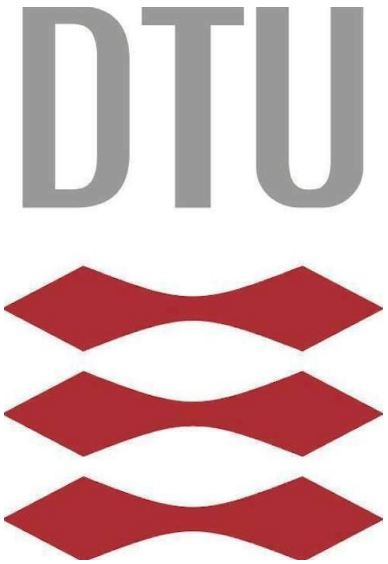
TECHNICAL UNIVERSITY OF DENMARK



A user manual for FleetYeeters

s194237 — David Ari Ostenfeldt
s194246 — Kristian Rhindal Møllmann
s194247 — Dennis Chenxi Zhuang
s194249 — Kristoffer Marboe
s194267 — Kasper Niklas Kjær Hansen

Machine-learning approach for real-time assessment of road pavement service life based on vehicle fleet data

June, 2024

Contents

1 Summary

This report is a user manual for our project. It aims to provide an overview of the Git repository we've set up and describes our thought process. While the main text gives a general overview and context, the accompanying Python notebooks delve into more technical details and document our experiments. This manual aims to help users understand and navigate the repository effectively.

Figure 1 shows the entire pipeline of the project. Each step along the way will be further explained in the relevant sections. In summary, the data is preprocessed to align targets and training data, then features are extracted and stored for selected signals. Finally, an MLP is trained and used to predict four key point indicators for the condition of the road.

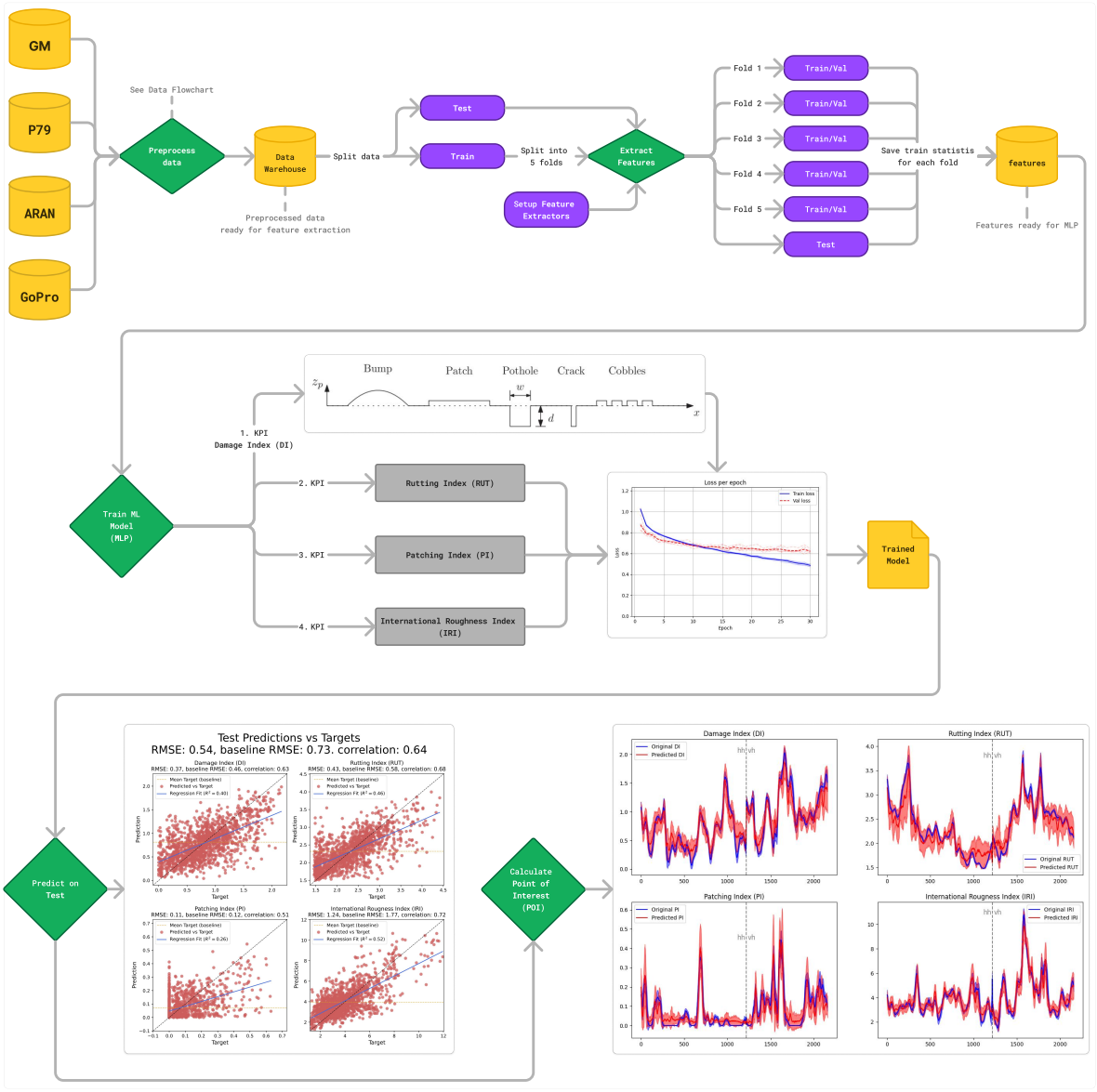


Figure 1: Flowchart of the entire project pipeline.

2 An introduction to the problem

Currently, the condition of the Danish road system is being monitored by measurements from the P79 and ARAN vehicles. However, given the limited amount of vehicles available, and the cost to operate them, getting data for large parts of the Danish road system is an arduous and time-consuming task.

Modern cars are equipped with numerous sensors. By analysing the data collected from these sensors during driving, we aim to infer the condition of the road. The goal of this project is to evaluate road conditions based on data from multiple cars driving over the same section of road. By doing so, we aim to gain insights into the road's condition and any potential issues.

3 The raw data

The raw data used in this project is described in "Live Road Assessment based on modern car sensors (LiRA): Practical guide" [1].

3.1 P79 - Profilometer

The P79 Profilometer is a vehicle equipped with 25 point-lasers to measure road surface profiles, collecting data on both longitudinal and transversal elevations. As the vehicle moves, lasers capture the road's surface elevation every 10 cm. Based on the two lasers directly in front of the front wheels of the vehicle, road conditions are estimated. The International Roughness Index (IRI) reflects road roughness using a quarter-car model simulation, while rut depth measures surface deformation along wheel paths. Both values are computed based on 10 meter intervals. This data is crucial for assessing road conditions and guiding maintenance decisions. For more details, see [1].

3.2 ARAN9000 - multi-functional road-testing vehicle

The ARAN9000 is a sophisticated vehicle designed to collect various road surface performance data. It is equipped with multiple systems: point-lasers in front to capture roughness and texture data, cameras on the windshield for Right of Way (RoW) images, and a Laser Crack Measurement System (LCMS) at the back to detail pavement surface conditions. The LCMS uses line-lasers to generate 2D and 3D images of the pavement to identify distresses in the road such as cracks, raveling, patches, potholes, and bleeding. This comprehensive data collection aids in detailed road condition assessment and maintenance planning, and in this project, the data is used to compute a damage index (DI) and patches index (PI) to asses these aspects of road condition. For more details, once again see [1].

4 The Processed Data

The purpose of this section is to give a brief overview of how data is handled. A more detailed description of each processing step can be found in our `notebooks/make_dataset.ipynb`.

What do we have? Data from different sources (GM, P79, Aran, GoPro) that are *not aligned*.

What is the goal? Combine data from the different sources into one file, adhering to the data-format expectations of our machine learning models.

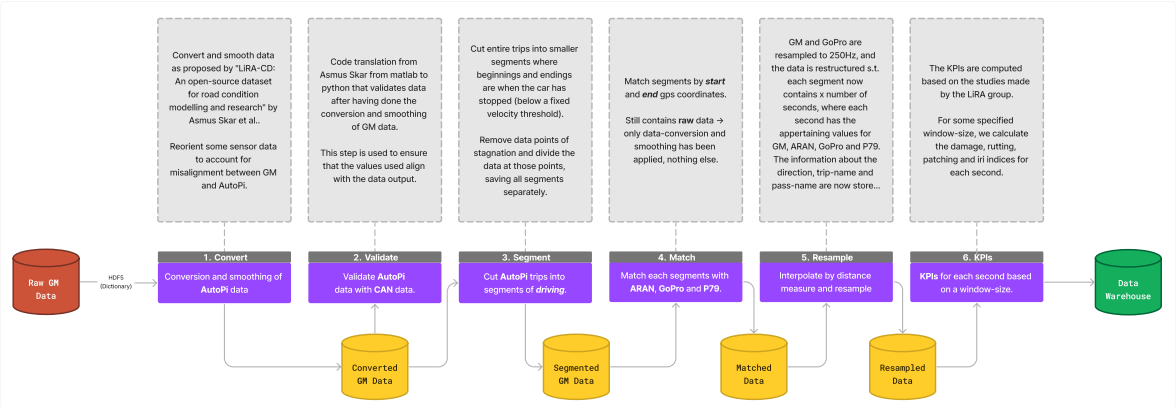


Figure 2: Flow-chart with explanations for each step in the data pipeline used for this project.

To achieve our goal, we made a data pipeline with six sequential steps as seen in Figure 2, going from raw data [1] to processed data.

Step one is to ensure that the sensor data measured directly from the CAN-bus and AutoPi in the GM cars match. To do this we convert the values as by the convention of Skar et al. [2], also further specified in the notebook. Additionally, to account for variability in data caused by measurement noise, LOWESS smoothing is done on selected columns in the converted data (mainly the acceleration measurements). Since the AutoPi and CAN-bus both measure longitudinal and transversal accelerations, we can compare the measurements to align any systematic deviations between the systems. Through this validation, it was found that the sensors were correctly aligned.

With the GM data being recorded continuously from start to finish for each trip and pass, it is unavoidable that the data will contain measurements from where the car is (close to) stationary i.e. is sitting in traffic or at a traffic light. Because these data points add no new information about the road condition, we decided to split the data into segments based on where the cars were found to be stationary. Doing this also allowed us to reconstruct the data structure from `direction -> trip-name -> pass-name -> measurements` to have the segments at the top-level, `segment -> {direction, trip-name, pass-name}, measurements`[1].

Next up, we match the reconfigured data structure of the GM data with that of P79, ARAN and GoPro. This is done based on GPS locations. In this step, we also create a sub-step in which we append some additional data to P79. In the P79 data, the columns used to calculate the IRI and RUT KPIs contain a lot of `NaN` values and are prone to outliers, thus to address this problem, we have an additional data-set that contains more accurate measurements of the road conditions. This simply results in P79 having four additional columns.

With the matched data, we notice that different sensors have different sampling frequencies, so to ensure that when doing a lookup at a specific point in a segment does not yield an empty value, we resample everything to 250Hz by linearly interpolating each sensor individually. Since each point in

[1] **NB.** the brackets '{}' indicate that the values inside the brackets are attributes of the HDF5 file for whatever is on the left side of the arrow.

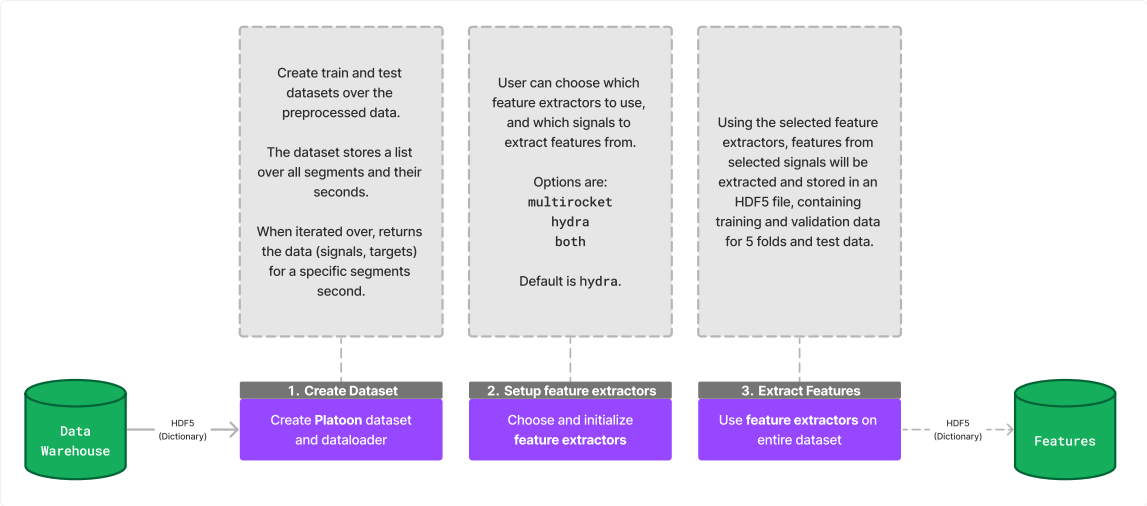


Figure 3: Feature extraction pipeline

the data set contains exactly one measurement from each sensor, the segments are split into second bits with 250 resampled measurements from each sensor. The structure of the training data thus becomes,

`segment -> {direction, trip-name, pass-name}, second -> measurements.`

Lastly, we calculate (slightly modified method from [1]) a damage-, rutting-, patching- and IRI-index as our four KPIs that help describe the road condition at each specified second. These KPIs act as labels for our ML models,

`segment -> {direction, trip-name, pass-name}, second -> measurements, kpi.`

5 Methods and models

With the data sorted, the next step of the journey is to predict the road's condition. To do that, we turn to two promising methods for feature extraction in time series data, namely MultiRocket and HYDRA [3, 4]. These two methods are used to extract features based on different combinations of vehicle sensors to see if any of these are more favourable than others. Based on the survey conducted by Middlehurst et al. [5], HydraMultiRocket performed marginally worse than the best model at the time, HIVE-COTE 2.0, however, it was roughly 200 times faster to train. This is a significant speedup, that allows for more experimentation. Subsequently, various simple MLP models are trained on the extracted features to predict the KPIs describing the road condition. Finally, the trained models are assessed using new, unseen data points.

5.1 Feature extraction

The two methods briefly discussed below are designed for time series classification, where the methods themselves are used to extract useful features from an entire time series, after which these features are classified into some fixed number of classes often using a simple linear classifier. We take almost the same approach, but instead of using the feature extractors in a time series classification setting, we use them in a time series extrinsic regression (TSER) setting to predict the

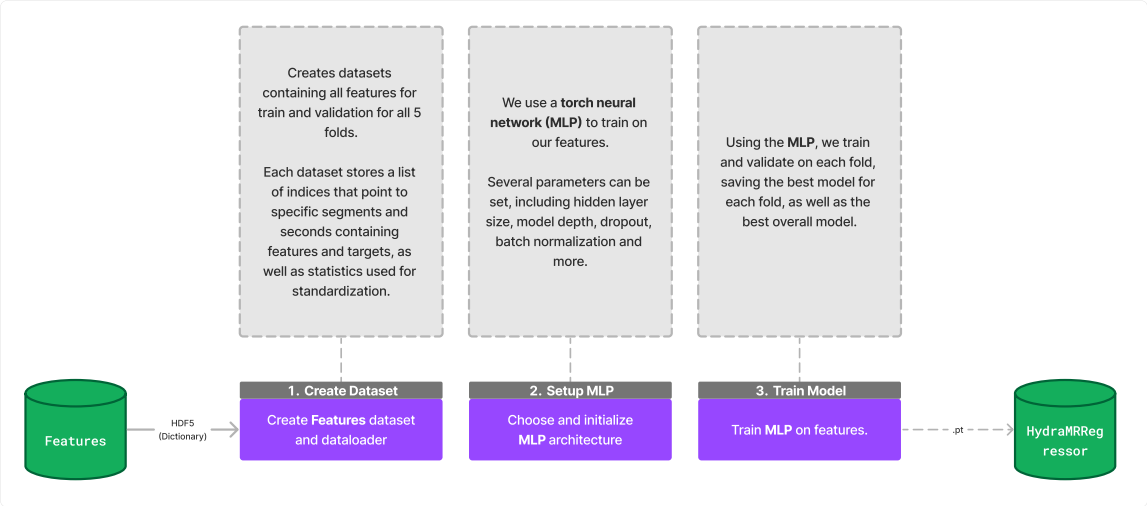


Figure 4: Model training pipeline

continuous KPIs - see [6] for details on TSER. The implementation can be found in the repository at `src/data/feature_extraction.py` with a demonstration notebook at `notebooks/feature/feature_extraction.ipynb`. Figure 3 shows the feature extraction pipeline.

5.1.1 MultiRocket

MultiRocket [3] builds on the concepts first presented in ROCKET [7] and MINIROCKET [8], where random convolutional kernels, combined with a simple linear classifier, showed state-of-the-art accuracy in time series classification, using a fraction of the computational expense of the existing methods. MultiRocket does so "by adding multiple pooling operators and transformations to improve the diversity of the features generated", while not only operating on the raw inputs but also on the first-order differences of the input signal. With MultiRocket, the user can specify how many features the method should produce - we have chosen 50,000 (which results in exactly 49,728 features) throughout testing, as this is the default value.

5.1.2 HYDRA

HYDRA [4] is presented as a simple and fast dictionary-based method for time series classification, that uses competing convolutional kernels, by combining concepts from the aforementioned ROCKET model and conventional dictionary methods. The authors note, that HYDRA is faster and more accurate than existing methods and that it can be combined with ROCKET-based methods to further increase performance. For the HYDRA model, the user can specify the number of groups, g, and the number of kernels per group, k, which, in turn, will impact the resulting number of features. Again we use the default values ($g = 64$, $k = 8$) and with a signal input length of 250, it will result in 5,120 features.

5.2 Training

Having extracted features, we can begin to investigate what information they hold. This is done by training simple MLP models on either of the two extracted features alone and together, for different

combinations of input features. Figure 4 shows the model training pipeline.

We investigated features extracted using *all features*, *only z acceleration*, *accelerations, speeds and yaw* and *most of the signals*. The reasoning behind this is that we theorised that information about, e.g., whether the passenger seat has its seat belt clipped in might not be too informative when predicting the road's condition. For each of these input feature combinations, we trained models on the extracted MultiRocket features, the HYDRA features and a concatenation of them both. All initial experiments yielded a lower loss when using only HYDRA features, an example of which can be seen in Figure 5. In the second to last "column", we see runs for each of the three feature extraction combinations, where all the best-performing runs traverse the HYDRA features node at the bottom. Additionally, the models trained on all input features consistently outperformed all other combinations of input features.

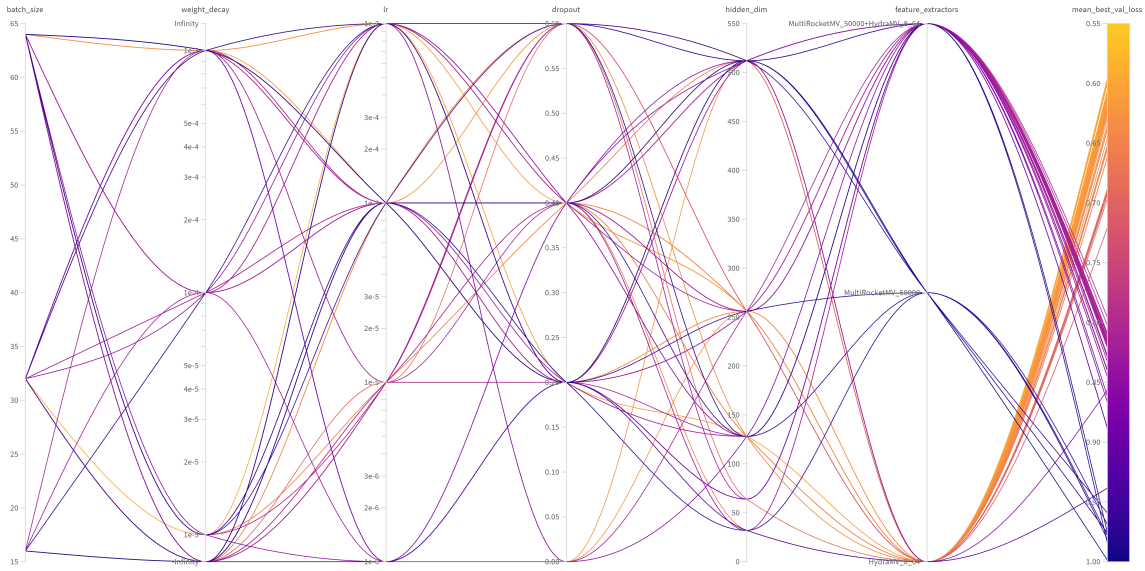


Figure 5: Weights & Biases sweep for features extracted on all inputs. Many different hyperparameter combinations are used with the mean best validation loss as the target across a 5-fold cross-validation.

This led us to investigate only the HYDRA further features while increasing the model complexity to squeeze the last juice out of the data, see Figure 6. From these tests, we obtained a minimum mean best validation loss, in the form of MSE, of approximately 0.58.

It was theorised that the model performed better with all input features because it included location data from GPS and odometer signals. Additional experiments were therefore conducted without these features to eliminate undesired information between the training and validation data that would not be accessible in real-world situations. Doing so decreased the performance, resulting in a minimum mean best validation loss of 0.70.

As a final caveat, in the latter stages of the project, a bug was found in the calculation of the IRI KPI which proved to be hindering the model in learning said KPI. After having fixed the bug, the models performed significantly better, reaching minimum mean best validation losses as low as 0.57 when training on HYDRA features based on all input features except those related to location.

The training script is located at `src/train_hydra_mr.py`, and the model can be found at

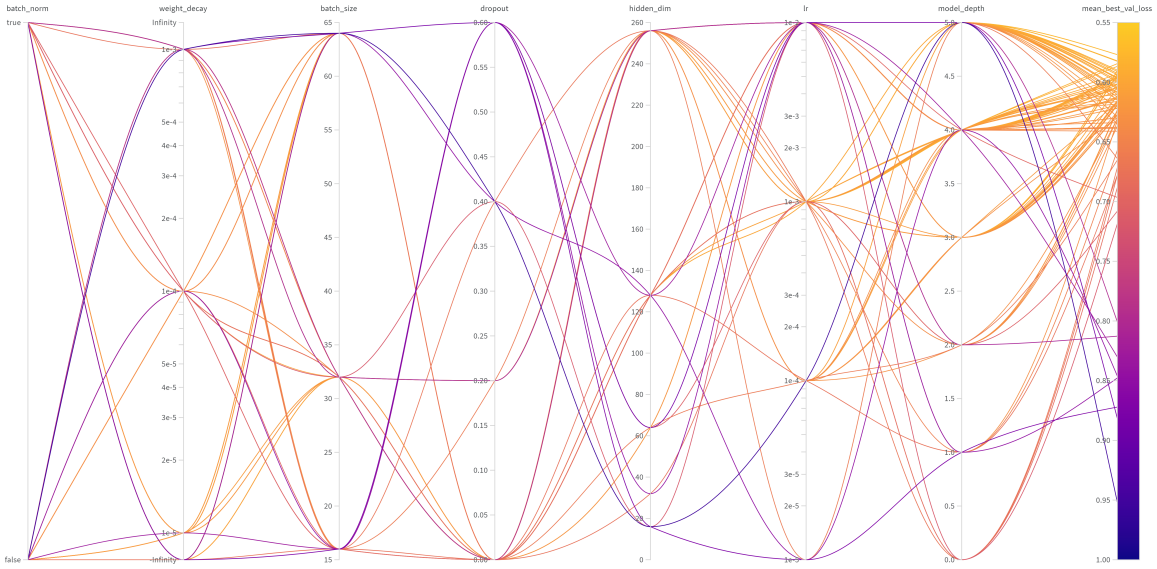


Figure 6: Weights & Biases sweep for HYDRA features extracted on all inputs. Many different hyperparameter combinations are used with the mean best validation loss as the target across a 5-fold cross-validation.

`src/models/hydramr.py`. A demonstration of the training script is available at `notebooks/train_model.py`.

5.3 Predicting

To finally evaluate the methods we picked out the model with the lowest validation loss from the training with the lowest mean best validation loss across all five cross-validation folds. An example of the best model trained on HYDRA features based on all input features except location data is seen in Figure 7. Here we see that the model captures some of the structures in the data, thus leading to relatively high correlations - especially for the damage and rutting index.

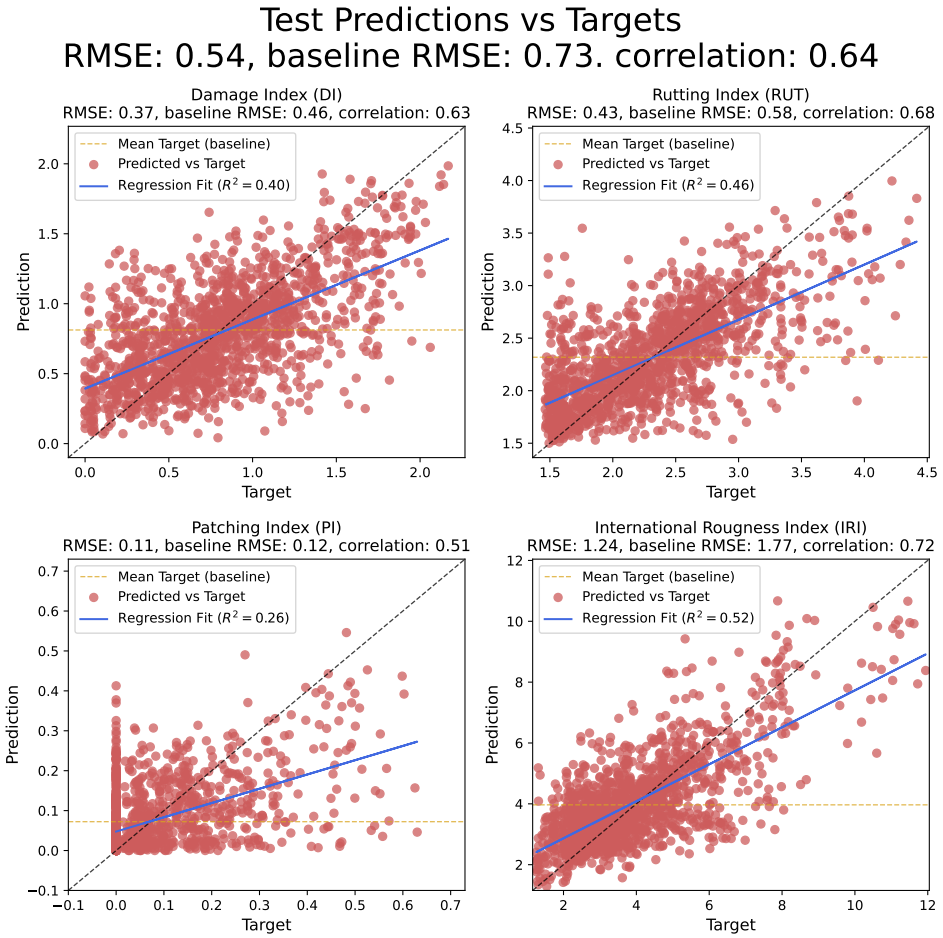


Figure 7: Test prediction across the four KPIs with RMSE and correlation coefficients. The model is trained on HYDRA features based on all input features except GPS and odometer.

6 Points of Interest

At the end of our pipeline is the transition from the predictions to the actual real-world coordinates, which can be used to create a heat map of the road's condition. The idea and implementation will be explained using Figure 8.

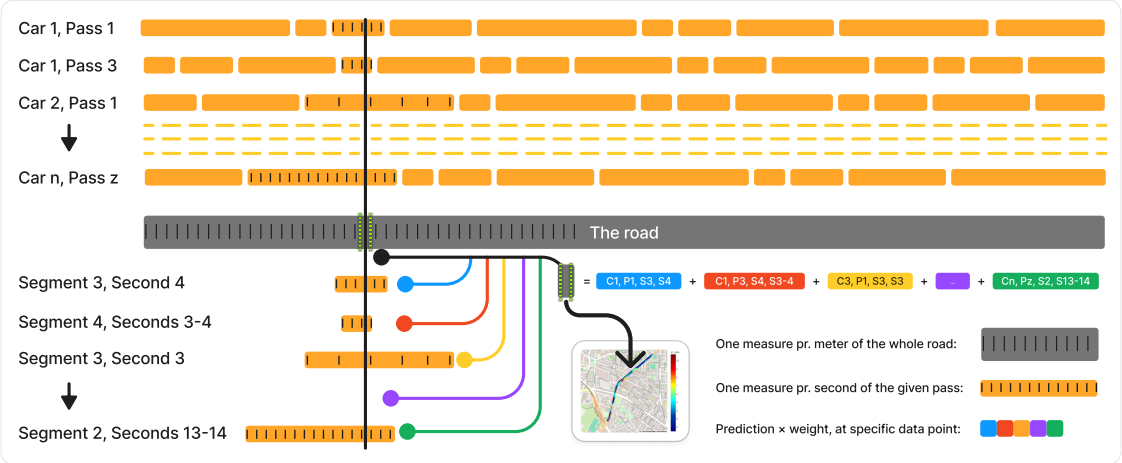


Figure 8: Overview of the AOI idea and implementation.

First, we will look at the upper half of Figure 8. The grey line in the middle resembles the entire right side of the road, which is covered in this study. It is divided into points of interest (POIs); one for each meter. The orange lines above are all the cars' passes, i.e. their drive from the start to the end of the road. These passes are divided into segments based on when the car had a speed below 5 km/h. In total, there are 218 segments, with the first 82 of them being on the right side of the road and the last 136 being on the left side. The black lines, |, in each segment represent the measurements which are calculated for each second. Notice that "Car 1, Pass 1, Segment 3, Second 4" (C1, P1, S3, S4) and (C2, P1, S3, S3), have different distances between their one-second measurements. That is due to the cars having different speeds, thus (C2, P1, S3, S3) have travelled further in one second than (C1, P1, S3, S4).

Now we are interested in looking at a specific POI, indicated by the green dotted lines, and figuring out what the condition of the road is there. The long vertical black line is there to showcase which seconds of the segments are of interest in each pass. Looking at C1, P1 we can see that it is the fourth second of the third segment which is the closest measurement to this. The implementation is made such that the POI will look for the two closest seconds in each pass, but a threshold is put in place such that the measurements have to be closer than 10^{-4} (11.1 meters) to be used. As there are 250 measurements each second only the closest location is used - one could have averaged instead. Each POI then has a number of measurements of interest (MOIs), on average 73, which can be used to calculate the condition of the POI.

Then the weights for each MOI are calculated based on their distances $\mathbf{d}$ to the POI. A small example is shown below.

$$\mathbf{d} = [2.54, 5.93, 8.63] \times 10^{-5} \tag{1}$$

$$\mathbf{w} = -\log\left(\frac{\mathbf{d}}{\sum \mathbf{d}}\right) \tag{2}$$

$$\mathbf{w}_{norm} = \frac{\mathbf{w}}{\sum \mathbf{w}} \tag{3}$$

$$\mathbf{w}_{norm} = [0.52, 0.29, 0.19] \tag{4}$$

The MOIs are then multiplied with their corresponding weights to obtain the prediction of POIs on the actual road.
Finally as shown in Figure 9 we map the POIs with colour according to their MOIs on a map to get a visual representation of the state of the road in both directions.

Disclaimer:
It is very important to note that these results show a huge overconfidence in the model's predictive ability, since the predicted KPIs are computed based on a combination of the entire dataset (train, validation and test), to enable the entire road to be shown as a heat map. Figure 10 illustrates how the predictions seem incredibly promising, but again it must be noted that most of the data is used for training, meaning that it is not possible to distinguish between actual performance and overfit to training data.
Instead, these figures are simply meant to illustrate how one could visually use the model, and show the capabilities of POI, when data from a new road is obtained and how the mapping from the time domain (second bits) to geo-location (longitude/latitude) would work in practice.

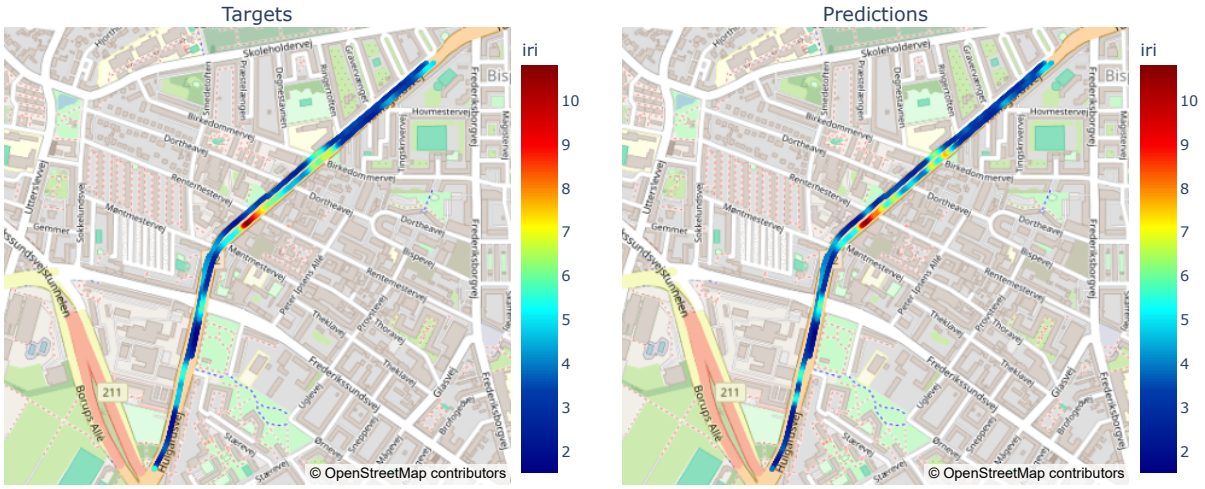


Figure 9: Mean target and prediction values for all one-second bits throughout the entire dataset (train, validation and test) plotted on top of a street view map.

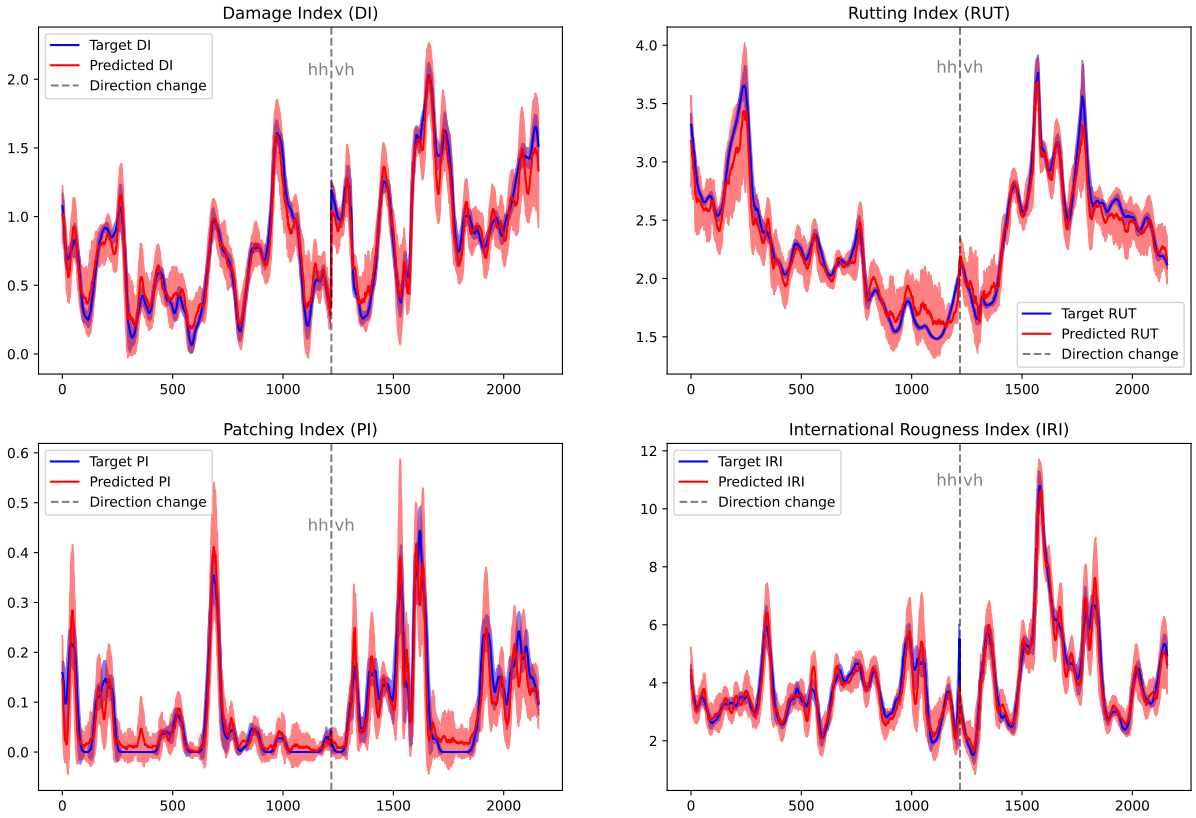


Figure 10: Target and predictions of each KPI metric plotted with ±1 standard deviation on all one-second bits in the entire dataset (train, validation and test).

7 Future work

Imagine you have a big jar filled with pearls and 50 people to help you figure out how many pearls there are in the jar. The 50 people are standing around the jar with different distances to the circle. Which strategy would give a more accurate answer:

1. Each person makes their own guess, and then you take each guess and multiply it with some normalised weight based on their distance from the jar.

2. The 50 people discusses the answer as a group, being aware of their different distances and angles to the jar, and the come up with one single answer.

In this project we have tried method 1, however we would encourage future work to investigate the performance of method 2.

Method 2 could work by predicting directly on the roads POIs using the (on average) 70 MsOI as input. There are some challenges to this like there are different amounts of MsOI for each POI. This can perhaps be solved by just reusing MsOI until all PsOI have the same number of MsOI.

7.1 Tuning feature extraction

We have only investigated extracting features from various signals, however the parameters of Hydra (8 kernels, 64 groups) and MultiRocket (50000 features) were kept to their defaults. However, these parameters can also be tuned and might provide better performance.

7.2 Fixing MultiRocket

Currently, some bugs in the MultiRocket feature extraction result in some features having standard deviations of 0. These features have been set to 0 as they provide no information for the MLP. A better solution would be to fix this bug, but we have not found one. An issue has been opened on the MultiRocket GitHub repository, awaiting an answer from the original authors. We suspect fixing this issue might improve the performance of MultiRocket and more closely align the performance of the model with the results found by the original authors of the Hydra-MultiRocket model, where the combination of Hydra and MultiRocket improved overall model performance.

7.3 Exploring different methods of feature extraction

Machine learning for time series is a rapidly evolving subject, and many papers discovering new methods are constantly being released. One place to start could be with HIVE-COTE 2.0 [9], or the more recently released ConvTran model [10].

The paper "*Deep Learning for Time Series Classification and Extrinsic Regression: A Current Survey*" by Fawaz et al., released in April 2024, [11] reviewed many models and provides a good overview of places to start. They dive into various improvements in the field in recent years, such as attention, transformers, self-supervised learning and many more.

An alternative approach could be to explore time series forecasting models and try to apply them to the extrinsic regression setting. Miller et al. created a survey of time series forecasting methods in January 2024 [12].

7.4 Using the GoPro images

This project has not made use of the GoPro data in the machine learning model, but the videos might provide additional information that could greatly increase performance through various computer-vision methods, such as convolutional neural networks or vision transformers. These features could be combined with the features from a TSER model, to provide greater context for the model.

8 Code Arguments

For reference, a table of all arguments for each script is provided. Arguments beginning with "`--`" are optional, and arguments without are required.

8.1 `main.py`

The main script that goes through the entire process from data preprocessing, feature extraction, model training, model prediction and finally validation between the newly trained model and our results.
Note that `main.py` *is meant to be used as a quick script that will reproduce our results.* ***Parameters are hardcoded****. If you wish to modify parameters, call each script on its own with the desired arguments.*

Table 1: Function Arguments for `main.py`

Argument	Type	Default	Description
`mode`	`str`	None	Specifies the mode to run the script in. Choices are: `all`, `make_data`, `extract_features`, `train_model`, `predict_model`, `validate_model`, `kpi`, . Running `all` executes all modes in sequence.
`--begin-from`	`store_true`	False	Starts the execution from the specified mode, inclusive. Useful for continuing a run from a certain step.

8.2 `make_dataset.py`

The script that preprocesses the dataset. It will go through 6 steps: Convert, validate, segment, match, resample and finally kpi. This prepares the data for feature extraction. To run it, the data is required to be in the data folder, with the file structure as outlined in the README.

Table 2: Function Arguments for `make_dataset.py`

Argument	Type	Default	Description
`mode`	`str`	None	Specifies the mode to run the script in. Choices are: `convert`, `validate`, `segment`, `match`, `resample`, `kpi`, `all`. Running `all` executes all modes in sequence.
`--begin-from`	`store_true`	False	Starts the execution from the specified mode, inclusive. Useful for continuing a run from a certain step.
`--skip-gopro`	`store_true`	False	Skips GoPro data in all steps. This is a flag argument that when set, excludes GoPro data from processing.
`--speed-threshold`	`int`	5	Sets the speed threshold for segmenting data, given in km/h.
`--time-threshold`	`int`	10	Sets the time threshold for segmenting data, given in seconds.
`--validation-threshold`	`float`	0.2	Specifies the normalized Mean Squared Error (MSE) threshold for validating data.
`--verbose`	`store_true`	False	Enables verbose output. This flag increases the verbosity of the script output for debugging and informational purposes.

8.3 `feature_extraction.py`

The script that extracts features using the desired model. Currently implemented feature extractors are HYDRA and MultiRocket for both univariate and multivariate time series.

Table 3: Function Arguments for `feature_extraction.py`

Argument	Type	Default	Description
`--cols`	`list[str]`	`['acc.xyz_0', 'acc.xyz_1', 'acc.xyz_2']`	Specifies the signals to extract features from. To manually specify signals, call like so: `--cols acc.xyz_2 spd_veh`. A list of all available signals can be found in Table 1 in [1].
`--all_cols`	`store_true`	False	Use all columns relevant for road pavement condition prediction. This includes GPS signals, accelerations, speed, odometer, steering, rpm, wheel pressure and torque, traction and brake torque.
`--all_cols_wo_location`	`store_true`	False	Use all columns relevant for road pavement condition prediction, except GPS and odometer, as they might provide skewed information about the road, that would not regularly be available (shortcut learning).
`--feature_extractor`	`str`	`both`	Which feature extractors to use for feature extractions. Choices are `both`, `multirocket`, `hydra`
`--mr_num_features`	`int`	50000	How many features should multirocket extract. Using the default will result in 49728 features.

Continued on next page

Table 3 – continued from previous page

Argument	Type	Default	Description
--hydra_k	int	8	The number of kernels in each group in HYDRA. When setting kernels to 1, HYDRA works more like ROCKET.
--hydra_g	int	64	The number of groups in HYDRA.
--name_identifier	str	''	An optional name identifier to add to the end of the stored features. Useful if you wish to extract the same features, but save them both.
--folds	int	5	How many folds to extract features for, for cross-validation purposes. Increasing this will increase runtime.
--seed	int	42	Seed for reproducibility.

8.4 train_hydra_mr.py

The script trains the HydraMultirocket regression model (MLP). Automatically detects the input size and output size based on provided data.

Table 4: Function Arguments for train_hydra_mr.py

Argument	Type	Default	Description
--epochs	int	50	How many epochs to train (each fold) for.
--batch_size	int	32	Batch size of dataloader.
--lr	float	1e-3	Learning rate of the optimizer (Adam).
--feature_extractors	list[str]	['HydraMV_8_64']	The feature extractors to use for training (loads the data from features.hdf5 based on this name).
--name_identifier	str	''	Name identifier to use for loading the features.
--folds	int	5	How many folds to use for cross-validation.

Continued on next page

Table 4 – continued from previous page

Argument	Type	Default	Description
--model_name	str	HydraMRRegressor	Model name to store the results under.
--weight_decay	float	0.0	Weight decay for the optimizer.
--hidden_dim	int	64	Amount of neurons in the hidden dimension(s).
--model_depth	int	0	Depth of the model (number of hidden layers).
--dropout	float	0.5	Dropout between each layer.
--batch_norm	store_true	False	Whether or not to use batch normalization.
--project_name	str	hydra_mr_test	The project the run should be stored under on WandB. Set wandb disabled or enter your project name here.

8.5 predict_model.py

The script uses a trained model to predict the test set.

Table 5: Function Arguments for predict_model.py

Argument	Type	Default	Description
--model	str	models/ HydraMRRegressor/ HydraMRRegressor.pt	Specifies the path to the model to load and use for prediction.
--data	str	data/processed/ features.hdf5	Path to the data file to load the features from.
--name_identifier	str	''	Optional name identifier to add at the end of the feature extractor (for loading the features)
--data_type	str	test	What type of data to load. Choices are train, val, test.
--batch_size	int	32	Batch size to use for prediction.
--plot_during	store_true	False	Plot predictions during prediction.

Continued on next page

Table 5 – continued from previous page

Argument	Type	Default	Description
--fold	int	-1	The fold to use for prediction (only works for train and validation). If set to -1, the fold will automatically be loaded from the information stored in the model.
--save_predictions	store_true	False	Save predictions to a file (used for validation).

8.6 check_hdf5.py

The script prints the structure of the specified hdf5 file. Can also print a summary of features.

Table 6: Function Arguments for check_hdf5.py

Argument	Type	Default	Description
--file_path	str	data/processed/features.hdf5	Specifies the path to the hdf5 to check.
--limit	int	3	The maximum number of items to print at each level.
--summary	store_true	False	Print a summary of the hdf5 file. NOTE: Only works for features.hdf5 files (created with the feature_extractor.py script.
--delete_model	list[str]	[]	List of models features to delete from the file features.hdf5 file. Example usage: -delete_model HydraMV_8_64.

References

[1] Asmus Skar and Tommy S. et al. Alstrøm. *Live Road Assessment based on modern car sensors (LiRA): Practical guide*. LiRA project, Denmark, January 2023. Published in connection with LiRA end-seminar January 16th 2023.

[2] Asmus Skar, Anders M. Vestergaard, Thea Brüsch, Shahrzad Pour, Ekkart Kindler, Tommy Sonne Alstrøm, Uwe Schlotz, Jakob Elsborg Larsen, and Matteo Pettinari. Lira-cd: An open-source dataset for road condition modelling and research. *Data in Brief*, 49:109426, 2023.

[3] Chang Wei Tan, Angus Dempster, Christoph Bergmeir, and Geoffrey I Webb. MultiRocket: Multiple pooling operators and transformations for fast and effective time series classification. *arxiv:2102.00457*, 2021.

[4] Angus Dempster, Daniel F Schmidt, and Geoffrey I Webb. HYDRA: Competing convolutional kernels for fast and accurate time series classification. *arXiv:2203.13652*, 2022.

[5] Matthew Middlehurst, Patrick Schäfer, and Anthony Bagnall. Bake off redux: a review and experimental evaluation of recent time series classification algorithms. *Data Mining and Knowledge Discovery*, April 2024.

[6] Chang Wei Tan, Christoph Bergmeir, François Petitjean, and Geoffrey I. Webb. Time series regression. *CoRR*, abs/2006.12672, 2020.

[7] Angus Dempster, François Petitjean, and Geoffrey I Webb. ROCKET: Exceptionally fast and accurate time series classification using random convolutional kernels. *Data Mining and Knowledge Discovery*, 34(5):1454–1495, 2020.

[8] Angus Dempster, Daniel F Schmidt, and Geoffrey I Webb. MiniRocket: A very fast (almost) deterministic transform for time series classification. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 248–257, New York, 2021. ACM.

[9] Matthew Middlehurst, James Large, Michael Flynn, Jason Lines, Aaron Bostrom, and Anthony Bagnall. Hive-cote 2.0: a new meta ensemble for time series classification, 2021.

[10] Navid Mohammadi Foumani, Chang Wei Tan, Geoffrey I Webb, and Mahsa Salehi. Improving position encoding of transformers for multivariate time series classification. *Data Mining and Knowledge Discovery*, 38(1):22–48, 2024.

[11] Navid Mohammadi Foumani, Lynn Miller, Chang Wei Tan, Geoffrey I. Webb, Germain Forestier, and Mahsa Salehi. Deep learning for time series classification and extrinsic regression: A current survey. *ACM Comput. Surv.*, 56(9), apr 2024.

[12] John A. Miller, Mohammed Aldosari, Farah Saeed, Nasid Habib Barna, Subas Rana, I. Budak Arpinar, and Ninghao Liu. A survey of deep learning and foundation models for time series forecasting, 2024.