D8.2 Improved Pipelines all Use Case Studies

DAPHNE

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 2.0

PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

This document reports on the extended use case pipelines that improve the runtime and/or accuracy by leveraging the DAPHNE system infrastructure.

In this deliverable, we present the extensions to the individual use case pipelines. We briefly show the benchmarking setup as well as preliminary benchmarking results.

D8.2 Improved pipelines all use case studies			
WP8 – Use Case Studies			
Type of document	R	Version	2.0
Dissemination level	PU		
Lead partner	KAI		
Author(s)	Benjamin Steinwender (KAI), Vytautas Jancauskas (DLR), Andreas Laber (IFAT), Marius Birkenbach (KAI), Bernhard Einberger (AVL), Daniel Krems (AVL)		
Contributors	Piotr Ratuszniak (INTP), Ilin Tolovski (HPI), Nils Straßenburg (HPI)		

Revision History

Version	Item	Comment	Author / Reviewer
V1.0	Merged documents of individual uses cases		Steinwender
V2.0	Incorporated reviewer comments		Steinwender

Executive Summary

The use case pipelines presented in the DAPHNE project are **enhanced by utilizing the DAPHNE** system infrastructure. Depending on the complexity of the initial pipeline description, the individual use cases either chose for implementing the pipeline in DSL or DaphneLib.

The WP8 project partners contributed to the development of DAPHNE system infrastructure by providing constructive **feedback** in the form of GitHub issues as well as pull-requests to the source code repository.

Preliminary benchmarking results show an improvement of the end-to-end runtime, which has been identified as the most crucial measurable of the individual use-case pipelines. For pipelines written in DaphneDSL, the **runtime decreased** to about 16% to 30% of the baseline implementation. For pipelines utilizing DaphneLib, results are not yet available.

Table of Contents

List of Abbreviations

Abbreviation	Meaning
APC	Advanced Process Control
API	Application Programming Interface
CFD	Computational Fluid Dynamics
CSV	Comma Separated Value
DOE	Design Of Experiments
DUT	Device Under Test
KRR	Kernel Ridge Regression
LCZ	Local Climate Zone
MSE	Mean Squared Error
RDP	Ramer-Douglas-Peucker
SBSE	Search Based Software Engineering
VW	Visvalingam-Whyatt

1 Introduction

Since the report on initial use case pipelines (deliverable D8.1), we have been working on enhancing the pipelines to utilize the DAPHNE system infrastructure. Naturally, this goes together with providing feedback to the technical work packages and/or by requesting certain features. Such are low-level primitives that operate on matrix data types like `idxMin` and `idxMax` but also high-level functions for data processing and machine learning like `randomForest`.

1.1 Feature requests & bug reporting

When working with the DAPHNE system, some parts of compiling the DSL script are still in an early stage. To overcome our workarounds, we regularly request for new features and sometimes also find bugs.

Overall, we have created 15 issues on GitHub, where 7 of them are already solved and implemented in the source code. Of the remaining open issues, some are already being implemented.

Solved:

- GH #477 [discussion] Running ./build-containers.sh without WSL internet connection
- GH #582 [feature] passing str variable to readFrame/readMatrix
- GH #583 [feature] return multiple values from UDF
- GH #589 [feature] idxMin, idxMax
- GH #599 [discussion] Daphne Lib Containers pip
- GH #602 [feature] DaphneLib Complex Computation exp(matrix)
- GH #618 [feature] passing string literals to a DAPHNE script

Open:

- GH #581 [feature] string comparison
- GH #584 [feature] creating a dataFrame from variables
- GH #587 [feature] reading parquet files
- GH #590 [feature] crossProduct
- GH #601 [feature] KernelRidgeRegression DaphneDSL - Algorithms Implementation
- GH #603 [discussion] Tensor Implementation
- GH #616 [feature] Daphne Lib: Numpy – Vector support
- GH #621 [feature] randomForest implementation

1.2 Open-source contributions

The people working in WP8 on the use-cases are the first users of the DAPHNE system infrastructure and thereby provide not only valuable feedback, but also contribute directly on the open-source GitHub repository:

1.2.1 GH #459: [CI] continuous integration

As we were using early versions of the main branch (partly not yet released snapshots of the repository) we wanted to know whether a specific commit is usable and thereby proposed an initial version of an automated build of the source code.

This initiative was taken over and enhanced by the technical work packages in a way that we now have (1) checks for every commit/merge to the main branch to check whether the introduction of a new feature breaks existing code and (2) a possibility to simply download the binary artifacts for execution on a supported system.

1.2.2 GH #550: Documentation hosting

The DAPHNE system infrastructure already contains a vast documentation in markdown format directly included in the git code repository. The drawback thereby is that such system is not easily searchable. An initial effort was undertaken by members of the WP8 team to provide an automated mechanism to render the static markdown files into a searchable web application and hosting the documentation publicly on the internet.

The result can be found here[1]. It allows users to search any content of the help files in a simple way. Furthermore, the documentation is split up into separate parts between users of the DAPHNE system (both DaphneLib and DAPHNE-DSL) and developers who want to contribute to the DAPHNE system.

1.2.3 GH #620: Documentation updates

While working on improving our pipelines to utilize DAPHNE, we also enhance the documentation for all.

1.3 Outline

In the following chapters, the individual use cases report on their status in more detail.

[1] https://daphne-eu.github.io/daphne

2 Earth Observation Case Study: Local Climate Zone classification (DLR)

2.1 Use Case Description

Local Climate Zones (LCZ) are a way to classify land based on its use. Unlike other land use classification schemes, it focuses on urban environments. It divides land in to 17 classes, 10 of which correspond to different built-up areas while the remaining 7 correspond to land without buildings on it. Those classes are listed in the image below this paragraph. While it was originally developed as an aid to study urban heat islands, it has found further use in various other research scenarios, for example, urban development, transport research, disaster mitigation and population assessment among others. At DLR, we use LCZ to facilitate research in such fields as the study of transport flows and building height estimation. It is thus of interest to us to have a way to acquire up to date LCZ classifications of any area on earth. For that, we need to rely on satellite observations of the Earth. Both radar and optical imagery can be used but we focus on optical data in this case. Ideally the user would specify a region of interest and a period and be able to retrieve an LCZ classification for that area that will look similar like the one in Figure 2.1.

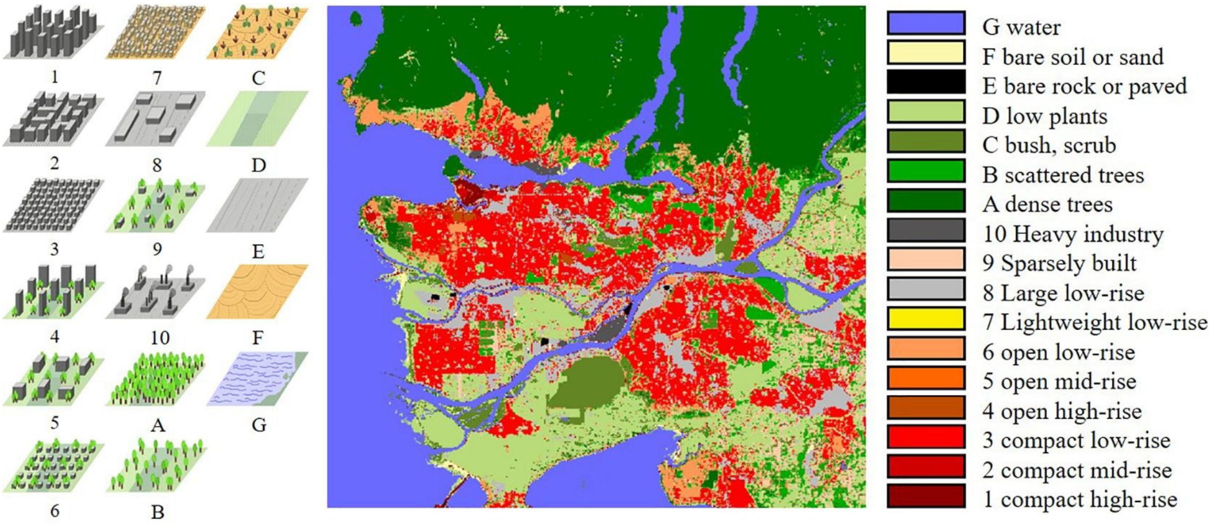


Figure 2.1: LCZ classification classes

At DLR we developed a pipeline to allow for LCZ classification of arbitrary geographical areas. It uses a ResNet deep learning model trained on the LCZ42 data set to classify land according to its LCZ label.

2.2 Use Case Extension

Since the deliverable D8.1, the pipeline was extended to support LCZ classification of the large geographical areas (including the full extent of the Earth). To this end a software framework was written that allows for efficient execution of user defined workflows on arbitrary geographical areas. The specified areas are divided into rectangular patches which are processed using a user specified Pypyr workflow described in a YAML file. After processing the results are optionally reconstituted into a larger image corresponding to the originally specified area. In our case, the workflow consists of a data acquisition phase and an inference phase. Data acquisition consists of a finding a time series of Sentinel-2 images for a given area and a given period as well as having a lower than specified cloud cover. From these images we construct a mosaic of cloud free pixels for each pixel of the image. This is done by taking pixels for which cloud probability is lower than the threshold and taking their medoid. The resulting image is then used as input for the inference stage. Inference is done using a ResNet based model trained on the LCZ42 dataset. We use Pytorch for the inference stage of the pipeline. After inference we get probabilities that given patches on the satellite image belong to a given LCZ class. These probabilities can then further be refined through Bayesian inference and the use of additional data sets, such as ESA WorldCover land use data set or the WSF 3D dataset of building height data. To this end we allow the user to define likelihoods specific to these data sets. A likelihood consists of probabilities of a specific feature appearing in the dataset if it belongs to a specific LCZ class. These probabilities are specified in a JSON file by the user. In Figure 2.2 we give an example of such a classification.

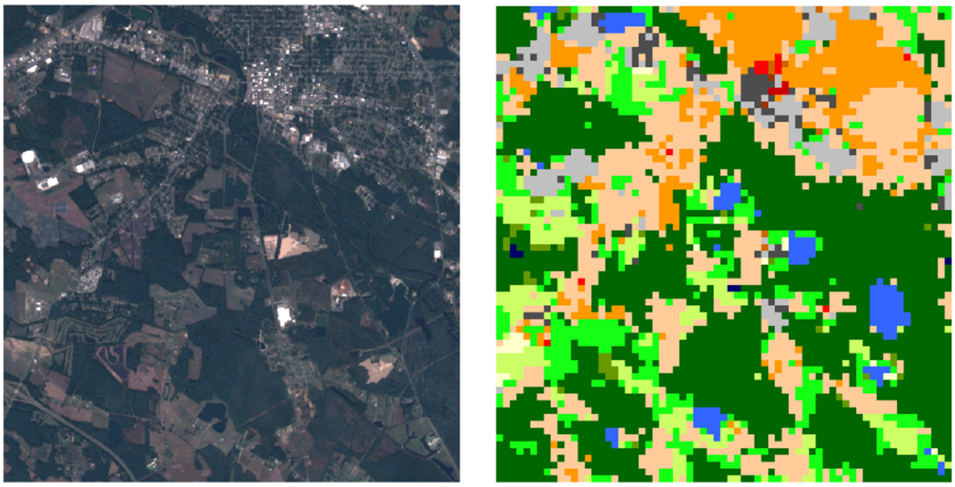


Figure 2.2: LCZ classification of an urban area

2.3 Evaluation

Figure 2.3 shows the Earth Observation Use Case results on the NVIDIA Tesla T4 GPU with 8.1 TFLOP/s and 16GB memory when using DAPHNE as well as other frameworks. We trained the ResNet20 model with the Adam optimizer for 100 epochs on So2Sat LCZ42 dataset. TensorFlow and TensorFlow XLA (both with GPU) show similar performance, which is dominated by I/O. TensorFlow XLA is a domain-specific compiler for linear algebra operations with similar goals to DAPHNE with regards to optimizing deep learning operations. Apache SystemDS (with GPU and read into FP64 and conversion to FP32) is about 1.2x slower, but 4x faster with multi-threaded I/O. DAPHNE's basic CSV reader and GPU integration read FP32 directly and show the best single-threaded performance. Vectorized read pipelines and task placement have potential for solid improvements. Further improvements to the Earth Observation Use Case could be possible with regards to data pre-processing and Bayesian inference stage of the pipeline.

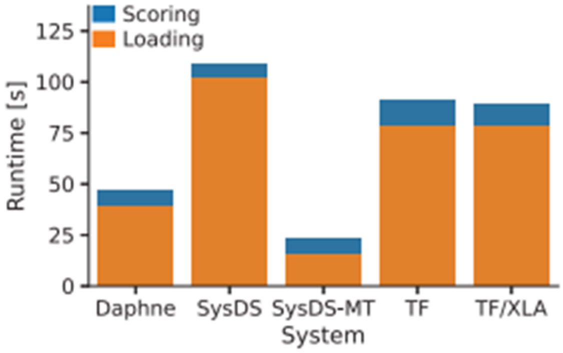


Figure 2.3: DLR pipeline results

3 Semiconductor Manufacturing Case Study: Optimizing the equipment stability and utilization (IFAT)

In deliverable D8.1 we described the underlying implantation process, the interim data engineering workflow, and the initial machine learning pipeline. Since then, we published a paper about the proof-of-concept of ion beam tuning prediction based on log files produced by the equipment [1]. Moreover, we have been working intensively on progressing the use case to make it ready for a productive implementation. It therefore had to under-go significant changes, which we will detail out in the later sections. Afterwards, we will showcase the DaphneDSL implementation for the core of the machine learning pipeline, this includes preprocessing, model training and scoring.

3.1 Overview of Improved Pipeline

For productive use of the developed AI model in Infineon's fabrication facilities (fab), we needed to substitute our existing data source - log files parsed directly by Python scripts - with a proper database that is supported by our IT department. In that regard we switched our input to be taken from a system called Advanced Process Control (APC). This also comes with a more stable and comfortable way of getting the data, i.e., from a data lake with the corresponding infrastructure and a data virtualization layer on top, that unifies the way data is accessed. As we learned more about our data, we updated the machine learning pipeline. Details can be found in the next two sections.

3.2 Data Engineering

We get our APC data from a Hadoop data lake instance. The data representation is in long format. Thus, we have a separate row for every feature tracked and wafer processed. This sums up to about 70 million entries per equipment and month. Via Python scripts we transform it into wide format, the typical (X, y) format, that we need for machine learning: (1) We pivot the dataset to have all the relevant data for an observation in a single row; (2) We sub-select the data, keeping only the last wafer per lot. We run the previously mentioned operations within our HPC cluster, due to the vast amount of data being handled. For preparing a dataset for one equipment, for a timeframe of 6 months, the resource usage summary shows values for maximum memory of up to 560 GB, with average memory usage of roughly 100 GB.

Prior to September 2023 the setup result of the tuning process which are used as the labels for supervised learning, were not tracked in APC. To compensate for missing labels, we merge our APC data with setup data from the proof-of-concept phase. The merge proved to be unmanageable within pandas because we kept running into an out-of-memory error. With the library pandasql[2] we were able to merge the APC sensor data with setup result data from the MySQL database. The data is then persisted as Python objects, utilizing the library joblib[3]. This method replaces our previous approach of using csv files. This allows for faster reloading into memory. On this data we can now run our machine learning pipeline.

[2] https://github.com/yhat/pandasql

[3] https://joblib.readthedocs.io

3.3 Machine Learning

Building up on the dataset generated within data engineering, we employ a couple of techniques on it to get a better performance of the machine learning model. An overview is shown in Figure 3.1.

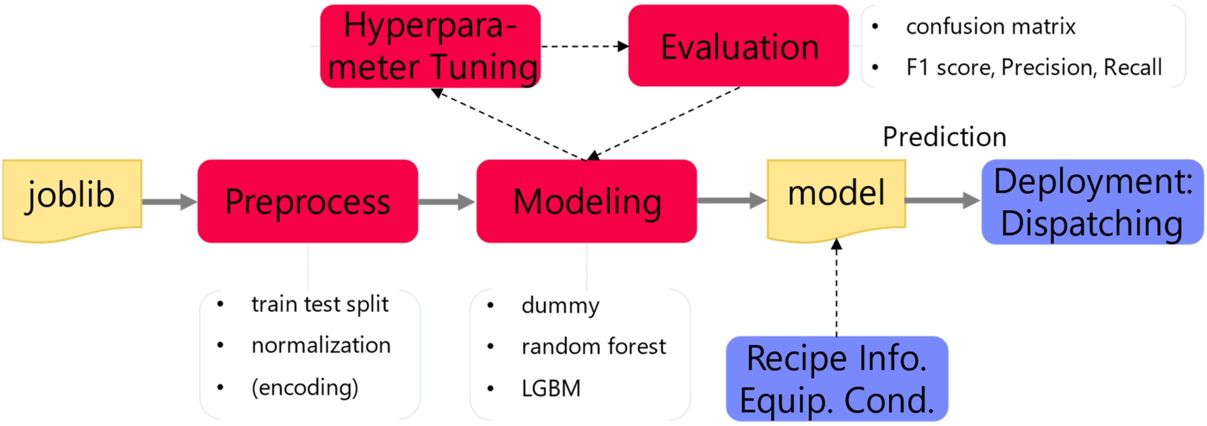


Figure 3.1: Improved Machine Learning Pipeline

3.3.1 Dataset

With the transition from the log files to the APC system we were confronted with a reduction of available features, from 2547 features down to 769. The observed reduction in the number of parameters can be attributed to the fact that the remaining features represent a carefully expert-curated set that monitors the quality of the equipment's manufacturing process. It is worth noting that despite this reduction in overall features, the predictive signal in the data has not been impacted negatively, as our performance metrics of the ML model have remained consistent, Moreover, we have introduced feature engineering to extract the most important components from a recipe: dose, species, and energy. We then replace the categorical species values (e.g., B for Boron) with the corresponding atomic mass units. As a result, we are left with a purely numeric dataset.

3.3.2 Preprocessing

We preprocess the data to prepare it for model training. We have a set of column filters that remove columns based on high NaN-portions, quasi-constant values, column-wise duplicates, and high correlation with other columns. This leaves us with roughly 650 feature columns.

We introduced an additional method for feature selection, which relies on the Boruta algorithm and Shapley values. The Boruta algorithm creates shadow features by shuffling the values within one column for each feature in the dataset. It then compares their importance scores to the original features. If the importance score of a feature is not significantly different from that of its shadow features, it is deemed unimportant and removed from the dataset. By default, it uses random forests as an underlying model. To get an estimate of the feature importance we can either use the feature importance scores inherent to the model or we can calculate the corresponding model-agnostic Shapley values. The resulting subset of features is then used for further processing and is usually in the range of 75 up to 100 features.

We split our data into train and test datasets. Imputation is currently not necessary because we get rid of missing entries by removing affected rows beforehand. Scaling is currently not improving the performance of the employed decision tree-based models, yet we know that it is necessary for e.g., neural network models.

3.3.3 Modeling

A dummy model, which always predicts the majority class, acts as a baseline as it reflects the status quo, in terms of how scheduling is currently handled within our fab. We dispatch lots, without taking the probability for tuning success into account. This is viable because tuning is successful in 80%+ of the cases. As a second level baseline we reference our results to random forests because they are known to be robust to noise and outliers in the data.

We evaluated a couple of models on the datasets and found that histogram-based gradient boosting classifiers perform very well on our dataset, Microsoft's LightGBM[4] and sklearn's HistGradientBoostingClassifier[5]. A master student tried to outperform them with various deep learning topologies with the help of TensorFlow/Keras[6], but was only able to match the same performance – with increased computational effort. In terms of hyperparameter tuning, we used optuna[7]. Yet, we could not achieve significant better results, as it tended to overfit the model to the training dataset.

Additionally, we developed a regression model to predict the tuning duration, as this would be straight-forward to integrate with our scheduling solution. While evaluating the business value of this use case, it turned out that most of the uptime loss is due to failed setups, not prolonged ones. Hence, the development of enhancements to the regression model has been put on hold for now. It is still valuable though, and we have again two baselines to compare the predictions with. Currently there is a set of static values in use for the setup matrix, which defines the cost for a specific recipe transition, i.e., switching from recipe A to recipe B. The second baseline is the Chebyshev distance between the most important parameters of a setup, dose, species, and energy. It remains to be checked whether our regression model can beat both baselines. This needs to be analyzed in the coming months.

3.3.4 Evaluation

For validating the classification models, we are analyzing the confusion matrix and monitor the F1 score, Precision and Recall. We have noticed differences in performance metrics from equipment to equipment, that could be explained by the volatility of the overall tuning fails in the corresponding datasets.

The regression models used to be evaluated on mean absolute error and mean squared error (MSE), but as mentioned before corresponding development effort is on hold.

3.3.5 Deployment

The productive deployment is still planned the same way, as described in the deliverable D8.1, section 3.3.5. It is essential to follow Machine Learning Operations (MLOps) practices while

[4] https://github.com/microsoft/LightGBM
[5] https://scikit-learn.org
[6] https://keras.io
[7] https://optuna.org

deploying an ML model in a production environment. This involves hosting the model within a containerization and orchestration platform to ensure stable and continuous operation. To ensure minimal to no delay in dispatching decisions, it is vital that the model receives live data from production equipment as quickly as possible. Regarding the process flow of the proposed productive solution, once a lot is moved out of the previous operation, it is added to the dispatch list of our implantation work center. When multiple equipment tools are available for processing the same recipes, we refer to it as a work center, and this is typically the case when the equipment is from the same equipment platform. Using a trained model, the ion beam tuning outcome is predicted with probability percentages for success/fail and respective duration estimates, and the lots are reranked based on these values along with other criteria like wait time, remaining time to due date, prioritization for development and customer demanded lots and lots needed for equipment checks. Our experiments during model training are tracked within MLflow[8]. We also register our trained models accordingly. The models can then be pulled from this registry for productive deployment.

Additionally, we created a conceptual interface between the machine learning pipeline and the dispatching system. It contains only the data needed for optimizing the schedule. The corresponding draft can be found in Table 1. If the value in the "Tune Success" column is 1, it indicates a successful tuning attempt, while a value of 0 indicates a failed tuning attempt.

Table 1: Interface between Machine Learning and Scheduling System

Equipment	Current Recipe	Next Recipe	Duration (min)	Tune Success
IMP01	X1-80E3B020A	X5-00E2B170A	2.8	1
IMP04	X3-22E4P020A	Y1-00E2B400A	4.2	0
IMP03	Y1-31E3B050A	Y2-50E4B130A	3.1	1
IMP01	Y6-11E5A150A	Y6-71E2P285D	5.3	1
IMP02	Y8-05E2P285D	..		

The development of the Python code for deployment has not been started yet.

3.4 DaphneDSL Implementation

In the above chapters we provided details on the Python implementation, which is going to be used productively within our fab. For benchmarking purposes, we maintain a decoupled Python implementation, which mirrors the DaphneDSL implementation as closely as possible. The corresponding code can be found in our internal GitLab repository, hosted by Know-Center and available to the consortium partners. The first version of our DSL scripts in the daphne folder can be run via a publicly available docker image from daphneeu/daphne, which can be found on the Docker Hub container registry[9]. Instructions on how to run the daphne container image are available in the public daphne-eu/daphne GitHub repository. Lastly, there is a data folder which contains the sample input data for running both benchmarking pipelines. With this ap-

[8] https://mlflow.org

[9] https://hub.docker.com

proach we can run comparable workflows in Python and DaphneDSL. The initial steps are performed within our productive Python implementation up to the point, where we export the preprocessed data to csv files, including the required meta files.

In a nutshell, we have been able to implement the core of the productive machine learning pipeline within DaphneDSL. We are just missing one more vital piece to and can benchmark it against Python, the Random Forest algorithm. Our benchmarking workflows begin with reading in the previously prepared csv files. After splitting this data into train and test sets, we optionally can run a normalization procedure. Afterwards, we plan to feed the data to the Random Forest algorithm. An optimized implementation of the Random Forest algorithm has been developed in Apache SystemDS and, that is currently being ported over to DaphneDSL implemented specifically for us, to help advance our use case. We then calculate the same set of metrics on the test dataset – F1, Precision and Recall. After the Random Forest is available, we can time both implementations, and compare them against each other in more detail.

3.4.1 Benchmarking Environment

For our tests we will use a single notebook (HP EliteBook x360 830 G6) and this environment:

- Hardware:
    - Processor: Intel Core i5-8365U @ 1.60 GHz
    - RAM: 16 GB DDR4 @ 3200 MHz
    - Storage: 475 GB NVMe SSD
- Software:
    - Operating System: Windows 10 Pro 64-bit (10.0, Build 19045)
    - Python 3.10.12; with scikit-learn 1.3.1

We have simple timing mechanism in place, that tracks the execution time, starting from the beginning of the script, until the completion of the last instruction. More detailed benchmarking information will be gathered with the UMLAUT framework from WP9.

3.4.2 Additional Identified Possibilities through DAPHNE

Running the Boruta algorithm for feature selection on the training set requires multiple hours to complete a predefined number of runs. For the productive Python implementation, the default number of runs is 20 but even after running 100 iterations some features remain tentative. This contrasts with the features that have either been rejected or accepted at some point. Speeding up this process would save a substantial amount of HPC resources.

4 Material Degradation Case Study (KAI)

At KAI, we develop accelerated stress test systems for semiconductor devices. Our use case is about investigating the degradation of power semiconductors. As described in D8.1, simulations offer valuable insights in degradation effects which are essential for lifetime models. The electrical measurements from the stress test systems need to be compressed before the actual simulation to reduce the computational effort. Suitable algorithms are the Visvalingam-Whyatt (VW) algorithm (see D8.1 Chapter 4.2.4) and the Ramer-Douglas-Peucker algorithm.

In this chapter, we show that:

- (1) we have extended the pipeline,
- (2) the DAPHNE system infrastructure is used for executing the use case and
- (3) the runtime of our use-case pipeline runs several times faster that its baseline implementation in Python.

4.1 The Ramer-Douglas-Peucker Algorithm

The Ramer-Douglas-Peucker (RDP) algorithm plays an important role at KAI since it has different properties and optimization targets than the VW algorithm. It optimizes for retaining critical points as maxima or minima while VW better retains effective areas. Like the VW algorithm, it is a lossy compression.

The RDP algorithm is a recursive algorithm which step by step chooses points which retain after reduction. To achieve this, an auxiliary line is drawn between the first and the last point. A tolerance band around this line decides which points are withdrawn. If there are points outside of this tolerance band, no points are withdrawn. Instead, the algorithm picks the point with the maximum normal distance to that line and splits the input series into two sub series. The algorithm is then applied to both slices independently. This process is repeated until each slice has no more points outside the tolerance band.

The RDP algorithm has the following parameters:

1. max_points
    - Maximum number of points after simplification
2. tolerance
    - Defines the tolerance band around the auxiliary lines (tolerance into one direction)

The RDP algorithm actually chooses points to retain step by step. It stops at the natural termination criterion (no more points outside tolerance bands) or if the max_points limit is exceeded.

4.2 Dataset Reorganization

The KAI dataset contains a collection of ~13.500 TDMS files. Since the DAPHNE system currently only supports reading CSV files, a reorganization of the data structure is necessary to run experiments.

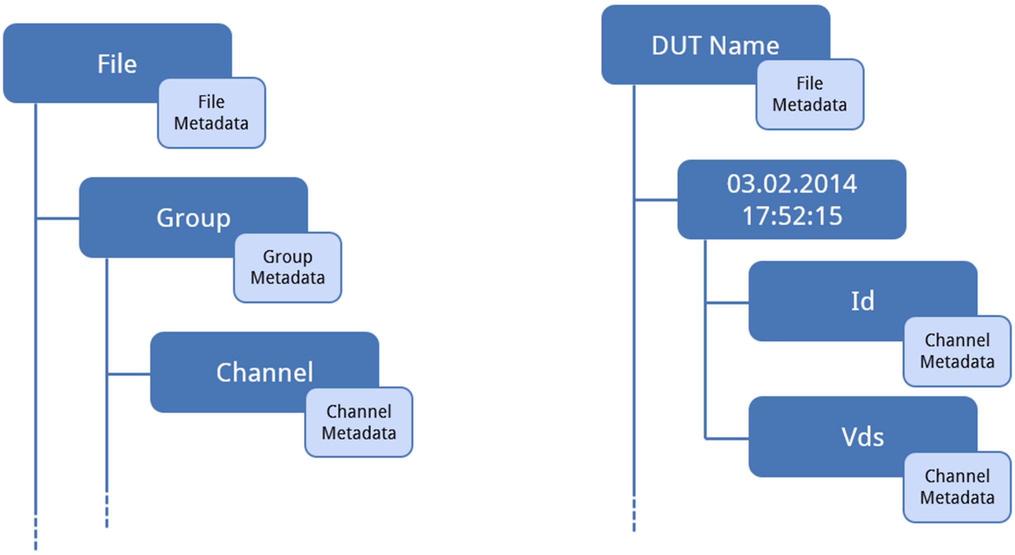
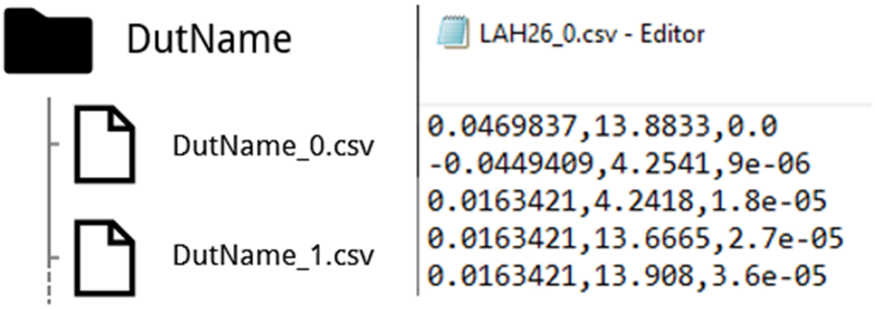


Figure 4.1: TDMS file structure

Figure 4.1 shows the general hierarchy of a TDMS file. Each file contains an arbitrary number of groups, a group in turn contains an arbitrary number of channels. Metadata can be attached to the 3 hierarchy levels (file, group, channel) each. The actual data is contained in the channels. As can be seen in Figure 4.1, within our dataset one TDMS file contains all measurement data which is related to one device under test (DUT). A group contains a datetime stamp, at which the measurement started. The number of groups varies across files. Each group contains exactly two channels, Id and Vds. Put in another way, two sampled waveforms with a duration of a few milliseconds. The sample points are equidistant, but the distance can vary across files. One TDMS file with 1000 recorded cycles (groups) would be represented with dimensions $[1 \times 1000 \times 2]$.



```
0.0469837,13.8833,0.0
-0.0449409,4.2541,9e-06
0.0163421,4.2418,1.8e-05
0.0163421,13.6665,2.7e-05
0.0163421,13.908,3.6e-05
```

Figure 4.2: Dataset CSV representation

CSV is a flat data representation and offers two hierarchy levels (file and column). To obtain an arbitrary number of additional levels, a directory hierarchy can be constructed. Figure 4.2 shows how the TDMS structure is mapped to a representation with directories and CSV files. The TDMS file is replaced by a directory with the same name. Each TDMS group is stored in an own CSV file with the respective enqueued sequence number in the file name. TDMS channels are represented as CSV columns. During this transformation process, the metadata attached to the TDMS files and channels gets lost. The distance between sample points is a relevant metadata key. To maintain this information, an additional column with the relative time is added to each CSV file. The reminder of the metadata is not important for the line simplification use case and is neglected for now.

4.3 DAPHNE Scope Pipeline Definitions

This chapter defines a pipeline which is fully implementable with DAPHNE and represents the Data Reduction component of the initial pipeline definition (D8.1 Chapter 4.1). The pipeline's I/O is adapted as CSV files are used for input and CSV files or stdout for output.

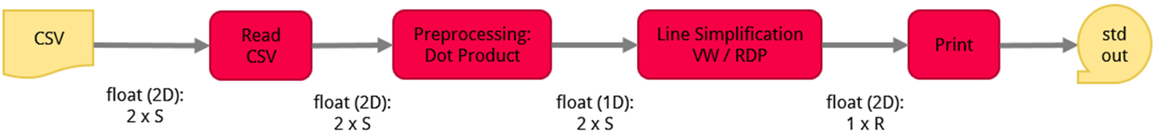


Figure 4.3: Line simplification pipeline for single file (pipeline P4.1)

Figure 4.3 defines the simple line simplification pipeline P4.1. A single CSV file is read (one measurement group in a set of tenth of millions) and forwarded to pre-processing and the actual line simplification algorithm. The output is printed to stdout.

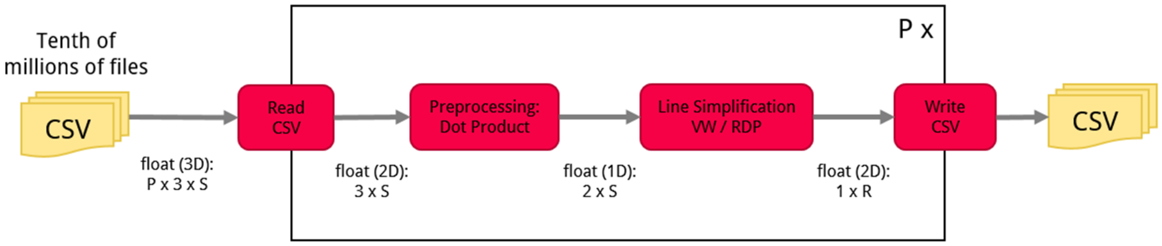


Figure 4.4: Line simplification pipeline for single TDMS file (pipeline P4.2)

Figure 4.4 shows the pipeline definition P4.2 with a larger dataset. A collection of CSV files is read and forwarded through the processing steps. The large number of results are stored in CSV files.

4.4 Python Baseline Implementation

We hold Python implementations of both P4.1 and P4.2, each for both mentioned line simplification algorithms, which serve as baselines. The RDP and VW algorithms are delivered as a KAI internal Python package. This makes use of the numpy[10] library for all linear algebra operations (abs, trapz, stack, concatenate, cross, max, argmax, reshape, append, sort). For reading and writing CSV files, the pandas[11] library is utilized (read_csv, to_csv).

4.5 DAPHNE DSL Implementation

As an improvement to our pipeline, we implemented P4.1 and P4.2 (each for both line simplification algorithms) with the DaphneDSL. This use case can utilize the following built-in DSL functions / features: readFrame / writeFrame, print, pow, sqrt, nrow, abs, aggMin / aggMax, idxMin / idxMax, cbind / rbind, sum, order, seq, script arguments, user defined functions (UDF).

User defined functions allow for a modular programming technique. Both, VW and RDP are modularized via UDFs with several utility UDFs. The following UDFs have been implemented:

[10] https://numpy.org

[11] https://pandas.pydata.org/pandas-docs/stable

- RDP algorithm (see section 4.1)
    - Cross product for R2
    - Frobenius norm between two points
    - Function to calculate all normal distances to the auxiliary line
- VW algorithm (see D8.1)
    - Calculation of triangle areas
    - Checking for termination criterion
    - Cumulative trapezoid

DSL scripts import above mentioned UDFs and execute them.

4.6 Preliminary Benchmarking

The current DSL implementation of our use case allows for preliminary experiments. Simple runtime tests can be executed without any benchmarking framework by making use of the Linux `time` command. The experiments are executed as `time /bin/daphne vw.daphne` and `time python3 vw.py` accordingly. The real time metric is extracted as experiment result. All of the following results were obtained from one single machine: Intel® Core™ i7-10700 @ 2.9 GHz, 16 GB RAM, Ubuntu 20.04 on WSL (on Windows 10 64-bit).

To evaluate the implementation of P4.1, a single measurement cycle was extracted as CSV from a previously selected TDMS file.

Table 2: Preliminary benchmarking result single waveform

Runtime in seconds	VW	RDP
Python Baseline	0.63	1.25
DaphneDSL	0.26	0.2

To evaluate the implementation of P4.2, one TDMS file was chosen and transformed into the adapted data structure as described in section 4.2. This sub dataset contains 9421 CSV files / measurement cycles. Each representing one input to the reduction algorithms. Hence the algorithms are executed 9421 times.

Table 3: Preliminary benchmarking results whole dataset

Runtime in seconds	VW	RDP
Python Baseline	200	110
DaphneDSL	630	34

For some reason unclear to us, the DAPHNE implementation of the VW-algorithm is slower in the case of running the reduction step for reading multiple files. This is subject to further investigations with the core developers.

4.7 Advanced Use Case Definition

In the course of the DAPHNE project, we at KAI developed an advanced material degradation use case [2]. This is about failure prediction of devices under test based on the raw measurement data contained in our dataset. A CNN model was developed which gets fed with measurement cycles in a tensor representation and predicts the remaining useful lifetime. The approach defines a regression problem with supervised learning.

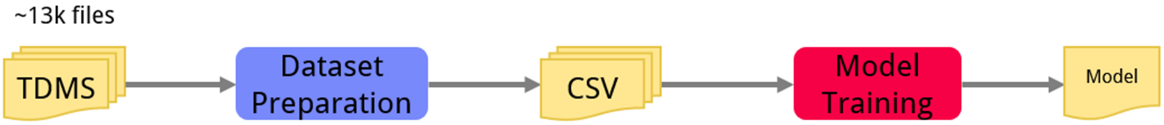


Figure 4.5: Advanced Use Case Training Pipeline

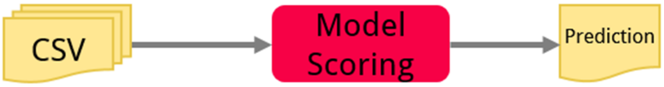


Figure 4.6: Advanced Use Case Inference Pipeline

Figure 4.5 defines the failure prediction pipeline. For the training, some data preparation steps are necessary which will take place outside of DAPHNE. The actual model training and inference (see Figure 4.6) offer the potential to be implemented with DAPHNE. A Python implementation based on PyTorch[12] is in place.

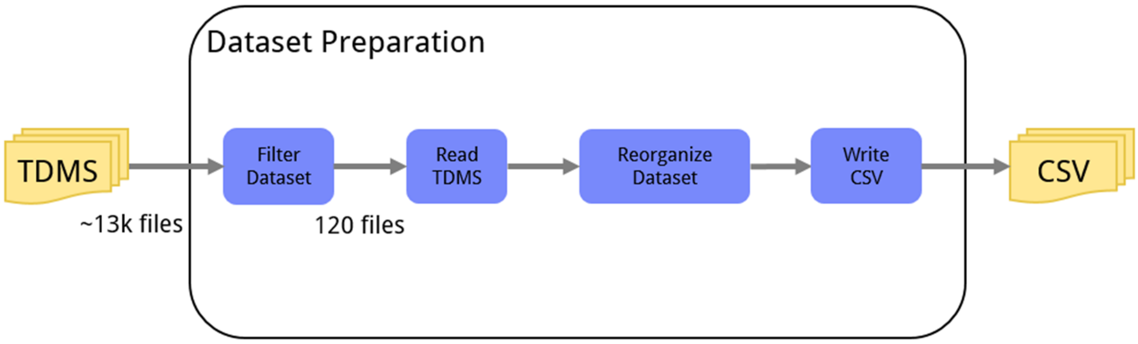


Figure 4.7: Advanced use case dataset preparation

Figure 4.7 shows the dataset preparation step. It needs to be restructured as defined in section 4.2 but it additionally needs to be pre-filtered.

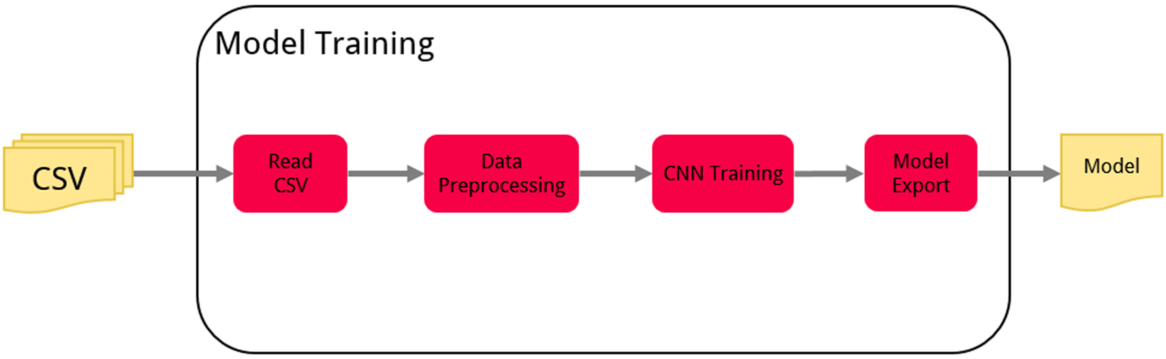


Figure 4.8: Advanced use case detailed model training

[12] https://pytorch.org

As can be seen in Figure 4.8, the training pipeline consists of a classic machine learning workflow. According to our color-coding scheme introduced in D8.1, the red boxes could be addressed completely by the DAPHNE system.

5 Automotive Vehicle Development Case Study: Ejector geometry optimization (AVL 1)

5.1 Summary of Pipeline Improvements

Several improvements have been provided to the initial ejector dimensioning pipeline since deliverable D8.1 to decrease user interaction and therefore increase efficiency as well as diversification of the Machine Learning tools.

Summarized the following major features and enhancements are now available:

- Full automatization of the pipeline – Active DOE Python script
- Additional Machine Learning pipeline (KRR method implementation in Python)
- DaphneLib Implementation of Machine Learning components

The Active DOE targets the maximization of automatization capabilities to guarantee an efficient dataset extension into new operating condition regions. An additional pipeline implementation in Python tackles the diversification of the workflow in order to stay flexible in accessing several Machine Learning methods besides the ones implemented in AVL's in-house tool Cameo. DaphneLib is used to test the speed up potential, which is important for upscaling the ejector dimensioning workflow to larger datasets or multiple feature variations to improve prediction accuracy.

5.2 Active DOE Implementation

The benefit of the Active DOE approach is the capability of fast ejector data generation with a low amount of user interaction. This is very important when completely new operating conditions are investigated in a new ejector dimensioning project. In such a case the prediction model does not perform optimal before seeing new data that corresponds to such operation conditions.

Another benefit is the possibility to start a DOE in times of low load on High Performance Clusters that can be utilized with this method to increase the ejector database without much user interaction for upcoming projects.

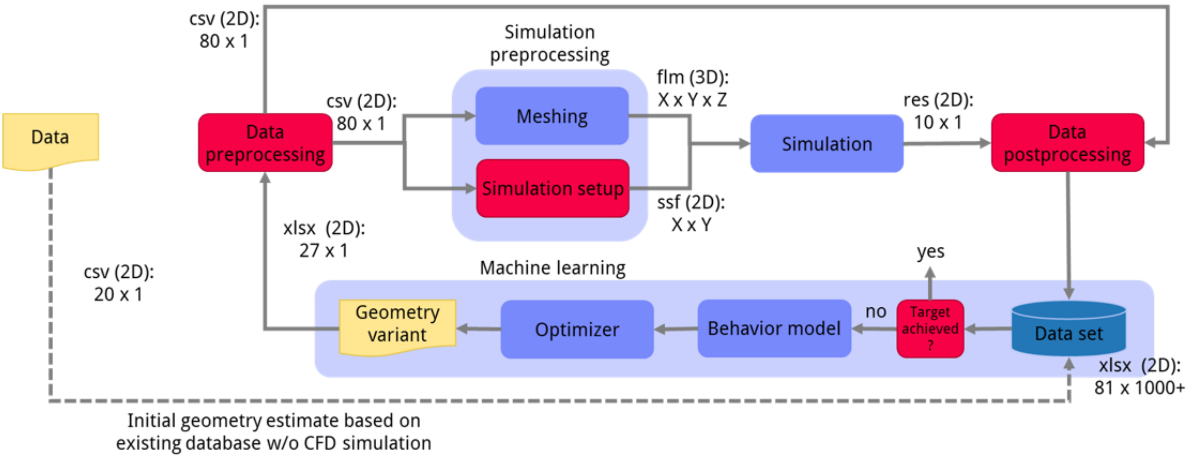


Figure 5.1: Ejector dimensioning pipeline

The pipeline, shown in Figure 5.1, describes the data-driven ejector layout. Initially, several manual steps were necessary:

- set up parameter space,
- building behavior model with dataset,
- perform optimization to operating conditions,
- initiate preprocessing and start simulations via scripts,
- initiate postprocessing via scripts.

A completely automated loop is called Active DOE and is targeted to increase the productivity of this Use Case. The implementation is realized with a Python script. On the way towards automatization the current pipeline has just one manual step left which is the Active DOE configuration. The configuration includes a parameter space definition, setting up the cluster on which the simulations should be performed and some other setup possibilities e.g., how many parallel simulations should be performed, how many loops should be executed. The configuration is set up via .json files which are loaded by the Python script.

5.2.1 Active DOE Components

The Active DOE is realized with a Python script that handles the interaction between the Machine Learning, executed on Windows (Cameo), and the simulation, executed on Linux.

Cameo provides the behavior model training, variation parameter distribution and the optimization to generate adequate design parameter configurations that finally will be simulated by CFD code to generate new data for the database.

On Linux side, the meshing of the candidates, setup and running of the 3D-CFD simulation and the postprocessing is performed. The simulation is carried out with AVL's in-house commercial CFD solver named Fire.

All these elements are managed by the active DOE Python script which is running on a Linux workstation.

5.2.2 Communication between Components

The communication to Cameo is established via a web API and for which a local host must be established on Windows side. Cameo provides a Python class "active_doe_client" that acts as a template for such tasks. The communication from Python to the web API is performed with the python library requests[13].

On the Linux side, the Active DOE Python script runs the shell scripts from the initial pipeline for preprocessing, postprocessing and starting of the simulation via the os[14] Python library. The simulation is carried out on a HPC system, the connection is established utilizing the SSH protocol.

[13] https://requests.readthedocs.io

[14] https://docs.python.org/3/library/os.html

5.3 Machine Learning Python Implementation

A Python script is developed in order to increase the accessibility of Machine Learning models provided by several Python libraries such as TensorFlow[15], scikit-learn[16] etc.

This capability can be seen as a diversification in our pipeline to increase flexibility in terms of predictive model architectures parallel to AVL's strong in-house tool Cameo, which does not provide the variety of models like the Python framework allows.

A Kernel Ridge Regression[17] model is set up in this Python implementation. Another key aspect is the identification of accurate model architectures in order to get them also implemented in Cameo. In the Python implementation we can identify various calculations that can be executed in DaphneDSL via the DaphneLib API.

In Figure 5.1, the related blocks that are coded in Python are: *Behavior model* and *Optimizer*.

The Python script is stored on the daphne GitLab repository that is maintained by Know-Center and available to the consortium partners.

5.3.1 Predictive Model Creation

Figure 5.2 shows the steps that are covered by the Python script in order to create a predictive model.

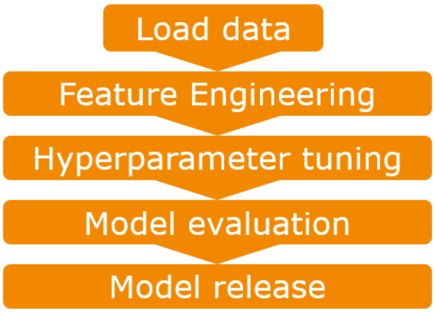


Figure 5.2: Model creation workflow

5.3.1.1 Load data & Feature Engineering

The first step is to load the dataset which is stored in a .csv file format to build up the feature matrix X and the label vector y. Features cover geometrical parameters and operating conditions, and the predicted label is the ejectors suction pressure. Afterwards the feature engineering is applied. Currently all features and labels are MinMax[18] scaled. In addition, a data cleaning is applied which aims to erase outliers to increase the predictive model quality. The data is split up in test data (20%) and trainings data (80%).

[15] https://www.tensorflow.org

[16] https://scikit-learn.org

[17] https://scikit-learn.org/stable/modules/generated/sklearn.kernel_ridge.KernelRidge.html#sklearn.kernel_ridge.KernelRidge

[18] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

5.3.1.2 Hyperparameter Tuning & Evaluation

The tuning is performed as a Grid Search utilizing the scikit-learn implementation[19] that features Cross Validation. The following model parameters are optimized.

- Kernel-function
- Regularization parameter
- Kernel parameter

The evaluation is performed on the test data by analyzing R2 score and mean squared errors.

5.3.1.3 Model release

The trained Kernel Ridge Regression instance and all scaler instances of the scikit-learn classes are stored utilizing the joblib[20] Python library.

5.3.2 Utilization of Predictive Model

Figure 5.3 shows the steps to utilize the predictive model.

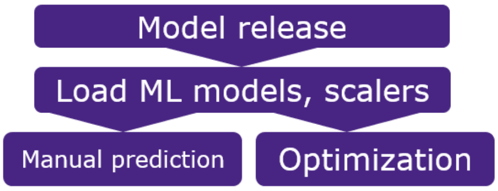


Figure 5.3: Predictive model utilization

At first the Kernel Ridge Regression model and all scalers are loaded. There are two different possibilities to utilize this Python script. The first one tackles the prediction of a specified geometry and operating condition and is represented by the block manual prediction. The second possibility is to optimize a geometry based on fixed operating conditions. The optimization is performed by a genetic algorithm that utilizes the following concepts:

- survival of the fittest,
- mutation,
- crossover,
- random candidate generation.

5.4 Daphne Lib Implementation

DaphneLib is a Python API that can gather computations and perform them with DaphneDSL.

With this approach it is not necessary to learn a new programming language and it opens the possibility to directly compute calculations in DaphneDSL from Python code.

To execute the Python workflow in combination with DaphneLib a container must be set up, this is described on GitHub[21]. Afterwards a virtual Python environment must be created in which all necessary Python libraries must be installed. (sklearn, joblib)

[19] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
[20] https://joblib.readthedocs.io
[21] https://github.com/daphne-eu/daphne/blob/main/doc/GettingStarted.md

In the Python workflow described in section 5.3, two parts can currently be executed utilizing DaphneLib. The first one is the MinMax scaling of the features and the second one is the prediction utilizing a trained Kernel Ridge Regression model.

The prediction is picked to be executed in DaphneLib due to the huge number of calls in the optimization progress. For the next deliverable more components will be implemented (e.g., model training) with DaphneLib to perform the benchmarking of the stand-alone Python workflow vs. the DaphneLib enhanced Python workflow.

6 Automotive Vehicle Development Case Study: Virtual prototype development (AVL 2)

In deliverable D8.1, we described the overall goal of this case study (i.e., to create process mining solutions) as well as the initial data processing pipeline setup consisting of 3 parts (data generation, model training, prediction). Since then, we have been incorporating DaphneLib into the existing pipeline implementation and did a first benchmarking, which will be described in more detail in the next section of this chapter. Moreover, we have conceived a possible solution to a major limitation of the current pipeline, that we can tackle with the expertise and technology available in the DAPHNE project consortium. Consequently, substantial conceptual work has been carried out to get this solution idea ready for implementation, which will be laid out in more detail in the latter sections of this chapter.

6.1 Overview of the improved Pipeline

This case study is dealing with a pipeline that consists of three parts that build on top of each other, as described in deliverable 8.1. Thus, work on the different parts of the pipeline needs to be done in sequence. Consequently, work has focused on the first part of the overall pipeline. The remainder of this chapter will also focus on the first part.

This sub-pipeline is shown in Figure 6.1. It creates time series data by running multiple simulations – each simulation contributes one data point – and putting the data points into a temporal order. It is implemented partly in proprietary AVL software (e.g., Model.CONNECT^{TM}) and partly in Python. The parts implemented in Python have been refactored using DaphneLib. Specifically, refactoring has focused on the "Create time series" step, since it is the numerically most time-consuming step in the sub-pipeline. This step is implemented in one Python script that executes a sequence of multiple array operations, with arrays consisting of typically 104 elements.

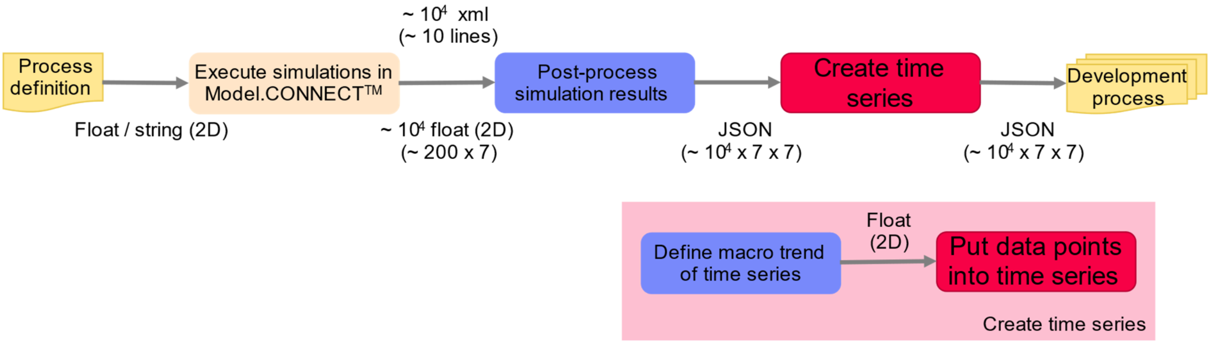


Figure 6.1: Pipeline for data generation

Array operations that were mostly using NumPy methods have been replaced with methods provided by DaphneLib (version 0.2). In general, the introduction of DaphneLib methods into the Python script was convenient, because individual lines of code could be changed while retaining the overall structure and workflow of the code. However, minor differences in numerical results have been observed compared to the original implementation, but these differences are not relevant for the pipeline.

The next steps will be to benchmark the pipeline on more capable hardware platforms (e.g., workstations or clusters) and improve the integration of DaphneLib into the sub-pipeline with the goal of decreasing the execution time of the sub-pipeline in comparison to the original implementation.

6.2 Reworking of Data Generation Sub-Pipeline

The major limitation of the sub-pipeline shown in Figure 6.1 is that the step "Put data points into time series" involves manual engagement of the user to create time series data of sufficient data quality, i.e., time series data that shows properties and trends as observed in real-world data. Thus, to increase the level of automation, this step shall be reworked by introducing other algorithmic approaches and shall be implemented using DaphneLib or DaphneDSL.

To overcome this major limitation, different approaches have been analyzed and genetic algorithms (cf. [3]) have been identified as the most promising and resourceful one. In collaboration with the DAPHNE project partner University of Maribor, which contributes expertise on genetic algorithms, the basic concepts of genetic algorithms have been thoroughly reviewed and the conceptual framework has been created which maps the defining features of genetic algorithms to this case study. In the following sections, the results of this analysis and the created framework will be described in more detail.

6.2.1 Review of automotive Product Development Processes

In the context of this case study, the goal and scope of automotive development processes can be characterized as follows:

- Find near-optimal solutions that fall within specified acceptable tolerances.
- Iterate and optimize the product design until all product requirements are satisfied.
- Balance competing product constraints and goals, e.g., balance product cost with product performance.

This is in accordance with the characterization of engineering problems provided by [4], which argues that engineering problems – including product development, as one specific example of an engineering problem – can be considered optimization or search problems.

In the research area of search-based software engineering (SBSE), this idea is used to cast software engineering problems as search / optimization problems and apply search / optimization techniques to solve them. In [5], which was used as a basis to develop the conceptual framework laid out in this and the following sections, this approach is used to automatically improve software code using genetic algorithms.

In SBSE, engineering problems are cast as search / optimization problems. Since the characterization of engineering problems that SBSE is based on also applies to automotive product development, the key idea here is to apply the approaches from SBSE, particularly metaheuristic search-based optimization techniques and genetic algorithms, to this case study in order to improve and automate the data generation sub-pipeline, specifically its "Create time series step".

Following [4], there are 3 requirements to consider when casting product development as a search / optimization problem:

- Representations of candidate solutions that allow for symbolic manipulation.
- Fitness functions that are defined in terms of the candidate solution representation and that can characterize good solutions.
- Manipulation operators to mutate one candidate solution into another candidate solutions. If manipulation operations include cross-over to produce child solutions from parent solutions, genetic algorithms are applicable.

In the following sections, these 3 generic requirements and their mapping to this case study will be analyzed in more detail to build up step-by-step the conceptual framework of how to apply optimization techniques and genetic algorithms to this case study.

6.2.2 Representation of Candidate Solutions

In the context of this case study, automotive product development can be simplified such that candidate solutions can be represented in terms of two numerical vectors: A vector $\overrightarrow{CV}$ and a vector $\vec{P}$. The vector $\overrightarrow{CV}$ contains a list of characteristic values (also often referred to as product key performance indicators or KPIs) and the vector $\vec{P}$ contains a list of product parameters.

Product parameters are numerical values that encode and are derived from product design, e.g., the vehicle mass, the vehicle wheelbase, or the coefficient of aerodynamic drag. All these exemplary parameters are derived from the product design, e.g., from the CAD model of the vehicle. Product design and consequently product parameters are subject to optimization.

Product characteristic values, on the other hand, are numerical values that encode the product performance, e.g., the electric driving range, the acceleration time to 100 kph, or the battery charging time. Product characteristic values are subject to product requirements and target characteristic values: The product design must be optimized such that the product characteristic values can meet the target characteristic values, which is the quintessential goal of product development.

The vectors $\overrightarrow{CV}$ and $\vec{P}$ are also linked with each other: The characteristic values are determined from the product design using tests or simulations, which means that $\overrightarrow{CV} = \vec{f}(\vec{P})$. In the context of this case study, an available vehicle simulation model will be used as the function $\vec{f}$.

6.2.3 Fitness Function

As explained in the previous section, product characteristic values are subject to target characteristic values and product design / product parameters must be optimized such that characteristic values meet associated target values. Consequently, a candidate solution’s fitness can be quantified by how close a candidate solution’s characteristic values come to the target values.

However, only measuring the deviation of a characteristic value to its target and summing up the deviations of all characteristic values is not correct, because quantifying target achievement is target-specific: For example, for the characteristic value “electric driving range” a characteristic value smaller than the target indicates an unfit candidate solution and a value larger than the target indicates an overachieving candidate solution, for the characteristic value “acceleration time” it is the other way around.

6.2.4 Manipulation Operators

To mutate one candidate solution into another candidate solutions, the numerical values of the parameters in vector $\vec{P}$ must be changed. However, these manipulation operations are not arbitrary but are subject to parameter-specific constraints. For example, the parameter "vehicle mass" cannot be changed to negative values.

Another manipulation operation is cross-over, i.e., creating a child solution from parent solutions while retaining features from all parents. Cross-over manipulation operations can be mapped to simultaneous engineering from real-world product development: Automotive product development is usually split up into several concurrent workstreams, e.g., mechanical design, thermal design etc. Each workstream works on optimizing a subset of $\vec{P}$, e.g., mechanical design only optimizes parameters related to the vehicle chassis. At discrete times during development, optimized candidate solutions from the different workstreams are merged to create a common baseline, which provides the common starting candidate solution for the next subset optimization in each workstream. The baseline child candidate solution can be created from parent candidate solutions coming from the different workstreams using cross-over.

6.2.5 Next Steps

The previous sections have described the framework of how the basic concepts used in optimization and specifically in genetic algorithms can be mapped to this case study / to the area of automotive product development.

The next step is to further extend and improve this framework in collaboration with the DAPHNE project partner University of Maribor, i.e., create a sub-pipeline architecture similar to the one shown in Figure 6.1 specifically for the step "Create time series". Since the implementation will likely be numerically demanding, the Daphne solutions must be leveraged and possibly need to be extended to cater to this case study's needs.

7 References

[1] A. Laber, M. Gebser, K. Schekotihin and Y. Yang, "Predicting Ion Beam Tuning Success in Semiconductor Manufacturing," in *2022 14th International Conference on Advanced Semiconductor Devices and Microsystems (ASDAM)*, Smolenice, Slovakia, 2022.

[2] M. Ghoneim, "Modelling Degradation of Semiconductor Devices for Lifetime Prediction based on Stress Test Data," Alpen-Adria-Universität Klagenfurt, Klagenfurt, 2023.

[3] S. Silva and L. Paquete, "GECCO '23: Proceedings of the Genetic and Evolutionary Computation Conference," Lisbon, 2023.

[4] M. Harman, "Search-based software engineering," *Information and Sofware Technology,* vol. 43, no. 14, 2001.

[5] B. R. Bruce, Reducing Energy Consumption Using Genetic Improvement, vol. Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15), New York: Association for Computing Machinery, 2015, pp. 1327-1334.