) using DASK client. Finally, the ScriptRunConfig was submitted to the experiment using the submit() method. This method executed the script on the 'mycompute' compute target and recorded the outcomes in the specified experiment. The results of the experiment can be seen in the Output+Log section of an experiment (displayed in the figure 3-13). The latter code shows the same script being run through Azure ML pipelines. [Mic23d]

```python
from azureml.core import Workspace
from azureml.core import Environment
from azureml.core import Experiment
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core.runconfig import RunConfiguration
from azureml.core.compute import ComputeTarget, AmlCompute

# Load the workspace
ws = Workspace.from_config()

# Load the environment
env = Environment.get(workspace=ws,
name="AzureML-lightgbm-3.2-ubuntu18.04-py37-cpu",version=49)

from azureml.core import ScriptRunConfig, Experiment

# create a compute target
compute_target = ComputeTarget(ws, "mycompute")

# Running a Script through ScriptRunConfig

exp = Experiment(workspace=ws, name="myexp")
# configure and submit your training run
config = ScriptRunConfig(source_directory='script',
                         command=['python', 'dask1.py'],
                         compute_target=compute_target,
                         environment=env)
script_run = exp.submit(config)
```

```python
# Running Script through Azure ML Pipelines

from azureml.pipeline.core import Pipeline, PipelineData
from azureml.core.runconfig import RunConfiguration
from azureml.pipeline.steps import PythonScriptStep

# create a new runconfig object
runconfig = RunConfiguration()
runconfig.environment = env

pipeline_step = PythonScriptStep(
    source_directory='script', script_name='my-script.py',
    #arguments=['-a', param1, '-b', param2],
    compute_target=compute_target,
    outputs=[output_data],
    runconfig=runconfig)

pipeline = Pipeline(workspace=ws, steps=[pipeline_step])
pipeline_run = Experiment(ws, 'my_pipeline_run').submit(pipeline)
```

Figure 3-12: Running a .py script (Left: Through ScriptRunConfig, Right: Azure ML Pipelines)

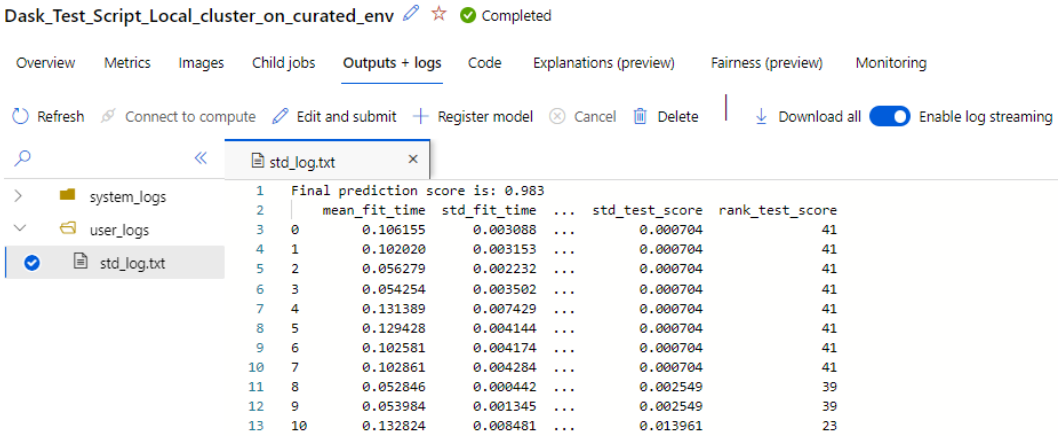


Figure 3-13: Results of Dask test script run

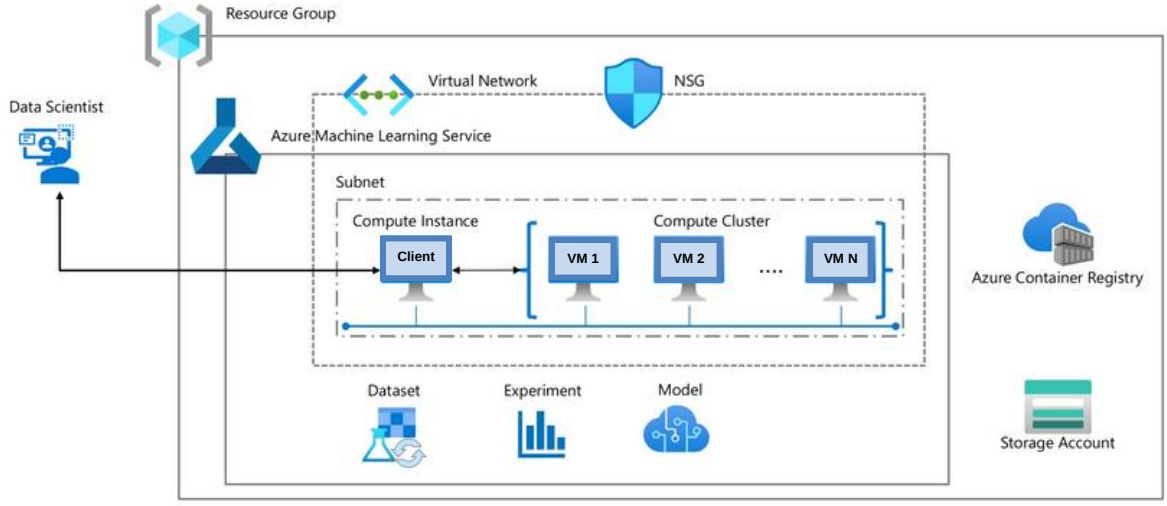


Figure 3-14: Overview: Running DASK on Azure VMC Cluster, Source: Adapted from [Ngu21]

The next step in the porting process is to run DASK cluster on multiple nodes with Azure Virtual Machine Cluster (VMC). The figure 3-14 presents an overview of the process followed while setting up and running a dask cluster on Azure ML cloud with multiple nodes or virtual machines (VMs). Initially, a virtual network, network security group, subnet, and compute cluster with multiple nodes were needed to be established. It is important to note that, Azure ML incurs additional costs when using multiple nodes for computation. To set up dask, the destination port range was manually set to the default dask ports (8786-8787) to allow the scheduler and workers to connect to the cluster on port 8786 of any provided IP address. The code shown in the figure 3-15 uses the AzureVMCluster class from the dask_cloudprovider.azure module to create an instance of the cluster. The parameters specified in the code include the resource_group, vnet, security_group, location, subscription_id, and n_workers. The n_workers attribute of AzureVMCCluster was set to 4, creating 4 workers on 4 VMs i.e., a single worker per virtual machine. The next challenge was to handle dependencies on multiple VMs. The same version of Dask and Distributed were needed to be installed on individual nodes. For that, it was important to ensure that the environment is consistent across all the VMs. To set same curated environment for all VMs in the Azure VMC cluster, a custom image was created with the necessary dependencies and packages. The process involved creating VMs using 'Scale set instances' option in the Azure Portal. Once the VMs were created, Dask was used to distribute computations across the cluster, and the custom image's environment was used by all VMs in the cluster. [Das23]

Once the Azure VMC Cluster with DASK was set up, the final step in the project was to execute pyMICE on the cluster with all the necessary dependencies. As pyMICE is an internal Bosch project, it was crucial to install it securely and privately within the Azure Machine Learning platform. It was needed to restrict the public access and use a curated repository of packages stored within a Bosch enterprise firewall.

Azure ML provides a secure approach to install private packages using wheel files. This is achieved using the static Environment.add_private_pip_wheel() method which permits the

addition of a private package to the workspace, which is well suited for development and testing purposes. Firstly, a .whl file for the pyMICE package was created by running 'python setup.py bdist_wheel', followed by adding it to the default workspace using the add_private_pip_wheel() method. Subsequently, a new private environment, named "my_private_env," was created, and the pyMICE package was securely added to it using CondaDependencies. Finally, a test script was executed using ScriptRunConfig, which tested the ability to import the pyMICE package and run it alongside its dependencies. [Mic23c]

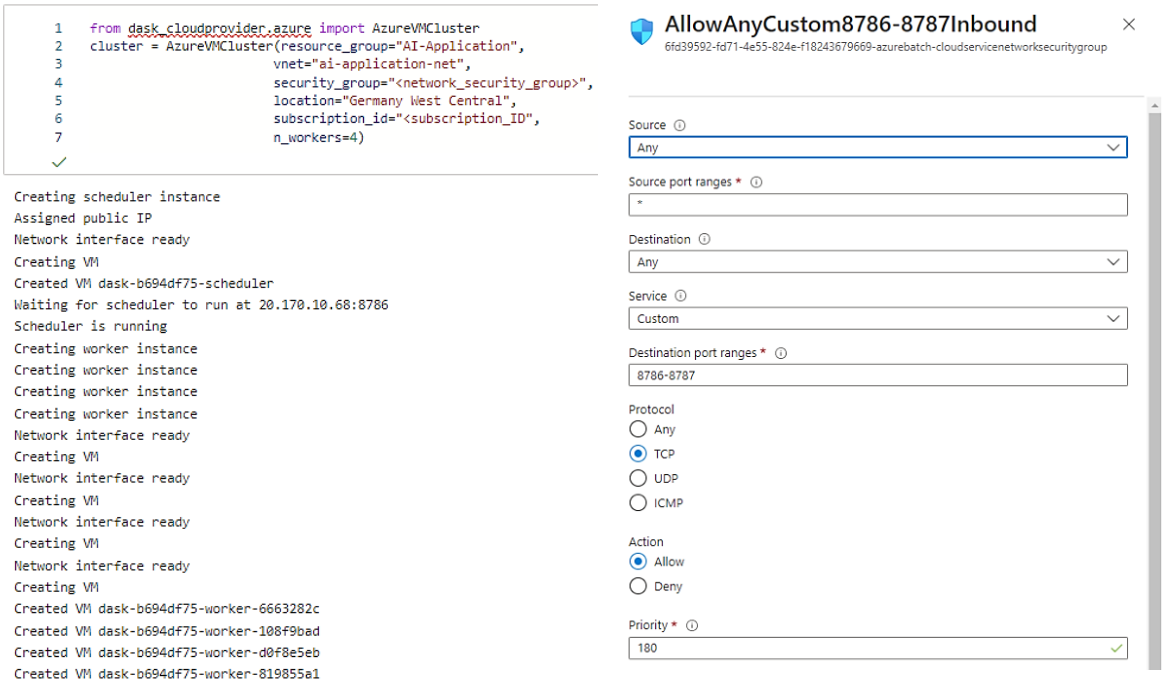


Figure 3-15: Implementation: Running DASK cluster on multiple nodes

4 Workflow and Implementation (PART II: ADAS Vehicle Sensor Data Analysis)

This section focuses on the implementation of part II 'ADAS Vehicle Sensor Data Analysis', with the primary aim of analyzing sensor data from vehicle measurements that consist of over 20 sensor positions per vehicle. Through this analysis, valuable insights can be gained into various aspects such as environmental loads within the vehicle, vehicle driving behavior, and the correlation between environmental measurements from different mounting positions within the vehicle with respect to the driving phase. The project has accomplished several essential tasks that are illustrated in the workflow presented in figure 4-1.

To begin with, the first step in the implementation process is to read and extract data from .mat file in python 3. This involved converting the file into a Python-readable format. This can typically be achieved using libraries such as scipy or mat73. However, it's worth noting that for .mat files with version 7.3 or higher, scipy converts the file into an HDF format instead of a dictionary. As a result, the h5py module must be used instead. To circumvent this issue, one can either downgrade the MATLAB version to 7.0 or lower, or use mat73 instead. In the current project, both options are provided by selectively changing the attribute value according to the input file version, allowing the .mat file to be read with scipy as well as mat73.

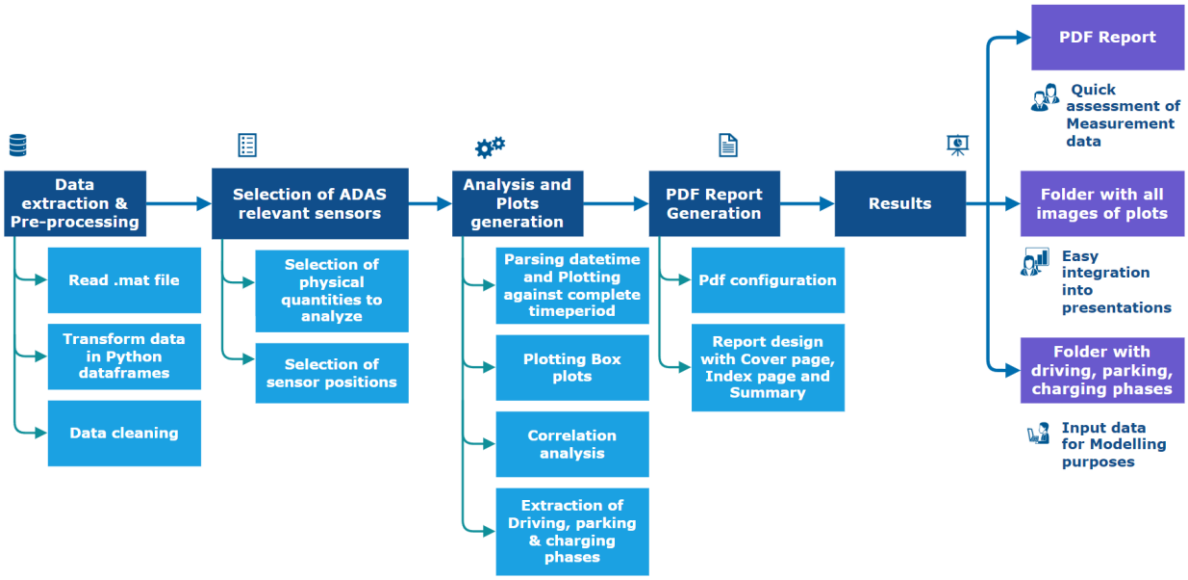


Figure 4-1: Workflow: ADAS Vehicle Sensor Data Analysis

After successfully converting the data into a Python-readable format, the subsequent step involved structuring the data using Pandas dataframe. The relevant columns were extracted from the generated dictionary, filtered, and sorted according to specific conditions, and indexed to facilitate easy retrieval and manipulation. Finally, the structured raw data was saved in the form of a dataframe for further analysis.

The next step in the data processing pipeline was data cleaning. This involved addressing missing values by either dropping rows with NaN values or performing interpolation to fill in

gaps where appropriate. Missing values can arise from various sources, such as errors during measurement, data transmission issues, communication errors, or sensor malfunction. To ensure the accuracy of the subsequent analysis, it was critical to handle these missing values appropriately. For datetime data, it was necessary to drop rows with NaN values, whereas for datasets with a large number of missing values, interpolation techniques were applied to fill in the gaps.

The subsequent step was the selection of physical quantities to analyze, which required providing a user interface for the user to choose the relevant physical quantities, such as temperature, humidity, and dew point. This process involved filtering the data by extracting columns with the selected quantities only. Despite appearing straightforward, it posed some challenges such as indexing and filtering. The reference list or sensors catalogue used to select sensors was different from the index of the data. Furthermore, some columns had identical names, making it difficult to differentiate them.

```
Selection of sensors from Temperature, Humidity, Dewpoint

Enter the number of parameters to analyze (1 to 3):  1

Enter the name of physical quantity: 1
(Choose from Temperature,Humidity,Dewpoint) Temperature

Sensor positions list for Temperature:
```

	Select_sensors_from
Sensor position 1	1
Sensor position 2	2
Sensor position 3	3
Sensor position 4	4
Sensor position 5	5
Sensor position 6	6
Sensor position 7	7
Sensor position 8	8
Sensor position 9	9
Sensor position 10	10
Sensor position 11	11
Sensor position 12	12
Sensor position 13	13
Sensor position 14	14
Sensor position 15	15
Sensor position 16	16
Sensor position 17	17

```
Enter the sensor position numbers for Temperature (separated by commas)
(type 'all' for all sensors): 1,3,5,7,9,17
```

Figure 4-2: Selection of ADAS relevant sensors

After identifying the relevant physical quantities, the subsequent step was to provide the user with the option to select the sensor positions for analysis. Depending on the type of vehicle and ADAS utilized, sensor positions may vary. For example, sensor positions may include the driver's seat, passenger seat, engine compartment, front camera, rear camera, side mirrors etc. After extracting the list of available sensor positions, the user could choose the desired positions for analysis. This selection process involved filtering the available sensor positions according to the vehicle modules or clusters.

Once the data has been extracted, pre-processed, and specific sensor positions have been selected, the subsequent stage involves conducting a conclusive analysis and generating plots. Before starting analysis it was necessary to parse datetime values. For this, a plotting function was created to parse datetime values from MATLAB timestamps using the Python datetime library. The function was then used to generate plots for each selected physical quantity and respective sensor positions against complete time period of a year. The figure 4-3 shows the plots for temperature and humidity of 17 different sensor positions against complete time period of 12 months.

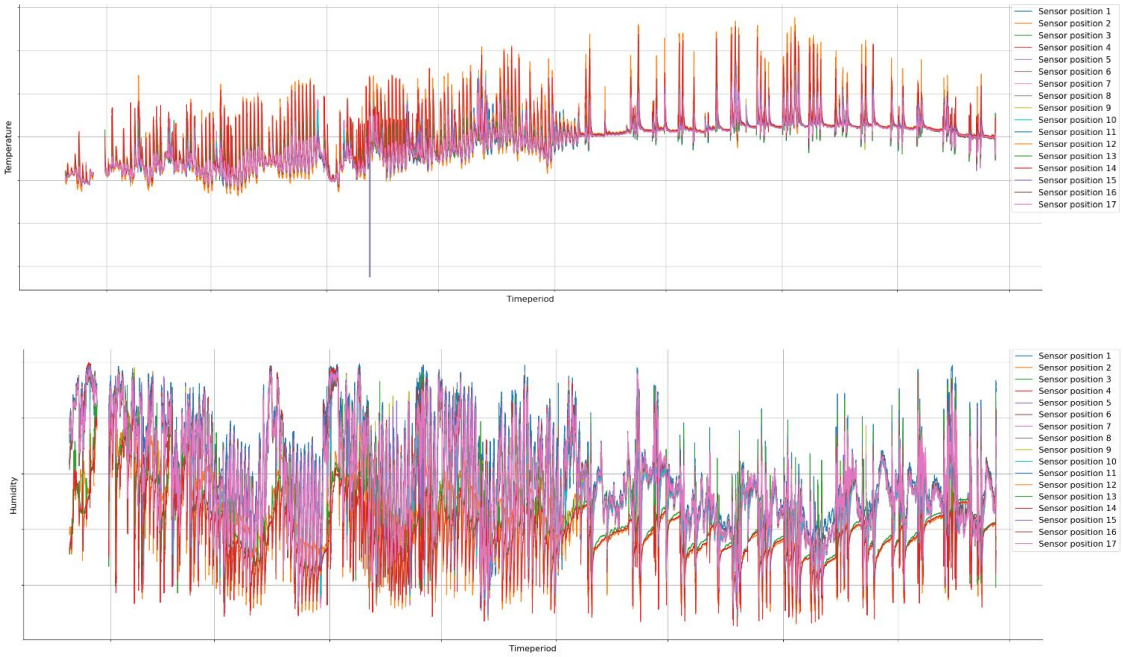


Figure 4-3: Plots of selected sensor positions against complete time period

In the next step, a statistical analysis is performed on the data by calculating the minimum, maximum, mean, and median values and plotted using the same function. The figure 4-4 shows the plot of statistical values for humidity sensor positions.

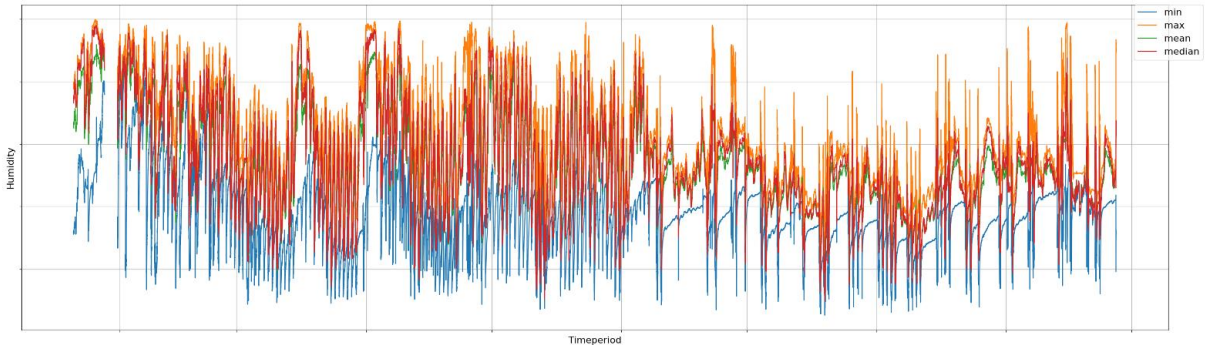


Figure 4-4: Plots of Min/ Max values against complete timeperiod

The next step in the analysis stage involved plotting of box plots for comparison between different physical quantities. Box plots are a commonly used tool in data analysis for visualizing the distribution of a dataset. They are especially useful for comparing multiple datasets and identifying differences in their central tendency, variability, and outliers. A box plot displays the distribution of a dataset using five key statistics: the minimum and maximum values, the first and third quartiles (also known as the lower and upper quartiles), and the median (or the second quartile). The box typically represents the interquartile range (IQR), which encompasses the values ranging from the 25th percentile to the 75th percentile of the dataset. The line inside the box represents the median. The whiskers of the box plot stretch from the box to minimum and maximum values within a certain range, which is often defined as 1.5 times the interquartile range. Data points beyond the whiskers are regarded as outliers and are plotted as individual points. [DS12]

In the context of this project, box plots are used to show comparison of the distribution of different physical quantities across multiple sensor positions, for example comparing the distribution of sensor positions from humidity and dew point. This can help identify patterns and anomalies in the data that may be useful for understanding the environmental influences or for detecting faults or irregularities in the sensor measurements with the help of outliers. In this project, seaborn library is utilized to create boxplots, which offers various options to customize the plot, such as modifying the color and style of the boxes, whiskers, and outliers.

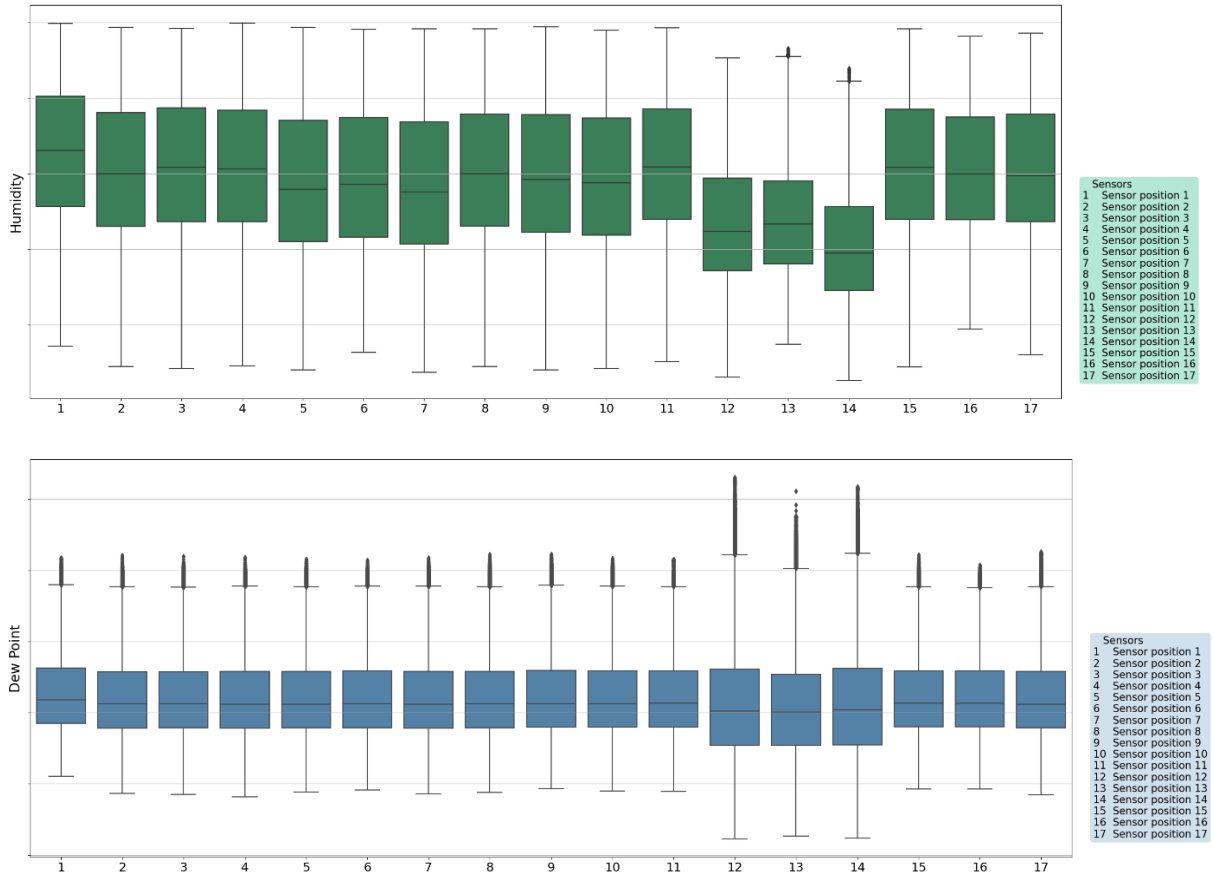


Figure 4-5: Boxplots of Humidity and Dew point sensor positions

The next task was to perform a correlation analysis using seaborn heatmaps. Correlation analysis is a statistical technique used to measure the strength and direction of the relationship between two or more variables [CK68] . In the context of ADAS sensor data analysis, correlation analysis can be used to identify relationships between different physical quantities or between physical quantities and parameters such as speed, acceleration, or ambient temperature etc.

Seaborn provides a convenient function heatmap() for creating correlation matrices and heatmaps with range of built-in color maps (cmaps) that can be used to customize the appearance of the heatmap. A heatmap is a graphical representation of a matrix of values, where each cell in the matrix is represented by a color that indicates the magnitude of the value. In the context of correlation analysis, a heatmap can be used to visualize the correlation coefficients between different variables, with custom colors for higher positive correlation, higher negative correlation, and no correlation. [Was21]

As part of this project, a correlation function was developed to conduct the correlation analysis on complete dataset without any filtering, as well as on individual phases of operation with filtering. The filtering process was used to distinguish between the phases such as driving, parking, charging etc. and involved removing certain data points from the dataset. The GUI of the final analysis can be seen from the figure 4-6. During the analysis, the GUI displays the filtered length and corresponding percentage in real-time, allowing users to closely monitor the effects of the filtering on the dataset. GUI also shows the overall progress and computation time needed using the tqdm library.

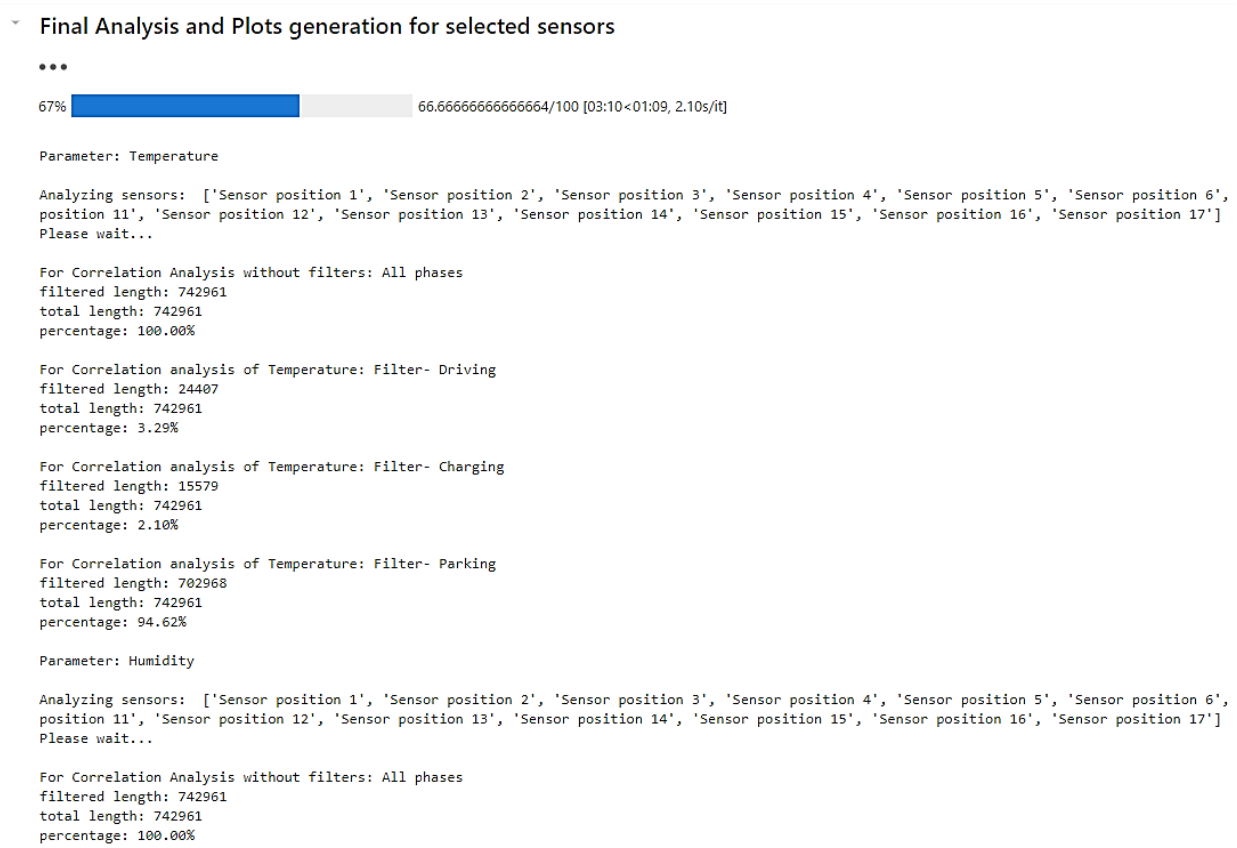


Figure 4-6: Final Analysis GUI

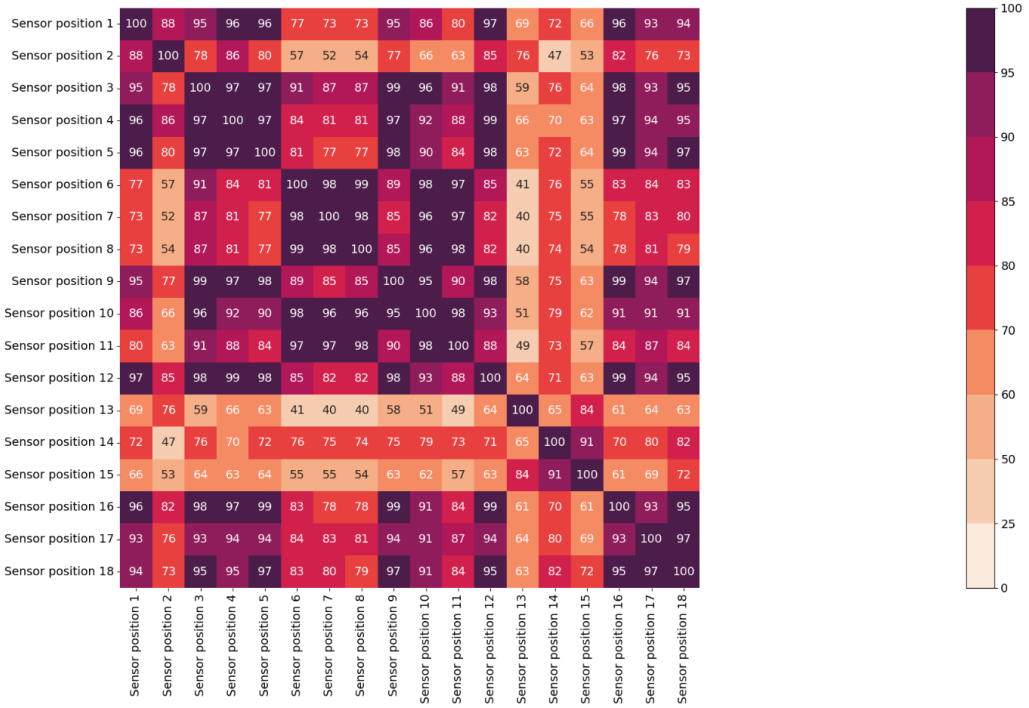


Figure 4-7: Correlation Analysis: Temperature

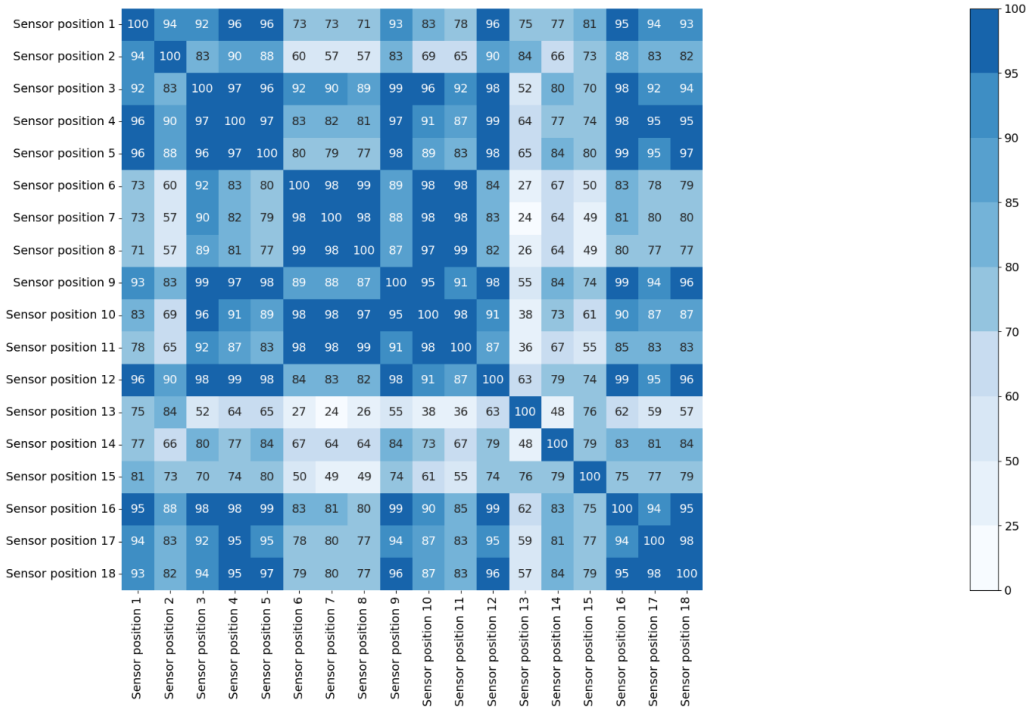


Figure 4-8: Correlation Analysis: Humidity

Figures 4-7 and 4-8 depict the heatmaps resulting from the correlation analysis conducted on physical quantities Temperature and Humidity. These heatmaps display the correlations between sensor positions as percentages, with darker colors indicating a higher positive correlation and lighter colors indicating either no correlation or a higher negative correlation. The visualization of this data in the form of heatmaps makes it easier to identify any patterns

or trends in the correlations between the different sensor positions, providing valuable insights into the behavior of the vehicle during the different phases of operation.

The final task of the analysis stage was the extraction of driving, parking, and charging phases. This step aimed to extract the driving, parking, and charging phases from the complete dataset. For this process, the functions and implementation of code over one of the reference vehicles was already available. However, the task was to integrate this code into the workflow and apply it to the selected vehicle data. The process involved imputing missing values, calculating the percentage of time spent in each phase to obtain an overview of the drives, sorting the time series data by datetime, and determining the time step differences between drives. To extract the driving, parking, and charging phases, the engine status is used as a criterion. Finally, the resulting phases were saved into separate CSV files, intended for further analysis and modeling purposes.

Once the data has been analyzed and the plots have been generated, the results can be compiled into a PDF report for quick and easy assessment. This project uses the FPDF (Free PDF) Python library for the automatic generation of PDF files. The FPDF library is open-source, simple, and easy to use. It provides basic functionalities for setting fonts, text, images, and graphic rendering in PDF documents. Due to its lightweight nature and fast processing speed, FPDF is an ideal choice for creating PDFs on the fly. [Rei12]

Initially, a PDF configuration file called "PDF configuration.py" was created. This file uses methods from the FPDF library and creates distinct classes for cover page, index page, summary page, and the main report with images. Each of these classes contains methods for generating chapters, subchapters, paragraphs, text, points, and images. The "Report design.py" file employs these classes to design each page of the report accordingly. Within the PDF configuration class, a constructor initializes the WIDTH and HEIGHT variables. The header method sets the font and creates a title for the report, while the footer method adds page numbers. The page_body method defines the body of the PDF page by setting the font, text size, and positioning for images, title_name, and comment_name. Finally, the print_page method was used to generate the report by adding a new page. The source code can be found in figure A-2.1 of Appendix A.

The next step in the results generation process is to automatize the report design procedure. In this step of the project, classes from a previously generated file were utilized to design and create a cover page, index page, summary page and main report. To arrange the text and images on these pages, methods such as add_blank_page(), add_chapter(), add_subchapter(), and add_image() were employed from their respective classes. To ensure proper formatting, a counter function was created to count the number of plots and distribute them per page while maintaining a maximum of three plots, three plot titles, and their comments per page. Due to the user-based selection involved, this process posed several challenges such as managing the positioning of images, text and numbers, counting page numbers, adding indentation, maintaining image quality, text styles and margins, and automatically generating the index page based on the selected ADAS sensors.

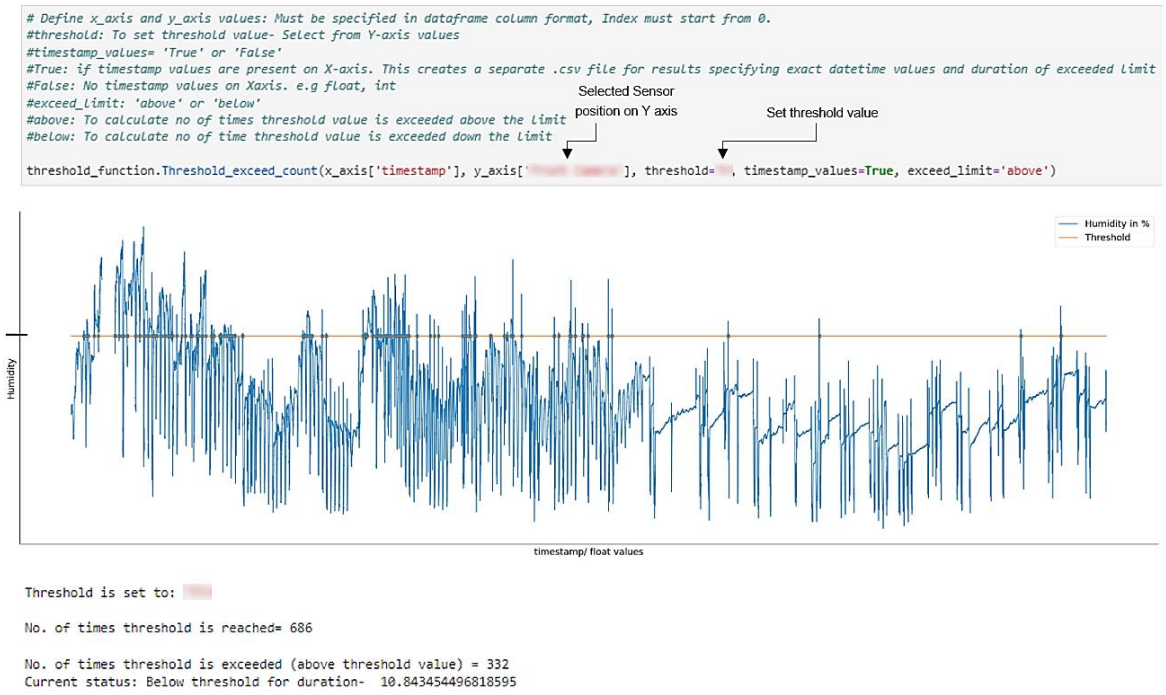
The final stage of the procedure required the preservation of the outcomes by saving the final merged PDF report, which consolidates various parts, such as the cover page, index page, and main report. This report proves to be significantly advantageous as it permits a quick evaluation of the measurement data, thus enabling an effortless comprehension and interpretation of the data. Additionally, the plots and summary tables were saved in a single folder, making it easier to incorporate them into presentations. Lastly, the driving, parking, and charging phases are saved as .csv files in a separate folder, serving as a raw data for further analysis and modeling purposes. This could potentially be used for CT system modeling or analyzing future mobility load profiles. The extracted phases provide information about how the vehicle was driven, parked, and charged over time, which can be used to understand the energy consumption patterns of the vehicle and its impact on the power system. Figure 4-9 below displays few selected pages of the generated report, providing an overview of its overall structure.

Figure 4-9: Overview of generated final PDF Report

5 Additional Tasks

In addition to the implementation of two parts, the internship encompassed two supplementary tasks. The first supplementary task involved designing a threshold function that computes the frequency and duration of temperature, humidity, or dew point exceeding the predetermined limit, both above and below it. The second task involved developing a driver visualization tool using MATLAB Graphical User Interface to visualize the driver groups with different drive profiles and road conditions and to provide a more readable and user-friendly graphical representation of driver groups.

As part of the first task, a function called "Threshold_exceed_count" was created to analyze the temperature, humidity, or dew point values of individual sensor positions. The results are saved in a .csv file, which includes the number of times the threshold was reached, the no. of times the line plot exceeded the threshold value above or below the limit, the list of all X-coordinates where the threshold value was exceeded, the list of all durations for which the line plot remained above or below the threshold value, and the total duration for which the line plot remained above or below the threshold value. The basic principle behind this function involves the calculation of all the intersection points of X and Y coordinates of the line plot, followed by calculating the minimum value of the range of every line having an intersection point, and then comparing the subsequent index values if they exceed the threshold values. This process involved several challenges and considering multiple criterions, such as where the threshold was reached but not exceeded, where the plot touched the threshold but stayed above only, intersection points with no index values, and arithmetic operations on timestamp values to calculate durations.




```
# Define x_axis and y_axis values: Must be specified in dataframe column format, Index must start from 0.
#threshold: To set threshold value- Select from Y-axis values
#timestamp_values= 'True' or 'False'
#True: if timestamp values are present on X-axis. This creates a separate .csv file for results specifying exact datetime values and duration of exceeded limit
#False: No timestamp values on Xaxis. e.g float, int
#exceed_limit: 'above' or 'below'
#above: To calculate no of times threshold value is exceeded above the limit
#below: To calculate no of time threshold value is exceeded down the limit

threshold_function.Threshold_exceed_count(x_axis['timestamp'], y_axis[' ], threshold= , timestamp_values=True, exceed_limit='above')
```

```
Threshold is set to:

No. of times threshold is reached= 686

No. of times threshold is exceeded (above threshold value) = 332
Current status: Below threshold for duration-  10.843454496818595
```

Figure 5-1: Task 1 - Threshold function: Example- Applied on Humidity values of selected sensor position

The results shown in Figure 5-1 demonstrate the application of the threshold function to the humidity values of the front camera sensor position. The function can calculate every intersection point even for large datasets, as demonstrated in this example with a dataframe having more than 700k rows and 125 columns.

In the second supplementary task, a MATLAB Driver Visualization Tool has been created with the motivation of enhancing the readability and user-friendliness of driver groups' information and extracting the maximum possible data. The tool has been designed to split drive timings in Weekday and Weekend drives, and consider a variable number of road conditions and driver groups. The development of the tool faced several challenges, such as visualizing data over Matlab using UI panels, UI tabs, and UI control feedback, converting values into Matlab Datetime format, as well as handling missing data and empty cells. The tool has been made robust to handle a variety of driver groups, including multiple variants and exceptions, variable roadway conditions and number of drives. Additionally, the tool has been designed to effectively manage space and positions of text, legends, UI panels, and buttons using scrollable UI while dealing with a large amount of data. The tool's basic overview is depicted in Figure 5-2. On the left-hand side, it displays the driver names and the corresponding day drive names, such as weekdays or weekends. At the bottom, a dashboard is presented, which includes filters for road conditions, UI button to link the axes of all subplots. Finally, the tool also provides an ui button ‘Mark’ to mark all start and end points of all drives.

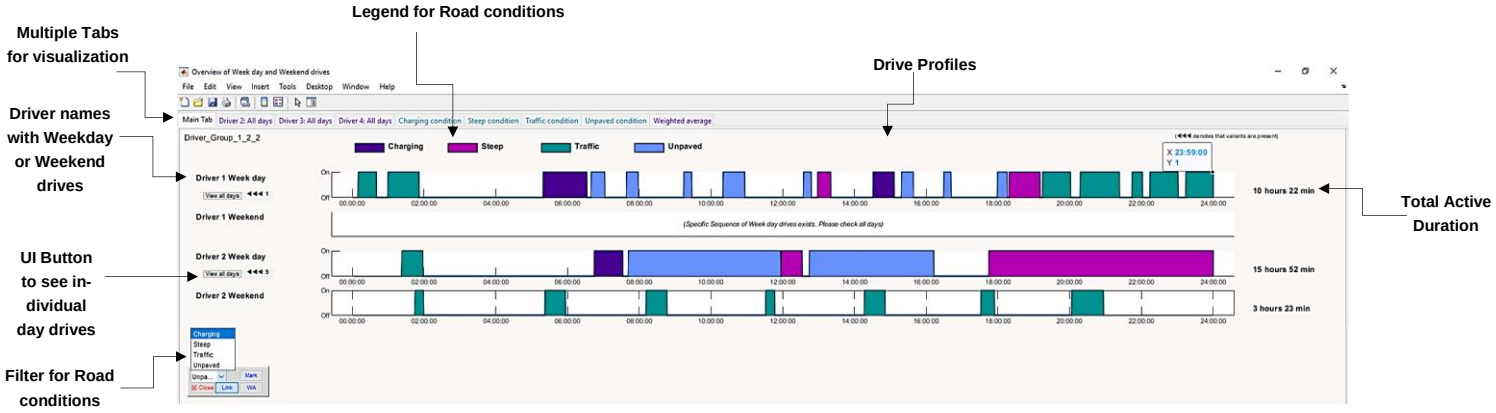


Figure 5-2: Task 2- Overview of MATLAB Visualization Tool: Main Window

6 Summary and Outlook

During the internship, the focus was on supporting the hardware development of ADAS systems through a series of tasks, which were divided into two parts.

The first part involved supporting AI-driven development with pyMICE, which required studying the knowledge discovery framework and applying AI-driven development in the context of autonomous driving. This included working on the use case of implementing the pyMICE package on the Automotive Camera's Joining Process with a focus on optimizing design and process parameters to achieve the desired values of outputs. The objective was to gain an understanding of the overall function and background behind pyMICE, which involves optimizing hyperparameters of machine learning models using the XGB-NSGAII algorithm and Pareto front approach. The implementation phase of the first part included various tasks, such as configuring pyMICE to ensure its compatibility across multiple operating systems and running it on DASK cluster with Python 3.6. The package was then migrated to Python 3.9 and tested again with DASK cluster to compare the performance after migration. The next task involved porting of pyMICE to the Azure ML cloud, which required setting up DASK cluster on both single and multi-nodes of Azure ML compute cluster. The subsequent step involved creating a custom curated Azure ML environment and running DASK cluster on multiple nodes with Azure VMC Cluster. Finally, pyMICE was deployed on the Azure ML cloud by installing it as a private package using .whl file.

The work described in this study centered on implementing the pyMICE package on the Automotive Camera's Joining Process. However, this knowledge discovery framework and AI-based product development process can be applied to other complex physical phenomena or mechanical processes as well to optimize the design and process parameters. Further research can explore the use of pyMICE in diverse domains, such as automotive manufacturing, robotics and automation, internet of things, or industry 4.0. Additionally, it would be worthwhile to investigate the performance of pyMICE on different platforms, such as AWS or Google Cloud, to identify the optimal platform for running the package. Integrating pyMICE with other tools and APIs could also improve its usability and accessibility. Moreover, in this project, the DASK cluster was executed on four virtual machines using Azure VMC cluster. However, in an improved version of the project, the curated environments and dependencies for individual VMs can be better managed by creating a custom image. This would help to ensure that each VM has the necessary software packages and configurations, leading to more efficient and reliable cluster execution.

In the second part of the project, the objective was to analyze sensor data from vehicle measurements. The data was transformed from a Matlab compatible file format to Python, pre-processed, and cleaned. Time series, statistical, and correlation analysis was performed on the data of multiple physical quantities such as temperature, humidity, and dew point. The results of the analysis have successfully visualized the environmental loads within a vehicle, vehicle driving behaviour and correlation between the environmental measurements of different mounting positions within a vehicle with respect to the driving, parking

or charging phases. Additionally, an automatized pdf report generation process was implemented to save the results. Furthermore, a threshold function was created that calculated the number of times the threshold value of temperature, humidity, or dew point was exceeded either above or below the specified limit and the exact duration for which the value remained above or below the threshold value. Finally, a driver visualization tool was developed using MATLAB Graphical User Interface to visualize the driver groups with different drive profiles and road conditions.

Throughout the internship, various technical skills were utilized, including distributed computing using DASK, developing tools with MATLAB, configuring Python packages, porting source code to the Azure ML cloud, performing statistical and correlation analysis, visualizing data using Python libraries such as seaborn and matplotlib, studying multi-objective optimization, evolutionary and HPO algorithms, investigating the impact of environmental loads on ADAS sensor mounting locations within a vehicle, and gaining a deeper understanding of automotive camera fundamentals. In addition, the internship offered the opportunity to improve social and communication skills by collaborating with teams and presenting findings to team members. The ability to work both independently and as part of a team was also developed. Overall, the internship presented challenging projects that demanded technical expertise and teamwork, resulting in the completion of assigned tasks and an improvement in various technical and communication skills.

References

[BHS21a] Botticelli, Hellmann; Stapf, Schünemann; Jochmann: AI-Driven Gasoline Direct Injection Development- A Knowledge-Discovery Framework for Comprehensible Evaluations of Complex Physical Phenomena, Springer, pp. 6, 2021.

[BHS21b] Botticelli, Hellmann; Stapf, Schünemann; Jochmann: Model Selection for Gasoline Direct Injection Characteristics Using Boosting and Genetic Algorithms, New York, USA: ACM ISEEIE, pp. 2- pp. 6, 2021.

[Roc18a] Rocklin: Dask Scaling Limits. URL: https://blog.dask.org/2018/06/26/dask-scaling-limits - 26.06.2018.

[Cri16] Crist: Dask & Numba- Simple Libraries for Optimizing Scientific Python Code, Austin, Texas: IEEE International Conference on Big Data, pp. 1- pp. 2, 2016.

[Roc15] Rocklin: Dask: Parallel Computation with Blocked algorithms and Task Scheduling, PROC. OF THE 14th PYTHON IN SCIENCE CONF. SCIPY, pp. 126- pp. 129, 2015.

[Roc14] Rocklin: Dask. URL: https://docs.dask.org/en/stable/ - 2014

[Roc18b] Rocklin: Dashboard Diagnostics. URL: https://docs.dask.org/en/stable/dashboard.html - 16.02.2018.

[Sig22] Signell: Dask Tutorial, URL: https://github.com/dask/dask-tutorial/blob/main/00_overview.ipynb - 10.07.2022.

[Das17] Dask developers: Dask-ML, URL: https://ml.dask.org/ - 2017.

[Pel22] Pelgrim: Scikit-learn + Joblib: Scale your Machine Learning Models for Faster Training, URL: https://www.coiled.io/blog/sklearn-joblib-dask - 15.03.2022.

[Wal21] Wallen: Time to Say Goodbye- Python 3.6 Is End-of-Life, URL: https://thenewstack.io/time-to-say-goodbye-python-3-6-is-end-of-life/ - 16.12.2021.

[Git08] Github community: About repositories, URL: https://docs.github.com/en/repositories/creating-and-managing-repositories/about-repositories – 2008.

[Inm05] Inmon: Building the Data Warehouse. 4th edition, Indianapolis, Indiana: Wiley Publishing, Inc, pp. 111- pp.390, 2005.

[PFH11] Pourmal, Folk; Heber, Koziol; Robinson: An Overview of the HDF5 Technology Suite and its Applications, USA: ACM Digital Library, pp. 1- pp. 3, 25.03.2011.

[DPA02] Deb, Pratap; Agarwal, Meyarivan: A Fast and Elitist Multi-objective Genetic Algorithm- NSGA-II, IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182- pp. 186, 2002.

[DS12] DuToit, Steyn; Stumpf: Graphical exploratory data analysis. ISBN 978-1-4612-9371-2. OCLC 1019645745, Springer, pp. 54- pp. 72, 2012.

[CK68] Croxton, Cowden; Klein: Applied general statistics. 3rd edition, London: Pitman, pp. 625, 1968.

[Was21] Waskom: seaborn- statistical data visualization, Journal of Open Source Software, 6(60), pp.1- pp.3, 06.04.2021.

[Rei12] Reingart: FPDF for Python. URL: https://pyfpdf.readthedocs.io/en/latest/index.html - 15.08.2012.

[Mic23a] Microsoft: What are Azure Machine Learning environments? URL: https://learn.microsoft.com/en-us/azure/machine-learning/concept-environments - 24.02.2023.

[Mic23b] Microsoft: Manage Azure Machine Learning environments with the CLI & SDK (v2). URL: https://learn.microsoft.com/en-us/azure/machine-learning/how-to-manage-environments-v2?tabs=cli - 04.04.2023.

[Ngu21] Nguyen: Library to turn Azure ML Compute into Ray and Dask cluster. URL: https://techcommunity.microsoft.com/t5/ai-machine-learning-blog/library-to-turn-azure-ml-compute-into-ray-and-dask-cluster/ba-p/3048784 - 30.12.2021.

[Das23] Dask developers: Microsoft Azure- Azure VMC Cluster. URL: https://cloudprovider.dask.org/en/latest/azure.html – 2023.

[Mic23c] Microsoft: Use private Python packages with Azure Machine Learning. URL: https://learn.microsoft.com/en-us/azure/machine-learning/v1/how-to-use-private-python-packages - 03.02.2023.

[Bot23] Botticelli: Development of a modular Knowledge-Discovery Framework based on Machine Learning for the interdisciplinary analysis of complex phenomena in the context of GDI combustion processes, Institut für Informationsmanagement im Ingenieurwesen (IMI), Karlsruher Institut für Technologie (KIT), pp. 08- pp. 76, 29.03.2023.

[Lan20] Langa: What's New In Python 3.9, URL: https://docs.python.org/3/whatsnew/3.9.html – 05.10.2020.

[Mic23d] Microsoft: Azure Machine Learning documentation, URL: https://learn.microsoft.com/en-us/azure/machine-learning/?view=azureml-api-2 – 2023.

[Act22] Active state: Managing Python Dependencies – Everything You Need To Know. URL: https://www.activestate.com/resources/quick-reads/python-dependencies-everything-you-need-to-know/ - 11.07.2022.

[Kin16] Kingi: A Brief Introduction to the Command Line and Git, Cornell University, New York, pp. 2, 2016.

[LJ16] Langtangen, Johansen: Creating Virtual Python Software Environments with Virtualenv, Department of Informatics, University of Oslo, Norway, pp. 2- pp. 3, 11.04.2016.

A Appendix

A.1 Part I: Supporting AI-driven development with pyMICE

A.1.1 Upgrade virtual environment to Python 3.9 and upgrade all dependencies to their latest version

Table A-1.1: List of pyMICE dependencies and migrated versions

Sr no.	Dependency	venv 3.6		venv 3.9
1	Dask	2.9.1	-->	2023.1.0
2	Distributed	2.9.1	-->	2023.1.0
3	DEAP	1.3.0	-->	1.3.3
4	Scikit-learn	0.21.3	-->	1.0.2
5	xgboost	1.0.1	-->	1.7.2
6	daskml	1.1.1	-->	1.9.0
7	daskjobqueue	0.6.3	-->	0.8.1
8	flask	2.0.3	-->	2.1.3
9	tornado	6.0.4	-->	6.2
10	paramiko	2.6.0	-->	2.12.0
12	numpy	1.17.2	-->	1.23.4
13	pandas	1.0.3	-->	1.5.2
14	scipy	1.3.1	-->	1.9.3
15	matplotlib	3.1.2	-->	3.6.2
16	seaborn	0.10.0	-->	0.12.1
17	yaml	0.1.7	-->	0.2.5
19	python	3.6.13	-->	3.9.15
20	pip	21.2.2	-->	22.3.1

A.1.2 Running DASK on Azure ML Curated environment using Script Run Config or Azure ML Pipelines

```
# DASK Test Script: Running Local cluster on Azure ML Curated Environment

# import modules
from distributed import Client, LocalCluster
import pandas as pd
import sys

# define local cluster
cluster = LocalCluster(n_workers=8, processes=False, threads_per_worker=1, memory_limit='8GB')
client = Client(cluster)

def run_grid_search():
    # Compute
    from sklearn.datasets import make_classification
    from sklearn.svm import SVC
    from sklearn.model_selection import GridSearchCV
    import pandas as pd

    X, y = make_classification(n_samples=1000, random_state=0)
    X[:5]

    param_grid = {"C": [0.001, 0.01, 0.1, 0.5, 1.0, 2.0, 5.0, 10.0],
                  "kernel": ['rbf', 'poly', 'sigmoid'],
                  "shrinking": [True, False]}

    grid_search = GridSearchCV(SVC(gamma='auto', random_state=0, probability=True),
                               param_grid=param_grid,
                               return_train_score=False,
                               cv=3,
                               n_jobs=-1)

    grid_search.fit(X, y)
    fit_results=pd.DataFrame(grid_search.cv_results_)
    grid_search.predict(X)[:5]

    Final_score= str("Final prediction score is: "+str(grid_search.score(X, y)))
    return Final_score, fit_results

# submit the function to the distributed workers
output= client.submit(run_grid_search)

# retrieve the result of the function execution
result = output.result()
df_result=pd.DataFrame(result)
print(df_result.iloc[0,0])
print(df_result.iloc[1,0])

# Close dask
client.shutdown()
client.close()
try:
    sys.exit(0)
except:
    print("Client closed")
```

Figure A-1.2: Dask test script: Running Dask local cluster on Azure ML curated environment

A.1.3 Deploying pyMICE on Azure ML cloud: Installing private package using .whl files

```
from azureml.core import Workspace, Environment
from azureml.core.conda_dependencies import CondaDependencies
from azureml.core import ScriptRunConfig, Experiment
ws = Workspace.from_config()

whl_url = Environment.add_private_pip_wheel(workspace=ws,file_path = "pyMICE-any.whl")
my_private_env = Environment(name="my_private_env")
conda_dep = CondaDependencies()
conda_dep.add_pip_package(whl_url)
my_private_env.python.conda_dependencies=conda_dep

# create a compute target
compute_target = ComputeTarget(ws, "mycompute")

exp = Experiment(workspace=ws, name="myexp")
# configure and submit your training run
config = ScriptRunConfig(source_directory='script',
                         command=['python', 'pyMICE_test_script.py'],
                         compute_target=compute_target,
                         environment=my_private_env)
script_run = exp.submit(config)
```

Figure A-1.3: Installing private package in Azure ML using .whl files

A.2 Part II: ADAS Evaluation: Analysis of Vehicle Sensor Measurements

A.2.1 PDF Report Generation

```
class PDF(FPDF):

    def __init__(self):
        super().__init__()
        self.WIDTH = 210
        self.HEIGHT = 297

    def header(self):
        self.set_font('Arial', 'B', 11)
        self.cell(self.WIDTH - 80)
        self.cell(60, 1, '<Tittle_of_the_report>', 0, 0, 'R')
        self.ln(20)

    def footer(self):
        # Page numbers in the footer
        self.set_y(-15)
        self.set_font('Arial', 'I', 8)
        self.set_text_color(128)
        self.cell(0, 10, 'Page ' + str(self.page_no()), 0, 0, 'C')

    def page_body(self, images,tittle_name,comment_name):
        # To set size of images, margins, tittles, comments according to the section
        if len(images) == 3:
            self.set_font('Arial', 'B', 8)
            self.text(10, 21, str(tittle_name[0]))
            self.set_font('Arial', '', 6)
            self.text(12, 24, str(comment_name[0]))

            self.set_font('Arial', 'B', 8)
            self.text(10, self.WIDTH / 2 + 1, str(tittle_name[1]))
            self.set_font('Arial', '', 6)
            self.text(12, self.WIDTH / 2 + 4, str(comment_name[1]))

            self.set_font('Arial', 'B', 8)
            self.text(10, self.WIDTH / 2 + 86, str(tittle_name[2]))
            self.set_font('Arial', '', 6)
            self.text(12, self.WIDTH / 2 + 89, str(comment_name[2]))
        elif len(images) == 2:
            self.set_font('Arial', 'B', 8)
            self.text(10, 21, str(tittle_name[0]))
            self.set_font('Arial', '', 6)
            self.text(12, 24, str(comment_name[0]))

            self.set_font('Arial', 'B', 8)
            self.text(10, self.WIDTH / 2 + 1, str(tittle_name[1]))
            self.set_font('Arial', '', 6)
            self.text(12, self.WIDTH / 2 + 4, str(comment_name[1]))

        else:
            self.set_font('Arial', 'B', 8)
            self.text(10, 21, str(tittle_name[0]))
            self.set_font('Arial', '', 6)
            self.text(12, 24, str(comment_name[0]))

    def print_page(self, images,tittle_name,comment_name):
        # Generates the report
        self.add_page()
        self.page_body(images,tittle_name,comment_name)
```

Figure A-2.1: FPDF source code