Fachhochschule Kärnten
Carinthia University of Applied Sciences
Applied Data Science

MASTER THESIS

Smart Meter Data Analytics to Enhance Grid Planning for KNG

Submitted in partial fulfillment of the requirements of the academic degree
Master of Science in Engineering, MSc

Author: DI Martina Schelander Bakk. techn.
Registration Number: 2210876010
Supervisor: Em. O.Univ.-Prof. Dr. Jürgen Pilz
Second supervisor: DI Dr. Daniel Posch, Kärnten Netz GmbH

Villach, August 2024

AFFIDAVIT (DECLARATION OF ORIGINALITY)

I hereby declare that

- I have independently written the presented Master thesis by myself;
- I have prepared this Master thesis without outside help and without using any sources or aids other than those cited by me; moreover, I have identified as such any passages taken verbatim or in terms of content from the sources used;
- in addition, I have fully indicated the use of generative AI models (e.g. ChatGPT) by specifying the product name and the reference source (e.g. URL);
- I have not used any other unauthorized aids and have consistently worked independently and when using generative AI models, I realize that I am responsible how the content will be used and to what extent
- I have not yet submitted this Master thesis in the same or similar form to any (other) educational institution as an examination performance or (scientific) thesis.
- I am aware that any violation ("use of unauthorized aids") violates academic integrity and may result in (academic-related) legal consequences.

Villach, 22.08.2024

Abstract

This thesis focuses on two main areas within the field of smart meter data analytics for managing low-voltage energy networks. First, it explores the possibility of accurately forecasting energy consumption using 15-minute interval data. Second, it delves into the identification of electric vehicle charging stations and heat pumps based on individual consumption profiles. The accurate energy consumption forecasts at a low-voltage network level are based on data obtained from KNG's Smart Meter infrastructure. This infrastructure comprises over 288,000 Smart Meters, with 55,000 meters set up for high-resolution 15-minute consumption records. Various forecasting models, including traditional statistical methods and advanced machine learning techniques, were employed. The results demonstrated that accurate forecasts are achievable, particularly when clustering low-voltage networks with similar consumption patterns. Supervised learning methods were applied for the second research question to classify consumption profiles using label data derived from external sources, such as customer contracts from KNG's ERP system. For electric vehicle charging stations, the identification process involved extracting features such as peak values, resulting in high accuracy for the classification task. Identifying heat pumps presented more challenges due to the different types of heat pumps and a lack of detailed label information about them. Despite these challenges, the classification achieved satisfactory results, though further improvements are possible. In conclusion, this thesis demonstrates that it is feasible to create accurate energy consumption forecasts at the low-voltage network level using 15-minute interval data and to identify specific consumer profiles for EV charging stations and heat pumps. Future work will focus on refining these models and exploring additional data sources to improve accuracy and applicability.

Keywords: Energy Consumption Forecast, Load Profile Disaggregation

Acknowledgements

I wish to express my sincere gratitude to my first supervisor, Prof. Dr. Jürgen Pilz, for his invaluable guidance and support throughout the completion of my Master's thesis. His expertise, insightful feedback, and commitment to aligning this work with the latest research have been crucial in shaping its direction and quality.

I am also deeply grateful to DI Dr. Daniel Posch from Kärnten Netz GmbH for his practical insights and assistance, which greatly enriched the applied aspects of my work.

I would like to extend my thanks to Kärnten Netz GmbH for the opportunity to collaborate, which allowed me to apply my academic knowledge in a real-world context. This partnership has been instrumental in the success of this research, and I deeply appreciate their support.

Lastly, my heartfelt appreciation goes to my family and friends for their unwavering support during this important endeavor.

Thank you all.

Table of Contents

List of Figures

List of Tables

1. Introduction

As Carinthia's primary power grid operator, KNG-KärntenNetz GmbH (KNG) currently manages over 288.000 Smart Meters as part of its Smart Meter infrastructure. The company aims to employ available Smart Meter data to promote and enhance the efficiency and sustainability of the Carinthian power grid and add value besides fulfilling its legal order. The availability of Smart Meter-based data offers new possibilities for grid planning, management, and operation. The Smart Meter infrastructure can significantly enhance various operational functionalities of the distribution network. These functionalities include generation scheduling, real-time pricing, load management, power quality enhancement, security analysis, fault prediction, frequency and voltage monitoring, and precise load predictions, which are crucial for supporting the energy market and strategic network planning and operation. Insights from this infrastructure can help in balancing demand and generation, preventing system imbalances and power outages and ensuring a more reliable and efficient grid operation. From the distribution network operator's view, it allows for the identification of capacity bottlenecks and managing available flexibilities in the network infrastructure. Therefore, there is an urgent desire to employ smart meter data to improve network operations of various kinds. The planning and operation of the grid can be improved through a precise demand forecast and identifying specific consumer profiles using the consumption time series. In this thesis, we employ Smart Meter data to address these two scenarios.

1.1 Motivation and Problem Statement

While numerous methodologies exist in the literature, KNG must consider the legal constraints that restrict the collection and utilization of power consumption data at daily and 15-minute intervals due to the legal framework in [1]. The operator's grid planning for private households is currently based on standard load profiles with no respect to the significant load profiles of heat pumps and charging stations for electric cars. A better understanding of this would allow KNG to operate its grid closer to the limits, potentially saving money and delaying grid expansion, thereby demonstrating the financial benefits of this proposal.

Loads can be forecast by considering the power consumption by customers and the power produced by all types of generations, including renewable and non-renewable, with the help of Smart Meters. Rapid penetration of renewable energy sources with high variability and uncertainty imposes new challenges to the operation of power systems, especially within low-voltage networks. Due to random fluctuations in weather conditions, the distribution network infrastructure may suffer from volatile generation. State-of-the-art distribution networks require Smart Meters to maintain a precise energy consumption forecast. Utilizing smart meter data can significantly elucidate these dynamics and help in planning and operating the network.

1.2 Project Scope

1.2.1 Consumer Profile Identification

Non-intrusive load monitoring (NILM) and load disaggregation are well-known techniques used for grid analysis. They can be used for customer segmentation, resource identification, rate recommendation tasks, and occupancy-based demand response systems. They can nudge consumers to defer their consumption to off-peak hours. This project's first objective is to analyze power consumption data to detect and identify heat pumps and electric vehicle charging stations. Customers are asked to disclose the specifications of the heat pump installed at the time of the contract request. However, there is no obligation, and only some customers comply with this request. Additionally, if changes are made to the installed systems, no update is sent, and neither does the information about the installed systems guarantee that those systems are used at full capacity and in the expected manner throughout the year. Nevertheless, because those consumers impose a heavy load on the network infrastructure, the automated identification of such devices immensely helps network planning and operation. The advances in electromobility and the ever-increasing number of electric vehicles and electric vehicle charging stations in Carinthia's infrastructure cause problems in ensuring power supply and correctly sizing the network infrastructure. Since January 2024, a legal obligation has been in place to register electric vehicle charging stations of any size with the distribution network operator [2]. We use Smart Meter data to identify typical vehicle charging loading profiles and any unregistered charging

stations. A key consideration is whether the labeled data and 15-minute interval information can adequately be used to develop an effective model to identify consumption patterns.

1.2.2 Energy Consumption Forecast

The second project objective is developing of a 48-hour power demand forecast for individual low-voltage networks within the power grid based on available Smart Meter consumption time series. Network operators prioritize identifying short-term demand peaks and predicting them in advance. KNG is strongly interested in predicting peak demands and free network capacities two days in advance. This time frame is generally considered a short-term forecasting range [3]. Short-term load forecasting is crucial in planning the load flow, avoiding overloading, and planning shared or higher-level load transport [4]. In the Carinthian network, the Austrian Power Grid (APG) is responsible for high-voltage load transport and demands accurate forecasting results from KNG. The prototype implementation shall anticipate demand and assist network system operators with strategic planning.

1.2.3 Project Limitations

The energy consumption forecast and the identification of consumer profiles heavily depend on high-resolution data. However, daily granularity is insufficient to make assumptions in either of those use cases. Therefore, this project is currently limited to Smart Meters operated in a high-resolution mode using 15-minute intervals to track the customer's consumption. This restricts the applicability of our approach to approximately one-quarter of all meters in KNG's network. Possible legislative changes in the future can change this [5]. However, we are currently bound to those restrictions. The restriction to 15-minute interval data also implies another restriction when focusing on low voltage networks, the smallest controllable network unit that contains all meters after a low voltage transformer station. Currently, only networks with enough 15-minute interval meters can be included in the forecasting process. We require at least three of those meters in a network to consider them in our process due to data privacy and volatility reasons. A single-meter forecast would not be reliable enough as a base for network planning and operation activities. This restriction leaves us with 4426 low voltage networks out of 7753 for our energy forecast scenario.

Due to this restriction on 15-minute interval data, we deemed it irrational to include the generated energy fed back into the network by consumers and producers in our forecasting amount. Including this data would restrict our dataset even more by requiring that a customer's feeder and consumption account are both operated in the 15-minute configuration, which is not always the case. So, we decided not to include this additional information. A legal change requiring all meters to become high-resolution data providers of 15-minute interval data will change the whole data source and must be considered as soon as this change occurs.

In this work, we determine the appropriate method for forecasting and profile identification based purely on the model's performance and not on the training or inference time of the individual approaches. We explain punctual observations regarding training times in the later stages of the forecasting, as those seem to be rigorous between the two leading approaches. However, this work does not intend to provide a detailed analysis of the calculation costs of the algorithms. One reason for this is that not all models were trained on the same hardware. Whenever the architecture allowed, we stayed on CPUs for model training and only switched to GPUs when required by the architecture itself.

2. Data Sources and Data Preparation

This chapter examines the available data sources, the exploratory data analysis, and the required preprocessing steps for the project's tasks ahead. First, KNG's Smart Meter infrastructure will be described to give the reader insight into the network elements required and the network topology to gather up-to-date energy consumption time series. Then, we closely examine the time series data by explaining the data structure and the different meter configurations in the network. We discuss the availability of weather information from the past and the possibilities of using future weather forecasts. Notably, label information is required for the time series classification tasks. We explain how this label information is gathered for the heating pumps and the electronic vehicle charging stations.

2.1 KNG Smart Meter Network Infrastructure

KNG's Smart Meter Infrastructure has been rolled out since 2017 and includes over 288.000 smart meters as of July 2024.

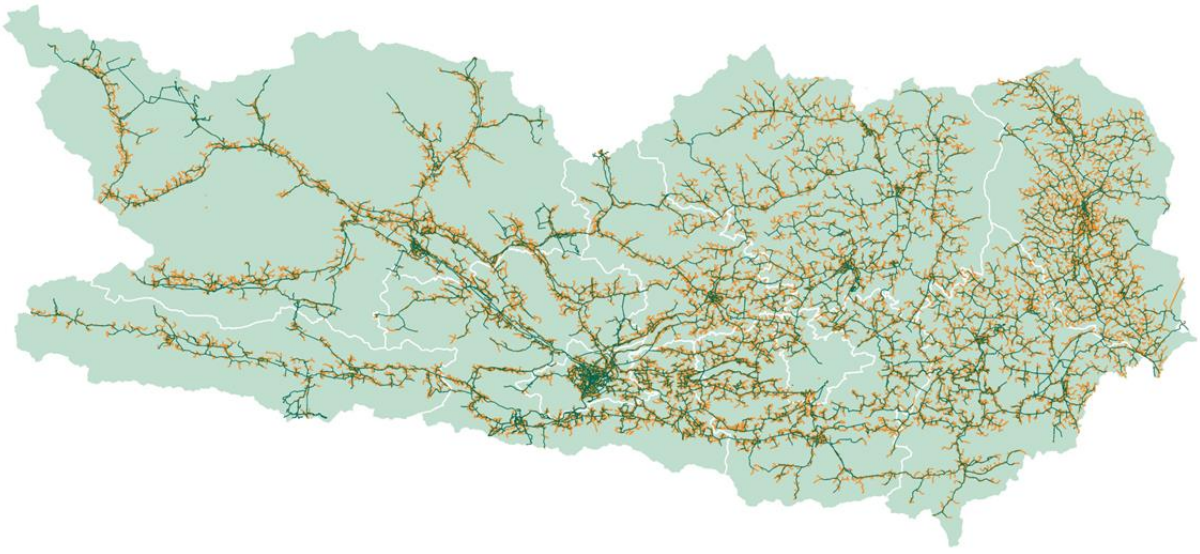

Figure 1 KNG's Middle- (green) and Low-voltage (yellow) network infrastructure

Figure 1 illustrates KNG's middle- to low-voltage network infrastructure. The green lines depict middle-voltage feeder lines, while the yellow ones illustrate low-voltage feeder line infrastructure. The Smart Meter data in this thesis is gathered on devices attached to the low-voltage network.

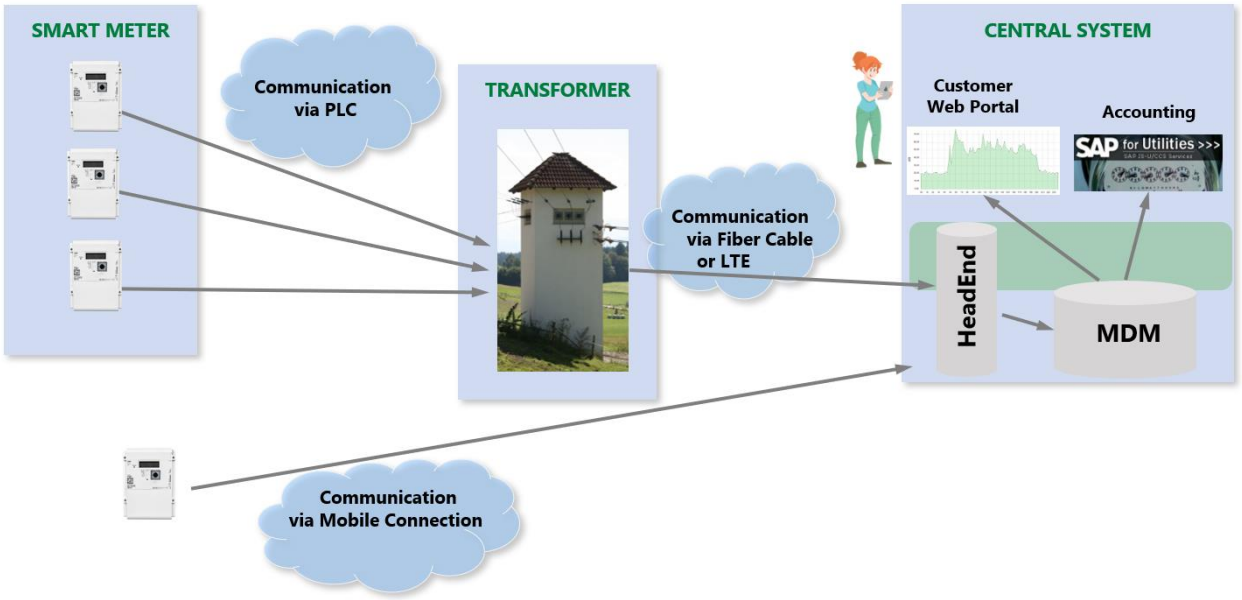

Figure 2 High-level Smart Meter network infrastructure

Figure 2 illustrates the high-level architecture of the Smart Meter infrastructure. The Meters transmit the consumption amounts once a day, using the Power Line Carrier Communication (PLC) protocol via a gateway located at the transformer building or using a mobile connection to the backend system. These transactions are triggered by the head-end system, which forwards them to the Meter Data Management (MDM) System once the data is received. The MDM System hosts an Oracle-based RDBMS responsible for data persistence of the consumption and various meter and system configurations. This database is the source for all consumption time series mentioned in the following chapters. Two distinct operational modes exist for all communicating Smart Meters. The first, Intelligent Metering Device Standard (IMS), reports the respective values for daily intervals, and the second one, Intelligent Metering Device Extended (IME), reports 15-minute consumption intervals producing 96 values per day. Roughly 55.000 of the 288.000 Smart Meters are operated in the 15-minute interval mode, creating over 5.2 million new consumption

records a day. The remaining 233.000 Smart Meters operate in the daily aggregation mode and produce one record per day, resulting in 233.000 new records per day. As of July 2024, the configuration modes are exclusively the customers' responsibility and can only be changed if explicitly demanded. There is currently no legal obligation to use a particular mode. The current meter configuration determines if a Smart Meter can be used for the planned use cases of nonintrusive load monitoring and energy consumption forecast. We decided not to include daily profiles as they hold too little information to create useful models.

Figure 3 Low-voltage network topology, including Smart Meters in the network

KNG operates over 7600 low-voltage networks in Carinthia. An example of such a local low-voltage network, including 37 Smart Meters, is illustrated in Figure 3. Every network has a unique consumption pattern. While we consider individual meters for the non-intrusive load-monitoring use case, we only use the summed-up consumption for a complete low-voltage network to create a consumption forecast, as the lowest forecast level of interest to KNG is the low-voltage level transformer. Notably, a consumption forecast for single meters or groups of individual meters in a network is not the objective of this work.

2.2 Power Consumption Time Series

The primary data source for this project is the 15-minute interval-based power consumption time series extracted from the meter data management system.

METER_BADGE_ID	CHANNEL_ID	INTERVAL_END_TIME	LP_VALUE_KWH	INTERVAL_LEN_SEC	DATA_SCOURCE_ID
735545633	29929466	2023.05.15 08:30:00	5,44	900	51
735545633	29929466	2023.05.15 08:45:00	11,92	900	51
735545633	29929466	2023.05.15 09:00:00	8,48	900	51
735545633	29929466	2023.05.15 09:15:00	10,32	900	51
735545633	29929466	2023.05.15 09:30:00	7,92	900	51
735545633	29929466	2023.05.15 09:45:00	6,64	900	51
735545633	29929466	2023.05.15 10:00:00	9,6	900	51
735545633	29929466	2023.05.15 10:15:00	9,84	900	51
735545633	29929466	2023.05.15 10:30:00	10,72	900	51
735545633	29929466	2023.05.15 10:45:00	13,92	900	51
735545633	29929466	2023.05.15 11:00:00	11,36	900	51
735545633	29929466	2023.05.15 11:15:00	13,04	900	51
735545633	29929466	2023.05.15 11:30:00	11,68	900	51
735545633	29929466	2023.05.15 11:45:00	11,44	900	51
735545633	29929466	2023.05.15 12:00:00	11,44	900	51
735545633	29929466	2023.05.15 12:15:00	13,76	900	51
735545633	29929466	2023.05.15 12:30:00	14,72	900	51
735545633	29929466	2023.05.15 12:45:00	12,4	900	51
735545633	29929466	2023.05.15 13:00:00	9,44	900	51
735545633	29929466	2023.05.15 13:15:00	13,44	900	51
735545633	29929466	2023.05.15 13:30:00	12,56	900	51
735545633	29929466	2023.05.15 13:45:00	13,12	900	51
735545633	29929466	2023.05.15 14:00:00	13,12	900	51
735545633	29929466	2023.05.15 14:15:00	10,64	900	51
735545633	29929466	2023.05.15 14:30:00	10,32	900	51
735545633	29929466	2023.05.15 14:45:00	11,2	900	51
735545633	29929466	2023.05.15 15:00:00	14,56	900	51

Figure 4 Smart Meter data for the 15-minute interval configuration from MDMS

An example profile for one specific meter is illustrated in Figure 4. Available variables are the meter identification (METER_BADGE_ID), the consumption channel configured on the smart meter (CHANNEL_ID), a timestamp indicating the end of the respective 15-minute interval (INTERVAL_END_TIME), the consumption value for the given interval given in kWh (LP_VALUE_KWH), the fixed interval length of 900 seconds (INTERVAL_LEN_SEC) and the source of the respective entry (DATA_SOURCE_ID) which could be either the original meter reading from the device or an estimated value for missing entries. This data is used as the basis for accounting and has already undergone a rigorous validation process. Therefore, the missing values and outlier detection tests are subordinate to the data analysis. The check for missing values in the

time series was negative. Outliers, or extreme values in the consumption profile, are actual valid values and are not to be considered as actual outliers produced by faulty processes or measurements.

Table 2.1 Descriptive statistics for the consumption (kWh) of a single low-voltage network

Network 2/540	
Time Stamp Count	35039
Mean	2.568482
Standard Deviation	1.321367
Minimum	0.0
25th Percentile	1.647
Median (50th Percentile)	2.211
75th Percentile	3.108251
Maximum	11.651738
99th Percentile	7.1

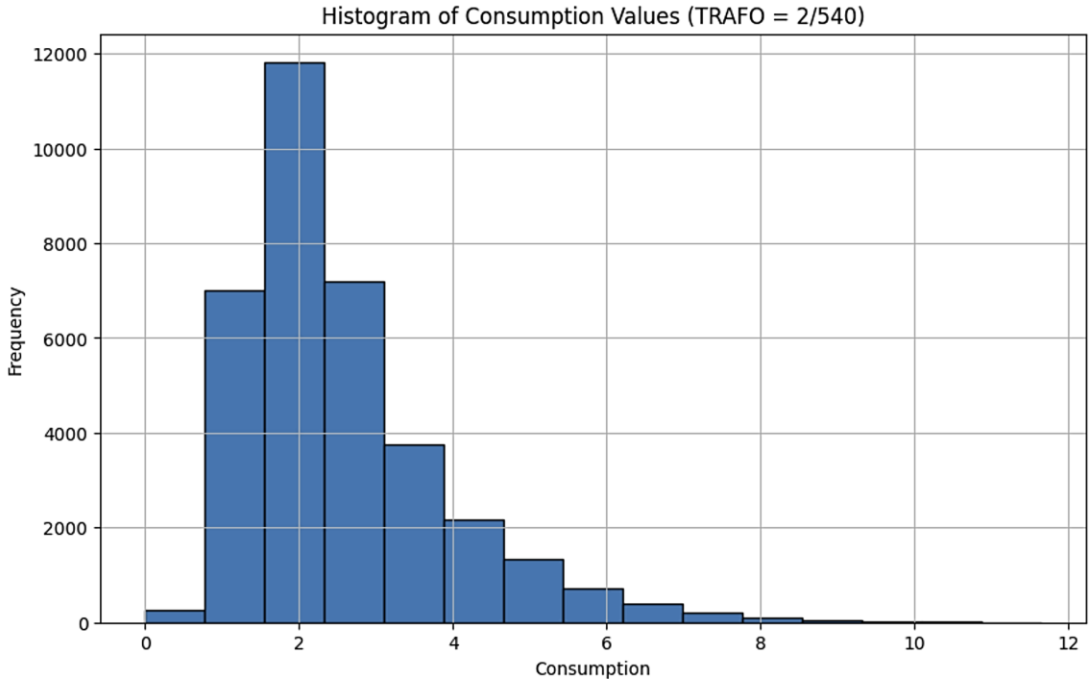

Figure 5 Histogram for the consumption values (kWh) for 2023 for low-voltage network 2/540

We conducted an exploratory data analysis and verification before any further preprocessing steps. See Table 2.1 and Figure 5 for the descriptive statistics and the 15-bin histogram for the yearly consumption values for the typical low-voltage network 2/540 for 2023. As expected, a slightly left-skewed distribution with a mean of 2.56 kWh is noticeable. The 75th percentile of 3.1 indicates that 75% of the consumption values are below that threshold. We added an uncommon but useful value to our statistics. The 99th percentile of 7.1 indicates that extreme values are rare, and only 1% of consumption values exceed that threshold. To understand when and why those extreme values may have occurred, we plotted the time series for the whole day where the maximum value occurred, namely 29th December (see Figure 6). The maximum occurred during the winter holidays, immediately after Christmas, when many people are very likely to be home with their families, and various electrical appliances and heating may be at their peak.

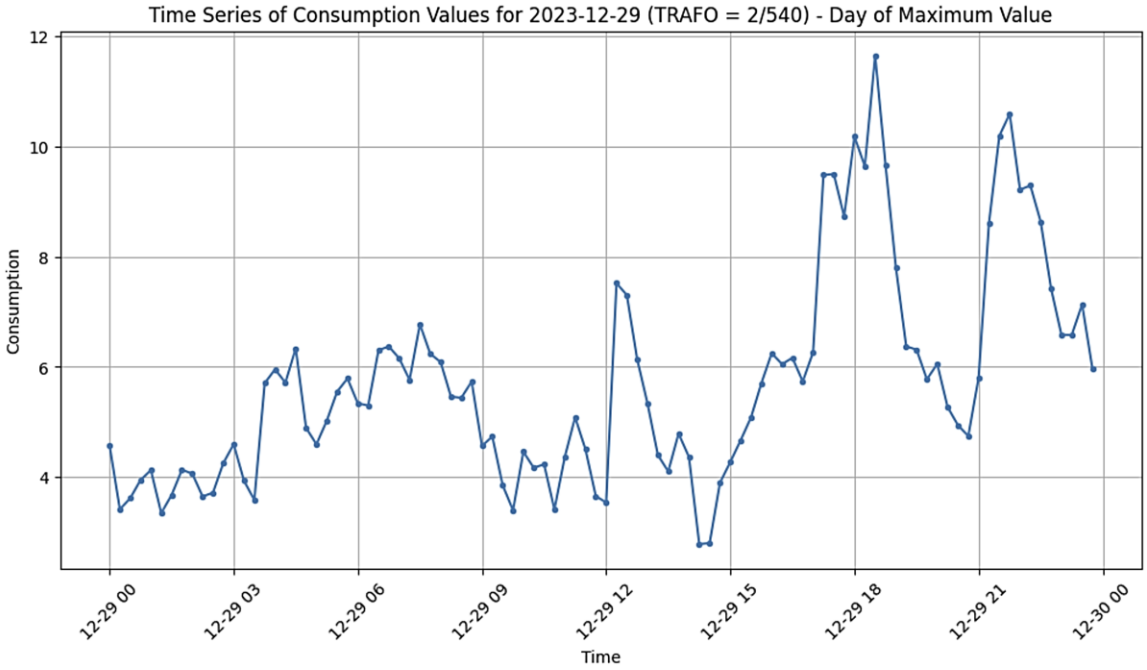

Figure 6 Time series consumption (given in kWh) plot for December 29, 2023, including the yearly maximum value

The maximum value of 11.65 kWh occurred at 18:30. It is not a single occurrence but embedded in a cluster of high values, indicating that it is a valid measurement and not an outlier.

2.3 Weather Data

KNG supports a personalized access account at GeoSphere Austria for current weather information and its network of weather stations to create a detailed weather forecast. A total of 18 weather stations spread across Carinthia are used for this project. Data is available in 15-minute intervals and can be derived for past periods for model training purposes. We first use the past data as additional regressors to assess their effect on the model types that easily support those additional regressors. If their application proves useful, they are added to all models as supporting regressors. See Figure 7 for the descriptive statistics of the weather data available at the weather station located in Villach.

	Villach_TL_Lufttemp_MWm	Villach_WG_Windgeschw	Villach_P_Luftdruck_MWm	Villach_GI_Globalstrahlung	Villach_RF_RelFeuchte
count	2976.000000	2976.000000	2976.000000	2976.000000	2976.000000
mean	11.240793	1.234509	954.837772	108.234459	80.834005
std	6.568167	0.864258	9.171546	190.223316	16.120337
min	-10.000000	0.000000	933.000000	0.000000	29.000000
25%	7.400000	0.600000	947.500000	0.000000	72.000000
50%	11.800000	1.000000	955.600037	0.000000	86.000000
75%	14.412501	1.600000	962.100037	140.000000	94.000000
max	32.799999	6.800000	979.700012	1032.000000	99.000000

Figure 7 Descriptive statistics for the available weather information for one weather station

The following parameters are available: air temperature, air pressure, wind speed, humidity, and global radiation.

3. Energy Consumption Forecasting: Methods and Applications

This chapter explores the current approaches in Smart Meter data-based consumption forecasting. Given that network clustering is a crucial preprocessing step in our scenario, enabling us to manage the number of individual low-voltage networks effectively, we also present our proposed solution for this clustering process. Finally, we discuss the use case scenarios we have defined to guide the preselection of models for our final application. This preselection is essential, as it is impractical to perform the final forecasting tasks across all clusters using every potentially suitable model. By narrowing down the model choices, we can significantly reduce the training and evaluation efforts, making the process more feasible.

3.1 Current Approaches in Smart Meter-based Energy Consumption Forecast

A wide range of methods are used in energy consumption forecasting, which spans classical statistical techniques such as ARIMA and Linear Regression to machine learning-based approaches, including neural networks, Recurrent Neural Networks (RNN), and methods based on Long Short-Term Memory (LSTM). The data aggregation levels in the literature vary from the very high resolution of 5 minutes [6] to intermediate intervals like 30 minutes [7] to even daily values. A simple approach is based on regression methods. The approach in [8] involves the use of Linear Regression for short-term forecasting horizons and a Random Forest approach for long-term forecasting. In [9], Multiple Linear Regression is used for multi-modal short-term forecasts. A semi-parametric additive model is used in [10] for a short-to-middle-term forecast. Other forecasting approaches include Discrete Decision Trees and Random Forests. Various feature selection methods were used in [11] before a Gradient Boosted Regression Tree was applied to them. A similar approach based on Gradient Boosting Decision Trees was used in [12]. In [13], a Random Forest algorithm was used on subgroups of consumers. Classical statistical models based on ARIMA were applied to energy consumption forecasts in [14] and [15]. The latter also

compares ARIMA, Support Vector Machines, and Artificial Neural Networks (ANNs). A very early approach to Smart meter-based energy consumption forecast from 2011 used Kalman Filters [16]. Their characteristic of taking new measurements into account and adjusting the outcome according to the new information proves to be a big advantage when faced with real-world forecasting tasks. Notably, [17] and [18] use Bayesian Learning for load forecasting tasks. The Bayesian method does not require stationarity and can handle high uncertainties. The approach by Roth et al. [17] also includes weather effects, holidays, and other dynamic factors. Some older approaches utilized Support Vector Regression for energy consumption forecasts. Support Vector Machines and Support Vector Regression require only a small amount of data to produce good results and have already been successfully applied in load forecasting [8]. Gajowniczek et al. [19] proposed a short-term forecast using Support Vector Machines and Multi-Layer Feedforward Neural Networks. In [20], Support Vector Regression was used to forecast the peak loads of Smart Meters, considering the previous day's average temperature and holiday status. In [21], Support Vector Regression and a multilayer perceptron were used for a consumption load forecast based on 30-minute consumption data. An online Support Vector Regression method was used in [22] for short-term forecasting. Consumption curves can be highly nonlinear and difficult to approximate, especially in the low-voltage and medium-voltage ranges [23]. ANNs are a promising approach for accurate consumption forecasting because they are suitable for nonlinear behavior and the extraction of complex features to model the relationship between the input and output. The application of ANNs in energy load forecasting was first proposed in 2013 by Asare-Bediako et al. [24], where consumption and weather data were used to create forecasts for hourly, minute, weekend, and holiday consumption patterns. Another ANN-based approach incorporating weather regressors is presented in [25]. An ANN-based approach applied to consumption clusters is presented in [26]. The usefulness of ensemble forecasting methods was demonstrated in [27]. ANNs, with more than five hidden layers are considered as deep learning methods. A few deep learning network architectures have been applied in consumption forecasting [28],[29],[30]. The application of RNN and LSTM-based deep learning approaches seems promising. They deliver good results in scenarios like this because of their ability to specifically learn the short-term and long-term dependencies in sequential data, such as time

series [31]. The approach in [32] implies that even low sampling rates can yield adequate forecast results. As the power grid Smart Meter infrastructure constantly changes in terms of roll-out status and configuration modes, integrating online learning capabilities into the system may be reasonable [33]. A sequence-to-sequence RNN (S2S RNN) method for load forecasting was presented in [34]. The sequence-to-sequence component is used to model the time dependencies, and two RNNs are used as the decoder and encoder components. In [33], an online adaptive RNN is proposed to allow for a continuous learning process that integrates new information over time. An LSTM-based approach that operates on clusters of similar daily load curves is presented by Jiao et al. [35]. Other architectures, including LSTMs for load forecasting, are presented in [36],[37],[38]. An interesting combination of genetic algorithms with LSTMs is presented in [36]. The forecasting of energy consumption involves the use of different methods, with no single algorithm being universally recognized as the best. The effectiveness of a specific method often depends on the particular use case, which is why we intend to experiment with a variety of traditional and modern techniques, comparing them to each other. Many of the approaches mentioned in the literature are from the period between 2014 and 2020, coinciding with the widespread adoption of electrical Smart Meters. This period marked the start of forecasting based on Smart Meter data. However, since then, numerous advancements have been made, especially in deep learning, and the introduction of new methodologies such as Transformers and Large Language Models (LLMs) for time series forecasting. Acknowledging these developments, we will also explore these state-of-the-art techniques to evaluate their potential in improving forecast accuracy.

3.2 Low-Voltage Network Clustering

As mentioned in Chapter 2.1, KNG operates over 7600 low-voltage networks in Carinthia. Creating a dedicated model per cluster is difficult because training and maintaining over 7700 models in our environment is not practically manageable. One of the major challenges in this project is the clustering of low-voltage networks according to their consumption patterns to reduce the number of forecasting models to a manageable number while still preserving the system's overall performance. Therefore, we used time series clustering to find homogeneous consumption

patterns in the individual low-voltage networks. In the first preprocessing step, we needed to decide which networks to include in the clustering and forecasting process. As mentioned in the project limitations, we were limited to networks with available 15-minute interval meters. Only approximately a quarter of KNG's meter infrastructure operates in that mode. Networks with only one or two meters in IME mode were discarded for General Data Protection Regulation reasons. Generally, the consumption of a single customer cannot be traced back, so we decided to exclude those networks. Considering all these restrictions, the remaining number of low-voltage networks to be clustered was 4426. Next, we needed to decide on an appropriate clustering method. In the first step, we had to pick an appropriate distance measure. Two common available options are classic measures, such as Euclidean distance, on the one hand, and elastic distance measures, such as Dynamic Time Warping [39], on the other, which use a form of realignment of the series before comparing them. The second step is choosing an appropriate clustering algorithm. The most common approach is K-means [40]. Another viable option is the K-medoids method, which is more robust in the presence of outliers. According to Holder et al. [40], the K-medoids performed better than K-means, but only for distances other than Euclidean. They also found that elastic distance measures, like Dynamic Time Warping, did not improve the outcome compared to Euclidean distance while being computationally more expensive. Considering these research outcomes, we decided to use K-means as a clustering algorithm for a simple Euclidean distance. We extracted the average 15-minute consumption data of all meters per low-voltage network for 2023 to consider seasonal variations, such as summer and winter month-specifics. The resulting 4426 yearly consumption time series were scaled using a *MinMax*-Scaler, and we applied the *tslearn.clustering* package *TimeSeriesKMeans* implementation [41].

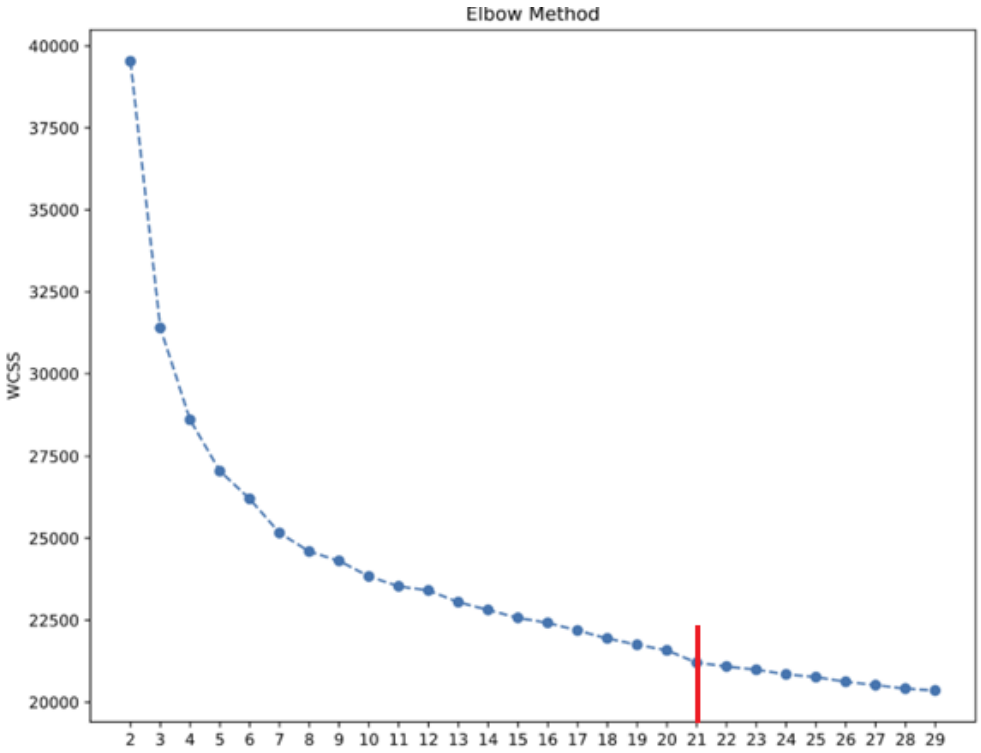

Figure 8 Elbow plot for the K-Means, Euclidean distance-based clustering to find a suitable number of clusters

To find a suitable number of clusters, we used an elbow plot for the within-cluster sum-of-squares (WCSS) (see Figure 8). Using the insights from the elbow plot, we created 21 clusters. See Table 3.1 for the number of networks per cluster. The decision for 21 clusters seemed to be a good trade-off between within-cluster similarity and computational effort. At 21, the elbow plot indicated a sudden decrease in the WCSS error value, making it a good candidate for the cut. Additionally, the number of networks per cluster spreads suitably for 21 clusters.

Table 3.1 Assigned low-voltage networks per cluster.

Cluster	0	1	2	3	4	5	6
# Low-Voltage Networks	76	535	135	258	42	824	14
Cluster	7	8	9	10	11	12	13
# Low-Voltage Networks	803	178	471	40	107	334	35

Cluster	14	15	16	17	18	19	20
# Low-Voltage Networks	68	387	90	5	6	14	4

The clustering identified two different consumption patterns that showed complete opposites in consumption behavior.

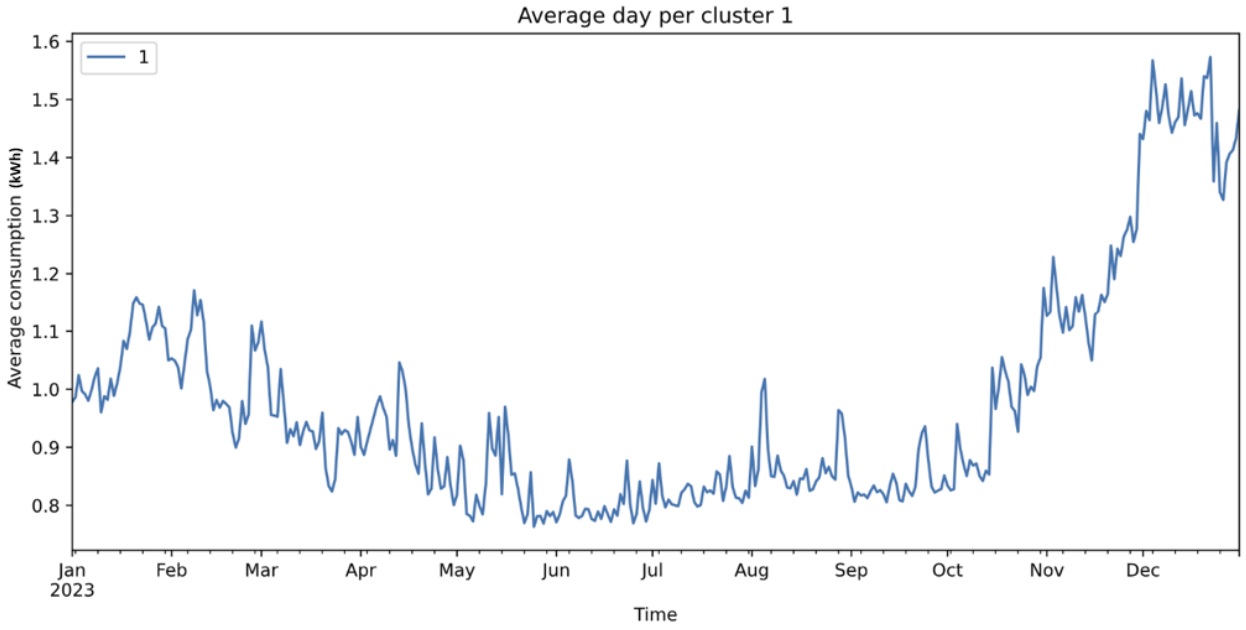

Figure 9 Cluster 1 Average daily consumption in kWh for typical consumption pattern

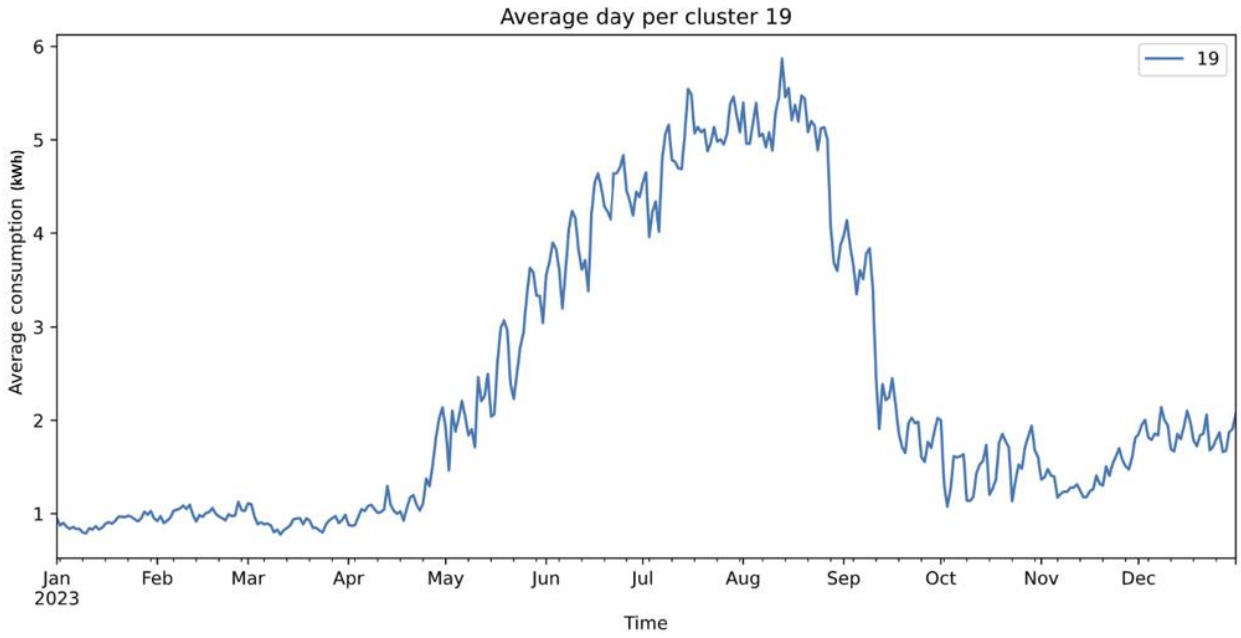

Figure 10 Cluster 19 average daily consumption in kWh for summer residence consumption patterns

The first group, including most networks, typically has high consumption during the winter months (see

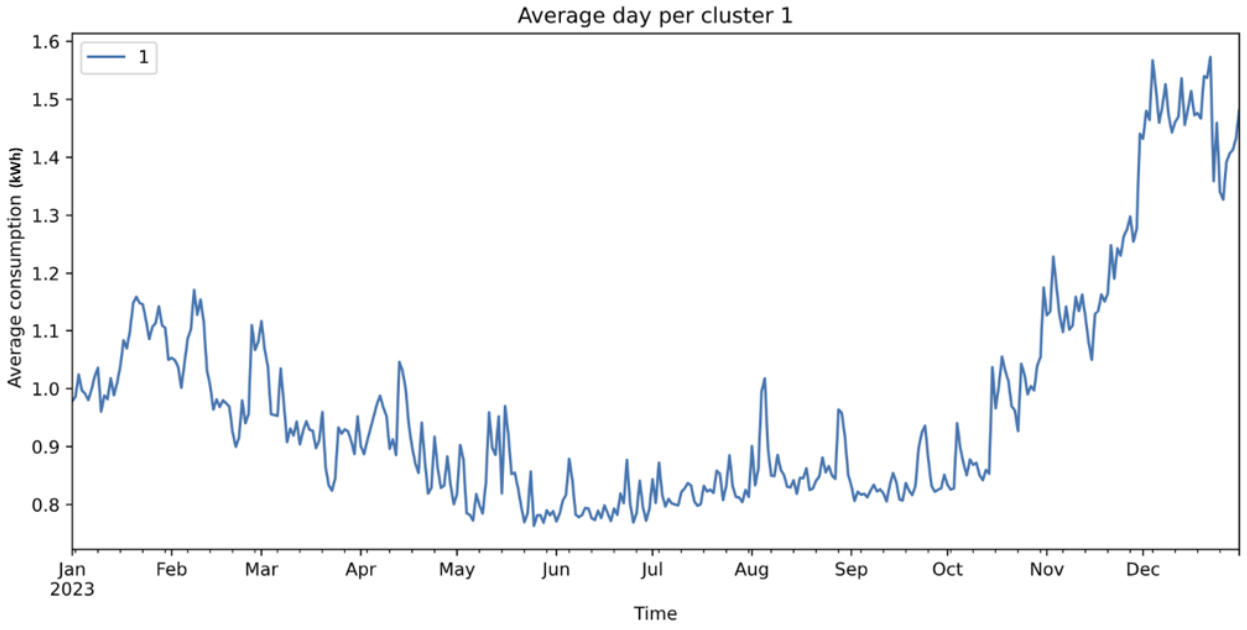

Figure 9), and the second group indicated typical summer residence areas, where consumption in winter was very low and increased significantly during the summer months (Figure 10). This

observation also supports the decision not to use Dynamic Time Warping and similar methods. Although the patterns may look similar, there is a big difference if those peaks occur in the summer or winter.

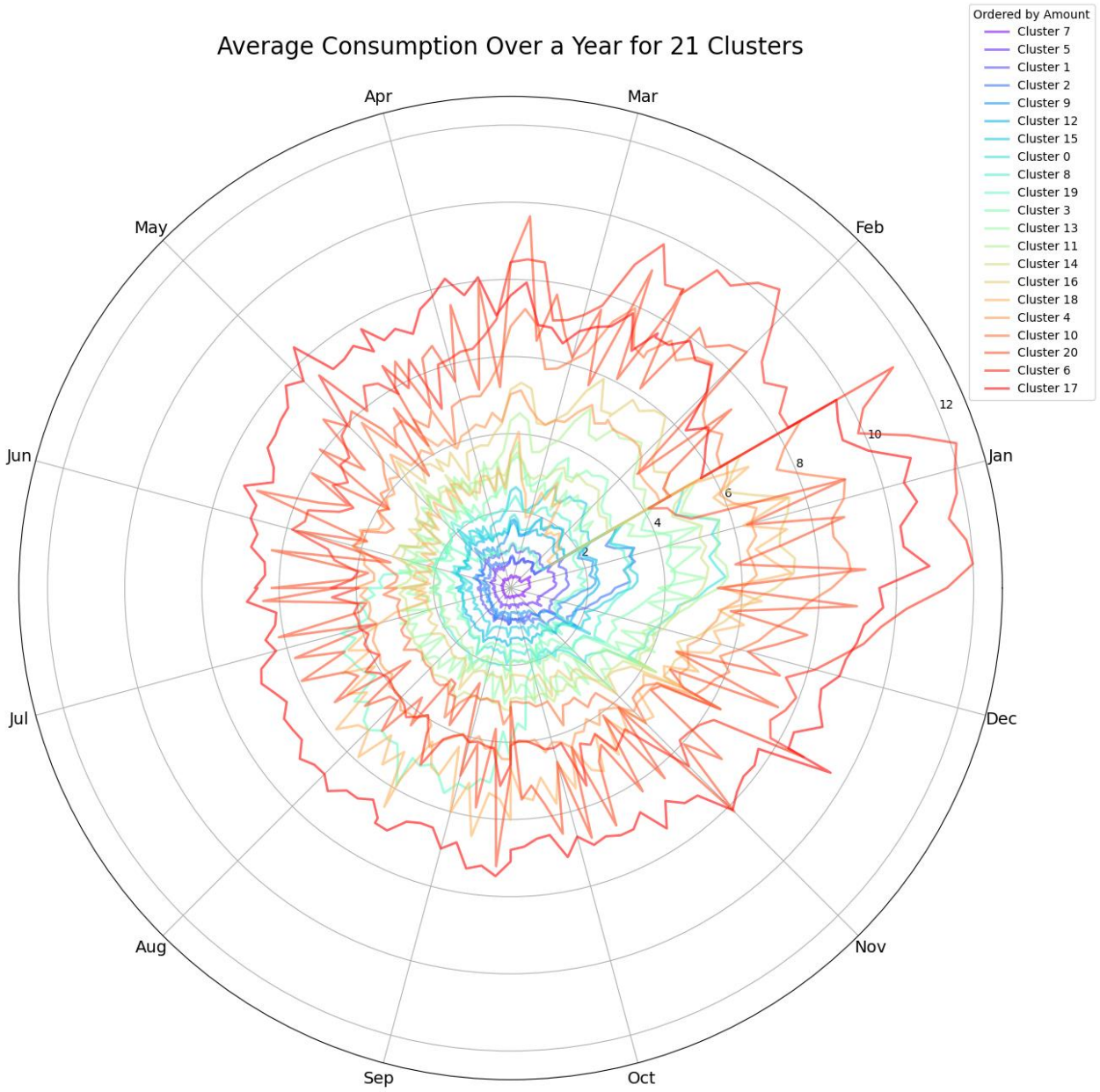

Figure 11 Polar plot of the average daily consumption over a year per cluster

Figure 11 illustrates a polar plot of all clusters' overall average monthly consumption over a year. Figure 12 is the respective plot for all clusters with a typical consumption pattern in KNG's

networks, including peaks during the winter months. Figure 13 illustrates atypical consumption patterns in clusters, including summer peaks or no seasonal dependence.

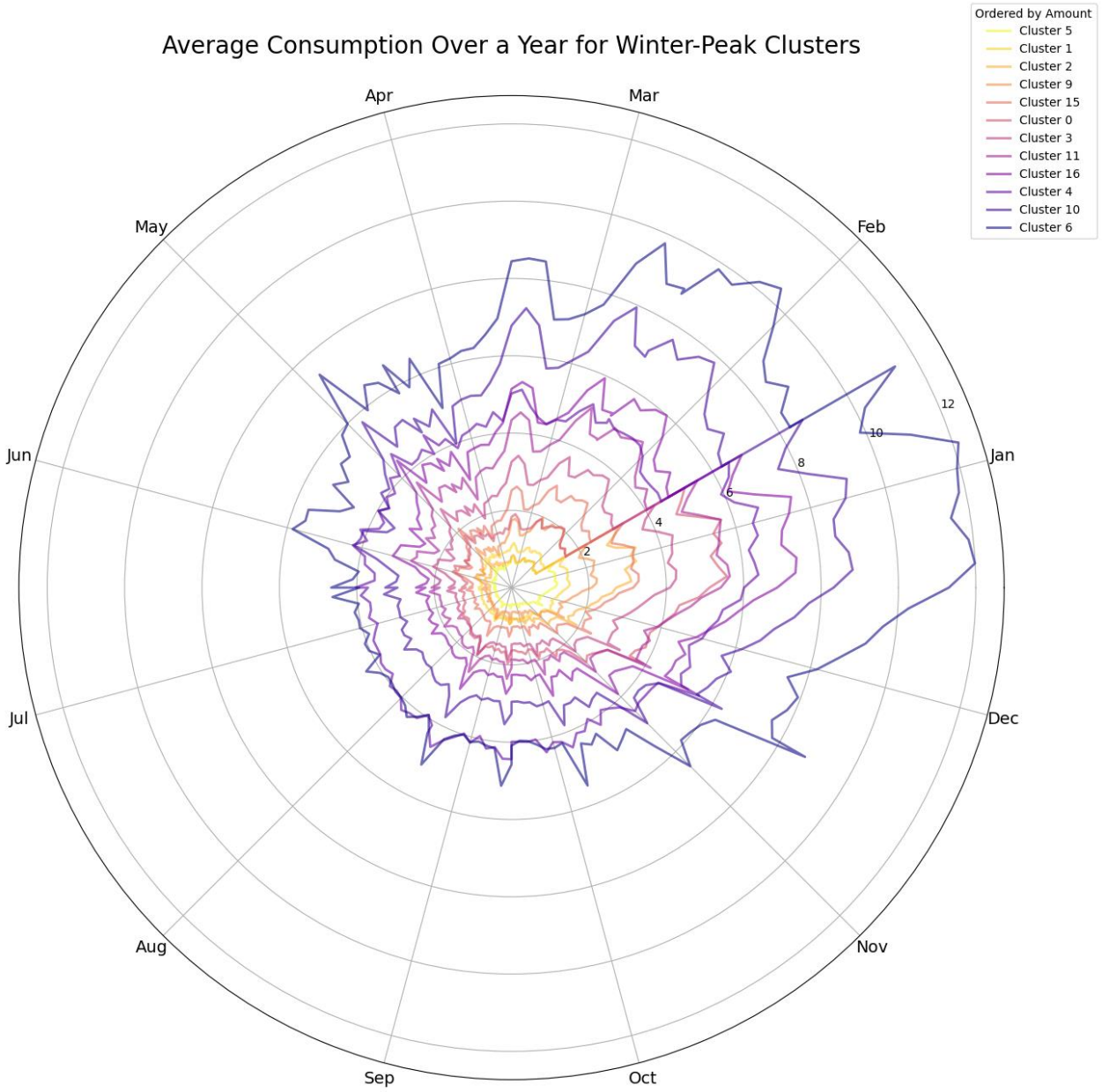

Figure 12 Polar plot for clusters with typical winter peaks

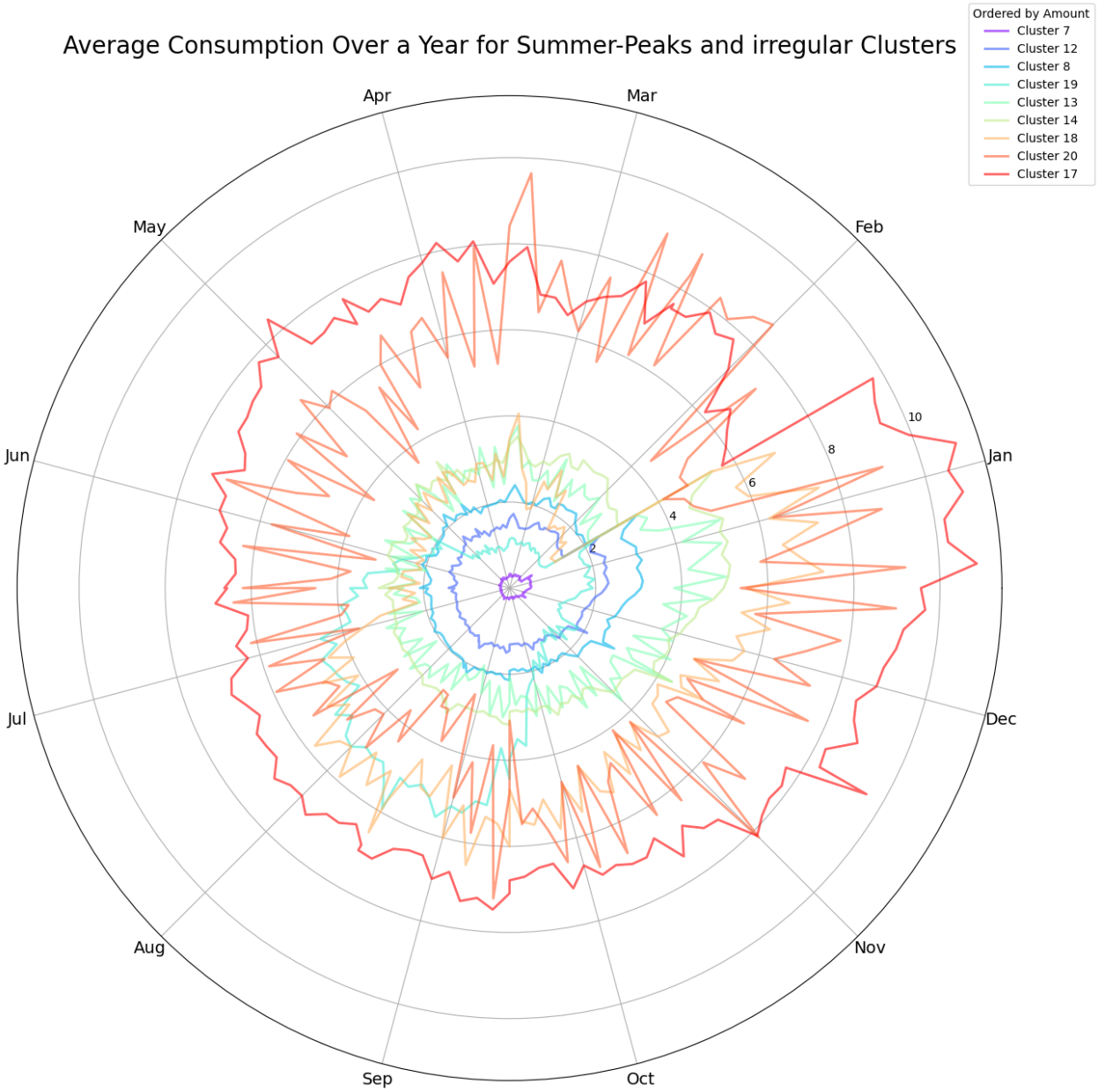

Figure 13 Polar plot for clusters with untypical consumption patterns

3.3 Consumption Forecast Evaluation Scenario

We needed to find an efficient evaluation scenario to determine the best model architecture for forecasting energy consumption. We chose clusters 5 and 19 to represent typical summer and winter annual consumption patterns, respectively.

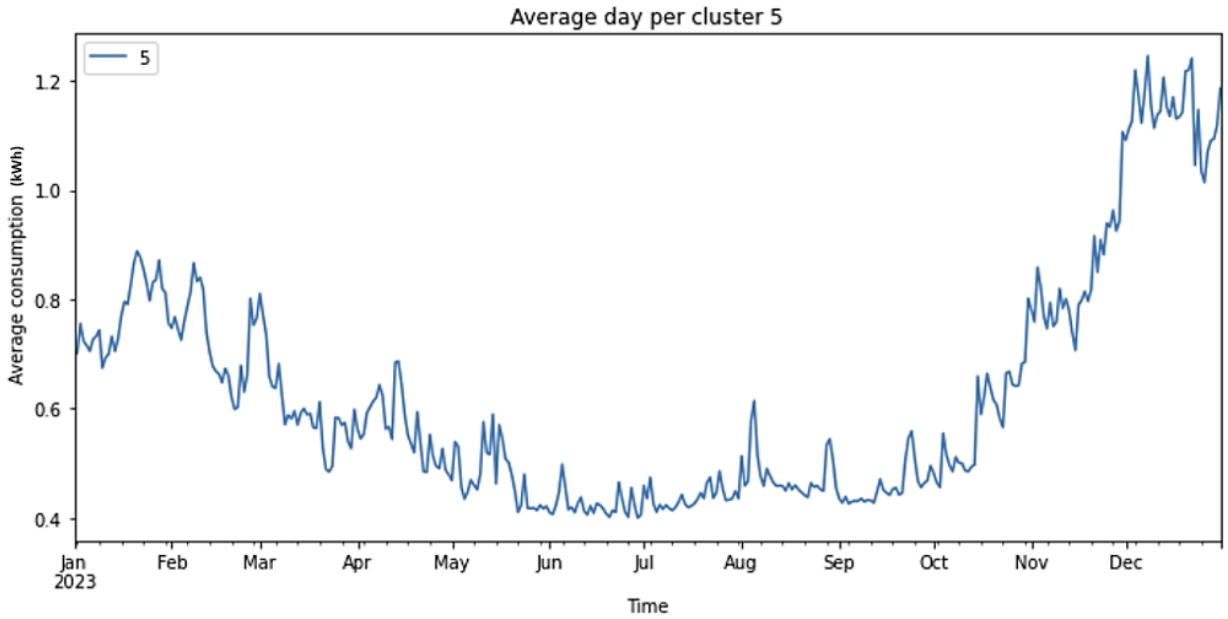

Figure 14 Average daily consumption of networks in cluster 5

Cluster 5 is the biggest cluster, and includes 824 individual low-voltage networks. Figure 14 illustrates the average daily consumption of the individual networks for 2023. Energy consumption peaks during the colder winter months and remains low during the summer.

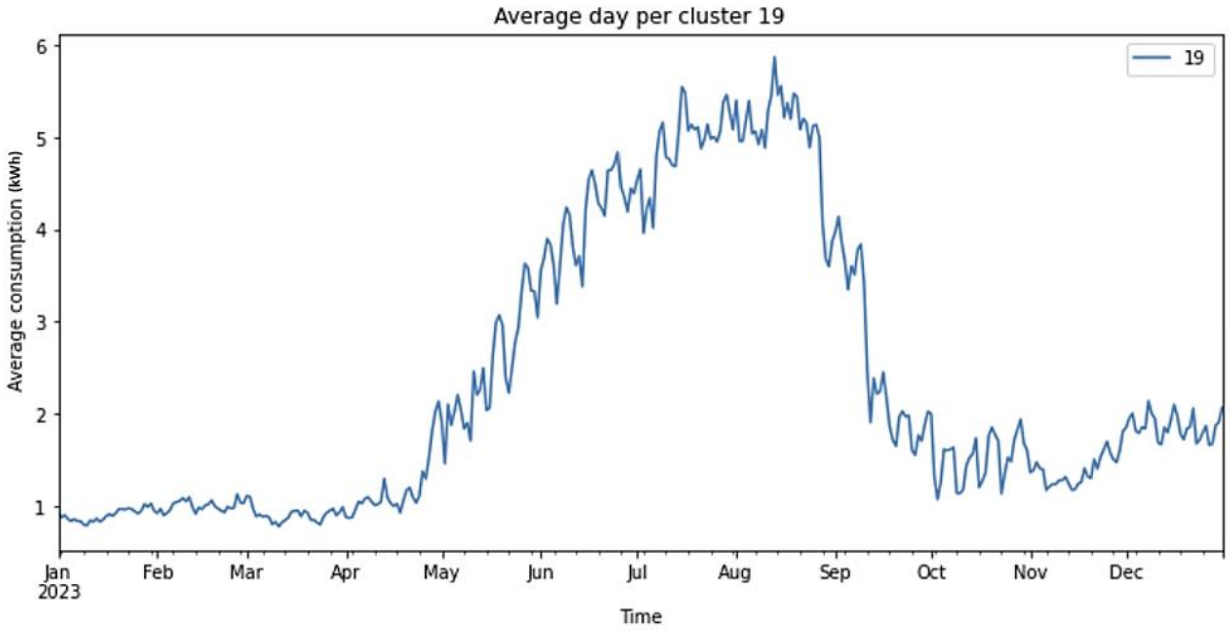

Figure 15 Average daily consumption of networks in cluster 19

Cluster 19 is a relatively small cluster of only 19 networks, but it has a unique consumption pattern with prominent peaks during the summer months (see Figure 15). The selected time frames of October 26th and 27th, 2023, which include an Austrian national holiday, indicate a distinct consumption pattern. The second scenario covers the weekend from October 28th to 29th, 2023, to represent a typical weekend. The third scenario spans from October 30th to 31st, representing normal workdays. The final selected time frame is from June 29th to 30th, 2023, representing a typical Thursday and Friday in the summer. This creates four different time frames for two clusters, resulting in eight evaluation scenarios. The training data always includes the data for the whole month. For example, the training data for the use case of October 27th-28th includes the data from October 1st to October 26th. All selected forecasting model architectures will be applied to all these scenarios to select the best-performing ones, which we will use to create the final models per consumption cluster. To compare the model results, we decided to compare the Mean Absolute Percentage Error (MAPE) and the Symmetric Mean Absolute Percentage (SMAPE) error in all evaluation scenarios due to their characteristics, such as scale independence and the fair treatment of over and underestimations. For a detailed analysis of possible evaluation metrics, see Chapter 4.6.

4. Time Series Forecasting Basics

This chapter explores the fundamental concepts of analyzing time series data. We first explain important properties, such as stationarity, and how to test for them. Next, we discuss the different forecasting horizons, followed by a section about the mathematical nature of forecasting results in terms of point versus probabilistic estimates. We conclude by discussing the metrics used to evaluate forecast results. This chapter provides the foundation to delve into the different methodologies in the following chapters.

4.1 Components and Time Series Decomposition

Exploratory data analysis is an important step in understanding time series data. It involves a decomposition process that separates a time series into its three components: the trend, the seasonality, and the residuals. This separation helps us create a preliminary visual understanding of the time series' nature and guides us to what preprocessing steps may be necessary to forecast its future values further.

The *statsmodels* Python package [42] offers a convenient method of performing time series decomposition. Figure 16 illustrates the results of the decomposition of two months of energy consumption data in one low-voltage network in Carinthia. The upper plot illustrates the original combined time series of the values. The second plot visualizes the trend, the third plot captures the seasonalities, and the third plot illustrates the remaining residuals. In the residual and seasonality plots, values range around the mean, resulting in positive or negative deviations from the mean. Noticeably, the trend plot clearly illustrates increasing behavior. The trend represents the time series values' slow-moving changes, gradual increases, or decreases. Therefore, the mean of the time series is not constant but changes gradually. This property causes major challenges in applying classical statistical models as they require the mean to be constant over time. Therefore, time series with a clear increasing or decreasing nature need to be changed before the data is fitted to the model. The seasonality component covers the seasonal pattern of the data where cycles repeat themselves over time. Notably, the data may have multiple seasonal patterns.

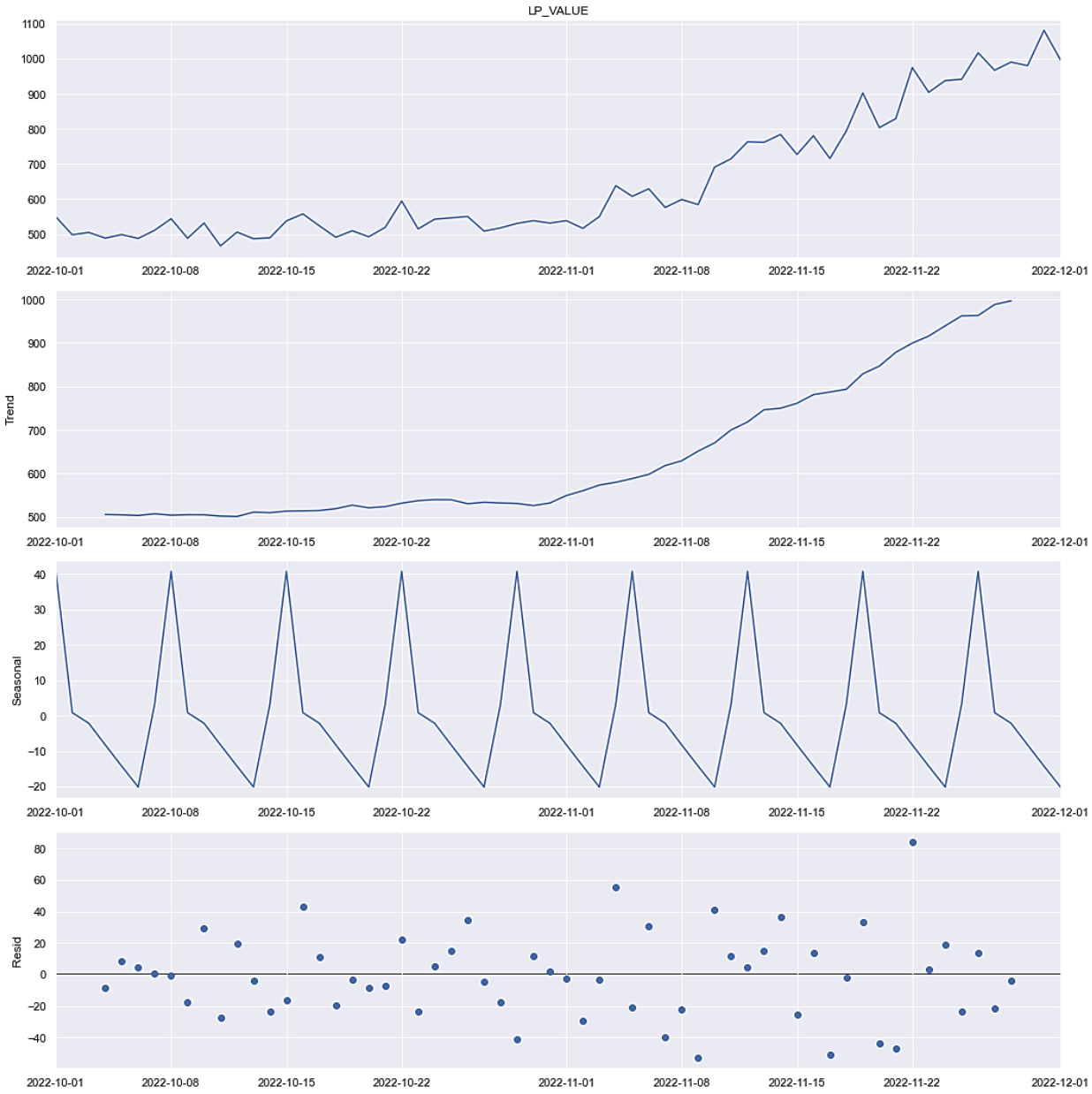

Figure 16 Time series decomposition for October/November 22 for one low-voltage network

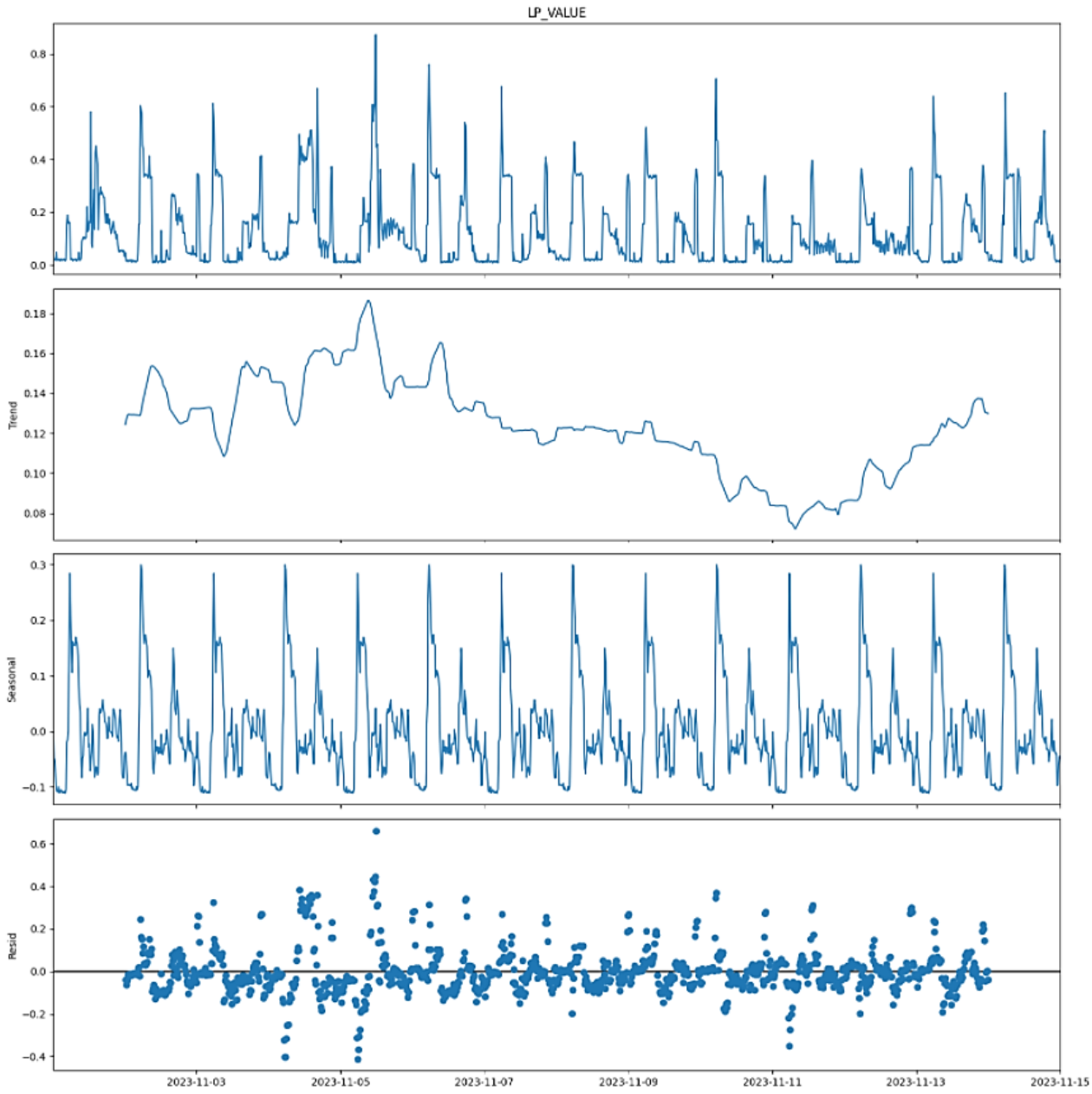

Figure 17 Time series decomposition plot for a two-week interval based on 15-minute consumption values revealing daily patterns

In our energy consumption use case, a weekly pattern repeating consumption patterns is visible. However, changing the observation interval to two weeks (see Figure 17), reveals that days also have distinct consumption patterns. Moreover, a third pattern based on yearly intervals associated with consumption that represents the actual seasons, such as summer and winter, is

present. This pattern becomes visible when the observation interval is increased. The last part of the time series decomposition is the residual plot, which reveals any behavior in the data that cannot be explained by the model and can be interpreted as unpredictable noise. The residual plot also reveals whether any regularity or pattern that the decomposition did not identify in the data is present, as the residuals are supposed to be white noise and not follow any pattern. Unsurprisingly, the three components need to be considered for any future forecast as trends and seasonalities from the past are likely to continue and should be reflected in the forecast [43].

4.2 Stationarity

While machine learning-based forecasting algorithms can work with stationary and non-stationary time series, classical statistical models, such as ARIMA, require the data to be stationary to produce valid forecasting results. Stationarity defines that the statistical properties of a time series, such as the mean, variance, and autocorrelation, do not change over time [44, p. 36]. The requirement for stationarity seems intuitive for classical statistical models, as the model parameters of Auto Regressive (AR) or Moving Average (MA) models cannot change over time, which would be required for time series with changing statistical properties. The mean and variance are not a function of time. Per definition, a stationary process is one where, for all possible lags, k, the distribution of $y_t, y_{t+1}, \ldots, y_{t+k}$ does not depend on it [43, p. 38]. A common statistical method to test for stationarity is the Augmented Dickey-Fuller-Test (ADF) [45]. It does so by testing for the presence of a unit root, which indicates that 1 is a solution of the process's characteristic equation. If it is present, then the time series is not stationary. The ADF test is a statistical hypothesis test, with the null hypothesis being that our time series is not stationary [44, p. 38]. In this work, we use the ADF test for stationarity before applying the SARIMA model in Chapter 6.1. Identifying stationarity is one thing. Removing it from the time series is the next step if the model requires it. Transformation, such as differencing, can remove the trend and seasonality to stabilize the mean. Differencing is, at its core, the calculation of the difference from one timestep to another. Differencing converts the time series of values into a time series of value changes over time [43, p. 186].

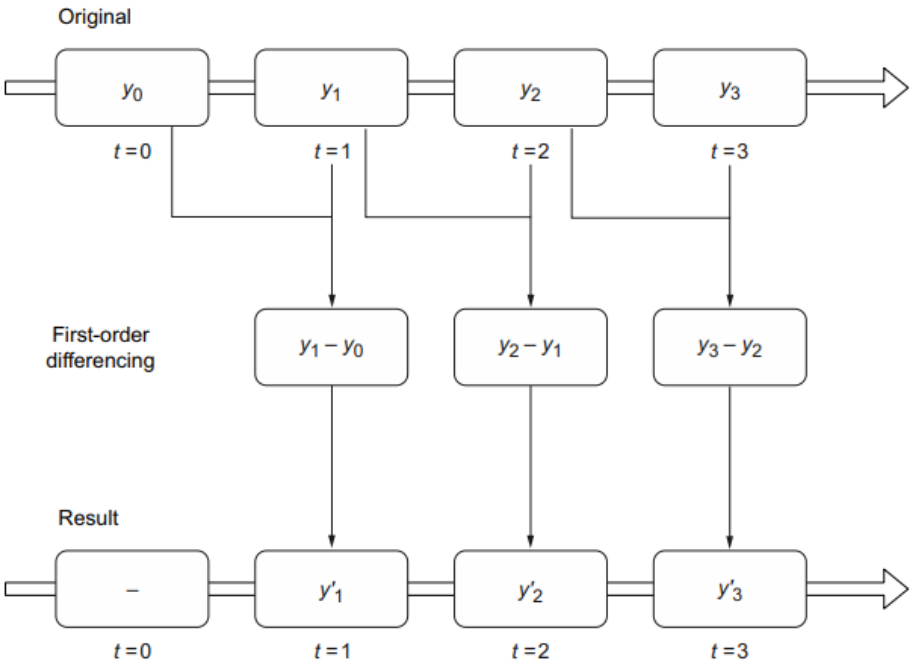

Figure 18 Visualization of the differencing transformation [44, p. 37]

Figure 18 visualizes a first-order differencing. Notably, every differencing transformation removes the first data point from the data because the difference to its previous step cannot be calculated.

4.3 Univariate versus Multivariate Time Series

A univariate time series refers to a time series in which just one variable is measured against time. Most of the use cases we analyze in this work are based on a univariate approach.

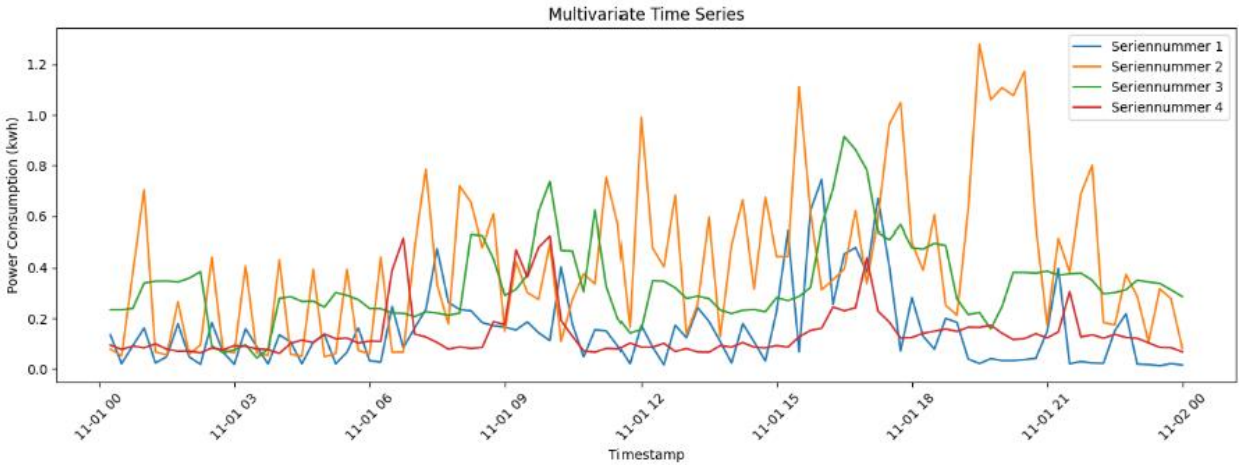

Figure 19 Multivariate time series of four measured variables

Multiple time series have multiple variables measured at each timestamp. Their analysis is sometimes challenging, as the measured variables are interrelated and have temporal dependencies. Figure 19 illustrates a multivariate time series problem with four measured variables for the power consumption for one day. The multivariate approach should not be confused with the univariate case when additional regressors are involved. When a time series is accompanied by other variables measured over time that are supposed to aid in predicting future values of the main variable, these additional time series are referred to as regressors. The future values of the regressors are meant to assist in predicting the main variable, but the regressors themselves are not forecasted [43, p. 21].

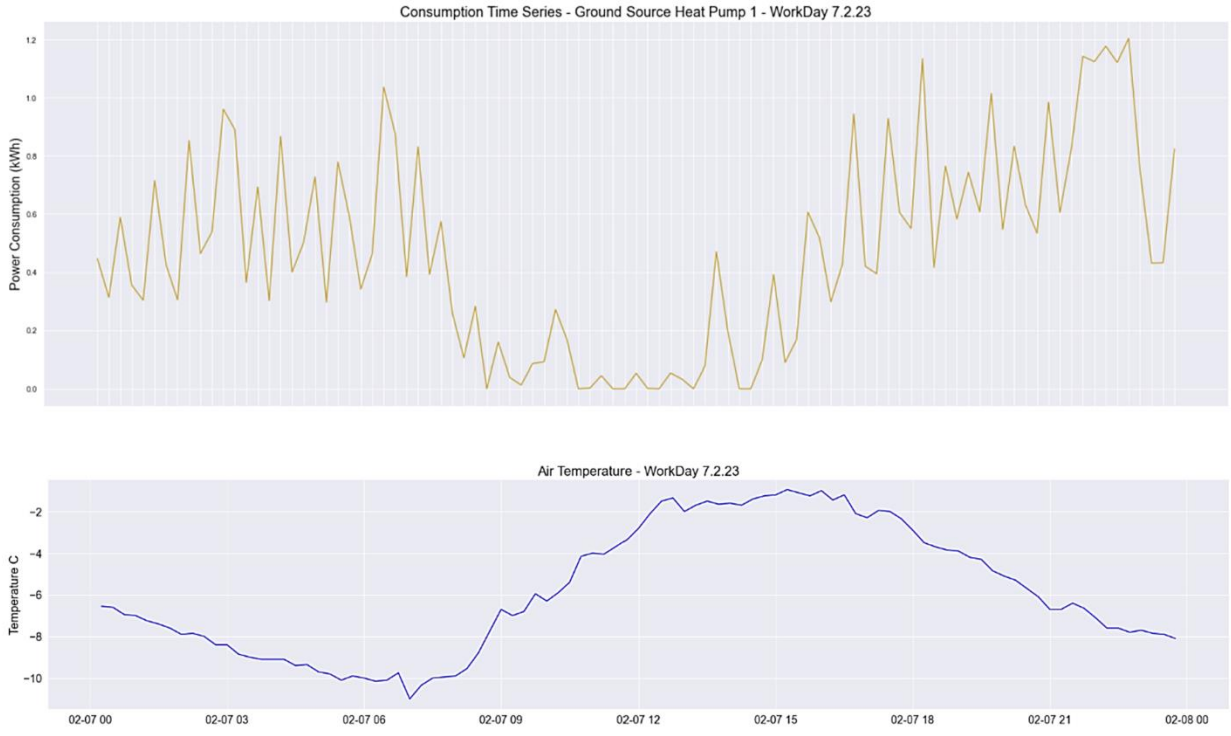

Figure 20 Univariate time series with an additional regressor

Figure 20 illustrates a case where an additional regressor for the air temperature in the same time frame accompanies a univariate time series of a day's power consumption.

4.4 Forecasting Horizons

Before building a forecasting model, one must select the appropriate forecasting horizons. Suppose the data granularity is 15-minute intervals, such as in our use case. Accordingly, we need to decide if the model will output the next 15-minute interval based on a past window size of data or if the model is supposed to forecast the next 48 hours, which would require 192 timesteps into the future.

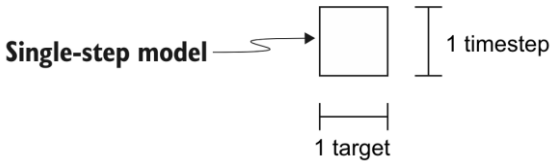

Figure 21 Output of a single-step model [44, p. 235]

The simplest model is the single-step model, which outputs a single value and a single timestep as the prediction. The output of a single-step model is usually a scalar (see Figure 21).

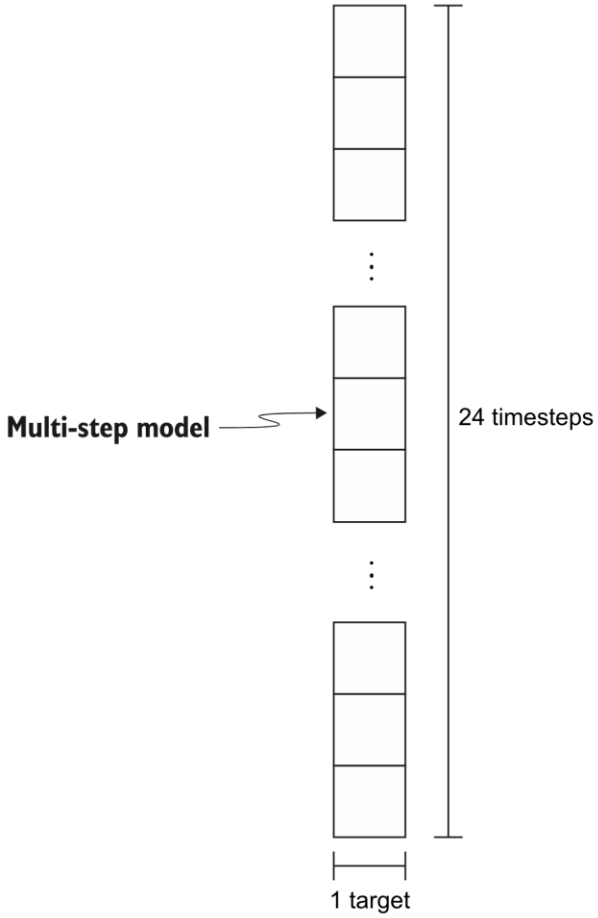

Figure 22 Output of a multistep model with output intervals of 24 times [44, p. 235]

The multistep model (see Figure 22) outputs predictions for one variable, such as the single-step model, but for multiple timesteps into the future in one forecasting step, which is a much more likely scenario from a practitioner's point of view.

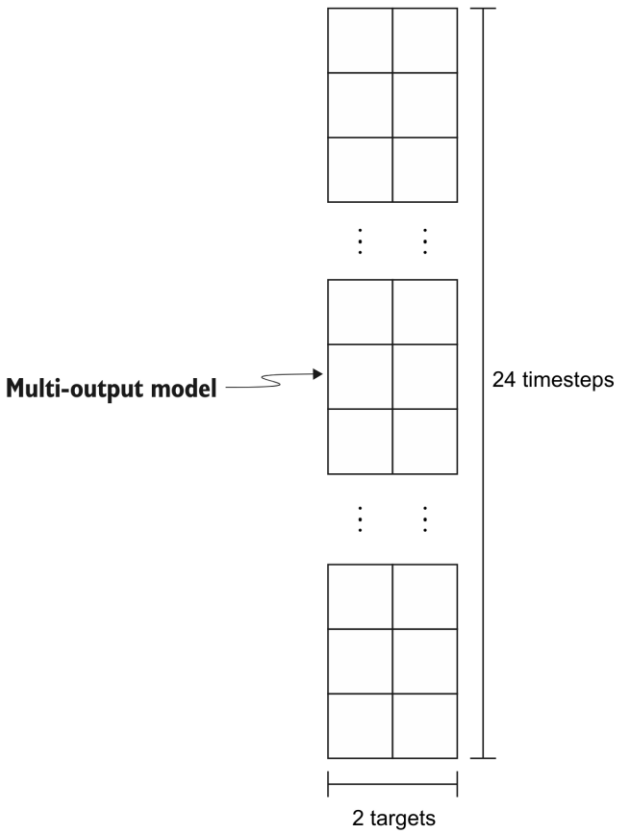

Figure 23 Output of a multi-output multistep model [40 p.236]

A third model option is available for a multivariate time series: the multi-output model (see Figure 23). It creates predictions for more than one target variable for multiple timesteps into the future. Therefore, if we were to create consumption forecasts for multiple low-voltage networks in one go, with one model for two days into the future, we would need a multi-output model. As mentioned, our use case scenario is based on two-day ahead forecasts based on single or multiple time series, built from aggregated multiple consumption profiles and including possible external variables. As a result, we will deal with univariate or multivariate, multistep forecasting models with or without additional regressors.

4.5 Probabilistic Estimates

Point estimates play an important role in predicting the future value of a target variable. However, comprehending the uncertainty associated with a forecast is also critical for decision-makers.

When forecasts have a high degree of uncertainty, model users may adapt their decision-making process or turn to alternative sources of information for additional insights. Not all forecasting models support the generation of probabilistic forecasts. Temporal Fusion Transformers (TFT) generate prediction intervals in addition to point forecasts by simultaneously predicting the percentiles at each time step [46]. For example, quantile regression can estimate the target variable's conditional median. Apart from the median, quantile regression can calculate any percentile, meaning the model can output a prediction interval around the actual prediction. Figure 24 illustrates what the percentiles for 5, 25, 50, 75, and 95% could look like in a regression problem.

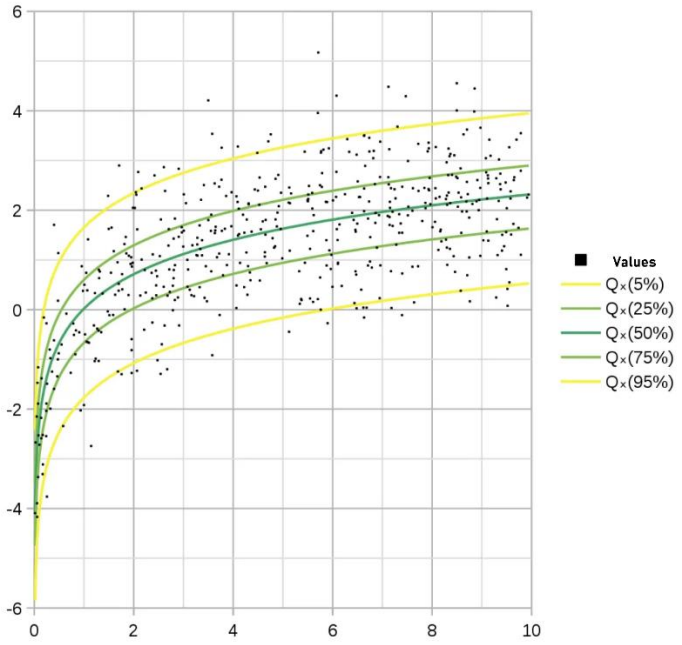

Figure 24 Example for percentiles in a regression problem [47]

The quantile loss function for an actual (y) and predicted value ($\hat{y}$) is defined as:

$$QL(y, \hat{y}, q) = \max\left[q(y - \hat{y}), (1 - q)(y - \hat{y})\right] \qquad (1)$$

With increasing q, overestimations are penalized more than underestimations. Therefore, if q is 0.75, overestimations are penalized by a factor of 0.75, and underestimations by a factor of 0.25.

However, quantile regression can become problematic when presented with the extreme tail of the response distribution. The corresponding parameter estimators may be highly imprecise due to data scarcity in the extreme tails, or there may be notable peaks in the data, to which quantile methods may react strongly [48]. Therefore, distributional regression methods have become increasingly popular recently. Distributional regression is a statistical method where we model the entire distribution of an outcome or dependent variable rather than just a single summary statistic like the mean or median, while they are more general than quantile regression by targeting either the complete conditional response distribution or more general features thereof. These methods are more flexible as they allow modeling of how covariates influence different aspects of the distribution, making them powerful tools for statistical analysis. In contrast to quantile regression methods, which are nonparametric, distributional regression methods are parametric. See Figure 25 for an example of the relationship between income and unemployment for individuals where complete conditional distributions (blue) and quantile distributions for the 5^{th}, 25^{th}, 50^{th}, 75^{th}, and 95^{th} percentiles (red) are given, estimated by a distributional regression approach.

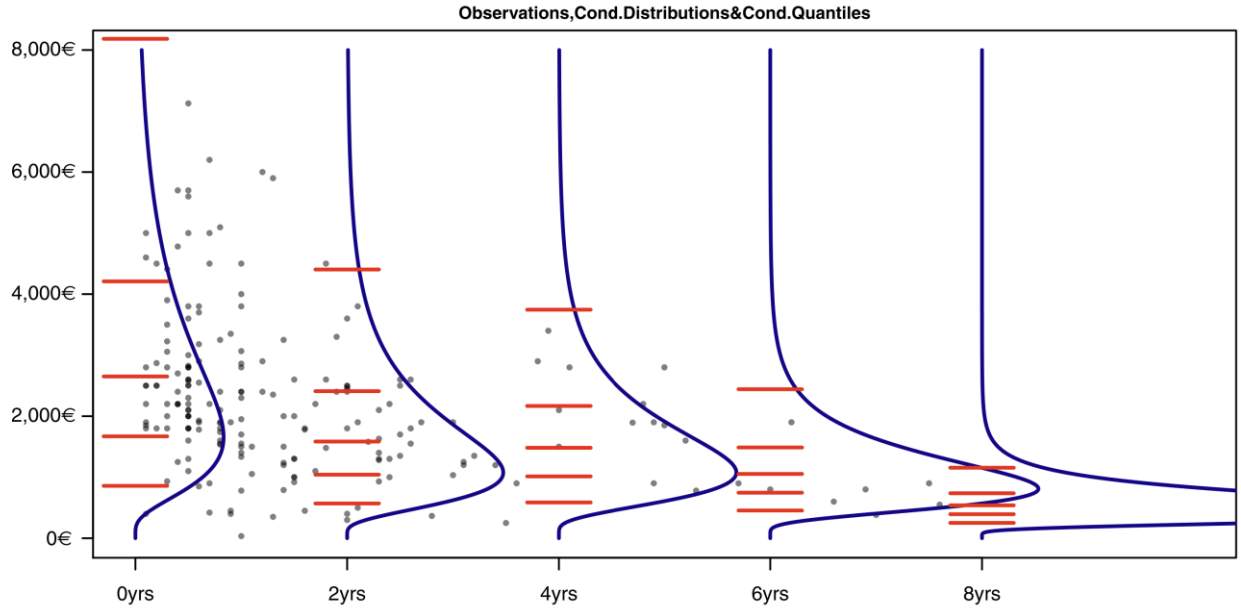

Figure 25 Parameter estimates with derived conditional distributions (blue) and quantiles (red) calculated using GAMLSS [49]

Distributional regression methods are an umbrella term that includes specializations known as generalized additive models for location, scale, and shape (GAMLSS), conditional transformation models and distribution regression density regression, and the already mentioned quantile regression [49].

Considering the importance of probabilistic estimates, we analyzed current methods in time service forecasting according to their ability to provide them. Table 4.1 provides an overview of the availability of probabilistic forecasts and user-defined quantiles for all algorithms used in this thesis and related algorithms to forecast energy consumption.

Table 4.1 Support of probabilistic Forecasts per model architecture

Algorithm	**Probabilistic Forecasts available in the standard architecture**	**User-defined Quantile Selection**
SARIMA [50]	yes	yes
Basic Kalman Filters [51]	no	no
Extended Kalman Filters [52] [53]	no	no
Ensemble Kalman Filters [54]	yes	yes
Bayesian Filters [55]	yes	yes
FB Prophet [56]	yes (Uncertainty Intervals)	indirectly (Using predictive samples)
LSTM [57]	no	no
Neural Prophet [58]	yes (Uncertainty Intervals)	yes
iTransformer [59]	no	no
PatchTST [60]	no	no
Temporal Fusion Transformer [46]	yes	yes

Lag Llama [61]	yes	yes
MOIRAI [62]	yes	yes
AutoGluon [63]	yes	Yes
TimeMixer [64]	yes	yes

4.6 Metrics in Time Series Forecasting

A systematic evaluation requires suitable performance metrics to compare the effectiveness of different forecasting approaches. The choice of the right metric depends on a variety of factors, such as a focus on the penalization of outliers in the forecast, calculation effort, interpretability, or the availability of a common scale. This chapter examines the most common metrics used in forecasting evaluation and examines their advantages and disadvantages [65]. In the following equations, *n* is the number of samples or timesteps in our use case, *A* is the actual value, and *F* is the forecast.

4.6.1 Mean Absolute Error (MAE)

While the mean absolute error is commonly used, it is not necessarily a forecasting-specific metric. It measures the absolute difference between the forecasted and actual values, representing the sum of the absolute errors. The lower the value, the more accurate the forecasting is. See Equation 2 presents the detailed calculation.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|A_i - F_i| \qquad (2)$$

The mean absolute error is a straightforward and easy-to-understand metric. It uses the same unit as the input values, making it even easier to interpret. The metric does not penalize outliers in the forecast, which may be a major disadvantage if the use case is sensitive to forecasts that are far out of range. Its biggest disadvantage is the scale dependency, making it impossible to compare the time series forecasting performance of use cases with different energy aggregation intervals [66].

4.6.2 Mean Squared Error (MSE)

The mean squared error is another commonly used metric in time series forecasting, like the mean absolute error, which is not a forecasting-specific error. It improves the mean absolute error by squaring the difference between the forecasted and actual values (see Equation 3).

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(A_i - F_i)^2 \qquad (3)$$

Squaring the difference heavily penalizes outliers in the prediction, which may be an advantage or disadvantage, depending on the use case. This leads to a mismatch in the units between the metric and the input value units, which may influence its interpretability. Another disadvantage is its scale dependency, which makes it unsuitable for comparing the forecasting results for different aggregation units or time intervals in the input data [66].

4.6.3 Root Mean Squared Error (RMSE)

The root mean squared error is a continuation of the mean squared error, where the square root of the result is calculated (see Equation 4).

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(A_i - F_i)^2} \qquad (4)$$

It preserves the mean squared errors characteristic of penalizing outliers while the result will have the same unit as the input value. It combines the advantages of the mean squared and absolute errors. However, it also shares the disadvantage of being scale-dependent [66].

4.6.4 Mean Absolute Percentage Error (MAPE)

The mean absolute percentage error is a forecasting-specific metric that measures the percentage difference between the actual and the forecasted value (see Equation 5).

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left(100 * \frac{|A_i - F_i|}{A_i}\right) \qquad (5)$$

Its major advantage is its unit-free and scale-independent characteristic, making it very easy to interpret and the go-to base metric for comparing different forecasting models. The mean absolute percentage error becomes unsuitable or unusable when the actual values are close or equal to zero [65, p. 45]. Another shortcoming of the mean absolute percentage error is that it penalizes negative errors more than positive errors. Both disadvantages lead to the development of symmetric evaluation methods [66].

4.6.5 Symmetric Mean Absolute Percentage Error (SMAPE)

The symmetric mean absolute percentage error is another forecasting-specific metric that was proposed by Armstrong in 1978 [67] (see Equation 6).

$$SMAPE = \frac{1}{n}\sum_{i=1}^{n}\left(200 * \frac{|A_i - F_i|}{|A_i| + |F_i|}\right) \quad (6)$$

The symmetric mean absolute percentage error solves the problem of the mean absolute percentage error becoming undefined when the time series values are equal to zero. It is a suitable alternative to the mean absolute percentage error when scale independence is required, and input values are very small. It is especially useful for cases where both overestimation and underestimation errors must be considered equally [66]. However, the symmetric mean absolute percentage error is less intuitive than the mean or absolute percentage error.

4.6.6 Mean Absolute Scaled Error (MASE)

The forecasting-specific scaled error metrics were proposed by Hyndman and Koehler [68]. The mean absolute scaled error can be calculated for time series without seasonality (Equation 7) and time series following seasonality with seasonal index *m* (Equation 8) with training data length *T*.

$$MASE\ (no\ seasonality) = \frac{\frac{1}{n}\sum_{i=1}^{n}|A_i - F_i|}{\frac{1}{T-1}\sum_{t=2}^{T}|A_t - A_{t-1}|} = \frac{MAE}{\frac{1}{T-1}\sum_{t=2}^{T}|A_t - A_{t-1}|} \quad (7)$$

$$MASE(with\ seasonality) = \frac{\frac{1}{n}\sum_{i=1}^{n}|A_i - F_i|}{\frac{1}{T-m}\sum_{t=m+1}^{T}|A_t - A_{t-m}|} = \frac{MAE}{\frac{1}{T-m}\sum_{t=m+1}^{T}|A_t - A_{t-m}|} \quad (8)$$

If the result is less than 1, the forecast is better than the average naive approach. It solves the shortcoming of the mean absolute percentage error by being able to handle actual values close to or equal to zero. It is scale-independent and symmetric by penalizing over and under forecasts equally. The only shortcoming of the mean absolute scaled error is that it is not intuitive and may make it difficult to communicate the results to end users [66].

4.6.7 Evaluation Metric Selection

In this thesis, we use the MAPE and the SMAPE to evaluate our time series forecasting models. MAPE is selected due to its ease of interpretation and scale independence, making it a widely understood and effective metric for comparing different forecasting approaches. However, recognizing MAPE's limitations when actual values are close to zero, we use SMAPE as a complementary metric. SMAPE addresses the issues with MAPE by providing a reliable measure even when forecasted values are near zero, ensuring a more accurate and balanced evaluation of our models.

5. Model Architectures for Time Series Forecast

In this chapter, we examine all methods used in this work to forecast energy consumption in the selected evaluation scenarios. First, their inner workings and characteristics are examined before they are applied to the use cases. The best-performing prototype models will then be taken as example implementations for the final use case of the energy consumption forecast. We start with classical statistical models as a baseline, as such models often deliver solid performance. We will continue with automated forecasting frameworks like FB Prophet and machine learning-based methods ranging from established methods like LSTM to new approaches using Transformer architectures. The last two sections of this chapter examine an LLM-based approach and the fully automized ensemble method, AutoGlueon TS, which was created to combine the best of all worlds.

5.1 Statistical Models for Time Series Forecasting

Classical statistical time series forecasting models have long been valuable tools in the field and are still widely used in various contexts, including academic research and industry applications. Statistical models, such as Auto Regressive models, Moving Average models, Auto Regressive Integrated Moving Average (ARIMA) models, and Vector Auto Regression (VAR), make strong assumptions about the statistical properties of the time series being analyzed.

Other methods in this category are exponential smoothing strategies, which originated in the 1950s from the work of R.G. Brown [69], [70] , C.C. Holt [71], and P.R. Winters [72] by focusing on the idea that most recent values are more important than past values. In this work, we use the Seasonal AutoRegressive Integrated Moving Average (SARIMA) model as our classical statistical approach. SARIMA was chosen because it is a widely used and well-established method in time series forecasting, and it is often considered a standard go-to model. The following chapter describes its components as a foundation for its application in the next step.

The second part of this chapter revolves around FB Prophet. Since FB Prophet is, at its core, a statistical method, it is discussed in this section. It is not a classical statistical approach but rather a new implementation that enables users to perform automated forecasts using strong default

parameters and removing the requirement for tedious preprocessing steps while working internally with Bayesian inference.

Classical statistical methods have major advantages, including their simplicity and transparency, which contribute to their explainability. They are suitable for even very small datasets for which typical machine learning-based approaches do not work due to the lack of training data. Even though they are very simple models, they perform very well compared to more complex deep learning models, with a good trade-off between performance and training time.

One of their drawbacks is that they cannot handle nonlinear dynamics in the input data, per definition. In cases where nonlinear dynamics are predominant in the data set, deep learning-based approaches might be a better alternative. They do not necessarily perform better with the provision of more training data. If the amount of training data is high, machine learning-based, especially deep learning-based methods, may also deliver better results [43, p. 203].

5.1.1 Seasonal Auto Regressive Integrated Moving Average Models (SARIMA)

In the 1970s, Box and Jenkins [50] developed a methodology for creating Auto Regressive integrated moving average (ARIMA) forecasting models, which greatly impacted research and industry alike and remain highly relevant to this day.

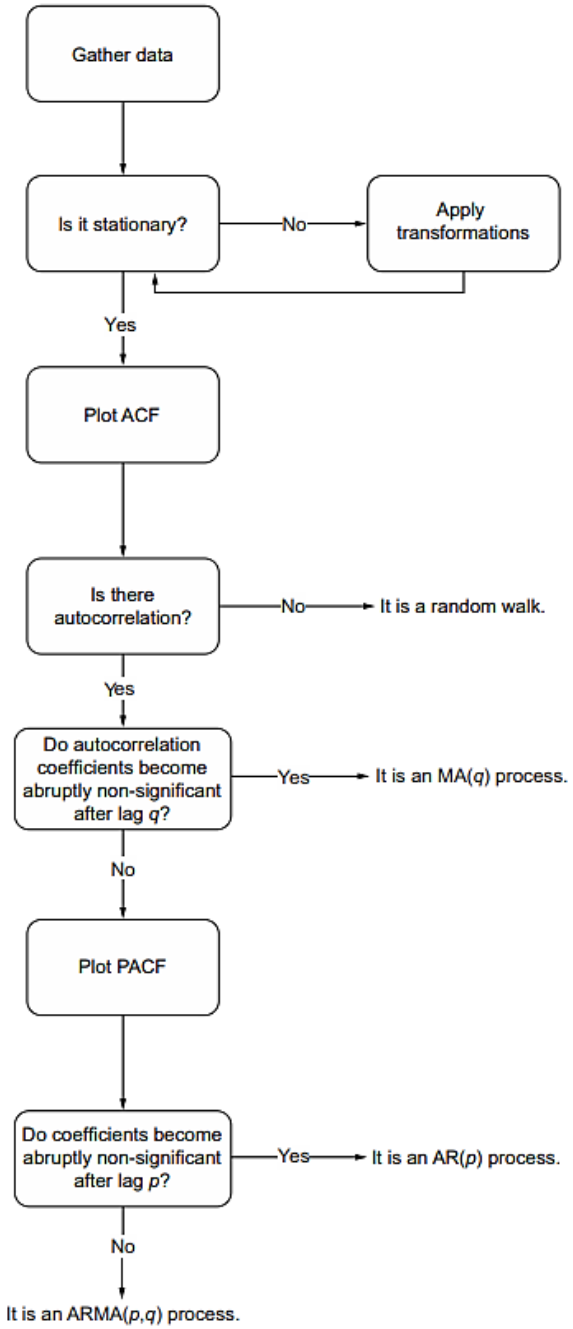

Figure 26 Workflow to identify a random walk, a moving average process MA(q), an Auto Regressive process AR(p), and an Auto Regressive Moving Average Process ARMA(p,q) [44, p. 107]

Figure 26 illustrates a cookbook-like approach to finding the appropriate model for the respective input data. In the first step, the data is checked for stationarity. This process step had already been discussed in Chapter 4.2. As soon as the ADF test verifies the stationarity of the data, the Autocorrelation Function (ACF) is used to check for autocorrelation in the input time series. The Autocorrelation Function measures the correlation of the time series with itself [44, p. 41]. It describes the linear relationship between lagged values of a given time series. The lag operator, also known as the backshift operator *B*, is used to simplify the notation of lagged values. It operates on individual time series points y and moves them back by one step every time it is applied (see Equation 9).

$$B^k\, y_t = y_{t-k} \qquad (9)$$

The lag operations in the following section are based on the Backshift operator *B*.

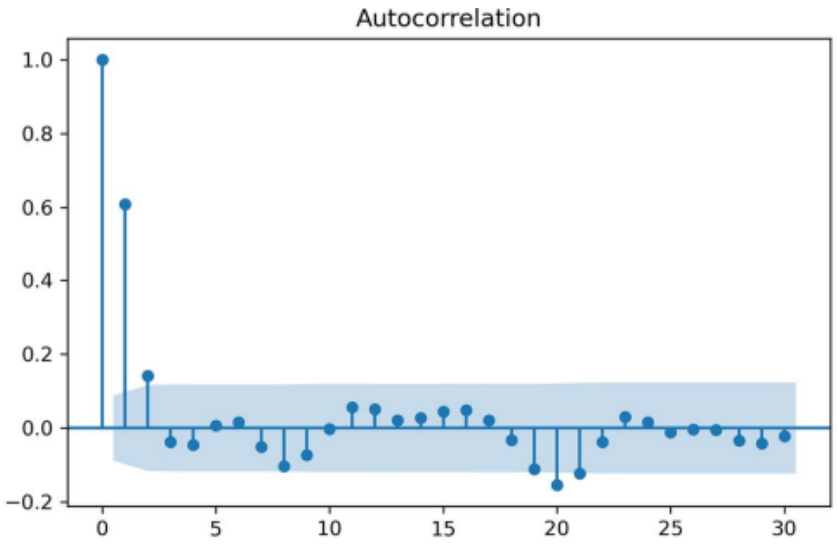

Figure 27 ACF for 30 lags of a time series [44, p. 69]

The ACF plot visualizes the autocorrelation coefficients per lag. To calculate the coefficient for lag 1, the coefficient between y_t and y_{t-1} must be calculated. The coefficient for lag 2 is calculated using y_t and y_{t-2}. In the ACF plot, the coefficient is the dependent variable, and the lag is the independent variable. The coefficient for lag will always be 1 because the linear relationship

between a variable and itself is always equal to 1 [44, p. 42]. Figure 27 illustrates the ACF plot for 30 lags. The light blue area marks the significance level. The first two coefficients are above the significance level, indicating an autocorrelation. The significant coefficients around the 20th lag are likely due to chance, as there are no significant coefficients between lags 3 and 19.

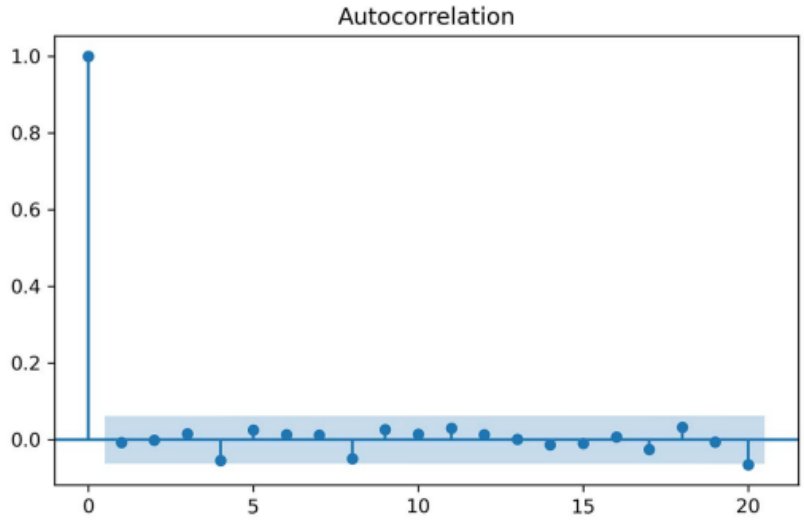

Figure 28 ACF of a Random Walk [44, p. 45]

The ACF plot in Figure 28 indicates that no autocorrelation is present, as no coefficients are above the significance level. This indicates that we are dealing with a random walk process, and statistical models such as ARIMA cannot be applied to generate forecasts. A random walk is a process where the time series changes upward or downward are equally high. It is a series whose first difference is stationary and uncorrelated. It moves completely at random [44, p. 35]. The present value depends on the previous value *yt-1, a* constant *C,* and a random number ϵ_t (see Equation 10) [44, p. 32].

$$y_t = y_{t-1} + \epsilon_t \qquad (10)$$

Once it is clarified that we are dealing with an autocorrelated time series and not a random walk, we can proceed, according to Figure 26, to determine the appropriate model. If the autocorrelation coefficients become non-significant after lag *q,* the data can be modeled using a Moving Average MA(*q*) process. In a Moving Average (MA) model, the current value is linearly

dependent on the current and past error terms. Error terms are assumed to be white noise, so they are mutually independent and normally distributed. In an MA(q) model, q denotes the order determining the number of past errors affecting the current value. The larger the order, the more past errors affect the present value. In an MA(1) model, only the current and the previous error terms are considered. The present value is a linear combination of the series' mean, the present error term ϵ_t, and the past error terms ϵ_{t-q} (see Equation 11). The impact of past errors is quantified using the coefficient θ_q [43, p. 63].

$$y_t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q} \qquad (11)$$

The linear dependence of past errors restricts the forecasting horizon in MA models. The error terms are not present in the data and must be recursively estimated. That means an MA(q) model can only forecast q steps into the future. For any additional step, the model will only forecast the mean. Using a rolling forecast can help avoid this restriction. Forecasts of MA models generally converge quickly to the process's mean [43, p. 185].

Returning to the decision process in Figure 26, we must still clarify what actions are required if the autocorrelation coefficients slowly decay or show a sinusoidal fluctuation. This indicates that an Auto Regressive process might be required. An Auto Regressive process is a regression of a variable against itself. The Auto Regressive model assumes that past values can predict the future. The present value of a time series linearly depends on its past values. In an Auto Regressive process, AR(p), p defines the process order, meaning the number of past values the current process depends on. The Auto Regressive process AR(p) can be expressed as

$$y_t = C + \varphi_1 y_{t-1} + \varphi_2 y_{t-2} + \cdots + \varphi_p y_{t-p} + \epsilon_t \qquad (12)$$

where y_t is the current value,

C is a constant value,

$\varphi_p y_{t-p}$ is the value of timestamp y_{t-p} whose magnitude is affected by the coefficient φ_p.

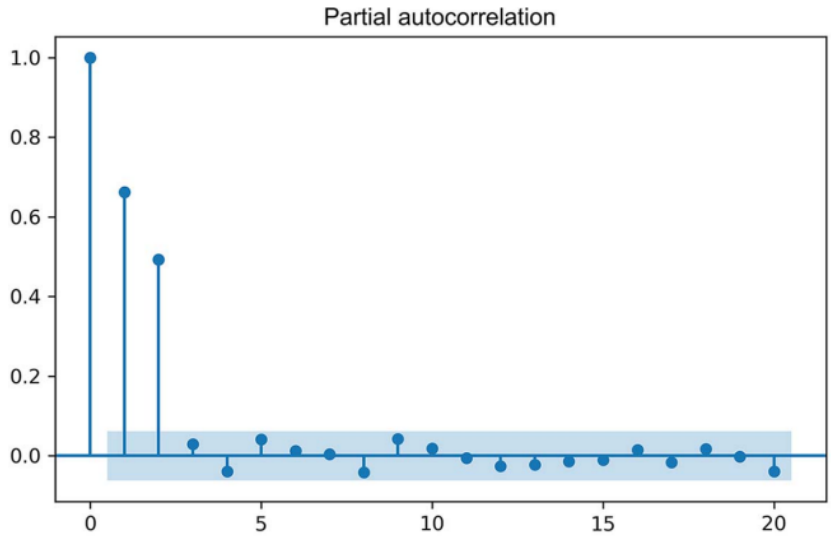

Figure 29 Partial Autocorrelation Function (PACF) for 20 lags [44, p. 91]

The Partial Autocorrelation Function (PACF) plot determines the correct order *p* of a stationary AR model. Like the ACF plot, the coefficients of the PACF plot will become insignificant after lag *p*. In Figure 29, the order of the AR process is 2. No lags are significant after lag 2. In an AR(2) model, understanding how the y_{t-2} value influences y_t is crucial. The ACF measures their correlation. Notably, y_{t-1} also influences y_t. Therefore, the autocorrelation does not measure the true effect of y_{t-2} on y_t. The PACF removes the effect of y_{t-1}. It measures the correlation between lagged values after the influence of correlated lagged values in between is removed.

Furthermore, the Auto Regressive model can be expressed similarly to the moving average model, where the linear equation refers to the present and past values of the time series rather than the present and past error terms. The MA process can very often be expressed as an infinite AR process. Likewise, AR processes can be considered as MA processes of infinite order [43, p. 181]. Referring to the diagram in Figure 26, we must decide how to proceed if neither the ACF's nor the PACF's autocorrelation coefficients become non-significant after a certain number of lags. In those cases, the data cannot be modeled using a pure Moving Average or an Auto Regressive model. If we are dealing with a stationary time series and both the ACF and PACF exhibit slowly decaying or sinusoidal patterns, then the time series is a stationary Auto Regressive Moving Average ARMA(*p,q*) process. The ARMA(*p,q*) process combines the Auto Regressive and the

moving average processes, where *p* is the order of the Auto Regressive part, and *q* is the order of the moving average part. See Equation 13 for the general model of the ARMA(*p,q*) process.

$$y_t = C + \varphi_1 y_{t-1} + \varphi_2 y_{t-2} + \cdots + \varphi_p y_{t-p} + \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q} \quad (13)$$

where $C + \varphi_1 y_{t-1} + \varphi_2 y_{t-2} + \cdots + \varphi_p y_{t-p}$ denotes the Auto Regressive portion,
and $\mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_q \epsilon_{t-q}$ denotes the Moving Average portion.

An ARMA(0,*q*) process is equivalent to an MA(*q*) process, whereas an ARMA(*p*,0) process is equivalent to an AR(*p*) [44, p. 105].

Using the ACF and PACF plots to choose the appropriate values for *p* and *q* becomes difficult as both plots display slowly decaying or sinusoidal patterns, so a quality metric, the Akaike Information Criterion (AIC), will guide the model selection process. The AIC measures the quality of a model by quantifying the relative amount of information loss when the data is fitted to the model. The lower the AIC, the lower the information loss, and the better the model. Additionally, the metric penalizes many parameters used by the model. See Equation 14 for the function of the AIC information loss criterion [44, p. 113].

$$AIC = 2k - 2\ln(\check{L}) \quad (14)$$

where *k* is the number of parameters of the model,
and $\check{L}$ is the maximum value of the likelihood function.

As previously mentioned, differencing can be performed to remove the trend component from a time series to make it stationary. It is easy to account for this transformation step in the ARMA model by including the number of times the time series must be differentiated to produce stationarity. The resulting model is the Auto Regressive Integrated Moving Average (ARIMA) model. The ARIMA model is specified by the parameters (*p,d,q*), where *d* defines the number of

times the time series must be differenced. Generally, all three parameters should not be too high to keep the model simple and prevent overfitting the data [43, p. 189].

While iterative processes, like the Box-Jenkins method, exist to fit the model manually, automated methods, such as *auto_arima()* from the Python *pmdarima* library [73], should be preferred for identifying the best-fitting parameter values.

If the data includes seasonal patterns, ARIMA can be extended to the Seasonal Auto Regressive Integrated Moving Average model (SARIMA), while incorporating multiplicative seasonality. It can be expressed as ARIMA(*p,d,q*)x(*P,D,Q*)*m*, where *m* specifies the number of time steps per seasonal cycle [43, p. 203].

5.1.2 FB Prophet

In 2017, the data science team at Facebook developed the Prophet algorithm [56], available open source in Python and R. It is a scalable, largely automized, fast forecasting algorithm that proposes strong default hyperparameters. It was one of the first time series forecasting libraries to support the democratization of time series forecasting approaches. It opened the field to a wide user group, including forecasting experts, practitioners, and domain experts with extensive expertise in their field but little to no knowledge about the mathematics and algorithms behind forecasting approaches. The authors claimed that few analysts can create highly accurate forecasts. As a result, the demand for high-quality forecasts is much higher than the amount that can be produced, leading to the conclusion that highly automated forecast libraries are needed to cover this gap [56]. It provides full automation for beginners and still supports fine-tuning and hyperparameter tuning capabilities for skilled experts. According to the authors, automated hyperparameter tuning works for many forecasting problems, and they claimed that the main problems in big-scale time series forecasts are not computational or infrastructural but the complexity and variety of forecasting problems and creating trust in the generated forecasts [56].

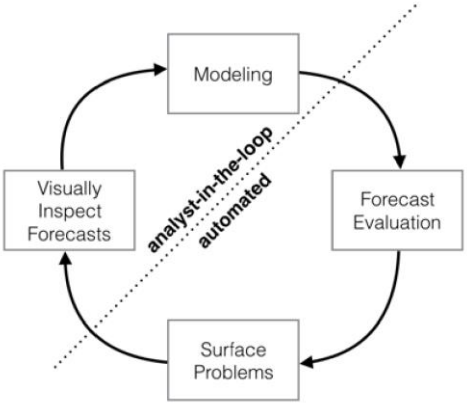

Figure 30 Analyst-in-the-loop by Facebook Prophet [56]

The "analyst-in-the-loop" approach emphasizes the separation of automated tasks, including forecast evaluation, and the handling of easy surface problems using hyperparameter tuning, leaving the task of bringing domain knowledge into the modeling up to the user (see Figure 30). Possible domain knowledge that users can bring in are capacities and external data on market sizes, specific points when the trends change, so-called changepoints, information about holidays and expected seasonality, and smoothing parameters influencing the calculation.

Prophet's internals are based on Bayesian inference, where a Markov Chain Monte Carlo algorithm samples from the posterior distribution of the model parameters. This allows for the generation of probabilistic forecasts instead of only point estimates [56]. Prophet builds on a decomposable, additive time series model based on [74] that includes trends, seasonality, and holidays (see Equation 15).

$$y(t) = g(t) + s(t) + h(t) + \in_t \qquad (15)$$

where *g(t)* is the trend,

s(t) is the seasonality, representing periodic changes,

and *h(t)* represents the effects of holidays and special events.

The error term $\in_t$ models all deviations that cannot be explained using the model. The generalized additive model [75] frames the time series forecasting problem as a curve fitting

problem and offers several advantages, such as easy decomposition, flexibility in accommodating seasonality with multiple periods, very fast fitting, easily interpretable parameters, and no requirement for regularly spaced measurements that reduce the need for imputation of missing values [56]. Prophet implements two trend models: a saturating growth model and a piecewise linear model. The non-linear, saturating growth model assumes a non-linear growth that saturates up to a carrying capacity. For example, the number of users of a certain app in a particular area is restricted by the number of people with Internet access. For saturating growth, the basic logistic growth model is used (see Equation 16) [56].

$$g(t) = \frac{C}{1+\exp\left(-k(t-m)\right)} \qquad (16)$$

where C is the carrying capacity
k is the growth rate
m is the offset parameter.

Because the capacity is not fixed, a time-varying capacity *C(t)* is replaced in the final model. The growth rate is not constant, and changes in the trend are modeled using defined changepoints. A vector of rate adjustments is incorporated into the model, leading to a piecewise logistic growth model. Prophet also includes an automatic changepoint selection given a set of candidates, but the number of changepoints must be chosen carefully to prevent overfitting. The seasonality component is modeled using the standard Fourier Series. Holidays and events are handled separately as they do not follow periodic patterns. Prophet allows for country-specific holiday lists and custom lists of past and future events. The effects of holidays are considered independent, and each holiday is assigned a parameter corresponding to the forecast change [56].

5.2 State Space Models

At their core, state space models are quite similar to classical statistical models. However, they address a few real-world concerns common to engineering problems, such as measurement error and how to incorporate prior knowledge into the estimates. They assume that you can never truly

measure the true value of a system but that it is possible to infer it using previously known information. State space models can be applied to deterministic or stochastic data in continuous and discrete cases. They allow for time-varying and changing coefficients and parameters, which allows the creation of models that gradually change their behavior. The stationarity of the input time series is not a requirement. All these characteristics make them a very flexible and versatile approach for time series forecasting. State space models estimate the underlying state based on the measured observations. The iterative process can be divided into three stages: Filtering, Forecasting, and Smoothing. Filtering uses the measurement at time *t* to update the internal state estimation at time *t*. It is a method of weighing the most recent measurements against past information. Forecasting uses the measurement at time *t-1* to predict the expected state at time *t*. It is the prediction of future states without further information about the future. This allows the expected measurement at time *t* to be inferred as well. Smoothing uses a range of measurements, which includes *t*, to estimate the true state at time *t*. It uses future and past information to estimate the state at a given point [43].

5.2.1 Kalman Filters and Their Extensions

The Kalman Filter is a well-established and popular method for predicting a system's past, present, and future states. It was developed in 1960 by R.E. Kalman at the Research Institute for Advanced Study, Baltimore [51]. It can smooth the data by modeling the time series as a combination of known dynamics and measurement error. Like every state space model, it incorporates new observations and previously known information to estimate the underlying system state. It does so by keeping only a smoothed value of the past and not all individual values, thus making it very memory efficient. This is one of the main reasons why one of its first uses was on the Apollo 11 mission for trajectory calculations, as the onboard computers could not handle more memory-intensive methods [43, p. 210].

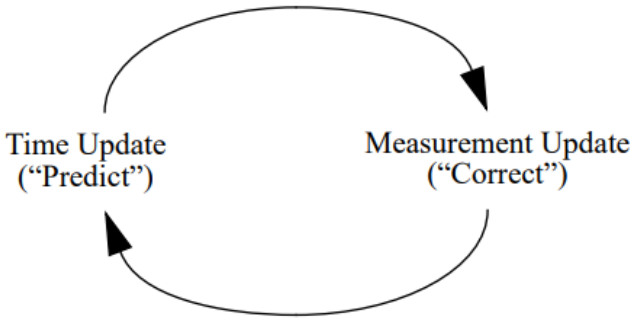

Figure 31 Discrete Kalman Filter cycle [76]

Figure 31 illustrates the iterative feedback control of the discrete Kalman Filter. The filter ('Time update') predicts a process state and receives feedback as a measurement update. The time update equations project the current state and error covariance to get a priori estimates for the next step. Then, the measurement update incorporates a new measurement to obtain an improved posterior estimate [76]. To apply the Kalman Filter, we start with a Linear Gaussian model with the initial state and observations satisfying the following two equations [43, p. 210]:

$$x_t = F * x_{t-1} + B * u_t + w_t \qquad (17)$$

$$y_t = A * x_t + v_t \qquad (18)$$

where x_t is the actual state at time t,

A, F, and B are all constant values,

$F * x_{t-1}$ is the state at the previous time step,

$B * u_t$ is the external force term,

w_t is the random error term,

y_t is the measured value that requires correction,

v_t is the measurement error.

If the problem follows these dynamics, the Kalman Filter can be applied. In the case of a time series forecast, y_t is the estimated future state, and x_t would be the real values that will be used

to update the system dynamics as soon as they become available. The iterative steps for the prediction and updating phase in the Kalman Filter are defined as [43, p. 210]:

Prediction:

$$\check{x}^{-}{}_{t} = F * \check{x}_{t-1} + B * u_t \qquad (19)$$

$$P^{-}{}_{t} = F * P_{t-1} + F^T + Q \qquad (20)$$

Filtering:

$$\check{x}_t = \check{x}^{-}{}_{t} + K_t * (y_t - A * \check{x}^{-}{}_{t}) \qquad (21)$$

$$P_t = (1 - K_t * A) * P^{-}{}_{t} \qquad (22)$$

where $\check{x}_t$ is the term to be estimated at timestamp t and $\check{x}^{-}{}_{t}$ the corresponding predicted term before it is updated,

P_t is the error covariance of the state (depending on a univariate or multivariate state, it is either a scalar or a matrix,

$P^{-}{}_{t}$ is the estimate for timestamp t before the actual measure for t is considered,

R is the measurement error variance (variance of v_t),

Q is the stochastic error variance (variance of w_t),

y_t is the measured value.

The calculation of the Kalman Gain K_t is

$$K_t = P^{-}{}_{t} * A^T * (A * P^{-}{}_{t} * A^T + R)^{-1} \qquad (23)$$

To start the process, estimates or known values for *R*, *Q*, the covariance matrices for measurement error and state stochasticity, and the state at time 0 are needed. Moreover, knowledge of the applied forces at time *t* is needed to determine *B* and the value u_t as well as knowledge of *A* and *F.*

One disadvantage of Kalman Filters is the number of hyperparameters that must be tuned. The high number of parameters and the associated fine-tuning may also lead to overfitting, especially when data is sparse [43, p. 210].

While the original Kalman Filters were designed to be applied to linear systems, the concept of applying the Kalman Filter to nonlinear systems by linearizing them has been proposed in [52]. This extension is the fundamental idea behind the Extended Kalman Filter. This approach allowed the Kalman Filter to be used in a broader range of applications. Several improvements to the extended Kalman filter have been proposed since, such as unscented transformations, which avoid the need for linearization by parameterizing the mean and covariance of the state distribution using a set of carefully chosen sample points [53].

The Ensemble Kalman Filter [54] differs from the original Kalman Filter and the Extended Kalman Filter by avoiding the need for linearization of system dynamics. Instead of linearizing, the Ensemble Kalman Filter uses an ensemble of system states to approximate the probability distribution to naturally handle nonlinearities. It uses a Monte Carlo method to propagate the ensemble members, making it capable of managing non-Gaussian error distributions. It is well suited for high-dimensional state spaces, as it requires only the computation and storage of an ensemble rather than a full covariance matrix, making it feasible for large-scale applications like weather forecasting.

In the context of Kalman Filters, it can be helpful to take a step back and consider the broader picture. Bayesian filters provide a general framework for estimating the state of a system using probabilistic methods [53]. They utilize Bayes' Theorem to update the probability distribution of the system's state as new measurements become available. This approach enables the inclusion of uncertainty and non-Gaussian noise in the estimation process. Bayesian filters generate probabilistic forecasts by continuously updating the probability distribution of the system's state. This updating process involves a prediction step, which moves the current state estimate forward in time using the system's dynamics, and an update step, which adjusts the state estimate based on new observations. By maintaining and updating a probability distribution, Bayesian filters offer a robust method for quantifying and managing uncertainty in state estimation.

5.3 Deep Learning Approaches for Time Series Analysis

Traditional time series forecasting models, such as AR models, have long been the go-to for many practitioners. However, they have inherent limitations. These models are designed to fit each time series individually, which can be time-consuming and inefficient for large-scale tasks. Moreover, they often require the manual selection of components like trends and seasonality, which can be challenging for practitioners without deep expertise. These drawbacks have created a demand for more advanced and automated approaches to time series forecasting.

Deep neural networks (DNNs) have been proposed as a viable alternative to address the limitations of traditional time series forecasting models. Recurrent Neural Networks (RNNs) are used to model time series autoregressively. Modern machine learning methods offer a way to understand temporal dynamics through a completely data-driven approach. Recent data availability and computational power advances have made machine learning an essential component in developing the next generation of time-series forecasting models. Deep neural networks learn relationships by employing several nonlinear layers to build intermediate feature representations. In the context of time series analysis, this process can be seen as encoding past information into a latent variable, z_t. The final forecast is then generated by using only this latent variable z_t.

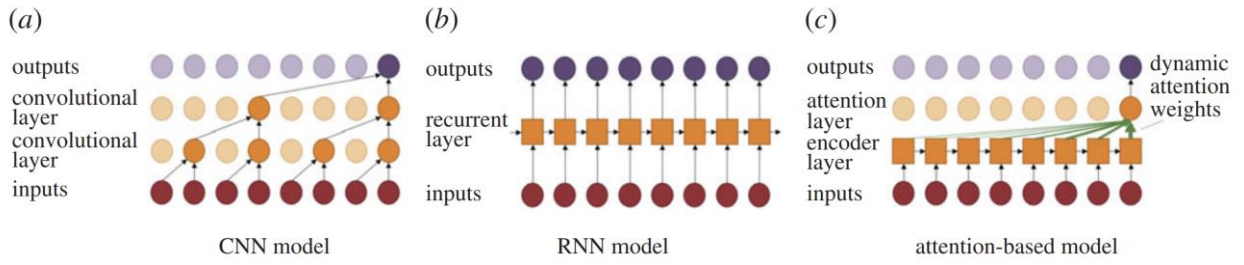

Figure 32 Different encoding architectures for time series modeling using DNN [77]

Figure 32 depicts three common deep learning architectures and the appropriate encoding approach. Temporal Convolutional Neural Networks (CNNs) operate under the assumption that relationships in the data are time-invariant, meaning they use the same filter weights consistently at each time step and across all time points. CNNs are limited to using inputs only within their

predefined lookback window, also known as the receptive field, for making forecasts. Consequently, it becomes crucial to carefully tune the size of the receptive field to ensure that the model effectively utilizes all relevant historical information. A single CNN layer can be considered equivalent to an Auto Regressive model [77].

Recurrent Neural Networks were designed for sequence modeling. They have a long history of applications in various natural language processing tasks. They are suitable for time-series data since time series can be seen as sequences of inputs and outputs. Central to the RNN architecture is an internal memory state within each cell, serving as a condensed historical data record. This memory state is continuously and iteratively updated with fresh observations at every time step. In that respect, they are like Kalman Filters, as they maintain a hidden state that is recursively updated over time. Due to their infinite lookback capability, RNNs often encounter difficulties learning long-range dependencies within data. They are known to be challenging to train due to the vanishing and exploding gradient problem, among others.

5.3.1 LSTM Neural Networks

The Long Short-Term Memory (LSTM) [57] is an improved version of RNNs that adds a cell state to solve the vanishing gradient problem that is a common problem with such networks. It occurs when gradients during training become extremely small as they backpropagate through many time steps, preventing adequate weight updates and making it difficult for RNNs to learn long-term dependencies. This usually leads to ineffective training and poor network convergence [78]. The added internal cell state allows the LSTM to retain a memory of past states carried via the cell state and control whether new information will be kept in the system's memory or discarded.

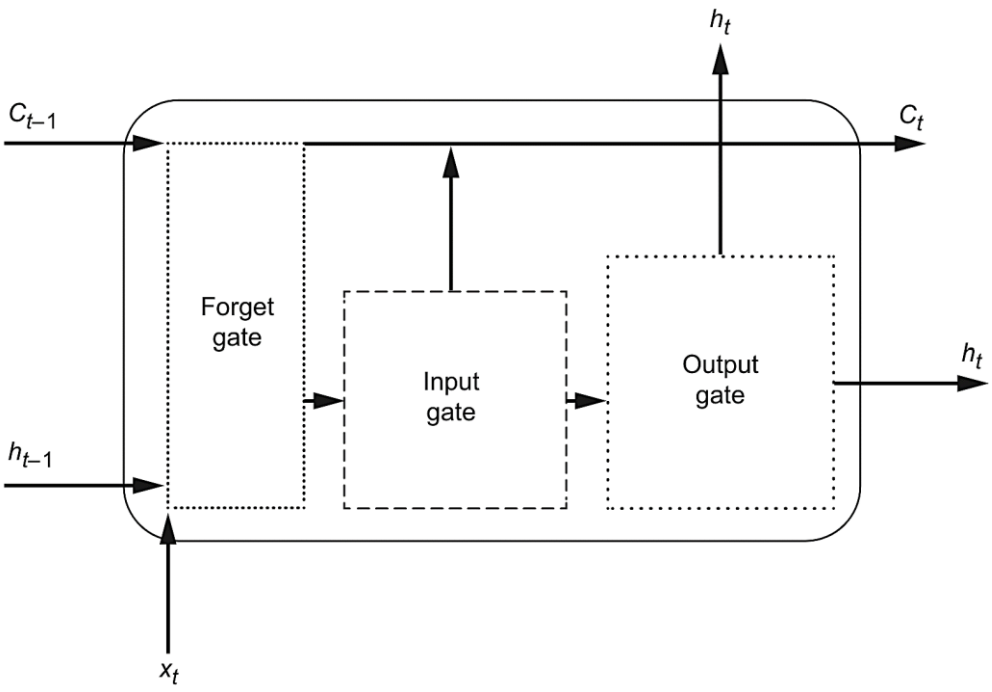

Figure 33 High-level overview of an LSTM cell [44, p. 290]

Figure 33 visualizes the high-level overview of a typical LSTM cell, where C_t is the internal cell state at time *t*, which allows the network to keep past information, x_t is the input unit for timestep *t* being processed, and h_t is the computed hidden state of the cell. This component is known from classical RNN architectures. The main building blocks of the architecture are the three gates, the forget gate, the input gate, and the output gate.

The forget gate (see Figure 34) decides what information from the past should be kept in the system. It combines the hidden state of the previous timestep with the new input data. It determines via a sigmoid function that outputs a value between 0 and 1 whether to keep the information in the system memory state *C* (see Equation 24) [79]. 0 would indicate forgetting the information, whereas 1 would indicate keeping the given information.

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f) \tag{24}$$

where σ is the sigmoid function,

W_f is the weight matrix of the gate,

h_{t-1} is the output of the previous time step,

x_t is the current input for time t,

and b_f is the bias for the gate.

The combination of the input values is forwarded to the input gate.

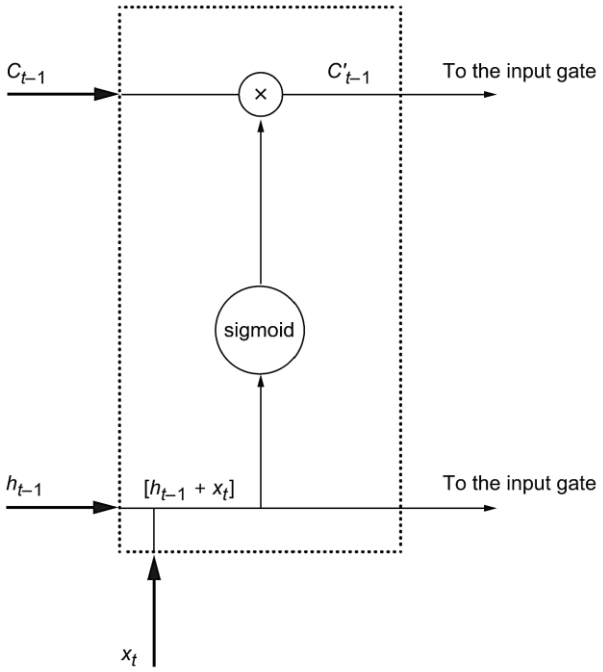

Figure 34 Forget gate of the LSTM cell [44, p. 291]

The input gate (see Figure 35) performs two tasks. It determines which values will be updated in the cell state, whereas the candidate cell state C_t represents a filtered version of the input data, prepared to be potentially added to the actual cell state (see Equations 25 and 26) [79].

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i) \tag{25}$$

$$\tilde{C_t} = \tanh(W_c.[h_{t-1}, x_t] + b_C) \tag{26}$$

where σ is the sigmoid function,

tanh is the hyperbolic tangent function that puts a value between -1 and 1 to regulate the network,

W_i is the weight matrix for the input gate,

W_C is the weight matrix for the candidate cell state,

b_i is the bias for the input gate,

and b_C is the bias for the candidate cell state.

The cell state update is the element-wise addition of the old state multiplied by the forget gate and the candidate state multiplied by the input gate (see Equation 27) [79].

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \qquad (27)$$

where $\tilde{C}_t$ is the candidate cell state,

C_{t-1} the previous cell state,

f_t the forget gate output,

and i_t the input gate output.

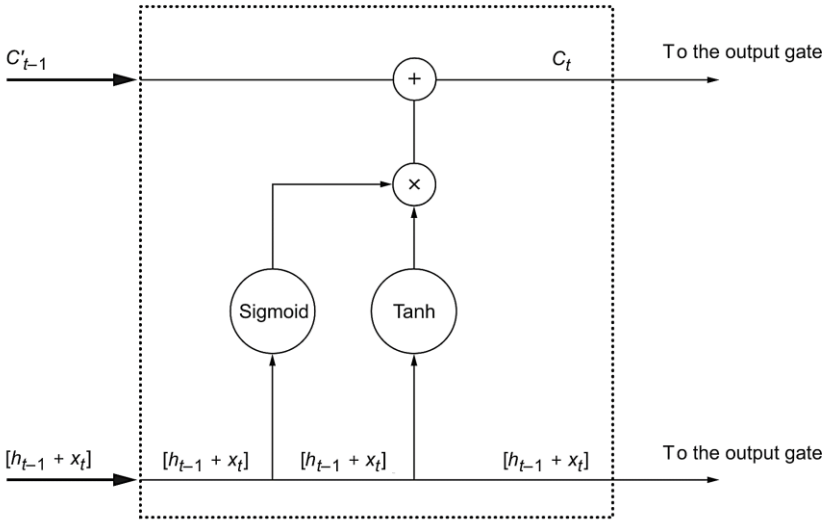

Figure 35 Input gate of the LSTM cell [44, p. 293]

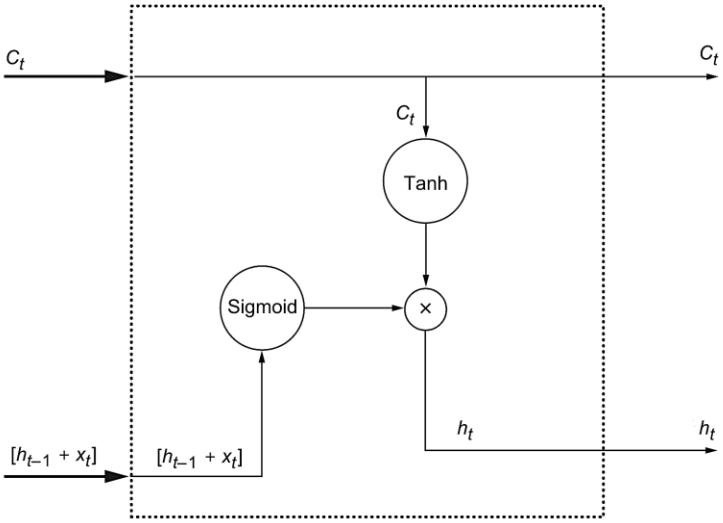

Figure 36 Output gate of the LSTM cell [44, p. 294]

In the final step, the output gate combines the previous hidden state and the new input data into a sigmoid function and multiplies it pointwise with the current hidden state, C_t. The result comprises the new hidden state, which will be passed on to the next network processing unit, possibly another LSTM cell, or the output layer [44, p. 290] (see Equations 28 and 29).

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o) \quad (28)$$

$$h_t = o_t * \tanh(C_t) \quad (29)$$

where σ is the sigmoid function,

tanh is the hyperbolic tangent function scaling the cell state,

W_o is the weight matrix,

and b_o is the bias.

All these architectural concepts make LSTMs powerful architectures for time series forecasting, addressing the handling of long sequences like recurrent neural networks while dealing with the

vanishing gradient problem, which is key in the training of the latter. Nevertheless, the training difficulties regarding the vanishing and exploding gradient problem sometimes remain. Studies have shown that while LSTM-based language models can handle an average context size of approximately 200 tokens, they effectively differentiate only approximately 50 nearby tokens, indicating a struggle with capturing long-term dependencies [46]. This is particularly relevant in real-world forecasting situations, which often exhibit long- and short-term repeating patterns.

5.3.2 Neural Prophet

In 2021, Neural Prophet [58] was proposed as the successor to Facebook Prophet. It followed the main characteristics of its predecessor in being user-friendly, supporting the democratization of forecasting libraries, and supporting interpretable results while solving the main problems of FB Prophet, including the lack of local context, which is an essential requirement for accurate short-term forecasting. Like its predecessor, it preselects very accurate default settings that lead to good out-of-the-box results while supporting customization options for domain experts. The initial version of Prophet was based on the probabilistic programming language Stan [80], which made extension and customization of the code very difficult. Neural Prophet is based on *PyTorch*, which makes it easily extendable. It supports the definition of custom loss functions as well as evaluation metrics. It supports the custom configuration of the non-linear layers of the internal feed-forward neural network, and the *PyTorch* Gradient Descent optimization engine makes the modeling process much faster. The authors labelled Neural Prophet a hybrid model, as it supports the classical statistical approach of its predecessor, new neural network components, and its autoregression and covariate modules to fit non-linear dynamics [58]. The Neural Prophet model comprises modular additive components with their individual inputs, processes, and configuration options, allowing the resulting model to be composed of any combination. The only precondition is that each model produces *h* outputs, where *h* defines the forecasting horizon, the number of steps forecasted into the future in a single step. The results are added up to the predicted values $\hat{y}_t, \dots, \hat{y}_{t+h-1}$. The additive model components are defined in Equation 30 [58]:

$$\hat{y}_t = T(t) + S(t) + E(t) + F(t) + A(t) + L(t) \qquad (30)$$

where $T(t)$ is the trend as time t,

$S(t)$ are the seasonal effects at time t,

$E(t)$are event and holiday effects at time t,

$F(t)$ are regression effects at time t for future-known exogenous variables,

$A(t)$are Auto Regression effects at time t based on past observations,

$L(t)$ are Regression effects at time t for lagged observations of exogenous variables.

Neural Prophet models the trend component using a classic approach, where the trend at time t_1 is defined by multiplying the growth rate by the time delta and adding an offset *m* (see Equation 31) [58]:

$$T(t_1) = T(t_0) + k * \Delta_t = m + k * (t_1 - t_0) \qquad (31)$$

The model changes the growth rate at several locations, resulting in an interpretable yet non-linear piecewise trend modeling. The number of changepoints for the trend is finite and is defined as the set of *C* of n_c changepoints $C = (c_1, c_2, \ldots, c_{n_c})$ where the trend is kept constant between the changepoints. It handles the seasonality using Fourier terms just like its predecessor Prophet does [56]. According to Equation 32, several Fourier terms are defined as sine and cosine pairs. This allows for multiple, overlapping seasonalities and non-integer periodicities, including a yearly seasonality with weekly data resulting in *p=52,18*.

$$S_p(t) = \sum_{j=1}^{k} (a_{j*cos}\left(\frac{2\pi jt}{p}\right) + b_j * \sin\left(\frac{2\pi jt}{p}\right)) \qquad (32)$$

A higher number of Fourier terms can model more complex patterns, but a too large number may lead to overfitting. Further, Fourier terms are restricted to modeling deterministic seasonal shapes that are fixed over time. The Neural Prophet model can automatically activate the required seasonality, such as daily, weekly, and yearly seasonality. It considers the data's granularity. The respective seasonalities are activated if the data resolution is higher. If the data is available in daily granularity, the yearly seasonality will be enabled if the data spans at least two

years. The weekly seasonality will be enabled if the data spans at least two weeks. The daily seasonality will not be activated as the granularity does not allow for intra-day seasonality. The autoregressive part of the model supports three different configuration modes. In the *LinearAR* mode, the AR-Net component of Neural Prophet is identical to the classical AR model. The AR order defines the number of past values used for regression. In the *DeepAR* mode, hidden layers are used in a fully connected neural network to model non-linear dynamics. This is, however, a trade-off between interpretability and forecasting accuracy, as the contribution of a particular past observation can only be observed as having relative importance. *SparseAR* approximates the correct AR order by setting it slightly larger than expected. Regularization is used to make model weights sparser. This makes model configuration more convenient. The concept of lagged regressors allows the correlation of other observed variables, the covariates, to the target variable. Lagged regressors' values are known for past periods and are unknown for future values, whereas future regressors are known for future periods. Like Prophet, Neural Prophets can model the effects of special events and holidays. The architecture enables users to define individual special events and use predefined country-specific holidays. Neural Prophet allows for very convenient and fast data preparation steps before training. It supports automated data normalization, based on either a MinMax-Scaler, two MinMax-Scaler versions working with quantiles, or a Standard Scaler, as well as a three-staged automated data imputation for missing data points that support bidirectional linear regression and a centered rolling average. For training, the default loss function is the Huber loss, also known as smooth L1-loss (see Equation 33), but individual custom loss functions can be defined using the *PyTorch* framework.

$$L_{huber}(y, \hat{y}) = \begin{cases} \frac{1}{2\beta}(y - \hat{y})^2, & for\ |y - \hat{y}| < \beta \\ |y - \hat{y}| - \frac{\beta}{2}, & otherwise \end{cases} \quad (33)$$

Here, β is equivalent to the Mean Squared Error. For values larger than β, it is equivalent to the Mean Absolute Error. Experiments using artificial and real-world datasets demonstrate that Neural Prophet improves the forecast accuracy by 55% - 92% for short-term to medium-term forecasts, whereas Neural Prophet performs identically or slightly better than Prophet for

datasets without lagged regressors, it performs significantly better than Prophet when lagged regressors are involved [58].

5.3.3 Time Mixer

The Time Mixer model is a unique multiscale mixing architecture proposed in May 2024 [64]. It decomposes time series data into different scales to capture fine-grained, microscopic, and broad, macroscopic patterns. The multiscale mixing approach assumes that time series data hold different information at different scales, which is helpful as it can help disentangle the variations in the multiple components of the input sequence. Time Mixer is a fully Multi-Layer Perceptron-based architecture that combines Past-Decomposable-Mixing to model a time series' seasonal and trend components and Future-Multipredictor-Mixing that aggregates predictions from different scales to utilize complementary forecasting capabilities. The resulting model achieves state-of-the-art results across a wide range of benchmarks, including both long-term and short-term forecasting tasks.

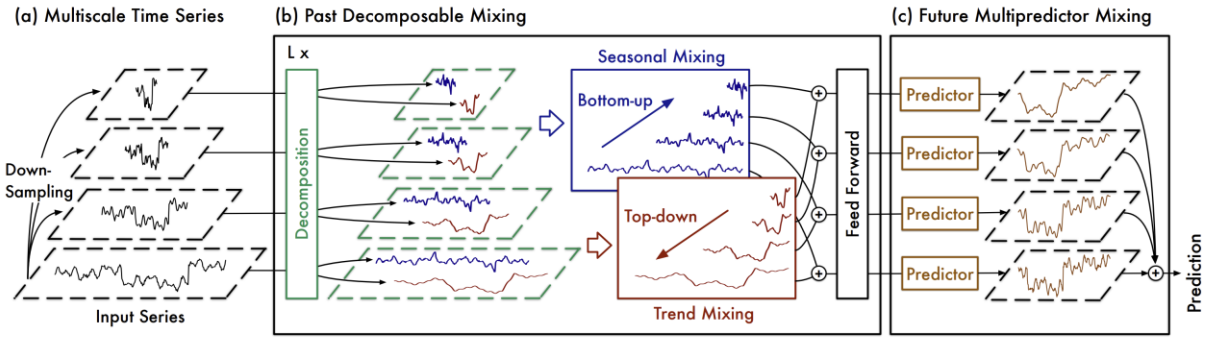

Figure 37 Time Mixer Overall Architecture [64]

Figure 37 depicts the overall architecture of the Time Mixer model. The first processing step is downsampling the input sequences to disentangle complex variations. The Past-Decomposable-Mixing Component decomposes the input into seasonal and trend parts before the Seasonal Mixing combines information from the low-level, fine-scaled time series upwards in a bottom-up manner. Trend mixing, on the other hand, is modeled as a top-down approach that aims to transfer macro knowledge from bigger scales to finer scales. In the final step, the Future-

Multipredictor-Mixing, an ensemble of predictors based on different scales, creates the final model results [64].

5.4 Transformer-Based Models

With the recent success of Transformer-based architectures in natural language processing tasks, such as language translation, text summarization, text generation, or sentiment analysis, there has been growing interest in adapting Transformer architectures to create time series forecasts. Originally developed for language processing tasks, Transformers have been adapted to handle time series data, offering a more nuanced approach to analyzing patterns over various timeframes [81]. Transformers have powerful sequence modeling capabilities and the ability to prioritize important information. Introducing attention mechanisms as an essential building block in Transformers has significantly improved the learning of long-term dependencies. Attention layers aggregate temporal features using dynamically generated weights, allowing the network to directly focus on significant time steps in the past, regardless of how far back they are in the lookback window. The original Transformer architecture was proposed in 2017 by Vaswani and his team [82] and is based on an encoder-decoder structure. Unlike LSTM or RNN models, which rely on recurrent structures, the Transformer architecture does not use recurrence. Instead, it incorporates positional encoding into the input embeddings, allowing it to model the sequence information in the data effectively. The model enables access to any part of the historical data, regardless of how far back it is. This feature potentially makes the Transformer more adept at capturing recurring patterns that involve long-term dependencies [83]. The original architecture comprises multiple identical blocks in both the encoder and decoder. The following section includes an overview of the basic building blocks of the original Transformer architecture and possible architectural adaptions for time series forecasting.

5.4.1 Standard Transformer Architecture

The main components of a Transformer are the encoder and decoder. Figure 38 depicts the original Transformer architecture, including the encoder on the left and the decoder on the right. Essential building blocks are the attention mechanism, positional encoding, the fully connected

feed-forward network, layer normalization, and residual connections. While the original Transformer architecture contains encoder and decoder components, encoder- or decoder-only architectures are available. Encoder-only models, like Google BERT [84], use bidirectional attention to identify context from both directions when processing an input sequence. This makes them particularly effective for tasks related to natural language understanding. On the other hand, decoder-only models use causal attention, which enables them to look back through a sequence to find context and predict the next word. These models are well-suited for natural language generation tasks, with OpenAI's GPT models being prominent examples [85].

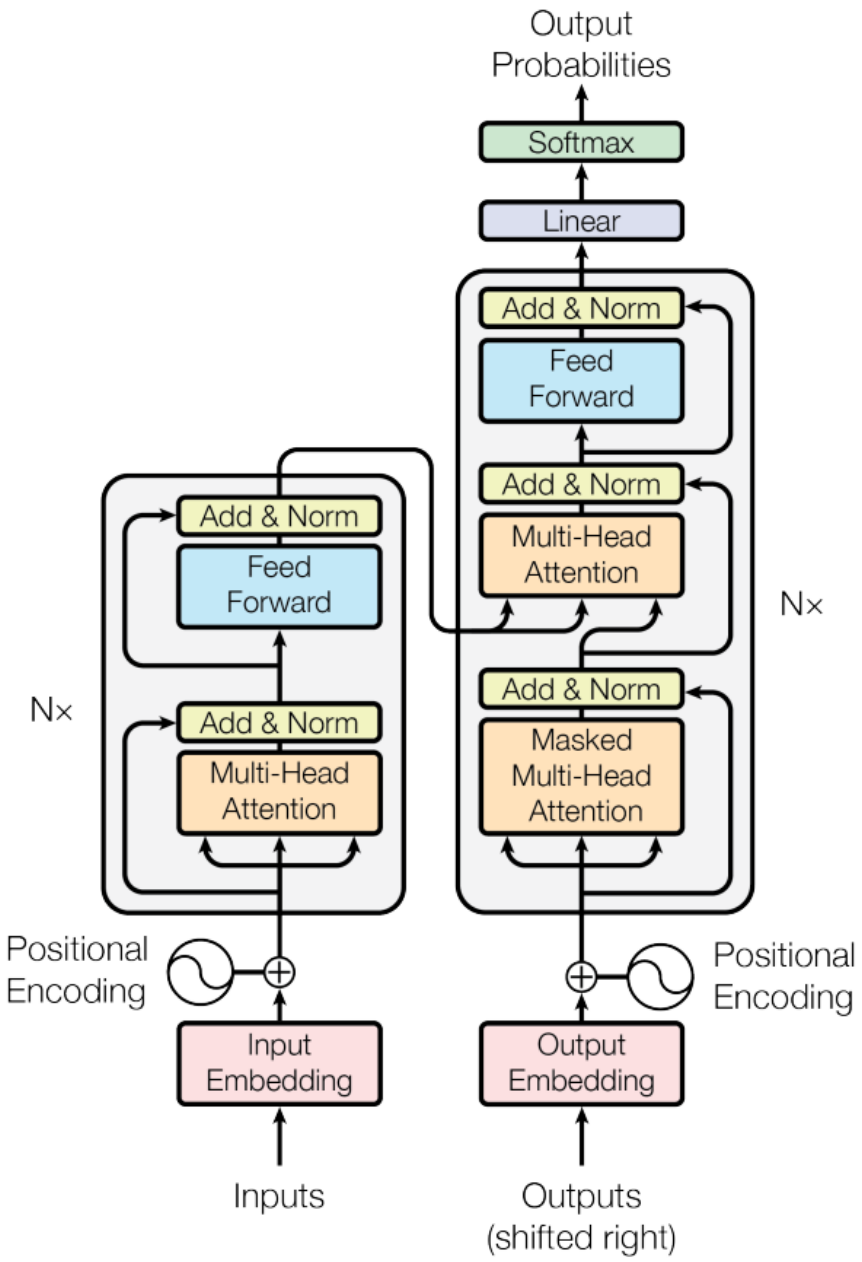

Figure 38 Original Transformer architecture, including the encoder (left side) and the decoder (right side) components [82]

5.4.1.1 Positional Encoding

In the context of encoding positional information in Transformers for time series analysis, there are three main categories [81]:

Vanilla Positional Encoding: This method, proposed in [82], involves adding standard positional encoding to the input time series embeddings before feeding them into the Transformer. While this technique captures some positional information, it is not fully optimized to exploit the unique characteristics of time series data.

Learnable Positional Encoding: To address the limitations of vanilla positional encoding, studies suggest learning positional embeddings directly from the time series data. This approach is more flexible and task-specific. The original TFT [46] employed an LSTM network to create positional embeddings, allowing a better understanding of sequential order in time series.

Timestamp Encoding: In real-world time series applications, timestamps (like seconds, minutes, holidays, and special events) provide valuable information. Traditional Transformers do not use this type of information. Architectures such as Informer [86], Autoformer [87], and FEDformer [88] propose encoding timestamps as an additional form of positional encoding using learnable embedding layers. This method makes the positional encoding more relevant and informative for real-world time series analysis.

5.4.1.2 Attention Mechanism

The attention mechanism is the main innovation in the Transformer architecture, making it so effective for sequential data. It decides how much focus is given to each word in a sequence, given every other word in the sequence. The self-attention (see Figure 38 Multi-Head Attention) component is used in the encoder and decoder parts of the network. Cross-attention (see Figure 38 Masked Multi-Head Attention) is only part of the decoder network. The decoder uses it to analyze the encoder output to focus on only on the relevant parts of the input sequence. Both self- and cross-attention calculate the scaled dot product attention. Every input X is projected into three vectors: the Query Q, the Key K, and the Value V using learned weight matrices W_Q, W_K, and W_V (see Equations 34, 35, and 36)[89].

$$Q = XW_Q \quad (34)$$

$$K = XW_K \quad (35)$$

$$V = XW_V \quad (36)$$

The query vectors, *Q*, and the key vectors, *K*, are dotted to calculate the attention scores. The result is a matrix of attention scores representing the attention scores of every pair of elements in the input (see Equation 37)[89].

$$Attention\ Scores = QK^T \qquad (37)$$

where Q is the query matrix,
K is the key matrix,
K^T is the transpose key matrix that allows for creating the dot product with Q.

To prevent the scores from becoming too large, they are scaled using the square root of the dimension of the key vectors, d_k, to help stabilizing the gradients (see Equation 38)[89].

$$Scaled\ Scores = \frac{QK^T}{\sqrt{d_k}} \qquad (38)$$

In the next step, the Softmax function is applied to the scores to convert them into probabilities. The Softmax function ensures that the scores sum up to 1 while preserving the information about the most important words in the input sequence (see Equation 39).

$$Attention\ Weights = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) \qquad (39)$$

In the final step, the attention weights from the previous step are used to create a weighted sum of the value vectors, *V* (see Equation 40)[89].

$$Output = Attention\ Weights\ .V \qquad (40)$$

The final output is a new word representation incorporating the context of the entire input sequence. As the input sequence might be very long, Transformers use a Multi-Head Attention mechanism, where the input sequence is split into multiple smaller parts, also known as heads, and processed independently using independent weight matrices for every attention head. After the processing, the outputs of the individual heads are concatenated to create a single output vector (see Equation 41).

$$MultiHeadAttention = Concat(Head_1, Head_2, \dots, Head_h)W_O \qquad (41)$$

The self-attention module of the original Transformer functions like a fully connected layer. Its weights are dynamically created, depending on the similarity of the input patterns. This design has the same maximum path length as a fully connected layer but with far fewer parameters. This efficient structure makes the self-attention module particularly effective for modeling long-term dependencies in data. Nevertheless, the canonical dot-product self-attention in the Transformer model, which matches queries against keys in the attention mechanism without considering local context, can make the model vulnerable to anomalies and create optimization challenges.

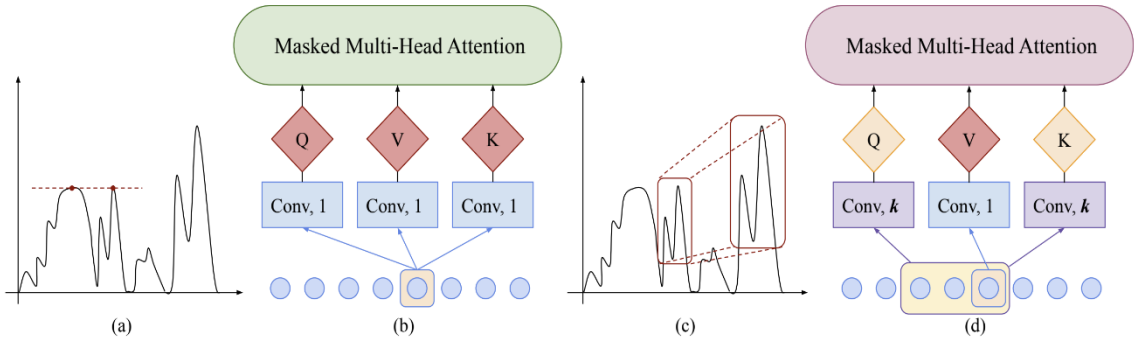

Figure 39 Comparison of canonical and convolutional self-attention layers

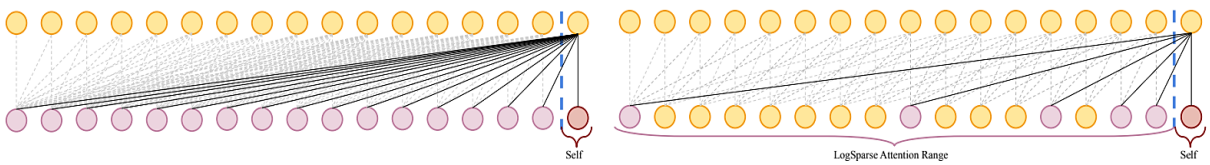

Figure 40 Log Sparse Attention

The [83] self-attention layers of canonical Transformer, where the similarities between queries and keys are computed based on point-wise values without considering local context (see Figure 39(a, b)), are being replaced by convolutional self-attention layers (see Figure 39(c, d)). Using convolutional self-attention, generated queries and keys are aware of local context and determine similarities based on local context information, including shapes. Additionally, a significant concern is the space complexity of the canonical Transformer. The self-attention module in the basic Transformer has a time and memory complexity of $O(N^2)$, where N represents the length of the input time series. This quadratic complexity becomes a computational challenge when dealing with long sequences and leads to memory bottlenecks when attempting to model long time series directly with fine granularity. Various adapted Transformer architectures have been proposed to address this problem. These models aim to reduce the quadratic complexity by introducing a sparsity bias into the attention mechanism c or exploiting the self-attention matrix's low-rank nature [86], [88]. The *LogSparse* Transformer proposed in [83] implements self-attention by allowing each cell to attend to itself and its previous cells only with an exponential step size. This approach must calculate $O(\log N)$ dot products for each cell in each layer only. Only $O(\log N)$ layers are stacked up, enabling the model to access every cell's information. As a result, the total cost of memory usage is $O(N(\log N)^2)$ (see Figure 40).

5.4.1.3 Position-Wise Feed-Forward Network

After the attention score calculation, the encoder and decoder components use a fully connected feed-forward network. The network consists of two linear transformations and uses the non-linear ReLU activation function to learn complex patterns (see Equation 42) [89].

$$FFN(x) = ReLU(xW_1 + b_1)W_2 + b_2 \qquad (42)$$

where x is the input from the attention score
W_1 and W_2 are weight matrices
and b_1 and b_2 are biases.

5.4.1.4 Layer Normalization

The Multi-Head Attention and Feed-Forward Network layers are stabilized using residual connections and layer normalization. Residual connections are added to address the vanishing gradient problem. The idea is to add a sub-layer's input to its output to preserve the original information (see Equation 43) [89].

$$Output = LayerNorm(x + Sublayer(x)) \quad (43)$$

where x is the sublayer's input.

A standard Layer Normalization is added to stabilize the training and ensure that each layer's inputs have a consistent distribution (see Equation 44) [89].

$$LayerNorm(z) = \frac{z-\mu}{\sigma+\varepsilon}\gamma + \beta \quad (44)$$

where z is the output of the residual connection (x + Sublayer(x)),
μ is the mean of the input z,
σ is the standard deviation of the input z,
ε is a small constant to prevent division by zero,
γ is a scaling parameter,
and β is a learned shifting parameter.

5.4.1.5 Encoder

Each block in the encoder contains a multi-head self-attention module and a position-wise feed-forward network. It transforms the input sequences into encoded representations. First, an input sequence *X* is encoded into a high dimensional space to capture its meaning, and a positional encoding is added to understand the position of each input element (see Equation 45). See [89] for details on the mathematics behind positional encodings in the original Transformer architecture.

$$X_{pos} = X + PE \quad (45)$$

where X is the input sequence,

and PE the positional encoding strategy.

In the next step, the Multi-Head Attention mechanism is applied to the encoded input sequence to focus on the meaningful parts (see Equation 46), and the Layer Normalization is applied afterward (see Equation 47). The result is passed through the Feed-Forward network (see Equation 48), as seen in Figure 38, and a final Layer Normalization to get the encoder output (see Equation 49) [89].

$$Z_{attn} = MultiHeadAttention(X_{pos}) \quad (46)$$

$$Z_{add\_norm\_1} = LayerNorm(X_{pos} + Z_{attn}) \quad (47)$$

$$Z_{ffn} = FFN(Z_{add\_norm\_1}) \quad (48)$$

$$Z_{encoder\_output} = LayerNorm(Z_{add_{norm_1}} + Z_{ffn}) \quad (49)$$

The $Z_{encoder\_output}$ is the final encoded representation and is fed into the decoder.

5.4.1.6 Decoder

In the decoder, each block has the same components already known from the encoder. It includes an additional cross-attention module between the multi-head self-attention module and the position-wise feed-forward network. The decoder uses masked self-attention, so the network cannot use future values during training. Also, a shifting operation is applied, to ensure that the decoder can only use tokens that have already been generated or predicted to maintain the autoregressive property. So to an original input sequence $Y=[y_1, y_2, y_3, \ldots, y_n]$, a "start of sequence"

token is added, resulting in an adapted input sequence looking like $Y_{shifted}=[<SOS>, y_1, y_2, y_3,\ldots,y_{n-1}]$. The following steps are very similar to what we already know from the encoder architecture. A positional encoding is added to the sequence. The resulting output is passed on to the masked multi-head attention (see Equation 50) and passed on to Layer Normalization (see Equation 51). The resulting sequence is then passed on to the Multi-Head attention, where it is matched to the encoder output and normalized again (see Equations 52 and 53) [89].

$$Z_{masked\_attn} = MaskedMultiHeadAttention(Y_{pos}) \tag{50}$$

$$Z_{add\_norm\_2} = LayerNorm(Y_{pos} + Z_{masked\_attn}) \tag{51}$$

$$Z_{cross\_attn} = MultiHeadAttention(Z_{add_{norm_2}}, Z_{encoder\_output}) \tag{52}$$

$$Z_{add\_norm\_3} = LayerNorm(Z_{add\_norm\_2} + Z_{cross\_attn}) \tag{53}$$

Finally, the sequence is passed on to the Feed-Forward Network and normalized (see Equations 54 and 55) [89].

$$Z_{ffn\_decoder} = FFN(Z_{add\_norm\_3}) \tag{54}$$

$$Z_{decoder\_output} = LayerNorm(Z_{add\_norm\_3} + Z_{fnn\_decoder}) \tag{55}$$

The final $Z_{decoder\_output}$ is then passed through a linear layer and a Softmax function to create the network's final output (see Figure 38).

5.4.2 Transformer Architecture Adaptations for Time Series Forecasting

Nevertheless, a recent study implies that simple linear models often outperform complex transformer-based time series forecasting based on the vanilla transformer architecture [59]. As

mentioned in [46], Transformers are not inherently unsuitable for time series forecasts. They are merely being used improperly. Therefore, either component and architecture adaptions or adaptions in the input embedding are necessary to increase the performance of Transformers based on time series forecasts. Existing, properly adapted Transformer-based architectures can be divided into four categories by their component and architecture modifications (see Figure 41). The different architectures are classified according to changes to the original transformer architecture and the individual architectural components.

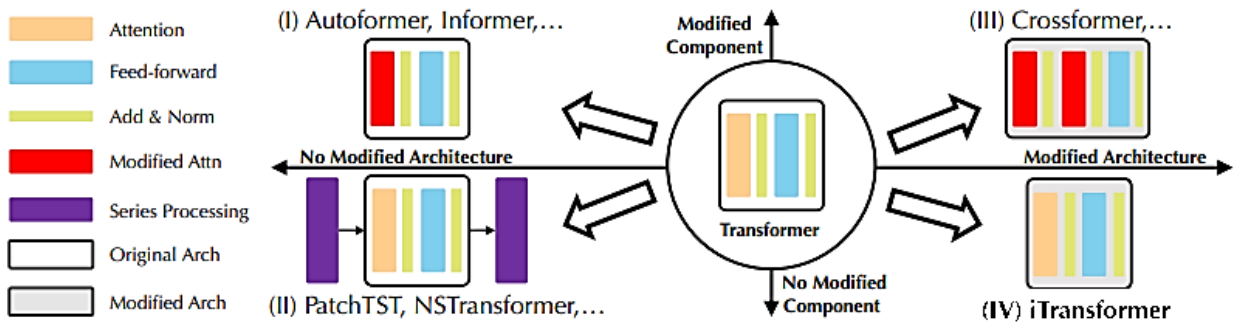

Figure 41 Transformer-based time series forecasting architecture categories [59]

The most common architectures are Autoformer [87], Informer [86], and FEDformer [88]. These architectures promote component adaptations of the original Transformer architecture, mainly in the attention module. The changes tend to improve the modeling of temporal dependency and optimization on long input sequences. PatchTST [60] and non-stationary Transformers [90] represent the second category of time series forecasters that address how time series are processed, such as channel independence, patching of input sequences, and stationarization. In the third category, the architecture and the components are adapted. The Crossformer architecture captures cross-time and cross-variate dependencies [91], whereas Temporal Fusion Transformers completely rethink the Transformer architecture, resulting in outstanding performance for point and probabilistic forecasts. iTransformer [92], as representative of the fourth category of time series transformer architectures, does not modify native Transformer components but applies the original component to the inverted input dimensions.

In the following sections, we explore the iTransformer, PatchTST, and Temporal Fusion Transformer architectures, which have been specifically adapted for time series forecasting.

5.4.3 iTransformer

Recent studies have started challenging the reliability of Transformer-based forecasters, at least for multivariate forecasting. Vanilla Transformer-based forecasters model global dependencies using temporal tokens. Each token is defined by multiple variates of the same timestamp, creating indistinguishable channels. The attention maps created on these temporal tokens, to capture temporal dependencies, could be meaningless. Given the numerical but less semantic relationship among time points, researchers have discovered that simple linear layers, like the ones in simple statistical forecasting methods, outperform complex Transformers in terms of performance and efficiency [59]. This problem is tackled by the iTransformer architecture, which takes the inverted view of input sequences and embeds the whole time series of each variate into one variate token to enable the identification of multivariate correlations (see Figure 42 [59]).

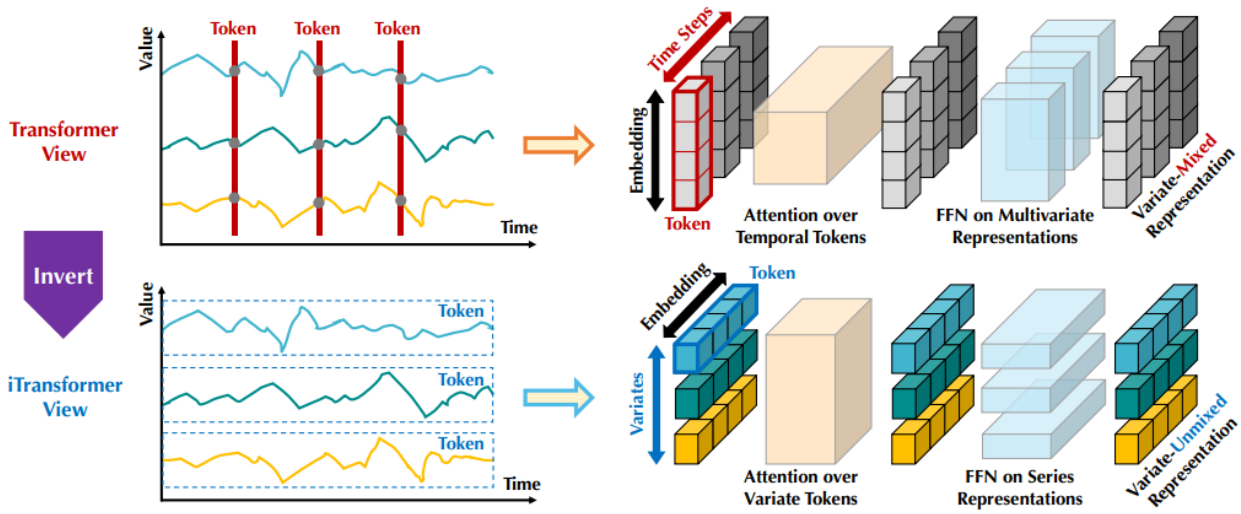

Figure 42 Comparison of temporal token creation in vanilla Transformers and iTransformer architecture

The independence of variables without the loss of their mutual information has been identified as one of the crucial aspects of creating accurate forecasts [93]. This observation has also been

the underlying motivation for the Crossformer architecture [91]. The iTransformer architecture is based on the lightweight encoder-only Transformer architecture [82] and focuses on representation learning and the correlation of multivariate series. The main building blocks are layer normalization, the feed-forward network, and the self-attention layer, see Figure 43 [59].

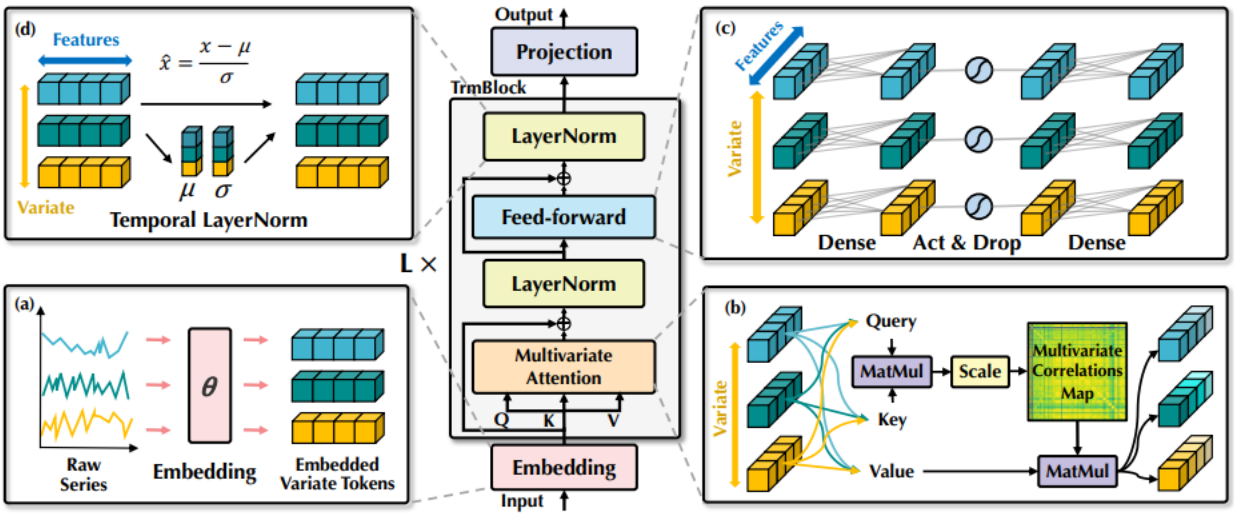

Figure 43 iTransformer Architecture [59].

The self-attention mechanism is applied to the embedded variate tokens to create multivariate correlation maps that reveal the variate-wise correlations. The module adopts linear projections to obtain the queries (Q), keys (K), and values (V) required for the attention layer. The feed-forward network handles series representation learning. It works as the basic building block for encoding the token representation. Layer normalization is applied after the multivariate attention mechanism and representation learning via the feed-forward network to normalize the multivariate representation of the individual timestamps. The token are normalized to a Gaussian distribution to minimize discrepancies caused by inconsistent measurements (see Equation 56) [59].

$$LayerNorm(H) = \left\{ \frac{h_n - Mean(h_n)}{\sqrt{Var(h_n)}} \middle| n = 1, \dots, N \right\} \qquad (56)$$

where $H = \{h_1, \dots, h_n\} \in \mathbb{R}^{N \times D}$ contains N embedded tokens of dimension D [59].

5.4.4 PatchTST

The Patch Time Series Transformer (PatchTST) was proposed in 2023 and delivered state-of-the-art results in time series forecasts [60]. The architecture supports multivariate time series forecasts using channel independence, a patching mechanism that extends the input tokens to a series of tokens to increase local semantic information, and two learning modes, supervised and self-supervised learning.

Channel independence guarantees that each input token contains information from only a single channel or time series variable. The individual channels share the Transformer backbone, but the forward processes remain independent. Figure 44 visualizes channel independence in PatchTST, with input length *L*, *M* univariate time series, and forecast length *T*. This concept resembles the inverted input transformation in the iTransformer architecture, in which the inverted input dimension guarantees that the timestamps of the individual variates are not mixed.

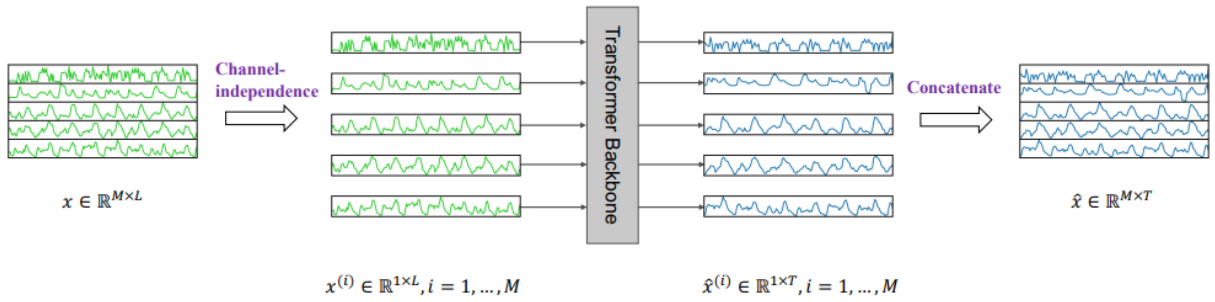

Figure 44 Channel independence in PatchTST [60]

Patching refers to replacing point-wise input tokens with patches of inputs to increase their semantic meaning. In time series forecasting, relationships between past and future time steps are important, and single time steps fail to reflect these relationships. Patches can overlap. Patching can be controlled using stride factors *S*, which work similarly to the convolution in CNNs. They define the number of timesteps between the beginnings of consecutive patches.

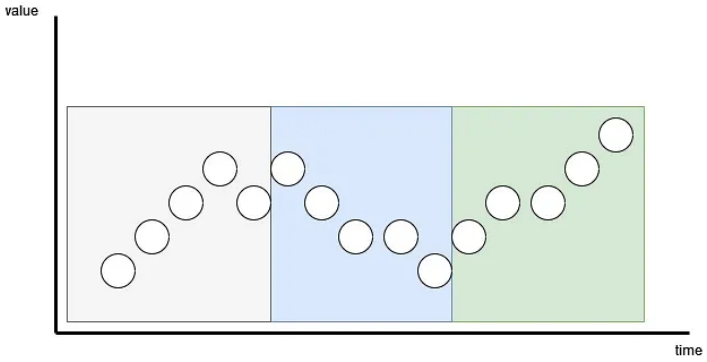

Figure 45 Patching Visualization for PatchTST

Figure 45 illustrates an example with 15 timesteps, a patch length of 5, and a stride of 5. The resulting three patches carry more semantic information than isolated, single timesteps. This design principle quadratically reduces the PatchTST model's space and time complexity. The Vanilla Transformer has a complexity of $O(N^2)$, with N being the number of input tokens. Without patching, N equals the input length L. With patching, N is reduced by the factor of stride S. This also enhances the model's capability of learning from longer lookback windows.

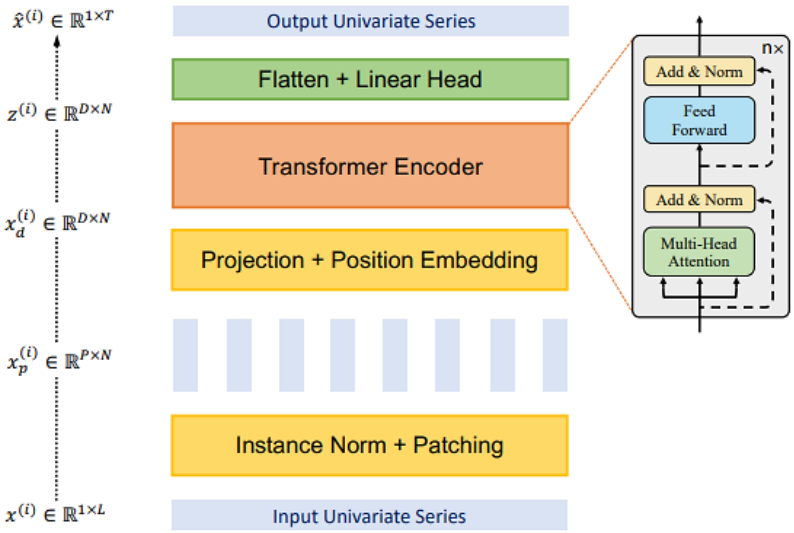

Figure 46 Patch TST Transformer architecture [60]

PatchTST also supports self-supervised representation learning to capture abstract representations. After patching, the model follows the Vanilla Transformer encoder architecture. No additional components were modified, as illustrated in Figure 46. After channel separation, every univariate channel is normalized. After patch segmentation and positional embedding, the data is fed into the transformer encoders and mapped to the latent representations. The loss function used is the Mean Squared Error [60].

5.4.5 Temporal Fusion Transformer

The Temporal Fusion Transformer (TFT) [46] is another Transformer architecture that enhances the Vanilla architecture to tailor it to the specific challenges encountered in time series analysis. According to [94], it outperforms current deep learning architectures in time series forecasting and is still considered state-of-the-art in time series forecasting. The two main reasons for its success are its ability to model attention as an interpretable mechanism and to capture long-term temporal dynamics while still using an LSTM-based decoder-encoder to model local dependencies. The architecture did not just copy the original Transformer model but adapted it successfully to the domain of time series forecasting [95]. It is structured as a multi-horizon forecasting model with static covariate encoders, a gating mechanism for feature selection, and a temporal self-attention decoder. This design enables the TFT to encode and select relevant information from various covariates for effective forecasting. Additionally, the model maintains interpretability by integrating global patterns, temporal dependencies, and event-related data into its framework. One of the advantages is the possibility of training a TFT model on thousands of univariate or multivariate time series and receiving multi-step predictions of one or more target variables as output. Those outputs can include prediction intervals. In the following sections, we investigate those characteristics more closely.

5.4.5.1 Supported Input Data Types

One of the major advantages of TFTs is their ability to handle various types of features. This is very useful because multi-step time series forecasting typically involves static, time-invariant covariates, known future inputs, and various exogenous time series that have only historical observations. This method becomes even more beneficial when there is a lack of prior knowledge

about how these elements interact with the target variable in the forecasting model [46]. Notably, four input types are supported.

Time-varying known: These temporal features are known for the past and future. Examples are holidays or special days.

Time-varying unknown: These features are only known up to prediction time. They are usually the target prediction variable.

Time-invariant real: These are exogenous static variables, such as available revenue values, yearly energy consumption values, or past periods.

Time-invariant categorical: The exogenous categorical features include values like product or customer IDs.

5.4.5.2 Handling of Heterogeneous Time Series

TFTs support training on multiple time series that can follow different distributions. To manage this, the TFT architecture divides processing into two separate parts: local and global processing. Local processing is dedicated to understanding the unique characteristics of specific events within individual time series. In contrast, the global processing component analyses the characteristics common across all the time series. This dual approach allows TFT to handle the diversity and complexity of multiple time series.

5.4.5.3 Interpretable Predictions

In recent years, there has been a shift from “black box models” to models offering greater explainability, particularly in production environments. Understanding and interpreting a model's decision-making process is highly valued. Sometimes, explainability is favored over model accuracy [47].

The Temporal Fusion Transformer offers interpretability in three distinct ways [46]:

Seasonality Interpretation: Through its unique, interpretable Multi-Head Attention mechanism, TFT assesses the significance of previous time steps, providing insights into seasonal patterns.

Feature Importance: The Variable Selection Network module in TFT determines the relevance of each feature, offering a detailed understanding of feature-wise significance.

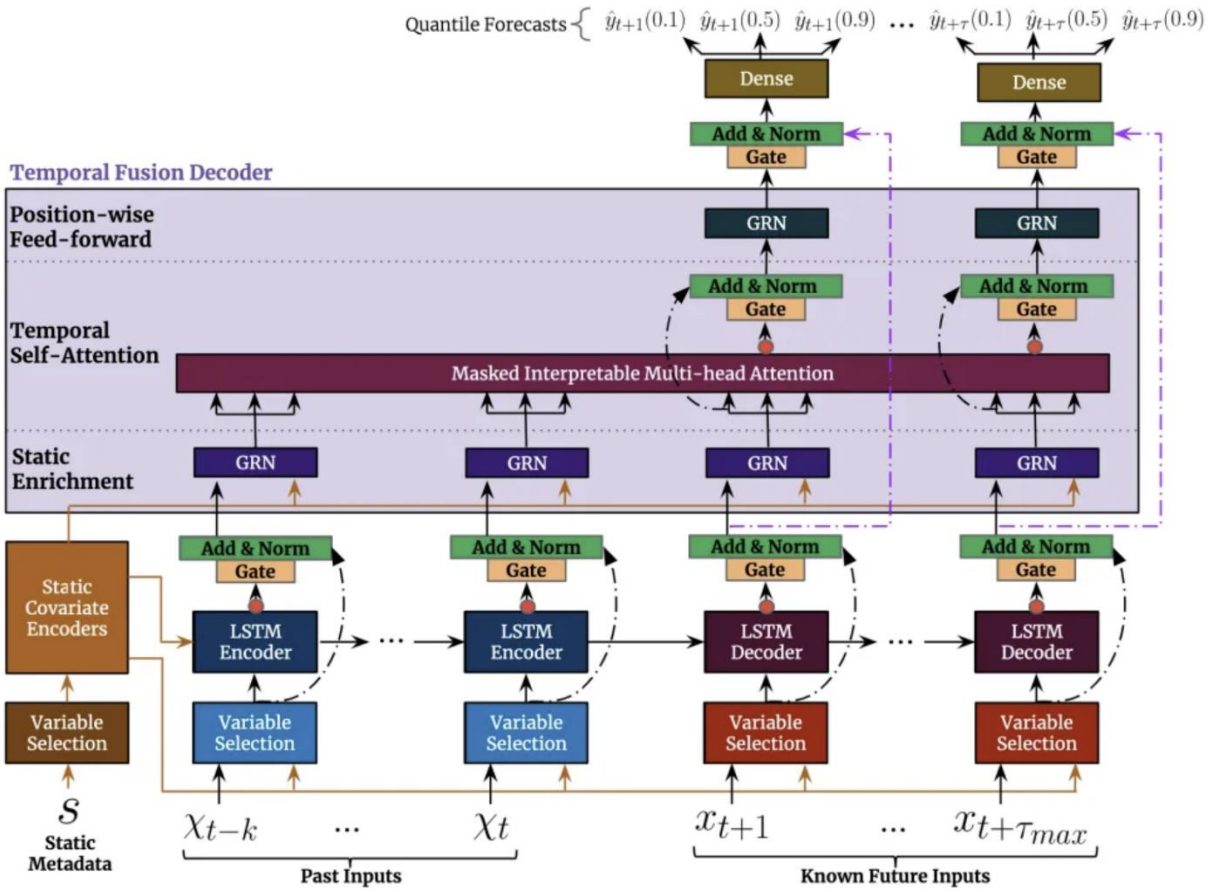

Figure 47 Temporal Fusion Transformer high level architecture [41]

Robustness to Rare Events: TFT allows the time series behavior to be analysed during rare or extreme events, demonstrating its capability to handle and interpret such occurrences.

5.4.5.4 Temporal Fusion Transformer Model Architecture

From a high-level perspective, the TFT is composed of four individual components.

Static Covariate Encoders: These encode context vectors utilized in other network parts.

Gating Mechanisms and Variable Selection: These mechanisms are employed throughout the network to minimize the impact of irrelevant inputs. They also facilitate sample-dependent variable selection.

Sequence-to-Sequence Layer: This layer processes known and observed inputs locally.

Temporal Self-Attention Decoder: This component is designed to learn and understand any long-term dependencies within the dataset.

Figure 47 depicts the high-level architecture of the temporal fusion transformer model as proposed in [46]. For a given timestep *t,* we have a lookback window of size *k* and a *τmax* step ahead window. Timestep *t* is $\in$ [*t-k .. t+τmax*]. The model uses past inputs *x* in the period [*t-k..t*] and future known inputs *x* in the period [*t+1 .. t+τmax*] and static variables *s*. The target variable *y* is predicted for the time window [*t+1 .. t+τmax*] [AT_1, AT_14].

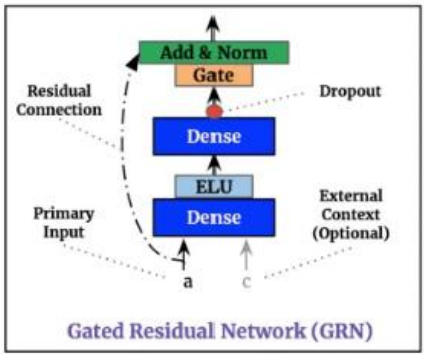

Figure 48 Gated Residual Network (GRN) [41]

The Gated Residual Network (GRN) is a basic building block repeatedly used throughout the TFT architecture (see Figure 48). It uses two activation functions, Exponential Linear Unit (ELU) and Gated Linear Unit (GLU), to select the most important features and understand which input transformations are simple and need complex preprocessing. The output is eventually passed through a standard normalization. The residual connection in the GRN enables the network to learn that the input should be skipped altogether. In addition, the GRN can handle static variables if required.

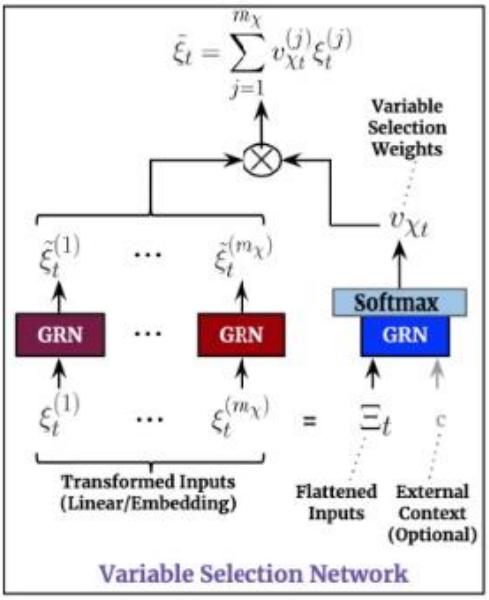

Figure 49 Variable Selection Network [41]

The Variable Selection Network (VSN) uses GRNs for filtering to distinguish between meaningful features and noisy ones (see Figure 49). The TFT model uses three instances of the VSN. At timepoint *t* a vector of past inputs (*Ξ_t*) is fed through a GRN and normalized using a SoftMax function. Meanwhile, each feature is passed through its own GRN, creating a processed vector per feature. The GRNs are the same for all features. The final output is created as a linear combination of *ξ_t* and *v*.

The LSTM Encoder Decoder Layer is used to create context-aware embeddings that work like the positional encodings of classic Transformers. The encoder handles the known past inputs, while the known future inputs are fed into the decoder. In an improved version of the Temporal Fusion Transformer, the LSTM component is replaced by a Gated Recurrent Unit (GRU) to learn long-term dependencies even more efficiently. The interpretable Multi-Head Attention, the self-attention mechanism common to all Transformers, is applied to learn long-term dependencies. The TFT architecture also provides feature interpretability.

5.5 Foundation Models

Despite the recent success of Large Language Models (LLMs) and their zero-shot and few-shot generalization, the new technology addressed the field of time series forecasting relatively late. The first foundation model for time series forecasting, TimeGPT, was released in October 2023 [96]. It opened the field of time series forecasting using API access to large foundation models. Lag Llama [61], ForecastPFN [97], PreDct [98], and MOIRAI [62] are other pre-trained foundation models designed for time series forecasting that were released recently.

5.5.1 Lag Llama

In January 2024, Lag Llama, the first open-source, general-purpose foundation model for univariate probabilistic time series forecasting, was released by a large team from different institutions like Morgan Stanley, ServiceNow, Université de Montréal, Mila-Quebec, and McGill University [61]. Lag Llama is inspired by the open-source large language model LlamaMA, released in 2023 [99]. It is based on a decoder-only Transformer architecture and was pre-trained on a large corpus of time series data from various domains to provide state-of-the-art zero-shot performance. Still, it allows for fine-tuning on a specific dataset to improve the few-shot performance. The model uses a flexible approach to handle time series data without depending on its frequency, which works well with new time series.

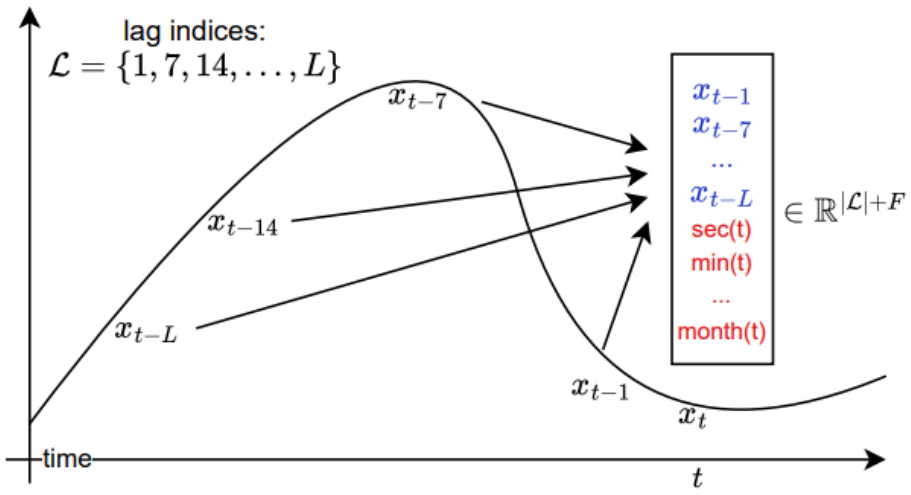

Figure 50 Tokenization in Lag Llama [61]

Lag Llama uses tokenization with lagged features from past values to model temporal dependencies. It automatically chooses appropriate frequencies, such as monthly, weekly, or daily, and builds lags accordingly. If the model's input data is a dataset with daily frequency, Lag Llama will build features using a daily lag (t-1), a weekly lag (t-7), and a monthly lag (t-30). The tokenization at timestep *t* for value *x* contains lagged features from an example set of lag indices $\mathcal{L}$. Past values of x_t (Figure 50, blue) and *F* temporal covariates constructed from timestamp *t* (Figure 50**,** red) are combined in a vector.

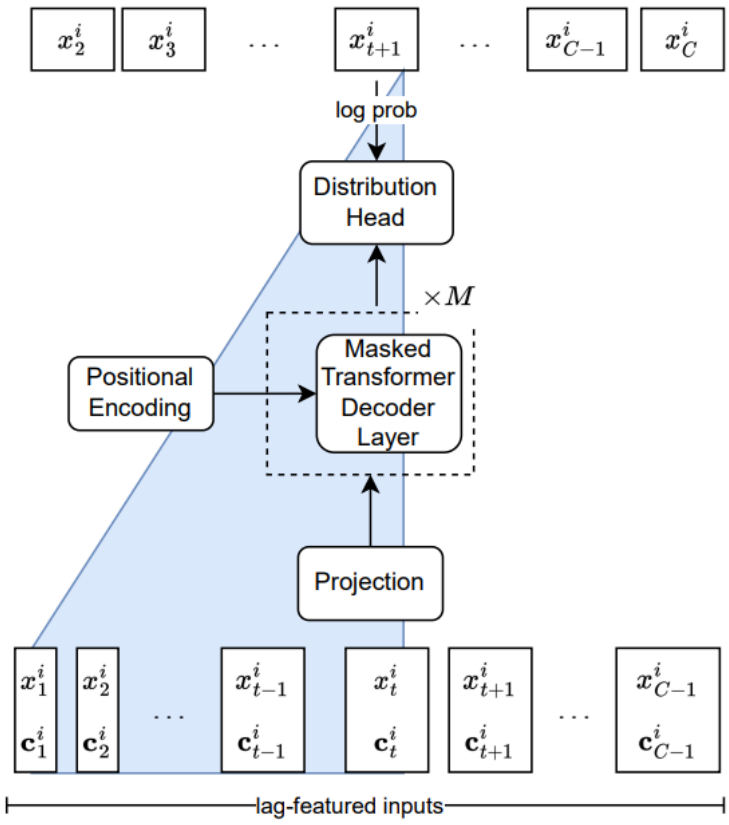

Figure 51 Lag Llama architecture [61]

Figure 51 depicts Lag Llama's architecture. The input token consists of a univariate time series *i* at timestamp *t* and additional covariates that contain the $\mathcal{L}$ lags. The input sequence is transformed through *M* linear projection layers that map the features to a hidden dimension of the attention module in the decoder. The features are then passed through the distribution head, the distinct last layer of a Lag Llama model, to predict the parameters of the forecast distribution of the next timestep. Autoregressive decoding allows the model to generate the rest of the forecast sequence until the length of the forecasting horizon is reached [61].

5.5.2 MOIRAI

The Masked Encoder-based Universal Time Series Forecasting Transformer (MOIRAI) [62] is a pre-trained Transformer model for zero-shot time series forecasting. It is labeled a Large Time Series Model and follows the concept of being a universal forecaster that is capable of creating probabilistic time series forecasts of multiple frequencies, such as hourly or daily data, any-variate

forecasts, allowing any dimensionality in the input data, and varying distributions. The foundation model was trained on the multi-domain Large-scale Open Time Series Archive (LOTSA) time series dataset, including over 27 billion datasets, created especially for MOIRAI training and made publicly available by the authors.

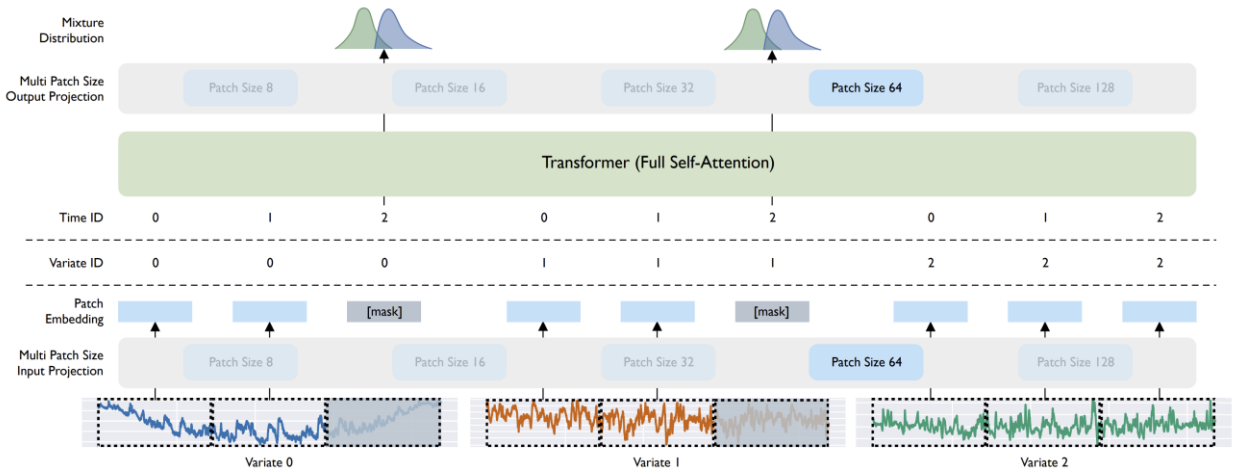

Figure 52 Overall MOIRAI architecture [62]

The MOIRAI model is based on the masked encoder architecture and employs a non-overlapping patch-based approach (see Figure 52). The improvements made to the underlying encoder-only Transformer architecture include various changes known from recent state-of-the-art LLM architectures, such as pre-normalization, RMSNorm as Layer Normalization, Query-Key Normalization, and SwiGLU activation in the Feed-Forward network architecture [62]. The first transformation step is flattening the multivariant input sequences, resulting in a single sequence for all variates. The model needs variable encodings to distinguish between the individual variates. Once this requirement is fulfilled, the model can create the Any-Variate Attention to handle an arbitrary number of variates [62]. The model Figure 52 depicts a use case where a 3-variate time series is flattened. Variates 0 and 1 are target series to be forecasted in the masked grey area, whereas variate 2 is a dynamic covariate, where the values are known for the forecasting horizon. The architecture's multi-patch size projection layers enable the model to handle a wide range of frequencies in one model. For low-frequency data, smaller patch sizes are used, whereas high-frequency data can be managed using larger patch sizes. A mixture of

parametric distributions was used for the MOIRAI pre-trained model to address the goal of allowing flexible distributions. The authors specifically proposed a Student's t-distribution, a negative binomial distribution, a log-normal distribution to model right-skewed data, and a low variance normal distribution [62]. The MOIRAI model has been evaluated in a zero-shot performance test on unseen datasets, where it could outperform even specifically full-trained Transformer architectures, such as transformer, PatchTST, and TFT [62].

5.6 Ensemble Method AutoGluon

The original AutoGluon framework was developed by Amazon Web Services and made open source in 2020 [100]. It was built on the design principle of delivering high-level automated machine learning algorithms that are robust and easy to use to enable users without limited expertise in the field to generate high-accuracy predictions. The field of automated Time Series forecasts is a new and growing research field. Deng et al. [101] proposed AutoPyTorch Forecasting, an automated deep-learning approach. In 2022, AutoTS [63] was released, and a similar approach was used for automated time series forecasting. The third of its kind, AutoGlueon-TimeSeries (AG-TS)[102], was released in 2023 on top of AutoGluon and provides probabilistic and point time series forecasts. It also includes other libraries from *Nixtla* [103], *StatsForecast* [104], and *MLForecast* [105]. One of the major advantages of AG-TS is its ability to provide probabilistic forecasts to consider uncertainty, which has been identified by Gneiting and Katzfuss [106] as a crucial factor for practical, real-world applications. Practitioners often struggle to select the right or most appropriate method for a particular use case in real-world situations. AG-TS is an ensemble method that combines various models from conventional statistical models and machine-learning-based forecasting approaches. It empowers users to combine various models in one training run and combine the results in a final step. The architecture is robust enough to handle the failure of individual models and can produce a valid result if a single model is successfully trained. The library uses numerous state-of-the-art models to construct the ensemble. All models can be categorized into one of two categories, local or global models. In local models, one model per time series is created. This category mainly includes classical

statistical models like the ARIMA Model and the ETS Model and their improved versions, the AutoARIMA Model, the AutoETS Model, the Croston Classic, Optimized and SBA Model, which are very useful for sparse data, the NaiveModel and the SeasonalAverageModel. The implementations of these models are provided by *StatsForecast* [107]. For global models, the model is fitted to the entire dataset. Every time series of the dataset is considered when training a single model. While increasing the training time, this approach benefits from cross-learning capabilities and provides faster prediction than local models, as it needs no retraining. Covariates are also naturally supported by global models. The global models in AutoGluonTS are either powerful deep learning approaches, like DeepAR [108], PatchTST [60], Temporal Fusion Transformers [46], or tree-based tabular models like LightGBM [109]. They first convert the forecasting problem into a tabular regression problem. The regression algorithms in AutoGluon are taken from the *Nixtla MLForecast* library [103].

AutoGluon offers four ensemble training modes that support different numbers and categories of pre-selected model types. *Fast_training* includes all simple statistical models and is the fastest training version that delivers immediate results. *Medium_quality* supports all statistical models as well as DeepAR. *High_quality* includes all options of the previous models as well as tree-based methods and automated statistical models. The mode *Best_quality* includes all options from the previous models plus more training copies of DeepAR. It delivers the most accurate predictions at the cost of the highest training time. The individual model results are ensembled as a convex combination of the individual model results. For probabilistic forecasts, the weighted average of the quantile forecasts of the models is computed [102], a method known as Vincentization [110].

6. Prototype Model Implementations

In this chapter, we examine the implementation of the described models and evaluate their performance on the selected use case scenarios described in Chapter 3.3. The two best-performing model architectures are then applied to all low-voltage network clusters, and in the final step, their performance on the individual networks is assessed. If not mentioned individually, the data preparation process includes creating *Pandas* data frames for the respective time frame per cluster, including additional regressors if used.

6.1 SARIMA

We used the *auto_arima* function of the *pmdarima* Python library for an easy, automated approach to the SARIMA model. *Pmdarima* wraps the popular *statsmodels* library, making it a convenient option for automated statistical modeling in Python. It provides automated SARIMA model creation, statistical tests, differencing, time series decomposition, and cross-validation utilities. As mentioned in the last chapter, statistical models make strong assumptions about the data, like the requirement for stationarity.

```
Augmented_Dickey_Fuller_Test(cluster_5_jun_source['CONSUMPTION' ],'CONSUMPTION')

Results of Dickey-Fuller Test for column: CONSUMPTION
Test Statistic                -8.907706e+00
p-value                        1.126068e-14
No Lags Used                   2.800000e+01
Number of Observations Used    2.851000e+03
Critical Value (1%)           -3.432646e+00
Critical Value (5%)           -2.862554e+00
Critical Value (10%)          -2.567310e+00
dtype: float64
Conclusion:====>
Reject the null hypothesis
Data is stationary
```

Figure 53 ADF Test confirming stationarity of the input data for cluster 5

We performed the Augmented Dickey-Fuller test on all eight source data frames to check for stationarity. Figure 53 depicts the test result on cluster 5 for June 2023. It displays a very small p-

value, indicating the rejection of the null hypothesis of non-stationarity. In the next step, a stepwise model is created for every use case using the *auto_arima* function of the *pmdarima* package. It simplifies finding the best ARIMA model for a given time series by efficiently selecting the optimal parameters. It reduces computation time by focusing on promising parameter combinations. The individual models are evaluated using the AIC information criterion. A lower AIC indicates a lower information loss and, therefore, a better model.

```
# create a stepwise model to find the best settings
# Seasonal is true
stepwise_model = auto_arima(cluster_5_oct_source_train,
                            seasonal=True,
                            m=7,            # Weekly seasonality
                            stepwise=True,  # Enable stepwise search
                            trace=True,     # If you want to see the progress/log
                            error_action='ignore',
                            suppress_warnings=True,
                            start_p=1, start_q=1, max_p=7, max_q=7)
```

Figure 54 Stepwise model selection using *statsmodels.auto_arima*

One of the remaining manual tasks is selecting the appropriate seasonality. The seasonal parameter and periodicity must be selected manually. See Figure 54 for the parameters for the cluster 5 analysis with a weekly seasonality.

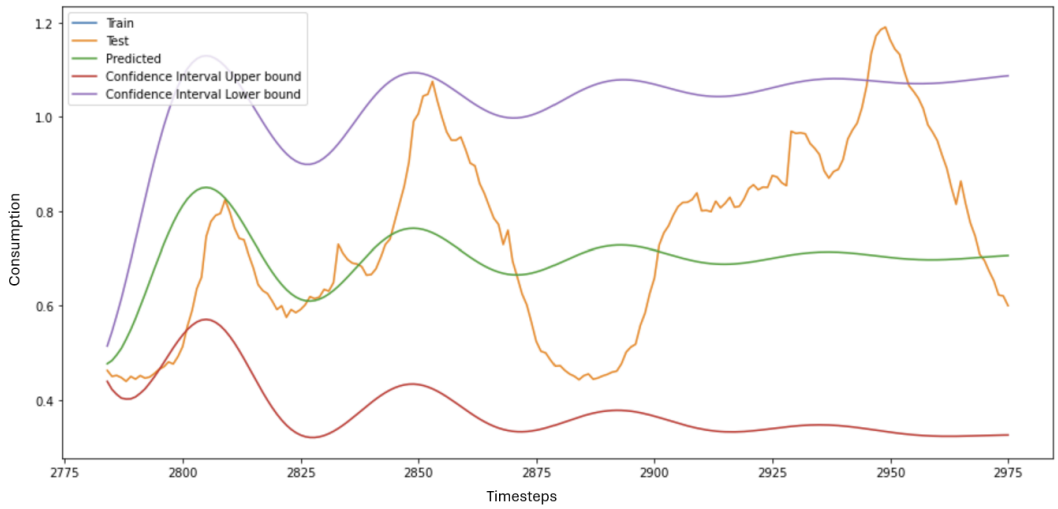

Figure 55 SARIMA Forecast results for Cluster 5 for October 30th to October 31st, 2023

After the model is created, the predict function of the stepwise model can be used to create forecasts and the respective confidence intervals for a defined number of periods. Figure 55 presents the result of cluster 5 for the last two days of October 2023. The MAPE and SMAPE for this use case were 22.12 and 21.60, respectively. For a detailed evaluation, see Chapter 6.13.

Comparing the actual versus the forecasted values reveals that the forecasts have less variance than the actual data. This reflects the forecasts as the means of predicted distributions, so it is only natural that their variability is lower than that of the actual data [43, p. 178].

However, one of the main problems with the approach is that multiple seasonal data, like in our use case, cannot be modeled in SARIMA. Because we have daily, weekly, and yearly seasonality, it seems likely that SARIMA fails to identify those overlapping seasonalities. Suppose a SARIMA model is applied to such a use case with multiple seasonalities, with only some being captured in the model. In that case, it often results in residuals that are not normally distributed and still correlated, indicating that the model cannot be used at all [44, p. 234].

6.2 FB Prophet

FB Prophet's application is straightforward due to strong parameter defaults. Prophet has only one prerequisite: the time dimension in the Pandas data frame must be named *ds*, and the dependent variable must be named *y*.

	ds	TRAFO	y	Villach_TL_Lufttemp_MWm	Villach_WG_Windgeschw	Villach_P_Luftdruck_MWm	Villach_RR_Niederschlag	Villach_GI_Globalstrahlung	Villach_RF_RelFeuchte
2971	2023-10-31 22:45:00	cluster_5	0.673369	5.25	1.45	956.450012	0	0.0	94.0
2972	2023-10-31 23:00:00	cluster_5	0.653239	5.20	1.70	956.700012	0	0.0	94.0
2973	2023-10-31 23:15:00	cluster_5	0.623167	4.95	1.70	956.649994	0	0.0	94.5
2974	2023-10-31 23:30:00	cluster_5	0.621075	4.80	1.40	956.799988	0	0.0	96.0
2975	2023-10-31 23:45:00	cluster_5	0.600484	5.05	2.60	956.950012	0	0.0	95.0

Figure 56 Dataframe preparation for FB Prophet

Figure 56 shows an example data frame with the correct naming and six additional regressors for the weather. Prophet's major advantage is its ability to identify seasonal patterns automatically. This is a strong feature for our use cases, as the energy consumption time series follows daily, weekly, and seasonal patterns.

```
# Create the model
# Prophet can automatically check the seasonal patterns
prophet_model = Prophet()
prophet_model.fit(cluster_5_oct_source_train)
future = prophet_model.make_future_dataframe(periods=192, freq='15min')
forecast = prophet_model.predict(future)
forecast = forecast[['ds','yhat']]
```

Figure 57 Model creation for the Prophet for October 30th to 31st

Figure 57 illustrates the code fragment used to create the future prediction data frame for the univariate forecast without additional regressors. The required forecast horizon is 192 timesteps, and the frequency is 15-minute intervals. The model stores the forecast results in the variable *yhat*. See Figure 58 for the results of the univariate forecast without additional regressors for cluster 5 for the weekday use case.

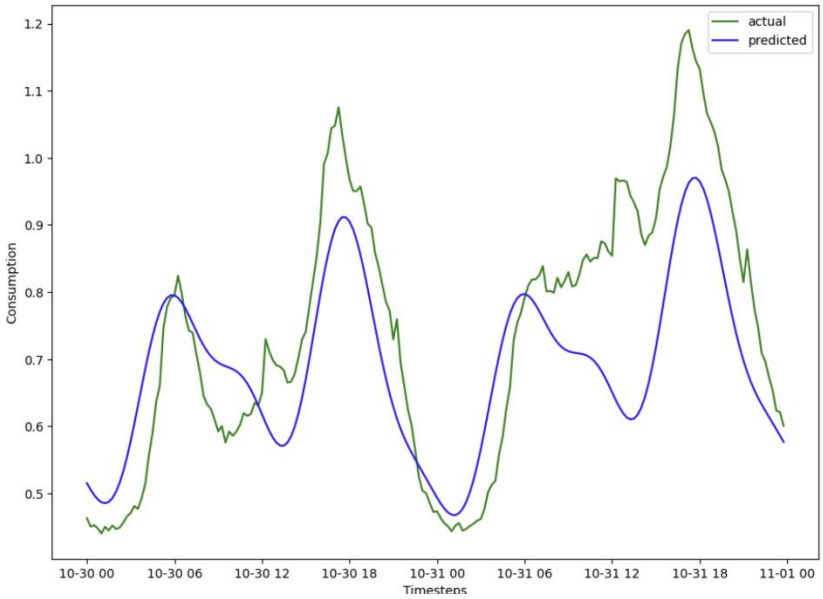

Figure 58 Univariate forecast without additional regressors using FB Prophet

It is easy to add additional regressors to the Prophet model. To evaluate the influence of past weather data as additional regressors, we incorporated wind speed, air temperature, and humidity into the model and compared the results to the univariate use case. We performed a distance comparison using Dynamic Time Warping and the Mean Absolute Error to select the most promising regressors from the five additional regressors: air temperature, wind speed, air

pressure, global radiation, and relative humidity. Dynamic Time Warping is an algorithm used to measure the similarity between two temporal sequences, which may vary in speed or length, by aligning them non-linearly to minimize the distance between corresponding points [111]. The idea was to compare the results for both distance metrics and choose the three regressors closest to the consumption time series.

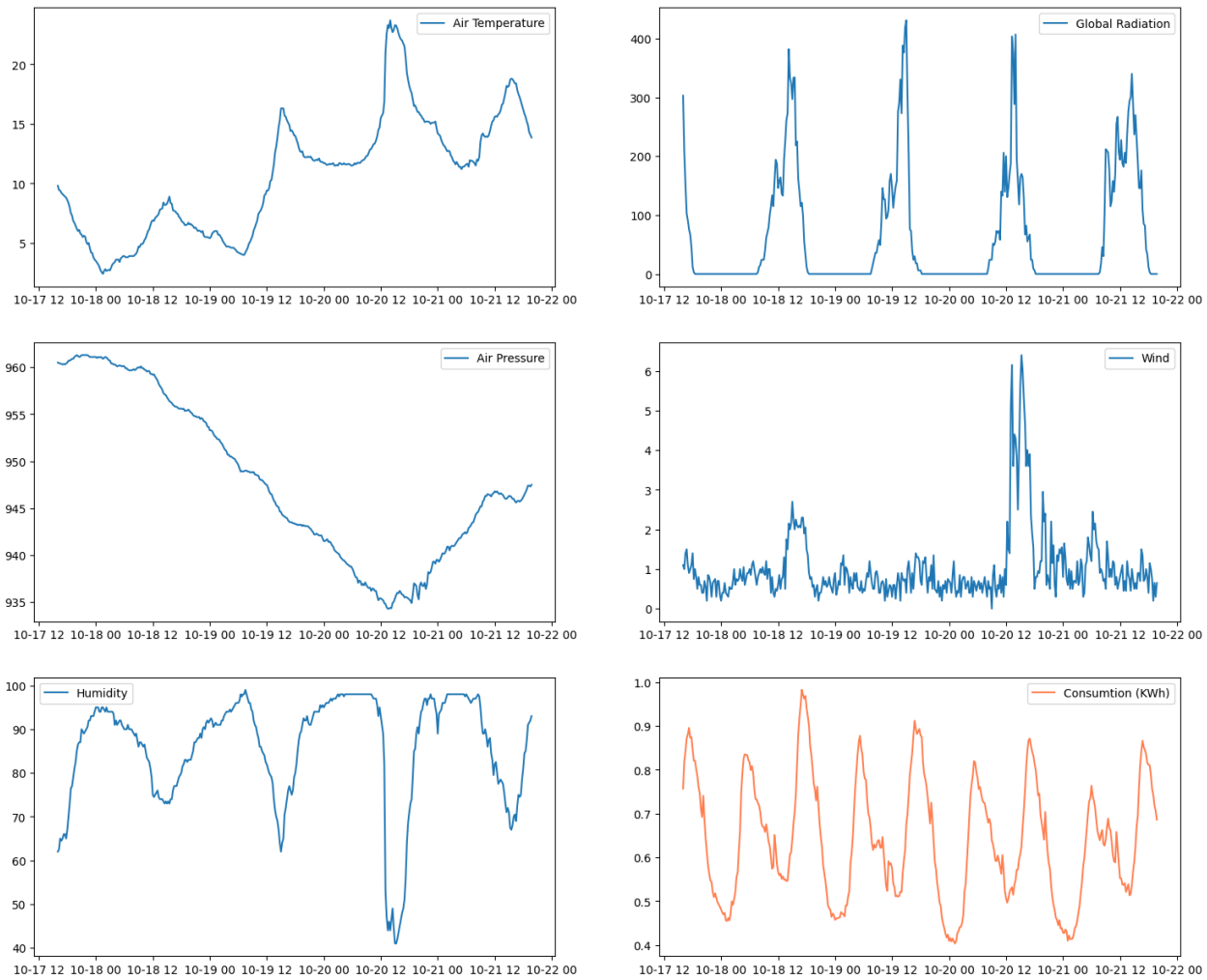

Figure 59 Weather data and consumption data comparison

Figure 59 compares the weather data and the consumption time series for six days in October 2023.

Table 6.1 Distance comparison using MAE and DTW for the additional regressors.

	DTW Distance	MAE Distance
Wind Speed	1.874292e+03	0.741219
Air Temperature	3.275469e+04	11.061762
Relative Humidity	2.388433e+05	80.256488
Global Radiation	3.224137e+05	108.253066
Air Pressure	2.839879e+06	954.260255

Table 6.1 depicts the results of the distance comparison. The results of DTW and MAE are equivalent. We added the three most similar regressors: wind speed, air temperature, and relative humidity.

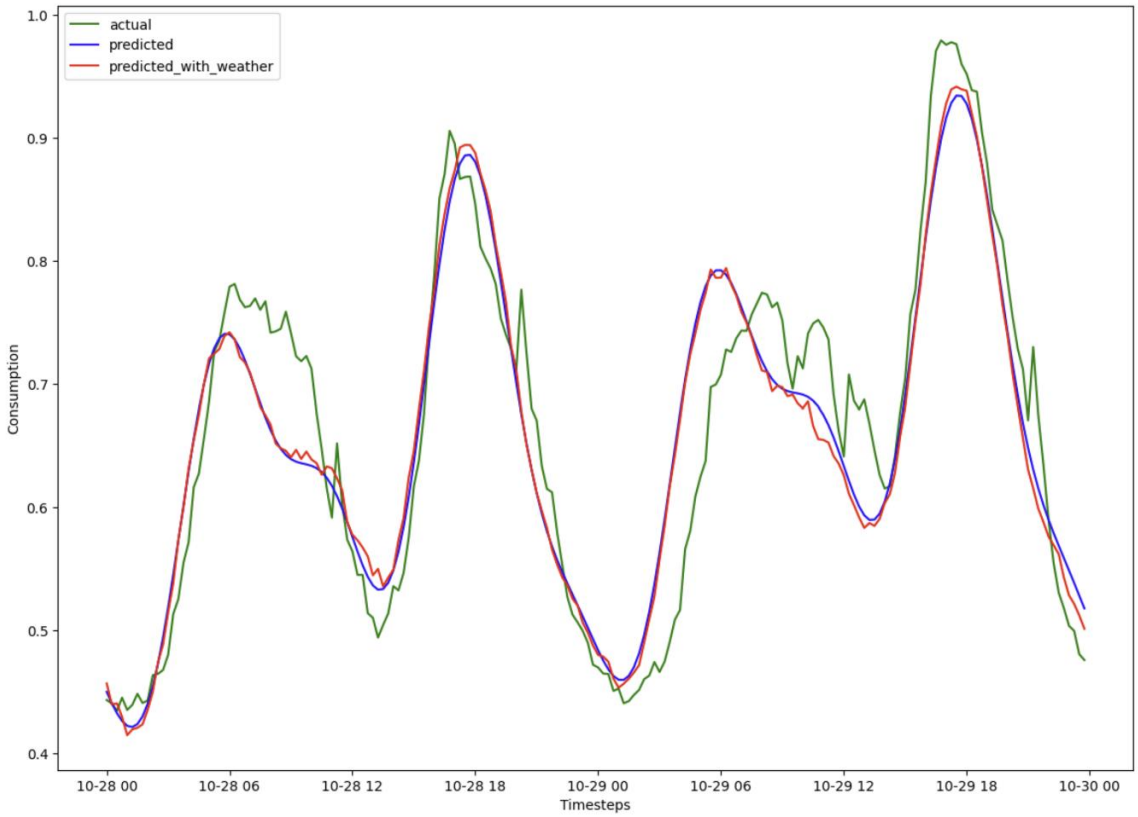

Figure 60 Comparison of consumption forecasts with and without regressors using FB Prophet for October 28th – October 29th.

Table 6.2 Comparison of consumption forecasts with and without regressors using FB Prophet.

	Cluster 5							
	October 26.-27.		October 28.-29.		October 30.-31.		June 29.-30.	
FB Prophet	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE
Without Regressors	6.51	6.61	6.61	6.55	13.94	14.47	8.51	8.09
With Regressors	6.16 (-0,35)	6.45 (-0,16)	6.79 (+0,18)	6.73 (+0,18)	12.91 (-1,03)	13.03 (-1,44)	9.66 (+1.15)	9.11 (+1.02)
	Cluster 19							
	October 26.-27.		October 28.-29.		October 30.-31.		June 29.-30.	
	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE
Without Regressors	17.54	15.59	19.52	20.20	16.45	18.25	11.75	10.85
With Regressors	16.01 (-1,53)	14.44 (-1,15)	19.31 (-0,21)	20.51 (+0,31)	13.87 (-2,58)	15.09 (-3,16)	14.12 (+2.37)	12.83 (+1.98)

The forecasting results with regressors improved the MAPE on average by 0,92% and the SMAPE by 0.9% for the October use cases. It decreased the MAPE by 0.61% and the SMAPE by 0.48% in the summer use case (see Table 6.2 and Figure 60). This led to the conclusion that providing past regressors for weather data can improve the forecasting accuracy of the Prophet models, which aligns with the results from the original paper on FB Prophet. Meanwhile, it does not guarantee such behavior, and adding additional regressors can decrease the performance in our use case, especially during the summer months. Our work is restricted by the availability of 15-minute interval data, and we decided not to consider additional feeder information with solar devices.

This may be the main reason why the influence of weather data may be limited. If we decide to include the feeder information in the future, this may change and must be reevaluated.

6.3 Kalman Filters

We used the *KalmanForecaster* implementation of the Unit 8 *darts.models* Python package [112] for the Kalman filter-based consumption forecast. This package offers a convenient way to use various time series forecasting methods, such as ARIMA, N-BEATS, Fast Fourier Transform, or Exponential Smoothing. It supports filtering methods like Gaussian processes and Kalman Filters. The only mandatory parameter of the Kalman Filter model is the size of the filter state vector, called *dim_x*. We selected a default value of 100 and created the forecasting results for all evaluation use cases. See Table 6.3 for the results per use case scenario.

Table 6.3 Comparison of consumption forecast with the *dart.models KalmanForecaster*.

Cluster 5							
October 26.-27.		October 28.-29.		October 30.-31.		June 29.-30.	
MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE
11.58	10.81	8.67	8.8	12.50	13.07	6.16	5.9
Cluster 19							
October 26.-27.		October 28.-29.		October 30.-31.		June 29.-30.	
MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE
13.81	14.93	17.82	19.45	27.63	22.83	8.68	8.97

The Kalman Filters' results were solid. They performed very well on the problematic Summer-Cluster use cases for Cluster 19 and especially Cluster 5, with a MAPE of 6.16% and a SMAPE of 5.9%. See Figure 61 for the visualization of the results of this use case.

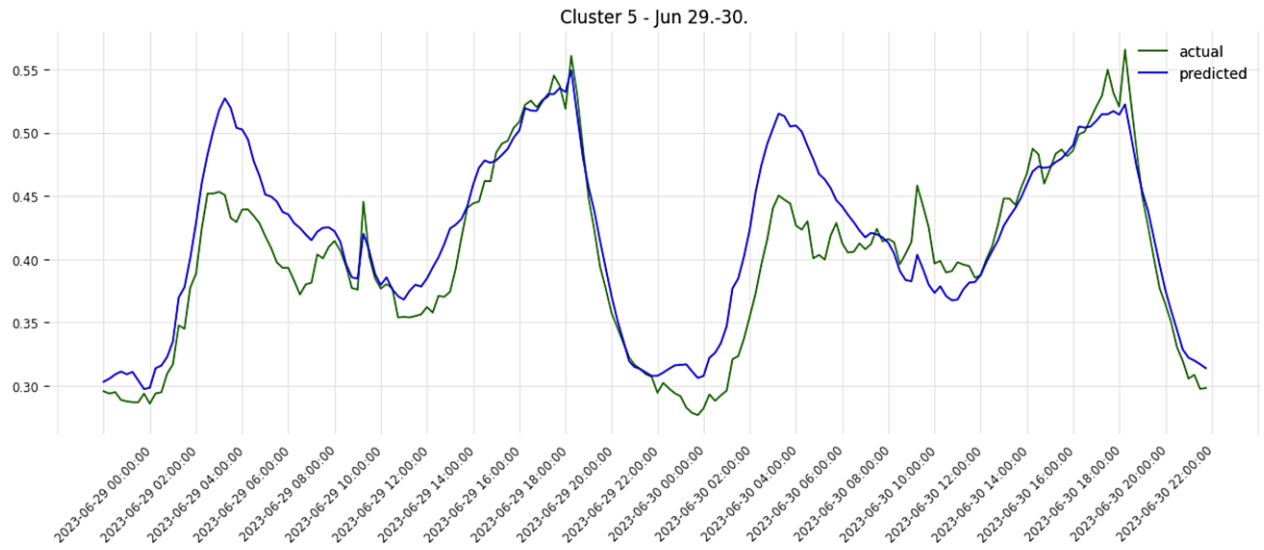

Figure 61 Kalman Filter forecasting results in kWh for Cluster 5 for June

6.4 NeuralProphet

The data preparation for the NeuralProphet starts with the same steps as the original FB Prophet. The time dimension and the dependent variable must be named *ds* and *y*, respectively. The model creation is very different, however, as the architectural concept of NeuralProphet is quite different from of its predecessor. The parameters are known from neural network architectures, like the number of training epochs, the batch size, the normalization strategy, and the respective loss function used for weight updates. Two characteristics it inherited from the original Prophet are the possibility of defining country-specific holidays and the easy setup for additional regressors.

```
npmodel_c5_oct_30_31 = NeuralProphet(batch_size=8, epochs=100, normalize="standardize", loss_func="Huber")
npmodel_c5_oct_30_31 = npmodel_c5_oct_30_31.add_country_holidays(country_name="DE")
```

Figure 62 Neural Prophet model parameters

See Figure 62 for the model parameters used in the evaluation use cases. We used a batch size of 8, set the number of training epochs to 100, and defined Huber Loss as the loss function. German-specific holiday settings were used. Again, we used wind speed, air temperature, and relative

humidity as regressor variables for a second modeling phase for univariate forecasts, including past regressors.

Table 6.4 Comparison of consumption forecasts with and without regressors using Neural Prophet.

Cluster 5	October 26.-27.		October 28.-29.		October 30.-31.		June 29.-30.	
	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE
Without Regressors	7.21	7.15	9.14	9.59	14.3	14.86	6.93	6.64
With Regressors	6.57 (-0.64)	6.67 (-0.48)	9.11 (-0.03)	9.57 (-0.02)	13.4 (-0.9)	13.7 (-1.16)	7.24 (+0,31)	6.92 (+0.28)
Cluster 19	October 26.-27.		October 28.-29.		October 30.-31.		June 29.-30.	
	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE
Without Regressors	23.87	20.14	20.32	22.44	11.18	11.7	7.87	7.5
With Regressors	21.92 (-1.95)	18.72 (-1.42)	20.3 (-0.02)	22.39 (-0.05)	11.34 (+0.16)	11.86 (+0.16)	9.94 (+2.07)	9.27 (+1.77)

While the additional regressors proved useful for the use cases in autumn, reducing the MAPE on average by 0.56% and the SMAPE on average by 0.49%, they failed to deliver better results in summer, similar to the original FB Prophet model. They negatively influenced the forecast on average by 0.88% for the MAPE and 0.74% for the SMAPE (see Table 6.4). See Figure 63 for the forecasting results for Cluster 5 in June, where adding regressors decreased the forecasting results.

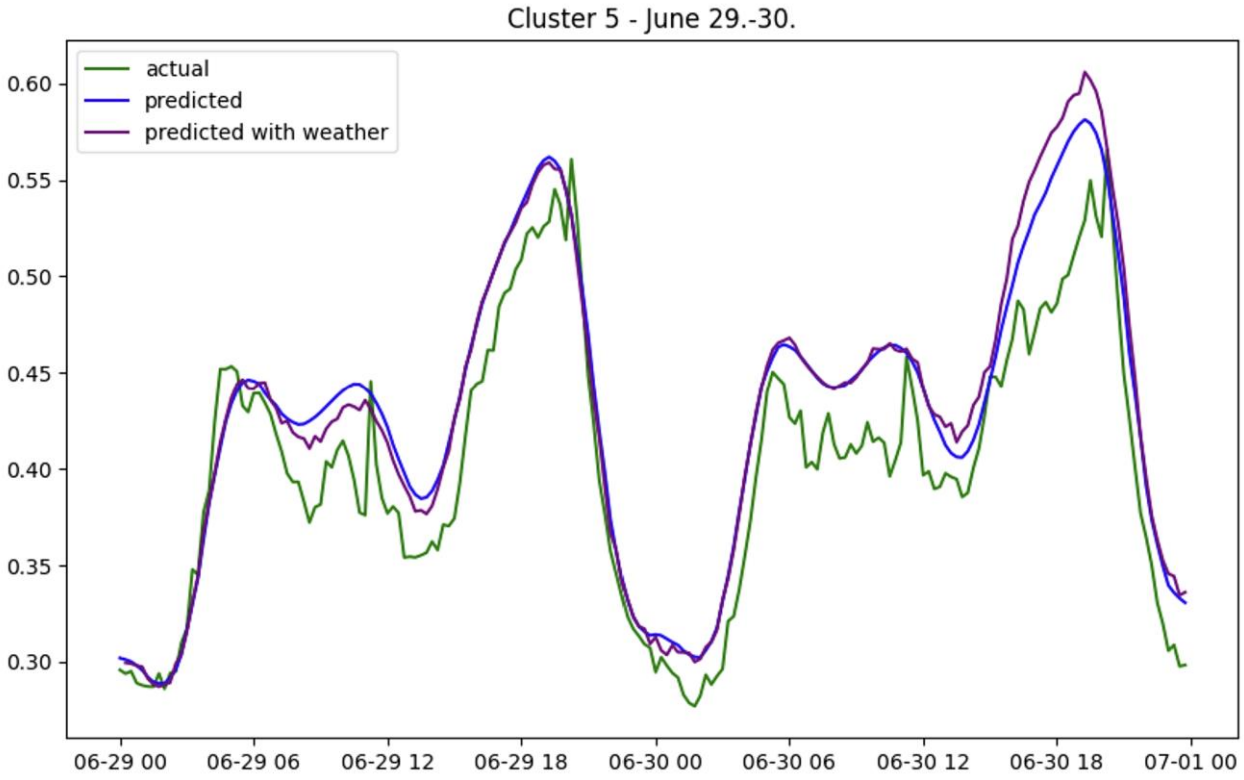

Figure 63 Neural Prophet forecasting results in kWh for cluster 5 for June with and without additional regressors

6.5 LSTM Neural Networks

We chose the classic *Tensorflow Keras* library for the LSTM implementation [113]. This library offers a convenient and widely used way to define and train neural network model architectures.

Layer (type)	Output Shape	Param #
lstm_18 (LSTM)	(None, 576, 100)	40,800
dropout_18 (Dropout)	(None, 576, 100)	0
lstm_19 (LSTM)	(None, 50)	30,200
dropout_19 (Dropout)	(None, 50)	0
dense_9 (Dense)	(None, 192)	9,792

Total params: 242,378 (946.79 KB)

Trainable params: 80,792 (315.59 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 161,586 (631.20 KB)

Figure 64 Best-performing LSTM architecture in the evaluation scenarios

In the initial step, we tested different model architectures on one of the evaluation use cases to understand different successful layer architectures before applying the best-performing architecture to all the datasets. The best-performing model architecture summary is illustrated in Figure 64. It contains two LSTM cell layers with 100 and 50 cells, respectively, with a 20%-layer dropout. It was trained for 150 epochs, including early stopping, with a patience value of 8. The optimizer was ADAM, and the chosen loss function was the MSE.

Layer (type)	Output Shape	Param #
lstm_32 (LSTM)	(None, 576, 100)	40,800
dropout_32 (Dropout)	(None, 576, 100)	0
bidirectional_4 (Bidirectional)	(None, 576, 100)	60,400
dropout_33 (Dropout)	(None, 576, 100)	0
bidirectional_5 (Bidirectional)	(None, 100)	60,400
dropout_34 (Dropout)	(None, 100)	0
dense_15 (Dense)	(None, 192)	19,392

Total params: 542,978 (2.07 MB)

Trainable params: 180,992 (707.00 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 361,986 (1.38 MB)

Figure 65 LSTM model using bidirectional LSTM cells

Surprisingly, the simple layout with two LSTM cell layers and no regularization outperformed the bidirectional LSTM cell architectures, which are known for performing well on time series data because they simultaneously scan the input from both sides (see Figure 65 and Table 6.5).

Table 6.5 Evaluation Scenario Results for Cluster 5.

	Best uni-directional LSTM Architecture (Figure 64)	Bi-directional LSTM Architecture (Figure 65)
MAPE	26.60	28.14
SMAPE	31.21	33.84

The overall performance of the LSTM architectures was disappointing and very dependent on the use case.

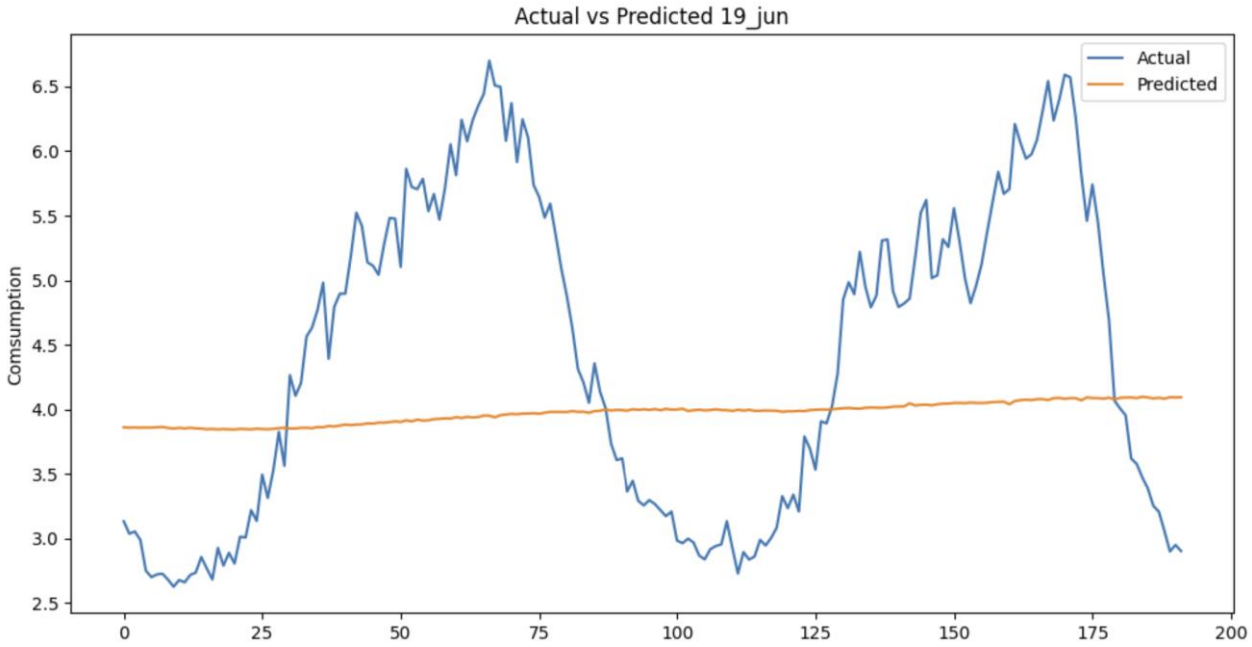

Figure 66 LSTM Results on Cluster 19 for June

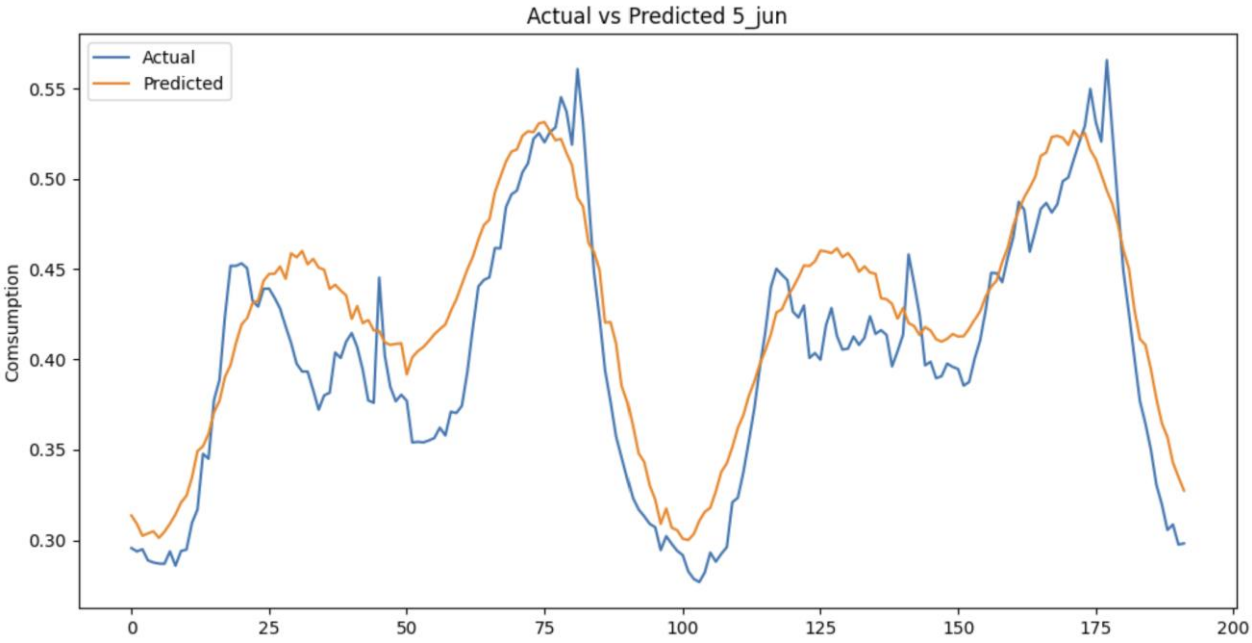

Figure 67 LSTM Results on Cluster 5 for June

The same architecture that proved quite successful for the use case in June on cluster 5 completely failed to learn on the training date for cluster 19 (see Figure 66 and Figure 67).

6.6 iTransformer

For the iTransformer model, we used the implementation provided by the *NeuralForecast* Python library, which is part of the open-source time series infrastructure *Nixtla* [103]. Because it is a very recent model architecture, only proposed in March 2024, the number of available implementations is limited. Still, *Nixtla* supports a very convenient version with standard functionalities from the *NeuralForecast* infrastructure, such as time series adapted cross-validation.

```
horizon = 192
model_5_oct_2627 = [iTransformer(h=horizon,
                                 input_size=3*horizon,
                                 n_series=1,
                                 max_steps=1000,
                                 early_stop_patience_steps=3)]
nf_5_oct_2627 = NeuralForecast(models=model_5_oct_2627,
                               freq='15T')
nf_5_oct_2627_preds = nf_5_oct_2627.cross_validation(
    df=cluster_5_oct_source_train_1,
    val_size=192,
    test_size=192,
    n_windows=None)
```

Figure 68 Model parameters for the *NeuralForecast* iTransformer model

The creation of the model is straightforward (see Figure 68). It includes providing the forecasting horizon, the input sequence size, and the number of series included in the model, as it supports multivariate forecasting solutions, early stopping patience, and the maximum number of training steps. In the *NeuralForecast* model, the frequency of the input data is set.

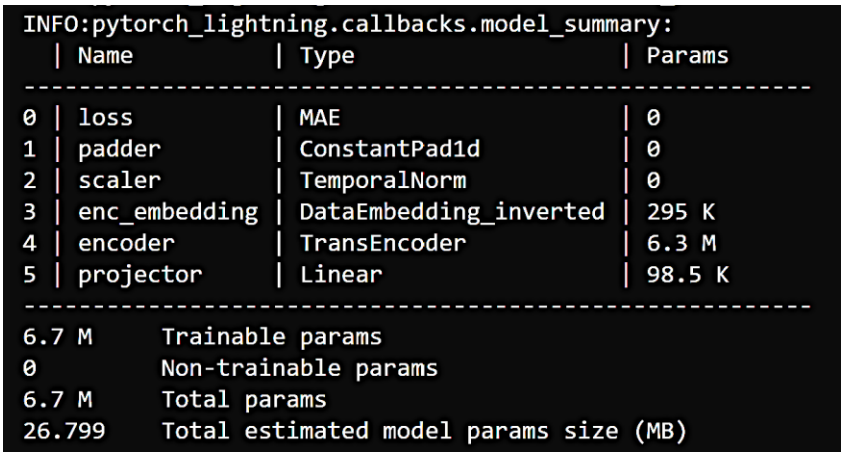

Figure 69 iTransformer model summary

Because it is a Transformer architecture, it is a heavy model with over six million parameters that must be trained on a suitable GPU (Figure 69). See the *PyTorch* lightning model summary for details. Our models were trained on an A100 GPU with 40GB of RAM.

The iTransformer model outperformed all other approaches tested in the difficult use case for Cluster 19 from October 28th to 29th. The overall MAPE and SMAPE for all models for that use case were 20.94% and 22.66%, making it the hardest use case to forecast. The iTransformer model reached an outstanding MAPE of 12.67% and 13.68%, respectively (see Figure 70).

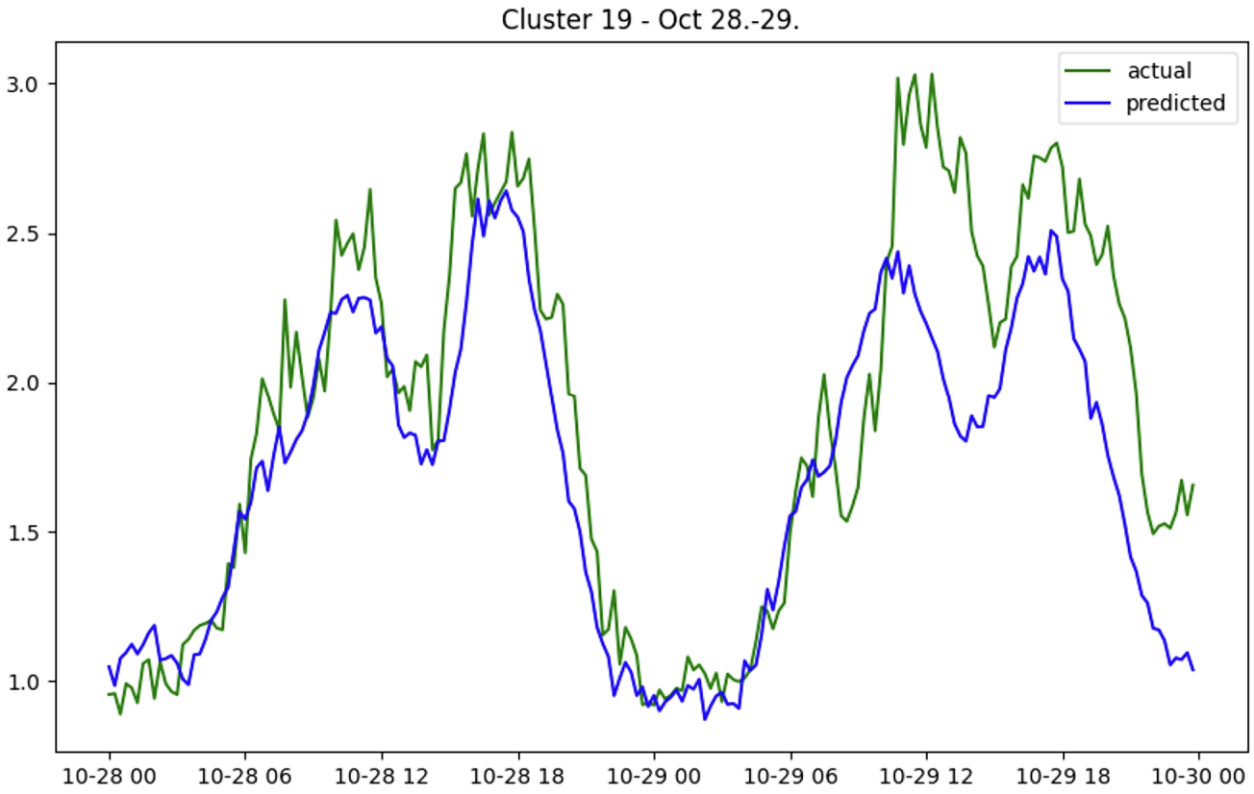

Figure 70 iTransformer forecasting results in kWh for Cluster 19 for October 28th – 29th

6.7 PatchTST

Once again, we used the *NeuralForecast* environment for the PatchTST Transformer architecture. The model creation is similar to that of the iTransformer, including defining the forecasting horizon and input sequence size and providing input data frequency via the *NeuralForecast* object. The time dimension and the dependent variable must be renamed *ds* and *y*. One difference between the iTransformer and the PatchTST architecture regarding the models' sizes becomes conspicuous on a model summary inspection (see Figure 71).

```
INFO:pytorch_lightning.callbacks.model_summary:
  | Name         | Type              | Params | Mode
-------------------------------------------------------
0 | loss         | MAE               | 0      | train
1 | padder_train | ConstantPad1d     | 0      | train
2 | scaler       | TemporalNorm      | 0      | train
3 | model        | PatchTST_backbone | 1.6 M  | train
-------------------------------------------------------
1.6 M     Trainable params
3         Non-trainable params
1.6 M     Total params
6.342     Total estimated model params size (MB)
```

Figure 71 PatchTST model summary

The TST backbone's total trainable parameters are only 1.6 million, making it much lighter than the iTransformer standard model. This makes the training faster while delivering accurate forecasts. The PatchTST model was especially strong for the use case for Cluster 19 in June. It had a MAPE of 7.4% and SMAPE of 7.04% (see Figure 72).

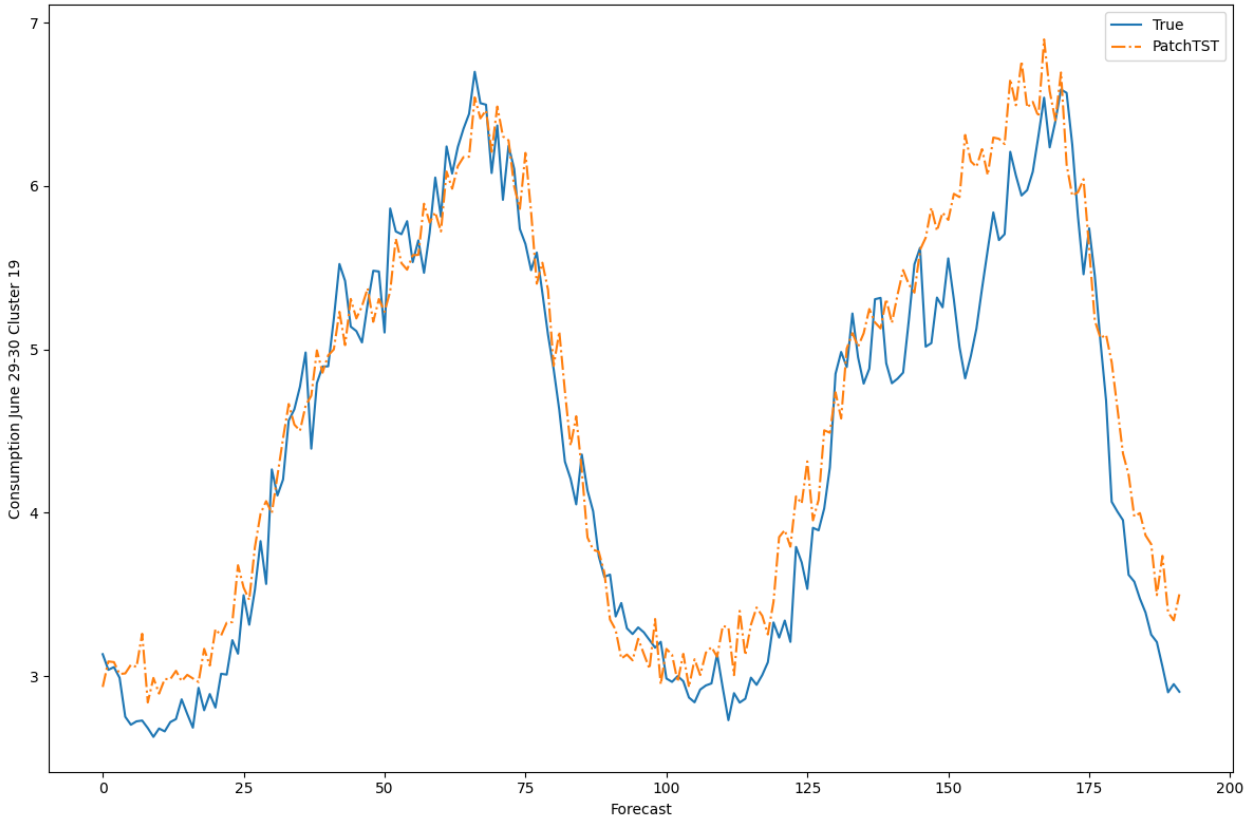

Figure 72 PatchTST forecasting results for Cluster 19 in June

For a detailed overview of the results, see Tables 6.8 and 6.9 in the final evaluation in Chapter 6.13.

6.8 Temporal Fusion Transformer

We used the Temporal Fusion Transformer [46] model in the *PyTorchForecasting* library and the *PyTorch* lightning implementation for the training framework. We also used the *Optuna* framework [114] to search for the best hyperparameter tuning of the TFT. The framework is designed for machine learning pipelines. It enables efficient sampling of hyperparameters and pruning of unpromising trials while outperforming classical approaches like *Grid Search* or unstructured approaches, such as random search. To create a TFT model for our use case, the following steps are necessary:

- Define training and test data sizes
- Define TimeSeriesDataset
- Create data loader objects from the train, validation, and test data
- Use *Optuna* to find appropriate hyperparameter settings
- Define Early stopping and a Logger for the training process
- Create a *PyTorch* lightning trainer object
- Create the Temporal Fusion Transformer
- Fit the Model using the data loaders

For the use case of October 30th - 31st for cluster 5, *Optuna* finetuning determined the following hyperparameter settings:

{'gradient_clip_val': 0.04804236445022774,
'hidden_size': 36,
'dropout': 0.11705886668173918,
'hidden_continuous_size': 33,
'attention_head_size': 3,
'learning_rate': 0.007943282347242816}.

```
training = TimeSeriesDataSet(
    cluster_5_oct_source[lambda x: x.TIME_IDX <= training_cutoff],
    time_idx="TIME_IDX",
    target="CONSUMPTION",
    group_ids=["TRAFO"],
    min_encoder_length=max_encoder_length//2,
    max_encoder_length=max_encoder_length,
    min_prediction_length=1,
    max_prediction_length=max_prediction_length,
    static_categoricals=["TRAFO"],
    time_varying_known_reals=["TIME_IDX","DAY","DAY_OF_WEEK", "MONTH", 'HOUR'],
    time_varying_unknown_reals=['CONSUMPTION'],
    target_normalizer=GroupNormalizer(groups=["TRAFO"], transformation="softplus"),  # normalization
    add_relative_time_idx=True,
    add_target_scales=True,
    add_encoder_length=True,
    allow_missing_timesteps=True
)
```

Figure 73 *PyTorch* Forecasting TimeSeriesDataSet Type Structure

The only import format this library accepts is the *pytorch_forecasting* TimeSeriesDataSet. First, the prediction length of two days and the lookback window size of one week were defined. In the next step, the *TimeSeriesDataSet* data type was defined. Figure 73 depicts the outline of the *TimeSeriesDataSet* type provided by *PyTorch*. The *PyTorch* Lightning Trainer can be used for training. We applied early stopping on the validation loss with a patience of 6 and a minimum delta of 0.0001.

```
trainer_2_AT = pl.Trainer(
    max_epochs=100,
    accelerator='gpu',
    devices=1,
    enable_model_summary=True,
    gradient_clip_val=0.04804236445022774,
    callbacks=[lr_logger, early_stop_callback],
    logger=logger)

tft = TemporalFusionTransformer.from_dataset(
    training,
    learning_rate=0.007943282347242816,
    hidden_size=36,
    attention_head_size=3,
    dropout=0.11705886668173918,
    hidden_continuous_size=33,
    output_size=7,  # [0.02, 0.1, 0.25, 0.5, 0.75, 0.9, 0.98]
    loss=QuantileLoss(), # probabilistic forecasts with loss per quantile
    log_interval=10,
    reduce_on_plateau_patience=4)

trainer_2_AT.fit(
    tft,
    train_dataloaders=train_dataloader,
    val_dataloaders=val_dataloader)
```

Figure 74 TFT Model definition and training pipeline

Figure 74 depicts the TFT hyperparameters and the training pipeline components. For the loss function, we defined the QuantileLoss to obtain prediction intervals instead of point estimates. An output size of 7 defines the following percentiles: 0.02, 0.1, 0.25, 0.5, 0.75, 0.9, and 0.98. The *PyTorch* Lightning Trainer object was created, and the model was fitted to the training data. The overall results were quite promising, as listed in Tables 8.7 and 8.8 in Chapter 6.13.

```
INFO:lightning.pytorch.callbacks.model_summary:
   | Name                                | Type                                | Params
-------------------------------------------------------------------------------------------
0  | loss                                | QuantileLoss                        | 0
1  | logging_metrics                     | ModuleList                          | 0
2  | input_embeddings                    | MultiEmbedding                      | 1
3  | prescalers                          | ModuleDict                          | 660
4  | static_variable_selection           | VariableSelectionNetwork            | 15.4 K
5  | encoder_variable_selection          | VariableSelectionNetwork            | 36.6 K
6  | decoder_variable_selection          | VariableSelectionNetwork            | 31.2 K
7  | static_context_variable_selection   | GatedResidualNetwork                | 5.4 K
8  | static_context_initial_hidden_lstm  | GatedResidualNetwork                | 5.4 K
9  | static_context_initial_cell_lstm    | GatedResidualNetwork                | 5.4 K
10 | static_context_enrichment           | GatedResidualNetwork                | 5.4 K
11 | lstm_encoder                        | LSTM                                | 10.7 K
12 | lstm_decoder                        | LSTM                                | 10.7 K
13 | post_lstm_gate_encoder              | GatedLinearUnit                     | 2.7 K
14 | post_lstm_add_norm_encoder          | AddNorm                             | 72
15 | static_enrichment                   | GatedResidualNetwork                | 6.7 K
16 | multihead_attn                      | InterpretableMultiHeadAttention     | 3.5 K
17 | post_attn_gate_norm                 | GateAddNorm                         | 2.7 K
18 | pos_wise_ff                         | GatedResidualNetwork                | 5.4 K
19 | pre_output_gate_norm                | GateAddNorm                         | 2.7 K
20 | output_layer                        | Linear                              | 259
-------------------------------------------------------------------------------------------
149 K     Trainable params
0         Non-trainable params
149 K     Total params
0.599     Total estimated model params size (MB)
```

Figure 75 TFT Model Summary

Figure 75 shows the model summary for our TFT architecture. This indicates that the overall model is comparably heavy and has a high training time. Additionally, this transformer-based architecture requires training on a powerful GPU.

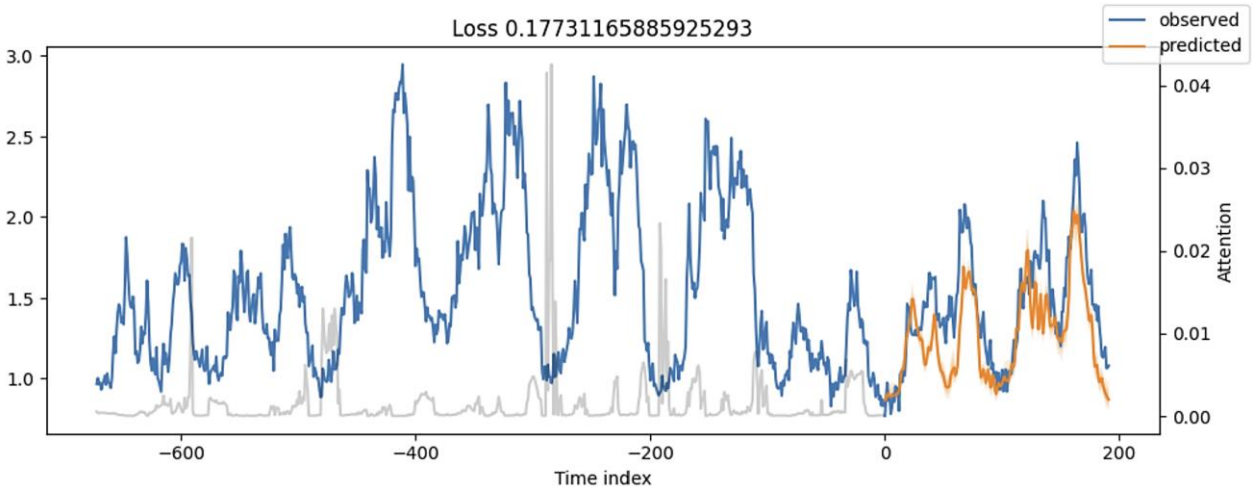

Figure 76 TFT forecast for Cluster 19, June 29th - 30th

One of the key advantages of the Temporal Fusion Transformer architecture is its built-in explainability, which allows for the visualization of multi-head attention values. Figure 76 illustrates the forecasting results for the last two days of June 2023 on cluster 19. The grey line plot, which represents the multi-head attention, enables the user to verify if day changes and week resets are identified correctly. This visualization is a powerful tool for controlling the model's learning process.

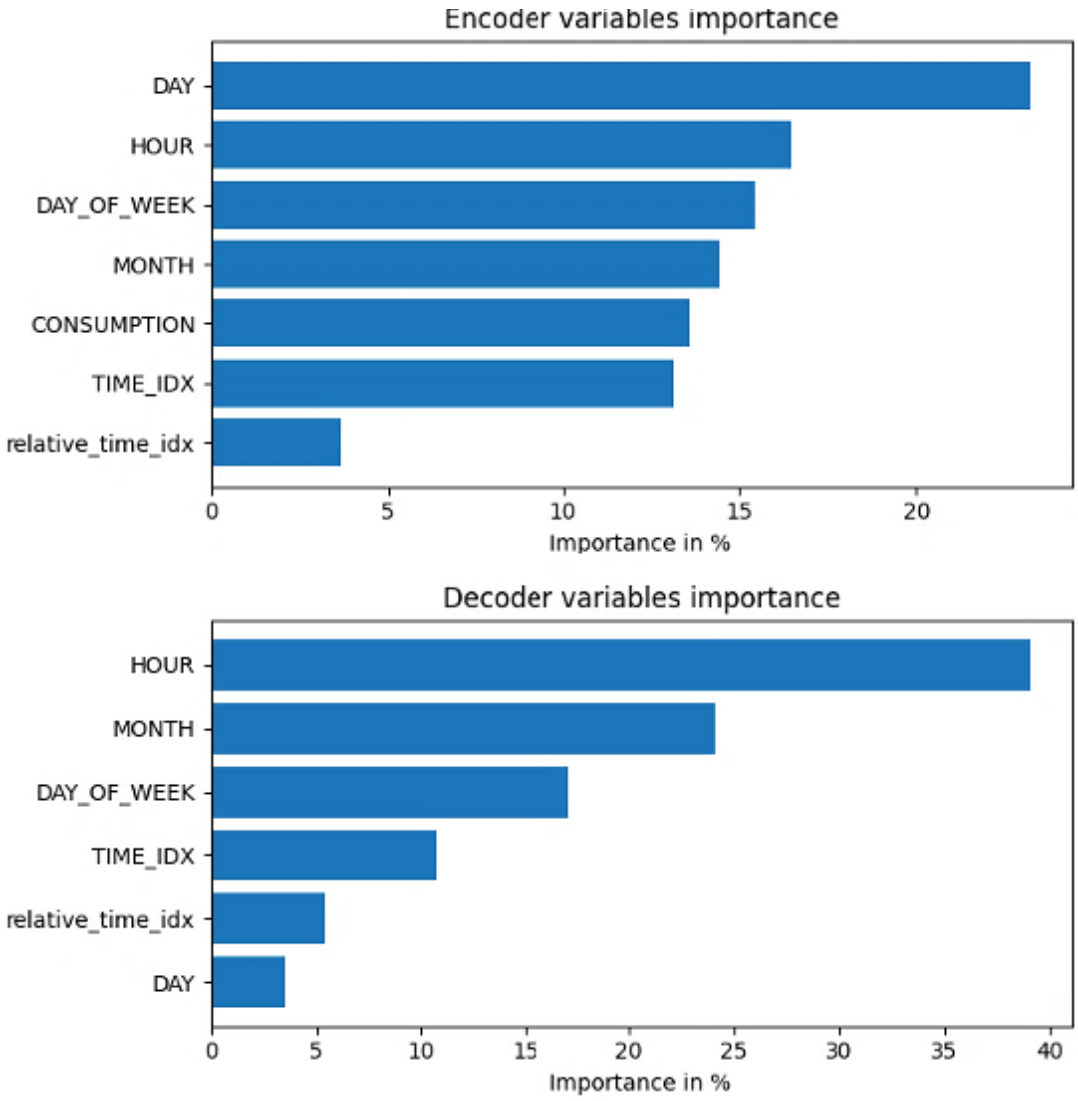

Figure 77 Feature Importance Plot for the TFT Model

To increase the explainability of the model results, we plotted the feature importance of the encoder and decoder variables to obtain a glimpse of what parameters were most useful to create the forecast (see Figure 77). It reveals that *day* and *hour* play important roles in the encoder and decoder, respectively, for creating the forecast.

6.9 Lag Llama

We used the open-source Lag Llama implementation available on GitHub [115] and downloaded the pre-trained model from Hugging Face [116]. The model requires the input data to be provided in the GluonTS format. The data format of GluonTS expects the three basic elements: start date, target data, and frequency of the data.

```
from gluonts.dataset.common import ListDataset
from gluonts.dataset.field_names import FieldName

def dataframe_to_gluonts(df):
    df['ds'] = pd.to_datetime(df['ds'])  # ensure datetime format
    unique_series = df.groupby('unique_id')

    dataset = []
    for name, group in unique_series:
        series_dict = {
            "start": group['ds'].iloc[0],
            "target": group['y'].tolist(),
            "freq": "15min"  # as data is in 15min intervals
        }
        dataset.append(series_dict)

    return ListDataset(dataset, freq="15min")

train_data_5_2627_oct = dataframe_to_gluonts(cluster_5_2627_oct_source_train)
```

Figure 78 GluonTS data format for the Lag Llama model

See Figure 78 for the conversion routine of our input data frames to the GluonTS format. We used Lag Llama in two ways: first, as a zero-shot model applied to our test data without any specific training, and second, as a fine-tuned version where we re-trained the foundation model on our

specific dataset for 150 epochs. The zero-shot prediction is straightforward. First, we defined a *LagLlamaEstimator*, and then a predictor was created, which can be used to create predictions.

Table 6.6 Evaluation results for Lag Llama Zero-Shot and fine-tuned version.

	Lag Llama Zero-Shot Model		Lag Llama Fine-tuned Model	
	MAPE	SMAPE	MAPE	SMAPE
Cluster 5				
October 30th-31st	14.66	14.83	10.31 (-4,35)	10.22 (-4,61)
October 28th-29th	14.03	13.87	5.63 (-8,4)	5.48 (-8,39)
October 26th-27th	16.73	17.63	19.72 (+2,99)	17.1 (-0,53)
June 29th-30th	15.89	15.85	14.62 (-1,27)	16.56 (+0,71)
Cluster 19				
October 30th-31st	30.75	32.63	43.16 (+12,41)	34.69 (+2,06)
October 28th-29th	21.99	23.26	17.67 (-4,32)	15.61 (-7,65)
October 26th-27th	20.53	19.52	23.87 (+3,34)	20.44 (+0,92)
June 29th-30th	20.58	19.19	11.19 (-9,39)	12.05 (-7,14)

See Table 6.6 for the performance results for the zero-shot and fine-tuned versions. The finetuning improved the MAPE by 1.12% on average and the SMAPE by 3.07%. Illustrating that the retraining of the Lag Lama model was successful and worth the extra training effort. The Lag Llama model outperformed all other approaches for the use case of Cluster 5 from October 28th

to 29th. It reached a MAPE of 5.63% and SMAPE of 5.48%. See Figure 79 for a visualization of the results, including the confidence intervals.

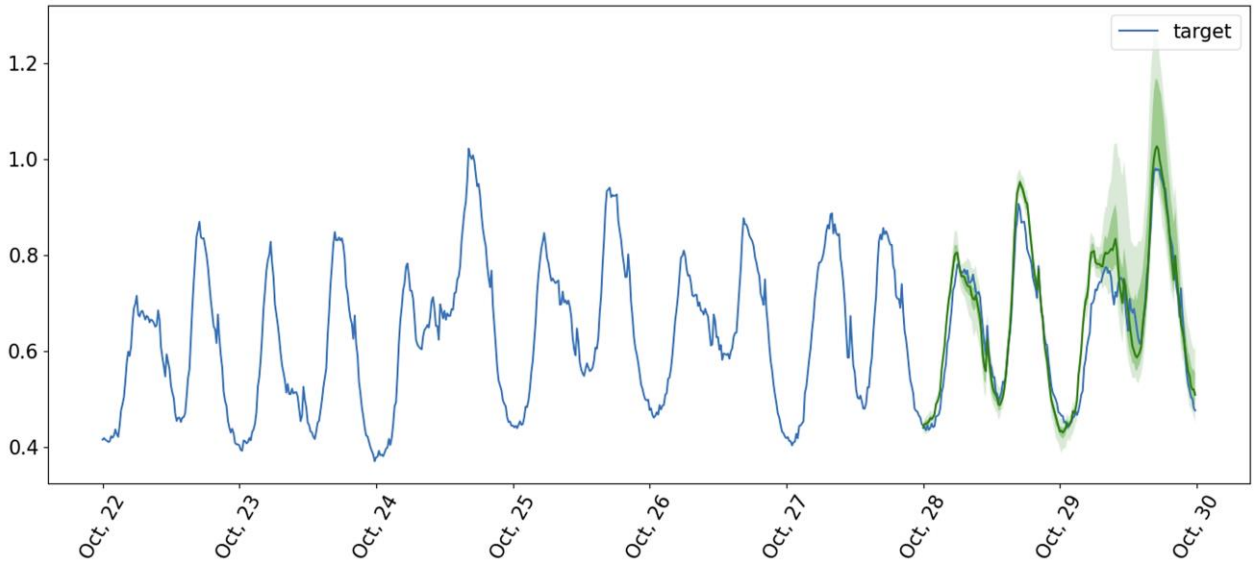

Figure 79 Lag Llama forecasting results for Cluster 5, October 28th to 29th, including confidence intervals (green)

6.10 AutoGluon

We use the official AutoGluon release [117] to apply AutoGluonTS. The framework requires the input data to be provided in the *autogluon.timeseries TimeSeriesDataFrame* type (see Figure 80).

```
complete_data_5_oct_26_27 = TimeSeriesDataFrame.from_data_frame(
    pd.concat([cluster_5_oct_source_train_1, cluster_5_oct_source_test_1], ignore_index=True),
    id_column="TRAFO",
    timestamp_column="READ_TIME"
)
```

Figure 80 TimeSeriesDataFrame type required by AutoGluonTS

It requires a timestamp column and a unique identifier for the individual time series, as it supports univariate and multivariate forecasts, to be defined in the parameters.

```
predictor_5_oct_26_27 = TimeSeriesPredictor(
    prediction_length=192,
    path="autogluon_5_oct_26_27",
    target="CONSUMPTION",
    eval_metric="SMAPE",
    freq="15T", #=15min interval data
    known_covariates_names=["HOUR", "DAY", "MONTH", "DAY_OF_WEEK"]
)

predictor_5_oct_26_27.fit(
    train_data_5_oct_26_27,
    presets="best_quality"
)
```

Figure 81 AutoGluonTS TimeSeriesPredictor parameter definition and fitting

In the next step, the predictor model must be set up, providing the data frequency, the target value, the evaluation metric to be used, and the known covariates that will be used (see Figure 81). The model is trained with the *Best_Quality*-fit, including the following models: SeasonalNaive, RecursiveTabular, DirectTabular, CrostonSBA, NPTS, DynamicOptimizedTheta, AutoETS, AutoARIMA, Chronos, TemporalFusionTransformer, DeepAR, PatchTST. After fitting, AutoGluonTS offers a convenient way to display the score leaderboard for all models fitted to the training data. Figure 82 depicts the score leaderboard for our prediction test run. The results are ordered by their score, ranging from best to worst. *Pred_time_val* depicts the time needed for inference, whereas the *fit_time_marginal* illustrates the time needed to train the respective model. Our case indicates the complexity and computational effort required for the Temporal Fusion Transformer and the DeepAR model. *Fit_order* illustrates the order in which the models are built and does not contribute in any other way.

	model	score_val	pred_time_val	fit_time_marginal	fit_order
0	WeightedEnsemble	-0.088997	0.143538	3.750497	13
1	PatchTST	-0.099535	0.042775	68.580393	12
2	RecursiveTabular	-0.110679	4.191140	20.060365	2
3	AutoARIMA	-0.122494	77.317367	46.335011	8
4	Chronos[base]	-0.126334	4.919766	14.832967	9
5	SeasonalNaive	-0.130799	1.084268	1.515686	1
6	TemporalFusionTransformer	-0.151406	0.056461	121.653022	10
7	DynamicOptimizedTheta	-0.151934	27.736756	28.007954	6
8	DirectTabular	-0.168668	0.100762	2.726562	3
9	DeepAR	-0.185312	0.978968	83.957987	11
10	NPTS	-0.231018	3.702806	3.492933	5
11	CrostonSBA	-0.275092	9.427228	9.892904	4
12	AutoETS	-0.410792	18.969099	19.030055	7

Figure 82 AutoGluonTS score leaderboard

6.11 TimeMixer

For the TimeMixer implementation, we again used the NeuralForecast Python library, which offers a convenient way to build a training and test pipeline. See Figure 83 for the straightforward model setup, training, and prediction. Considering the very short training times, ranging between 2 to 4 minutes for 500 epochs, the solid results proved TimeMixer to be a viable alternative for time series forecasting. For the overall results, see Chapter 6.13. The use case for June for Cluster 5 was especially promising, with a MAPE of 7.63% and a SMAPE of 7.26% (see Figure 84).

```
horizon = 192
freq = '15min'

timemixer_model = TimeMixer(input_size=horizon,
                            h=horizon,
                            n_series=1,
                            scaler_type='robust',
                            batch_size=64,
                            max_steps=500,
                            early_stop_patience_steps=3)

models = [timemixer_model]

nf = NeuralForecast(models=models, freq=freq)
```

Figure 83 Time Mixer model setup using NeuralForecast

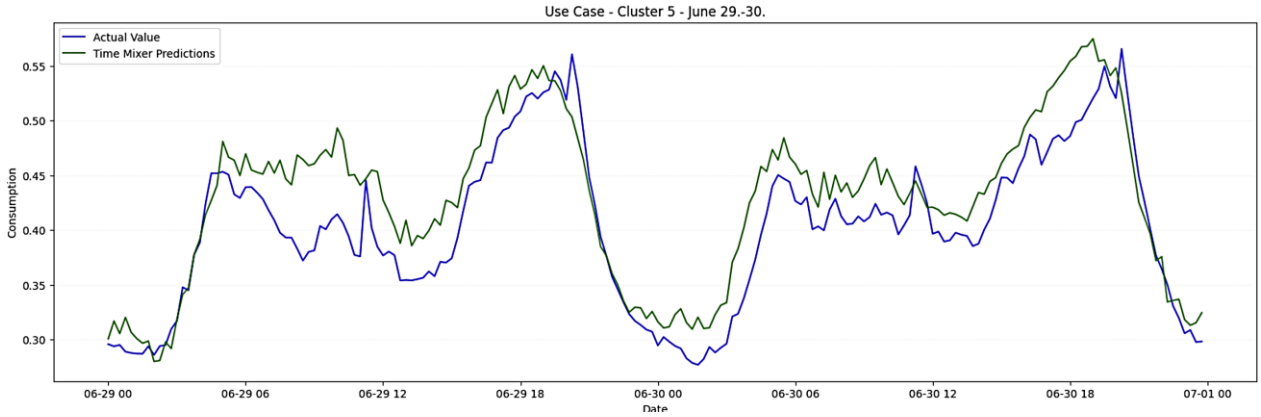

Figure 84 Forecasting results for cluster 5 for June using TimeMixer

6.12 MOIRAI

The pre-trained MOIRAI model is particularly strong in zero-shot performance. It is available in three different versions: the small, base, and large versions. The number of layers, attention heads, and respective parameters differ. We chose to work with the largest model size, comprising 24 layers, 16 attention heads, and 311 million parameters [62]. We implemented the model available in UNI2TS [118] in the release version 1.1.R for our implementation. The input dataset must be transformed into the gluonts format before it can be used (see Figure 85).

```
ds = PandasDataset.from_long_dataframe(
    complete_data,
    item_id="unique_id",
    target="y",
    freq="15min",
    timestamp="ds")

train_gluon, test_template = split(
    ds, offset=-TEST
)

test_gluon = test_template.generate_instances(
    prediction_length=prediction_length,  # time steps for each prediction
    windows=TEST // prediction_length,  # number of windows
    distance=prediction_length,  # number of time steps between each window
)

train_pandas_df = gluon_to_pandas(train_gluon)

forecasts_moirai_large = get_moirai_forecast('large', test_gluon)
moirai_forecast_df = moirai_forecasts_topandas(forecasts_moirai_large,
                                               forecast_col_name="moirai_forecast_large")
```

Figure 85 Data preparation setup for MOIRAI

The model displayed outstanding zero-shot performance for all use cases but performed best in the use cases for June scenarios for both clusters. It outperformed all other approaches tested (see Figure 86 and Figure 87). For the overall results, see Chapter 6.13.

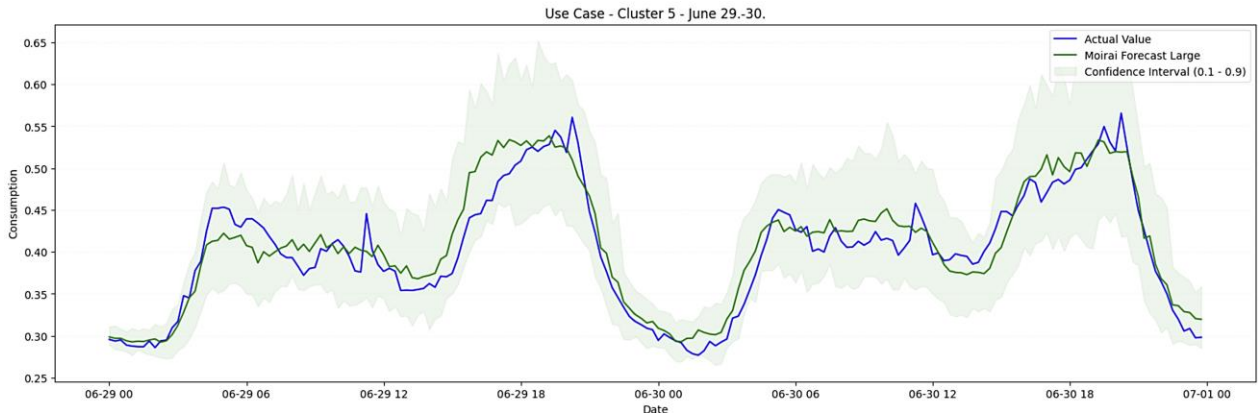

Figure 86 MOIRAI forecasting results for Cluster 5 for June

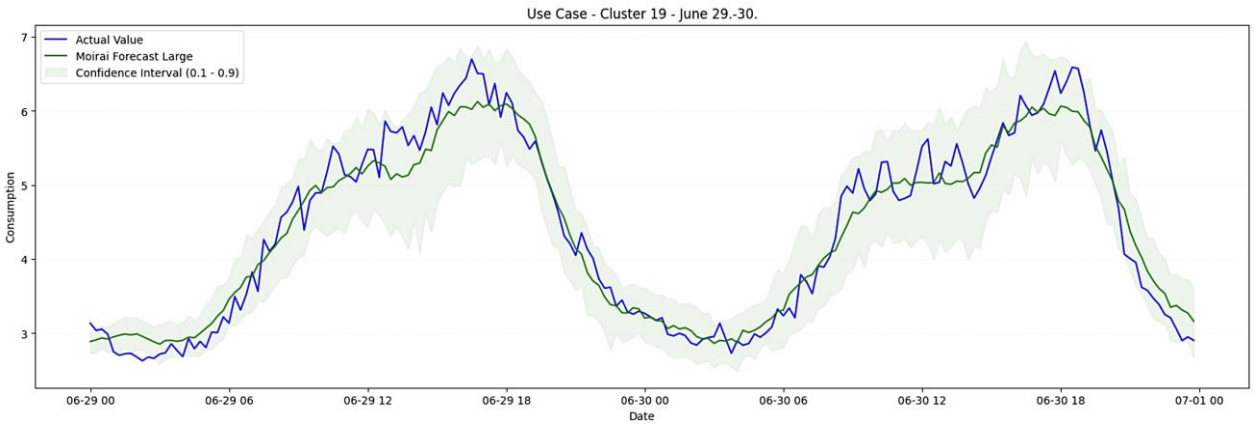

Figure 87 MOIRAI forecasting results for Cluster 19 for June

6.13 Evaluation Scenario Results

On common observation, the results for the typical cluster with consumption peaks during the winter were much more accurate than for the cluster with summer peaks. See Table 6.7 for the overall error for all use cases.

Table 6.7 Average MAPE and SMAPE for all models applied.

	October 26.-27.		October 28.-29.		October 30.-31.		June 29.-30.	
	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE
Cluster 5	11.95	11.75	10.6	10.64	14.84	15.94	9.14	8.85
Cluster 19	18.77	18.32	20.94	22.66	22.35	20.31	13.93	14.18

There seems to be high volatility in consumption in summer peak networks, making reliable forecasting hard. This is true for all methods and approaches used in this work. Another observation is that the energy consumption for a typical autumn weekend in a winter-peaked network, represented by our scenario for the 28th and 29th of October for cluster 5, could be forecasted very accurately, whereas holiday forecasts exhibited a higher percentage error. This effect is not visible for the summer-peaked network. Tables 6.8 and 6.9 display the overall

performance of the best methodology for all use cases. The best-performing approach per use case is highlighted. There is no clear, overall best method. Prophet and its successor, Neural Prophet, show good overall performance, but classic approaches like Kalman Filters also deliver very good results in certain scenarios. The pre-trained Large Time Series Model MOIRAI has remarkable zero-shot performance, making it a viable option for scenarios with no model training time or limited training data. While iTransformer and PatchTST each lead the ranking in one use case, the overall results of Temporal Fusion Transformers are remarkable, even if this method did not deliver the best results in any of the individual use cases.

Table 6.8 Evaluation Scenario Results for Cluster5. The best method per use case is highlighted.

	Cluster 5							
	October 26.-27.		October 28.-29.		October 30.-31.		June 29.-30.	
	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE
Sarima	16.83	16.77	18.27	16.84	22.13	21.6	14.44	13.49
Prophet	13.94	14.47	6.61	6.55	**6.51**	**6.61**	8.51	8.09
Kalman Filters	11.58	10.81	8.67	8.8	12.50	13.07	6.16	5.9
Neural Prophet	**7.21**	**7.15**	9.14	9.59	14.3	14.86	6.93	6.64
LSTM	15.04	16.08	19.91	18.63	24.32	28.95	7.47	7.17
iTransformer	10.94	10.50	10.53	10.73	16.05	17.59	9.91	9.33
PatchTST	12.43	11.64	7.46	7.22	13.78	14.29	7.83	7.42
TFT	9.53	10.41	7.03	7.27	9.28	9.50	12.43	11.64
Lag Llama	19.72	17.1	**5.63**	**5.48**	10.31	10.22	14.62	16.56
AutoGluonTS	9.41	8.94	9.93	10.68	16.54	18.65	9. 25	8.27
TimeMixer	8.43	8.6	11.31	11.77	14.31	15.59	7.63	7.26
MOIRAI	8.37	8.54	12.79	14.14	18.12	20.36	**4.55**	**4.49**

Table 6.9 Evaluation Scenario Results for Cluster19. The best method per use case is highlighted.

	Cluster 19							
	October 26.-27.		October 28.-29.		October 30.-31.		June 29.-30.	
	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE	MAPE	SMAPE
Sarima	31.12	35.65	22.28	22.96	23.56	21.48	28.54	35.94
Prophet	17.54	15.59	19.52	20.20	16.45	18.25	11.75	10.85
Kalman Filters	13.81	14.93	17.82	19.45	27.63	22.83	8.68	8.97
Neural Prophet	23.87	20.14	20.32	22.44	**11.18**	**11.7**	7.87	7.5
LSTM	17.13	18.41	34.62	39.72	17.51	19.66	25.69	26.41
iTransformer	21.42	18.4	**12.67**	**13.68**	29.62	24.23	8.63	8.25
PatchTST	17.18	15.37	16.55	18.59	29.86	24.52	7.4	7.04
TFT	13.85	15.24	17.63	15.58	13.02	14.24	7.29	7.21
Lag Llama	23.87	20.44	17.67	15.61	43.16	34.69	11.19	12.05
AutoGluonTS	**14.27**	**13.29**	23.8	28.28	14.99	13.93	32.78	27.50
TimeMixer	14.28	14.48	21.52	24.28	23.32	19.76	12.55	13.63
MOIRAI	16.92	17.99	26.95	31.23	18.00	18.46	**4.9**	**4.9**

We ranked the five best models for each use case to find the best models for each. The results are in Table 6.10. Temporal Fusion Transformers perform well in nearly all use cases while never leading the ranking for a single use case. This may illustrate their general applicability, where results are never outstandingly good but never disappoint, proving to be a good general method for all clusters with various characteristics.

Table 6.10 Five best-performing models per use case

	1st	2nd	3rd	4th	5th
Cluster 5					
Oct 26-27	NeuralProphet	MOIRAI	TimeMixer	AutoGluonTS	TFT
Oct 28-29	Lag Llama	Prophet	TFT	PatchTST	KalmanFilters
Oct 30-31	Prophet	TFT	Lag Llama	KalmanFilters	PatchTST

Jun 29-30	MOIRAI	KalmanFilters	NeuralProphet	LSTM	TimeMixer
Cluster 19					
Oct 26-27	AutoGluonTS	KalmanFilters	TimeMixer	TFT	PatchTST
Oct 28-29	iTransformer	TFT	Lag Llama	PatchTST	KalmanFilters
Oct 30-31	NeuralProphet	TFT	AutoGluonTS	Prophet	MOIRAI
Jun 29-30	MOIRAI	PatchTST	TFT	NeuralProphet	iTransformer

To evaluate the use cases' overall performance, we calculated the average MAPE and SMAPE, including their variance, to get a final overall evaluation result for all methods (see Table 6.11). We created a scatter plot to visualize the model performance concerning the differences in overall MAPE and SMAPE (see Figure 88). It also shows how close the leading architectures are to each other. Prophet's and Neural Prophet's performance is almost identical, and SARIMA and LSTM-based yielded poor results.

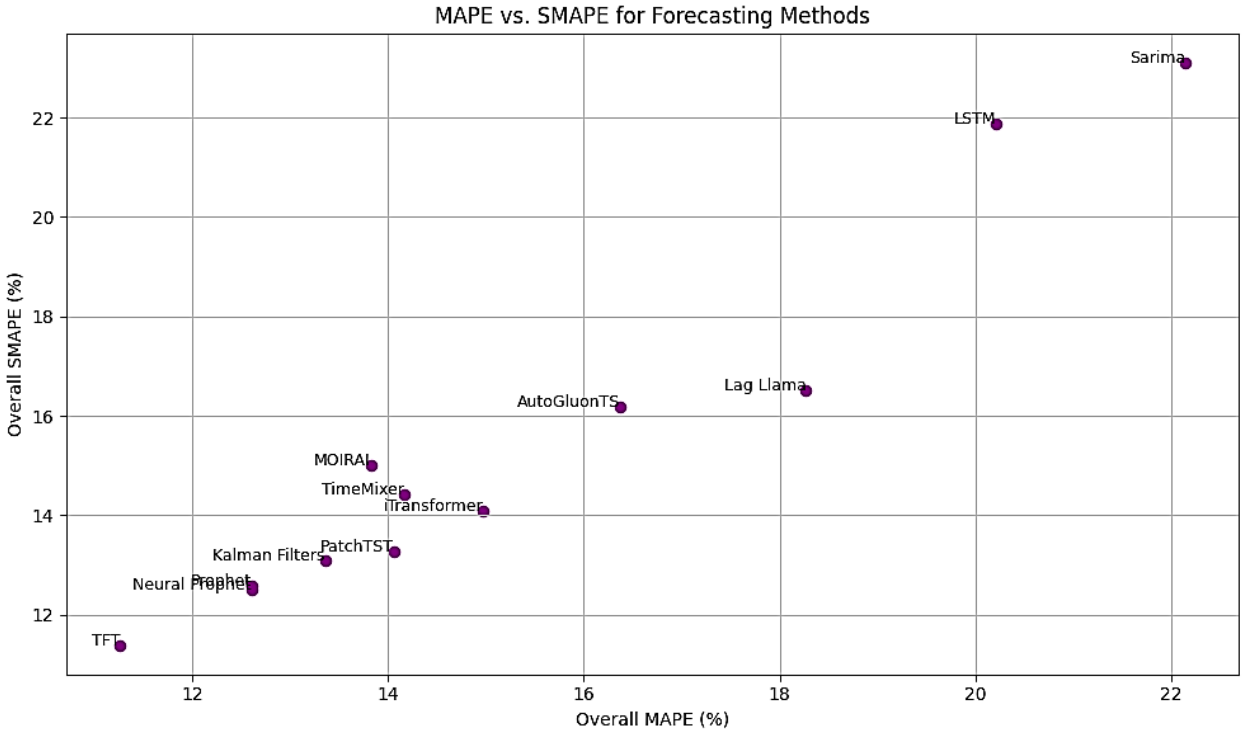

Figure 88 Overall MAPE and SMAPE results for all models

Table 6.11 Overall Performance Evaluation for all clusters and use case scenarios.

Rank	Method	Overall MAPE	Overall SMAPE	MAPE Variance	SMAPE Variance
1	TFT	11.25750	11.38625	13.188364	11.363255
2	Neural Prophet	12.60250	12.50250	41.009021	37.116136
3	Prophet	12.60375	12.57625	25.618112	28.285741
4	Kalman Filters	13.35625	13.09500	46.196198	32.996571
5	MOIRAI	13.82500	15.01375	59.175971	81.141998
6	PatchTST	14.06125	13.26125	56.434041	38.944212
7	TimeMixer	14.16875	14.42125	32.042841	31.413184
8	iTransformer	14.97125	14.08875	52.062155	30.679241
9	AutoGluonTS	16.37125	16.19250	67.349413	62.757479
10	Lag Llama	18.27125	16.51875	134.273527	75.461984
11	LSTM	20.21125	21.87875	65.781927	95.189612
12	Sarima	22.14625	23.09125	32.299884	71.267155

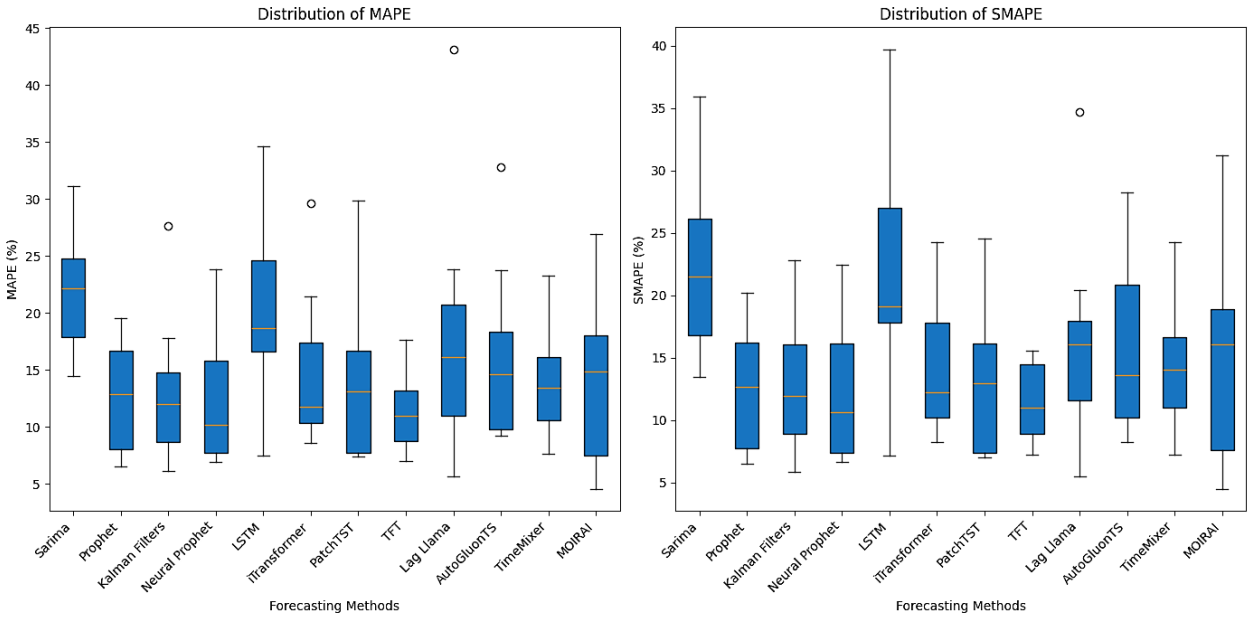

Figure 89 Box plots for the model performance regarding MAPE and SMAPE

Figure 89 illustrates box plots for the individual model performances. The high variance in the LSTM results illustrates that the performance was very unstable. Lag Llama has similar behavior with extreme outliers, so the model's overall performance was not good, even though some use case results were outstandingly good. Temporal Fusion Transformers, Prophet, and Neural Prophet exhibited stable performance without outliers and relatively low variance. Based on this evaluation, we will proceed with TFT and Neural Prophet to perform a two-day forecast for the remaining clusters. Because Prophet's and Neural Prophet's performance was similar, we would only choose the latter for the final evaluation step on all clusters.

7.Performance Evaluation on Network Clusters

We use the two best-performing architectures to forecast a new evaluation scenario for all network clusters. In the final evaluation step, the performance of the trained models is evaluated based on the individual low-voltage network energy consumption time series, as opposed to the overall cluster result. This verifies their applicability on the individual networks and, in the best scenario, proves our clustering approach successful. The use case for the final evaluation is in November 2023. We forecast the timeframe from 13.11.2023 to 14.11.2023 with roughly a month prior as training data, starting on 16.10.2023. Temporal Fusion Transformers and Neural Prophet both support multivariate time series. This is a major advantage, for we can model the individual low-voltage network consumption amounts as separate time series without creating averages over all networks and breaking it down to the individual networks for the prediction scenario again. Naturally, this comes with the downside of higher training times.

7.1 Cluster Results for Neural Prophet and TFT

We trained a separate Neural Prophet and Temporal Fusion Transformer model for each cluster and, based on the training data, performed a 48-hour forecast. See Appendix A.1 for details on the final model implementation.

Table 7.1 Overall Performance Evaluation for all clusters for Neural Prophet and TFT.

Cluster	Number of networks	Temporal Fusion Transformer		Neural Prophet	
		Overall MAPE	Overall SMAPE	Overall MAPE	Overall SMAPE
0	76	**29.68**	**27.15**	35.32	31.59
1	535	**39.26**	**34.50**	43.22	38.53
2	135	**37.02**	**34.65**	51.91	43.03
3	258	**30.35**	**28.29**	31.37	30.31
4	42	**22.93**	**21.80**	34.24	27.47

5	824	**51.87**	**42.58**	54.50	47.96
6	14	**19.96**	**21.03**	24.36	23.08
7	803	**53.33**	**47.42**	66.06	60.10
8	178	**27.42**	**25.95**	31.51	29.16
9	471	**45.63**	**41.67**	47.40	41.26
10	40	**20.86**	**20.11**	21.82	22.36
11	107	29.36	29.27	**27.73**	**27.19**
12	334	**31.39**	**29.22**	34.93	32.07
13	35	**25.69**	**24.46**	43.18	35.80
14	68	**25.51**	**24.09**	27.14	24.89
15	387	**35.47**	**32.52**	36.38	34.34
16	90	26.16	24.96	**26.42**	**26.02**
17	5	**15.74**	**14.83**	19.09	17.68
18	6	**14.56**	**13.79**	54.82	38.66
19	14	**37.14**	**35.65**	40.10	38.84
20	4	**38.06**	**26.74**	54.81	41.11
Overall		**30.35**	**30.03**	38.25	33.87

See Table 7.1 for the overall MAPE and SMAPE results for both methods. This time, we must consider that we train one specific model for a whole cluster and then use this model to perform individual forecasts for each low-voltage network in that cluster. As expected, the overall MAPE and SMAPE is higher than in the initial method tests, where we only forecasted a univariate case. The results for the TFT are slightly better, with an overall MAPE of 30.35 and an overall SMAPE of 30.03, compared to 38.25 and 33.87 for the Neural Prophet models. Only in 2 cases did the Neural Prophet model outperform the Temporal Fusion Transformer models.

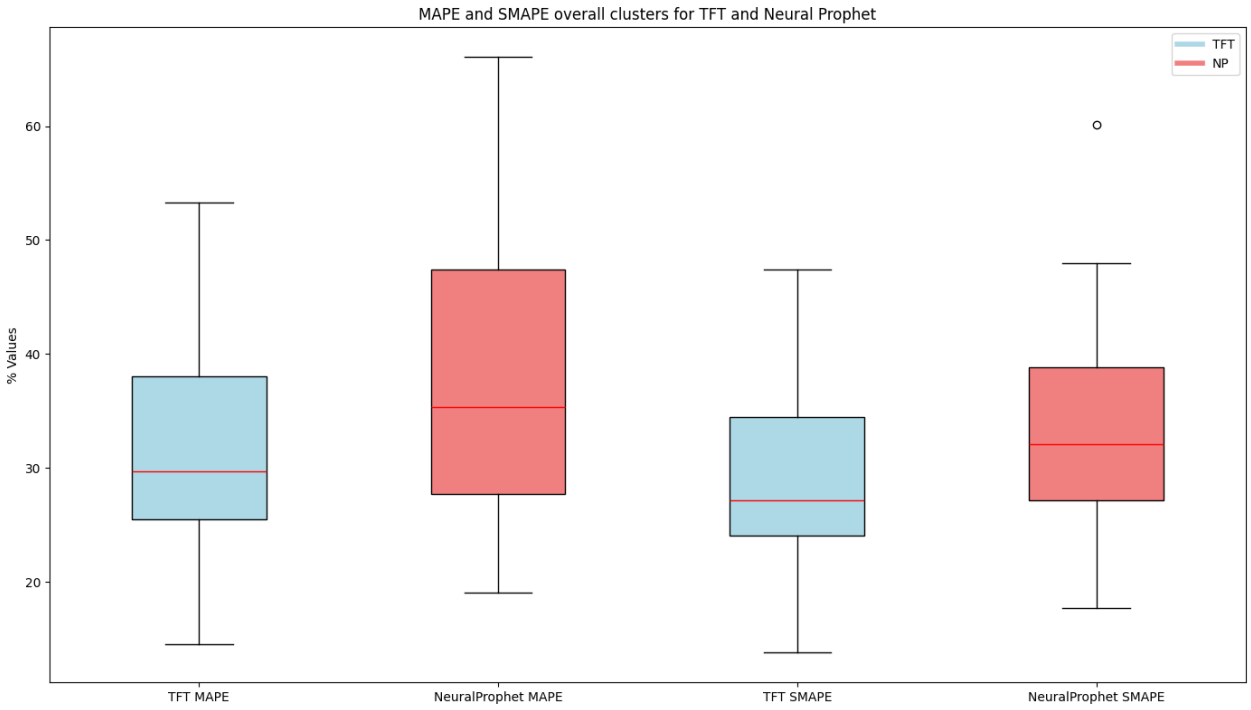

Figure 90 Box plots for overall results of MAPE and SMAPE for TFT and Neural Prophet

See Figure 90 for a box plot of both methods' overall results for MAPE and SMAPE. The median value for the TFT models is lower for both metrics. The TFT models generate more stable results, but both models create acceptable results for most networks and prove that our clustering approach successfully combined networks with similar characteristics. However, the results for smaller clusters appear to be generally more accurate and indicate that a higher number of clusters and even higher specialization may be required if the results need improvement. To investigate the relationship between cluster size and forecasting accuracy, we created a scatter plot that visualizes the MAPE and SMAPE results for every cluster as a function of the number of low-voltage networks within the cluster (see Figure 91). The plots show a positive correlation between cluster size and forecasting error, although there are some outliers with large forecasting errors but small cluster sizes.

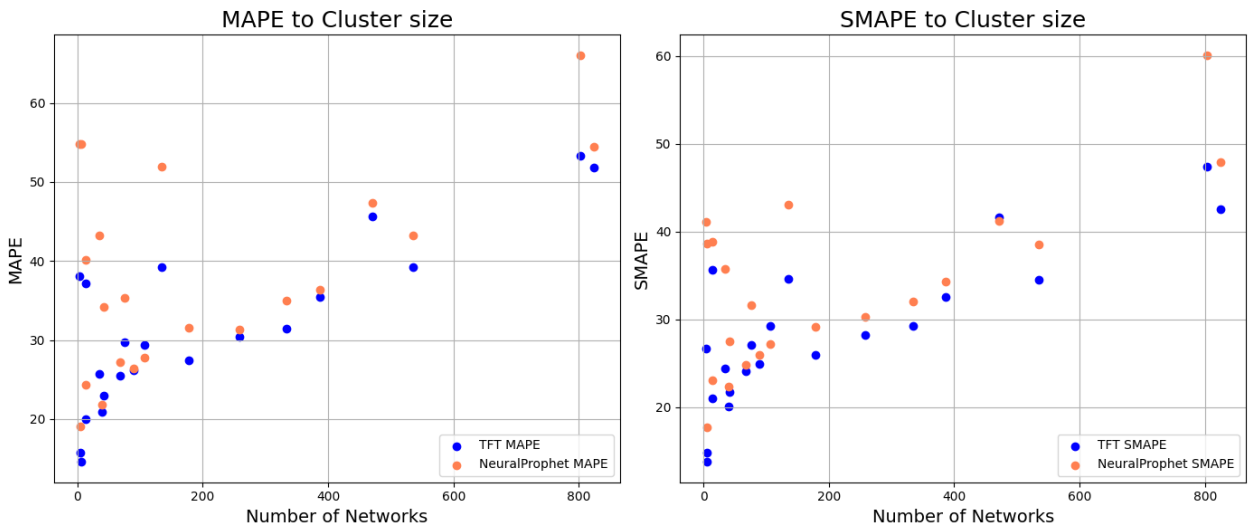

Figure 91 Forecasting accuracy vs cluster size

In the next step, we examine the detailed results for the individual networks within the clusters. In some cases, poor performance in individual networks highly influences overall results. Investigating the variance of individual network results within a cluster might be interesting, especially for the bigger clusters with rather poor results. We intend to find out if these results are inaccurate due to outliers. If the overall results suffer mainly from the poor performance of outliers, removing those networks from the cluster and building specific models, especially for those networks, would make sense.

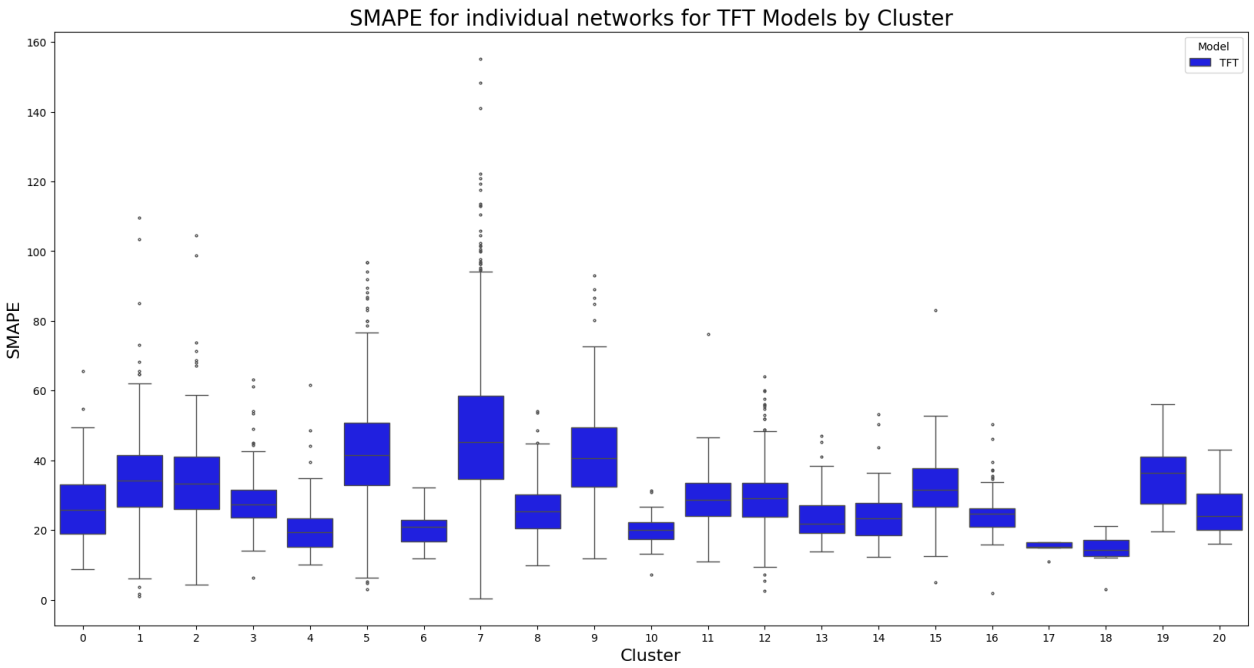

Figure 92 SMAPE for the individual networks per Cluster for TFT models

Figure 92 visualizes a box plot of the individual cluster results for the SMAPE. While some models perform very well with little to no outliers, others indicate that it may be useful to separate the clusters even further to account for the individual characteristics of the networks. Clusters 1, 2, 5, 7, and 9 are good candidates to be separated in the future, and for cluster 7, outlier detection and the training of individual models for 'problem' networks may be required.

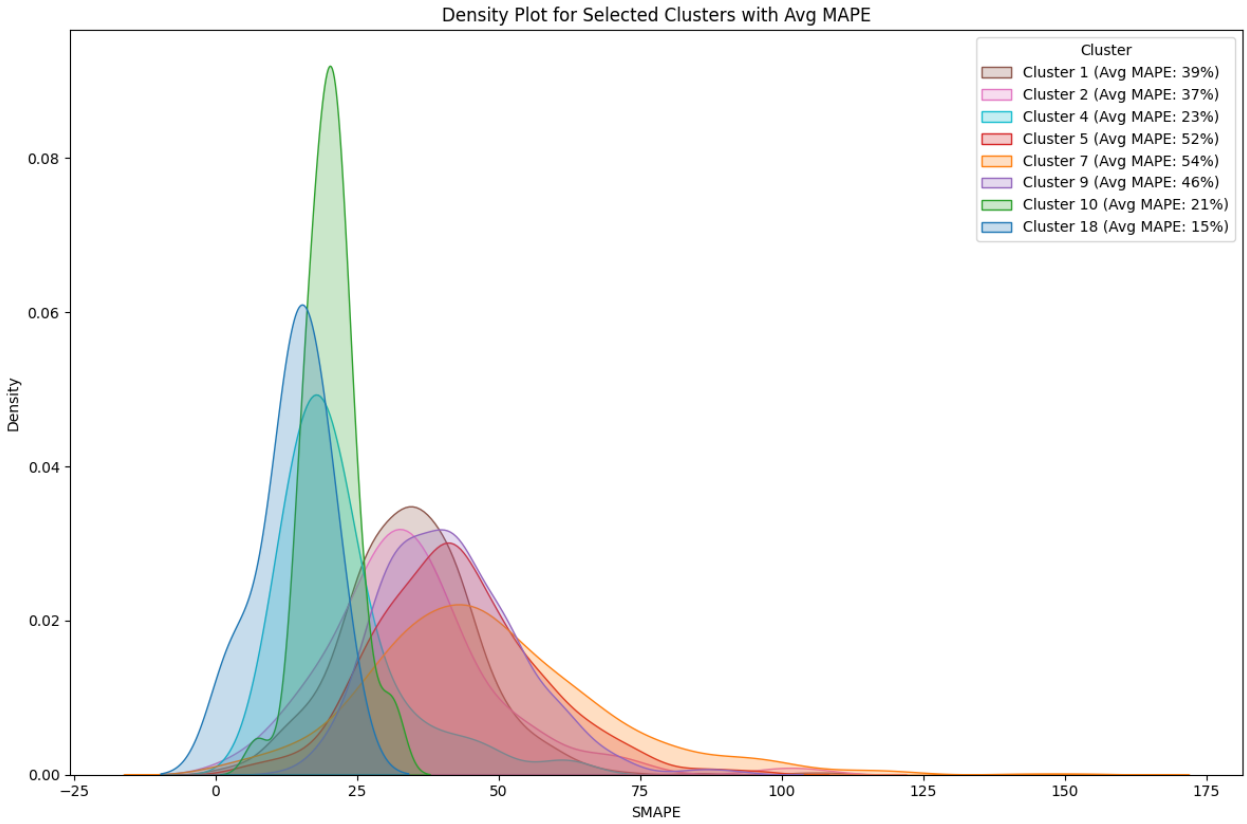

Figure 93 Density plot for best and worst performing clusters for TFT models

The density plots in Figure 93 illustrate the juxtaposition of very good cluster performance, such as those in clusters 4, 10, and 18, with poor cluster performance in clusters 1, 2, 5, 7, and 9. From the analysis point of view, examining the detailed results for cluster 7 will be most interesting, as it has the highest variance and contains very good and poor individual network results.

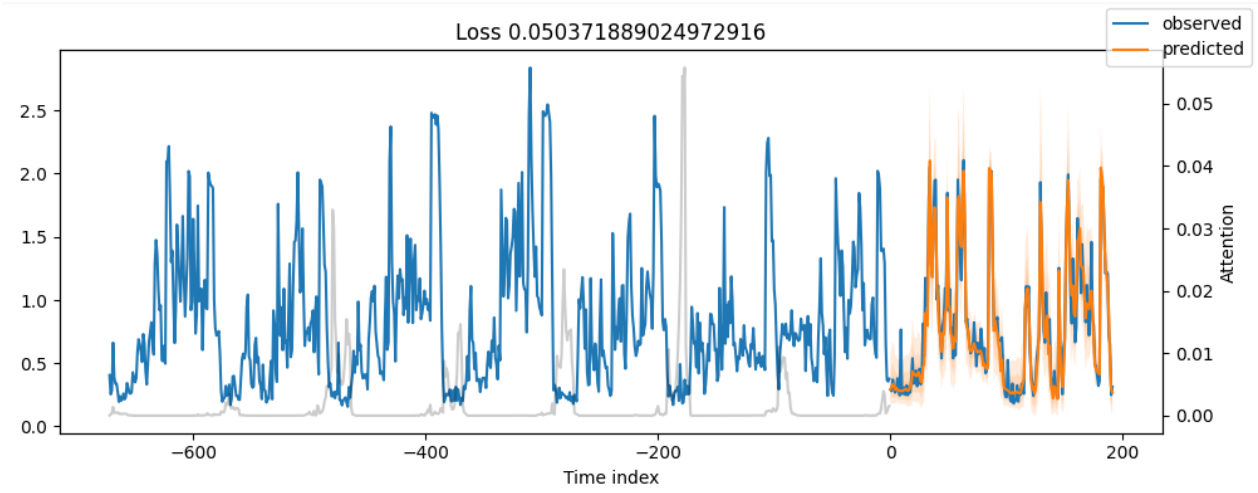

Figure 94 48h-forecast for a single network with MAPE 13.07%.

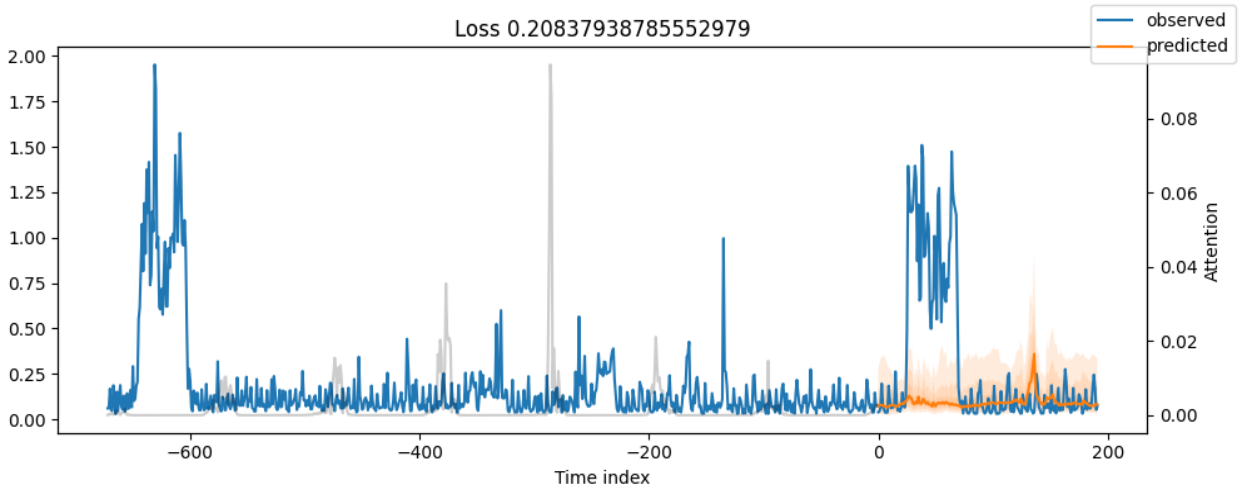

Figure 95 48h forecast for a single network with MAPE 75.65%.

See Figure 94 for an example of a very accurate forecasting result with an MAPE of 13.07%. The model appropriately captured the network's characteristics. Sudden and unexpected changes in the consumption profiles impose problems on the forecasting accuracy, as seen in Figure 95, where the model failed to forecast the peak during the first day of the forecasting interval, where the actual consumption amount far exceeded the predicted confidence intervals. Notwithstanding, the results are satisfying overall, and reducing models to the cluster number seems to be a promising approach.

7.2 Model Retraining Requirements

An important question in any real-world forecasting scenario is when a model that performed well in the past may be outdated and needs retraining. This is not because the model was inaccurate but simply because the ground truth, the real-world situation, has changed, and the model needs to be updated to reflect those changes. In our case, this could be attributed to different consumption behavior, a change in network topology, or a fast-growing network infrastructure. We continued to forecast until the end of 2023 using both models to determine when the forecasting performance decreased so that the required retraining could be performed.

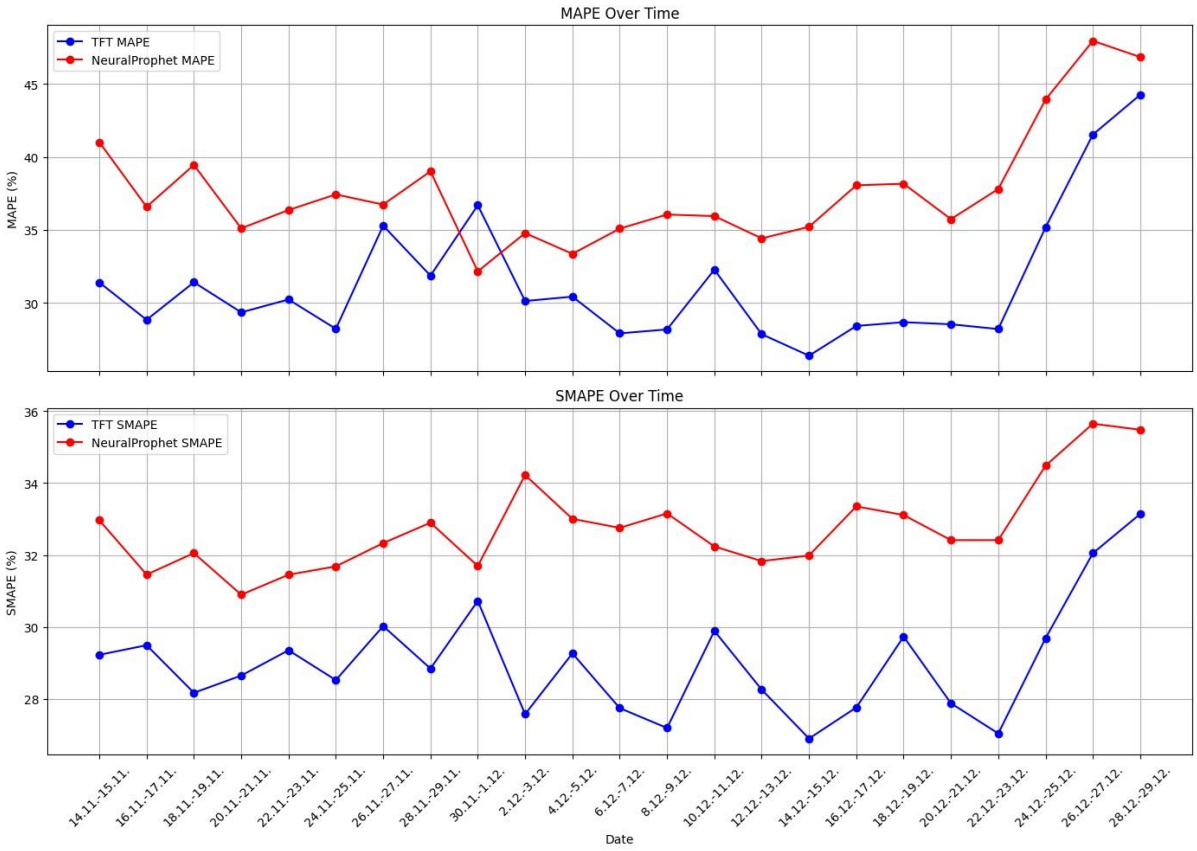

Figure 96 Continuous forecasting with the trained model for one month using TFT and Neural Prophet models

The forecasting results for a whole month are visualized in Figure 96. The MAPE and SMAPE results for TFT models are generally lower than for Neural Prophet. Both models' performance decreased by the end of the month. This could be because the end of December has unique consumption patterns, or models must be retrained after approximately a month. We believe both explanations account for the decrease at the end of the month. Consequently, the re-training effort is of interest when choosing a final forecasting approach, as models must be refined in a timely manner. Based on our consecutive forecasting tests, we suggest to retrain the models after one month.

Table 7.2 Comparison of training time and inference time for cluster 12 of average size.

Cluster	Number of networks	Temporal Fusion Transformer			Neural Prophet		
		Overall SMAPE	Training Time	Inference Time (48h)	Overall SMAPE	Training Time	Inference Time (48h)
12	334	**29.22**	10h 49min 36s	**3.85 s**	32.07	**13min 49s**	1min 5s

When training and especially re-training a forecasting model, one aspect to consider is the required training and inference time. Even though Temporal Fusion Transformers seem to deliver better overall results than Neural Prophet, the complexity of the architecture, which results in immense training effort, is a significant drawback of this method. See Table 7.2 for a direct comparison of the training and inference times for the TFT model and the Neural Prophet model for an average-sized cluster. We chose cluster 12, including 334 low-voltage networks, a typical average-sized cluster for our scenario. The forecasting results for both models are similar, but the training times differ immensely. It is very questionable if an decrease in SMAPE of 2,85% is worth a training time of approximately 11 hours compared to 13 minutes. We emphasize that the final decision for a particular model should not be solely based on performance but also on the required training time. In our use case, Neural Prophet can become a viable alternative when modeling performance for particular clusters decreases rapidly, and retraining becomes an immediate necessity.

8. Time Series Classification and Consumer Profile Disaggregation in Smart Meter Data

In this chapter, we first explore the current approaches to consumer profile disaggregation based on Smart Meter data. Then, we explore our two specific use case scenarios: identifying electric vehicle charging stations and detecting heat pumps within Carinthia's distribution network.

8.1 Current Approaches in Time Series Classification for Energy Profile Disaggregation

Electrical Load Disaggregation and Non-Intrusive Load Monitoring (NILM) handle the disaggregation of the total energy consumption measured by smart meters into appliance-specific consumption patterns. NILM helps identify various home appliances, their time of usage, and their respective energy consumption. Identifying household appliances via NILM is done in multiple ways, as described in [119]. It can be performed as a supervised [120],[121] and unsupervised approach [122],[123],[124]. Supervised NILM methods perform best on the smart meter data they were trained on, usually on buildings known in the training phase. This does not apply to transfer learning in unseen houses, making unsupervised methods an option. Nevertheless, our approach is supervised and semi-supervised because label information can be extracted from external sources, including customer contracts. The availability of current and voltage values makes the application of classical edge detection [125] or time-domain feature extraction [126] possible. The automatic feature extraction and segmentation of six common household appliances, such as coffee machines, fans, and fridges, has been studied in [126]. They are identified by their usage patterns using a novel hybridization of segmentation, time-domain feature extraction, and SVM-based machine learning algorithms. A similar deep learning-based approach for household appliance identification has been studied in [125]. Devlin et al. used a sampling rate of 10 seconds for household electricity consumption. One drawback of our available dataset is the lack of current values for analysis and the sampling rate of 15-minute intervals, making the application of these methods impossible. The requirement for basic labels in

household application identification led to initiatives like those in [127], where a database including 225 different appliances from 15 different categories of devices was made available to the public. The approach described in [123] applies to our use case. Kim et al. applied load disaggregation to low-resolution smart meter load profiles, incorporating weather information and using unsupervised methods to identify clusters of device types. The objective was to identify the profiles of so-called HVAC (heat, ventilation, and air-conditioning) devices in a so-called Independent Component Analysis (ICA). The available data had a 15-minute granularity, which is equal to our resolution. Building occupancy detection is another time series classification task based on Smart Meter data. In [128], the ABODE-Net deep learning model was proposed. It uses a parallel attention block for building occupancy detection using smart meter data. An LSTM- and CNN-based real-time occupancy detection algorithm was proposed in [129]. With the widespread adoption of Smart Meter infrastructures, there has also been a growing interest in the ability to classify households based on their electricity consumption data, paving the way for more informed decision-making and strategic planning in the energy sector. This classification of individual households into separate clusters is analyzed in [124], [130], [131], [132], [133]. While many other approaches exist, the ones mentioned here try to predict specific household characteristics, including demographic information, household income, and cooking habits from their everyday electricity consumption. While this may raise privacy concerns, energy providers and distribution network operators argue that this information enables the development of intelligent business applications and can help consumers reduce their energy consumption [134].

8.2 Smart Meter Data-Based Consumer Profile Identification

This chapter explores the time series classification tasks for identifying the network's electric vehicle charging stations and heating pumps. We examine the label generation using external data sources from KNG's Enterprise Resource Planning and Customer Information systems. We then decide on an appropriate model approach for the classification tasks and evaluate the overall results as the final step.

8.2.1 Label Preparation

A major challenge of supervised learning-based methods is the scarcity of labeled data sets, as device-level load profiles are rarely available. Due to the available customer energy contract information, we can work with a dataset where a specific signal in the profile is guaranteed. However, the disaggregation task is still required to separate it from other household appliances active in the same timeframes. The initial timeframe that is available covers November 2023 to February 2024. The following sections examine the respective sources for this time frame. After label engineering and preparation, we reduce the time frame used for classification to a suitable interval, where the different methodologies can be applied with a trade-off between accuracy and computational effort.

8.2.2 Prerequisites for Classification Tasks

We had to extract information about existing electric vehicle charging stations and heating pump contracts from the ERP and Customer information systems to create meaningful labels. We also had to create negative labels for charging stations and heat pumps. The natural choice to do this was selecting consumption profiles where attaching or building heat pumps would be impossible. This included typical city apartment buildings and single-phase meters inside such buildings. We selected and subsequently extracted the consumption profiles for 213 meters. These will be used as negative label sets for both scenarios.

8.2.3 Electric Vehicle Charging Stations Labelling

Since January 2024, there has been an obligation to register vehicle charging stations regardless of their size. Before January, registering devices below the power of 11 kW was possible but not mandatory. We extracted a list of 533 known charging stations all over Carinthia. The data included the owner's name and customer identification, the date of completion, the address information, and the associated low-voltage network to which the station is attached. In the first step, we had to merge that information to find the respective Smart Meter in the MDM System to find the respective consumption patterns. The second step included a filtering stage where only Smart Meters with a 15-minute interval configuration were selected. This restriction was necessary to work with meaningful high-resolution patterns. A single daily consumption value would not be sufficient for our classification task. The resulting positive label set for electric

vehicle charging stations includes 170 positive labels. Because the mere existence of a registered charging station does not guarantee actual loading cycles within our observation timeframe, we decided to continue with a further label engineering and validation step.

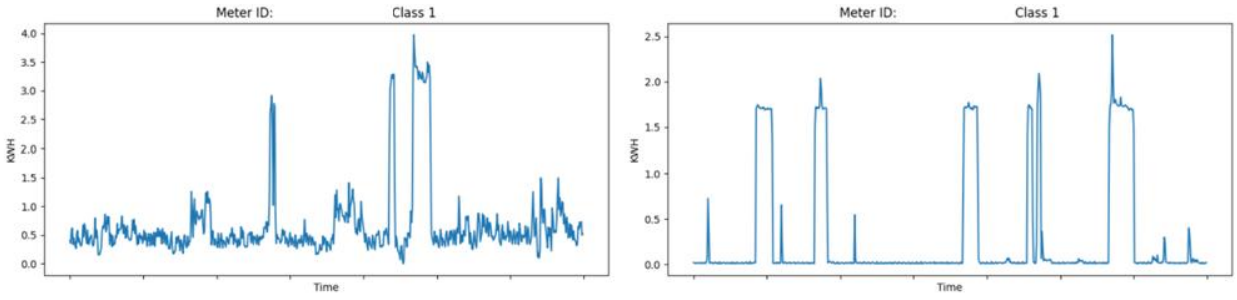

Figure 97 Time series with positive charging station labels depicting typical peaks

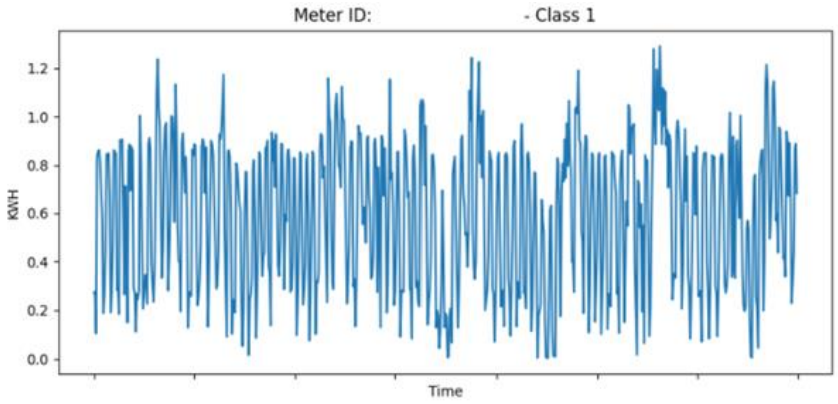

Figure 98 Time series with positive charging station label without clear charging cycles

Some time series exhibit clear charging patterns with their associated peaks on visual inspection (see Figure 97), whereas others lack those clear indicators of loading cycles (see Figure 98). This may be due to existing batteries at the customer's site, or the charging station not being used regularly. Nevertheless, we must remove such time series from the positive label set to ensure accurate learning for our models. We first used automated feature extraction for weekly observation patterns to remove ambiguous time series from the positive label set.

```
features = []
for meter_id in meters_of_interest:
    meter_data = filtered_df[filtered_df['METER_BADGE_ID'] == meter_id]['LP_VALUE_KWH']
    max_val = meter_data.max()
    peak_to_peak_val = meter_data.max() - meter_data.min()
    relative_distance = peak_to_peak_val / (meter_data.max() - meter_data.min()) if meter_data.max() != meter_data.min() else 0
    height_threshold = meter_data.mean() + 2 * meter_data.std()  # threshold: mean + 2*std
    peaks, _ = find_peaks(meter_data, height=height_threshold) # from scipy.signal import find_peaks
    num_peaks = len(peaks)

    is_outlier = 1 if num_peaks < 1 else 0

    features.append([meter_id, max_val, peak_to_peak_val, relative_distance, num_peaks, is_outlier])
```

Figure 99 Feature extraction for outlier detection

The features we extracted for all the time series were the maximum peak, the distance between the maximum and minimum peak of the time series, the relative distance between the maximum and minimum peak to account for different peak heights, and the overall number of peaks in the observation timeframe, which we set to at least one (see Figure 99). For this, we used the *scipy.signal find_peaks* method. The height threshold was calculated as the relative value of the series' mean value plus two times the standard deviation. After the feature extraction step, we scaled the data using the *MinMaxScaler*. We then used an Isolation Forest model with a default number of 100 trees to remove the outliers from the dataset. An Isolation Forest is a common method for anomaly detection. It works by selecting a random feature and then selecting a split value between the maximum and minimum values of the selected feature [135]. This process is repeated to create an isolation tree. The fewer splits required to isolate a data point, the more likely that it is an anomaly. As a result, 17 time series were removed from the positive label set as possible outliers. The final number of positive charging station labels was 157.

8.2.4 Heating Pump Labelling

Creating labels for the Heat Pumps posed more challenges than those for the EVC use case. There is no obligation for consumers to register their devices. Consumers can acquire a specific electricity supply contract if they own heat pumps, but this is not a requirement. In the first processing step, we extracted the customer information of customers with heat pump contracts and joined it with the information on the respective Smart Meter using the MDM System. In the next step, we reduced the list to the Meters with a 15-minute interval configuration. We then

merged the information about those existing heat pumps with the information on available feeder contracts for those customers. With solar devices or other electricity producers at the same measuring point, it seems likely that the consumption profile differs markedly between these two types of customers. Out of the 3315 original labeled time series, we could only use 1844 with no feeder information at the respective measuring point. We reasoned that the 1471 Smart Meter consumption profiles for heat pumps with additional energy feeders at the same measuring point would not provide expressive profiles that could be identified.

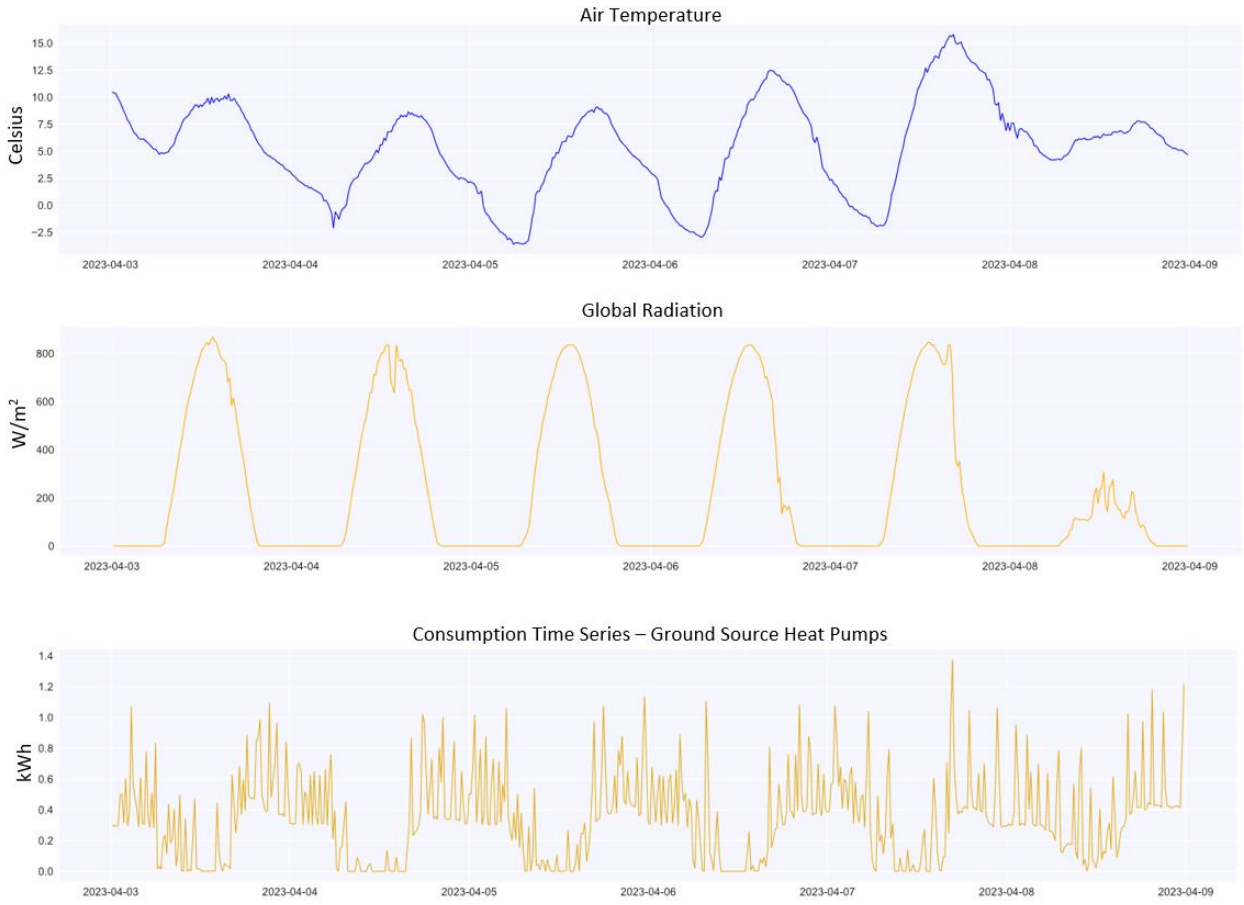

Figure 100 Air temperature, global radiation, and ground-source heat pump consumption profile for one week in April

Another highly influential aspect is the current weather situation. Heat pumps have different consumption profiles depending on the weather and air temperature. We added the weather

information to the respective profiles. The two parameters we chose were the air temperature and the global radiation (see Figure 100).

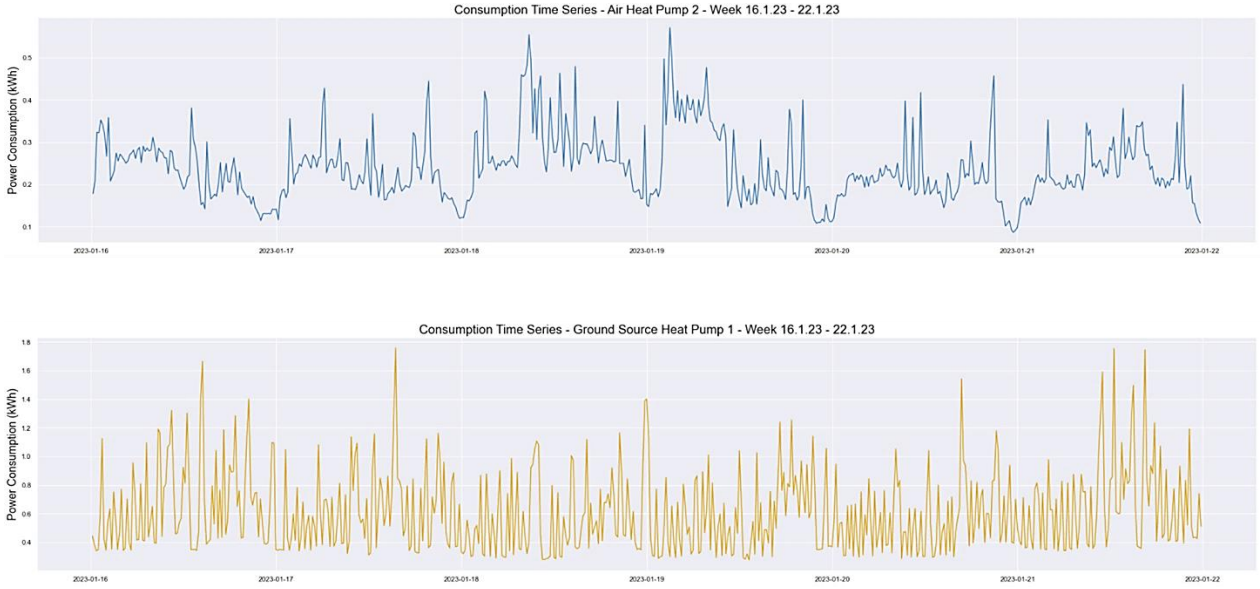

Figure 101 Ground-source vs. air-source Heat Pump Consumption Profile for one week in January

We also encountered a problem with the label generation for heat pumps. Two distinct types of heat pumps are available on the market. Ground-source and air-source heat pumps differ in their heat sources and operational methods. Ground-source heat pumps use the stable temperature of the ground or groundwater, whereas air-source heat pumps extract heat from the outside air. Air-source heat pumps are usually constructed using an internal inverter, while ground-source heat pumps use no inverters. Inverter and non-inverter heat pumps differ significantly in their energy consumption profiles due to the way they control the compressor's operation, which is the component responsible for regulating temperature. See Figure 101 for a visualization of a typical ground-source heat pump and an air-source heat pump for the same week in January. Whenever a customer requests a respective energy supply contract for heat pumps, they should submit the specification sheet of the device installed in the house. Still, there is neither a requirement nor a subsequent control process at the distribution network operator to determine if the data was submitted correctly. Moreover, the data sheets are only available for a small

portion of customers, and there is currently no automatic processing performed on those data sheets. A total of 985 installed heat pump data sheets are currently available in portable document format. The customer ID and the building ID identify the data. The data sheets contain information about the type of heat pump installed, but until now, they have not been systematically analyzed. We employed the *Pymupdf* Python library to parse the PDF files automatically and extracted distinguishing German words like “Erd”, “Grund”, and “Luft” to identify the type. The extracted information allowed us to create separate label sets for Ground-Source and Air-Source heat pumps. However, many of the analyzed attachments did not contain the required information: some were not proper data sheets, and others had limited readability. After merging the respective Smart Meter with the customer ID and building identifier, and merging it with the availability of 15-minute interval data, we had a label set of 213 negative labels, 5 Ground-source heat pumps, and 33 Air-Source heat pumps. This label set was not large enough to train a classifier, but those type-coded labels were useful for a semi-supervised approach. We had a very small number of labeled time series and many unlabeled ones, which was ideal for semi-supervised learning.

8.3 Time Series Classification Models

In this chapter, we employ our labeled EVC and heat pump datasets to create supervised and semi-supervised classification models. The methods are evaluated with regard to accuracy and computational efficiency.

8.3.1 EVC Classification

The consumption profile of a single household with an electric vehicle charging station has a unique pattern, featuring prominent peaks with a certain regularity.

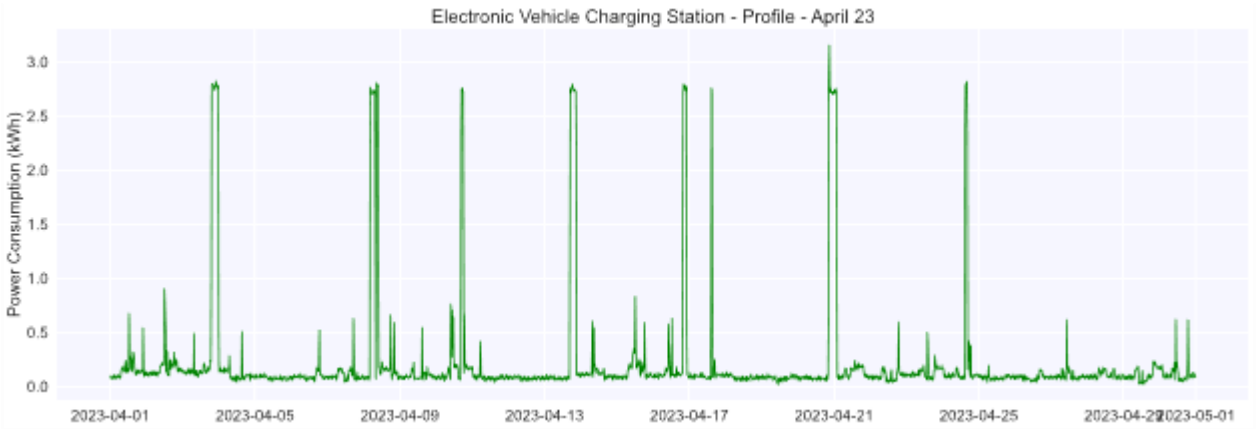

Figure 102 Typical EVC consumption pattern

Figure 102 illustrates a typical consumption pattern for such a consumer. Based on visual inspection, we assumed that feature extraction methods, in combination with easy classification algorithms applied to those features, would yield good results. To employ more sophisticated approaches, we also tried *Tslearn* Shapelet models to determine if the unique charging cycle shape could be identified in a complete consumption curve. Because we are only interested in the presence of an EVC station, we must distinguish between positive and negative cases to compare with the registered stations. It is a simple binary classification problem.

8.3.1.1 Feature Extraction and Random Forest Classifier

Our first approach includes feature extraction and a subsequent classification algorithm. The features we extract for the observation timeframe are the minimum peak value, maximum peak value, peak-to-peak ratio between minimum and maximum, and the series' mean. After feature extraction, we applied a Random Forest classifier, including 500 trees, to perform a binary classification. The overall results of the model were very promising. Considering the computationally simple approach, applying these methods in future implementations is reasonable.

Table 8.1 Evaluation metrics for feature extraction and Random Forest Classifier

	Precision	Recall	F1-Score	Accuracy	Support
Class 0	0.88	1	0.94		44
Class 1	1	0.79	0.88		29
Overall	0.94	0.9	0.91	0.92	

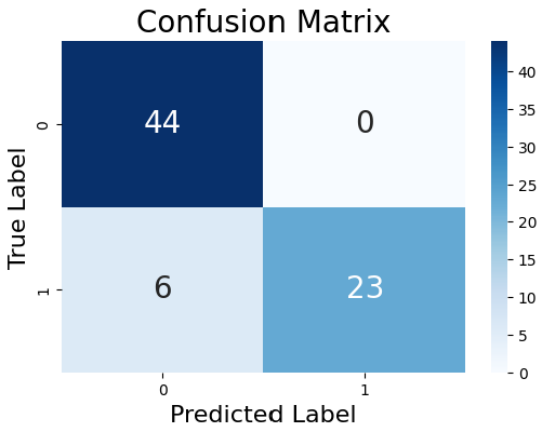

Figure 103 Confusion matrix for the feature extraction and Random Forest Classifier

Table 8.1 presents the model's overall results. The confusion matrix (see Figure 103) indicates that no negative cases were predicted as positive ones, corresponding to a false positive rate of 0. Six instances of false negatives were present, indicating that a positive label was not recognized as positive. We will closely examine some of these misclassifications.

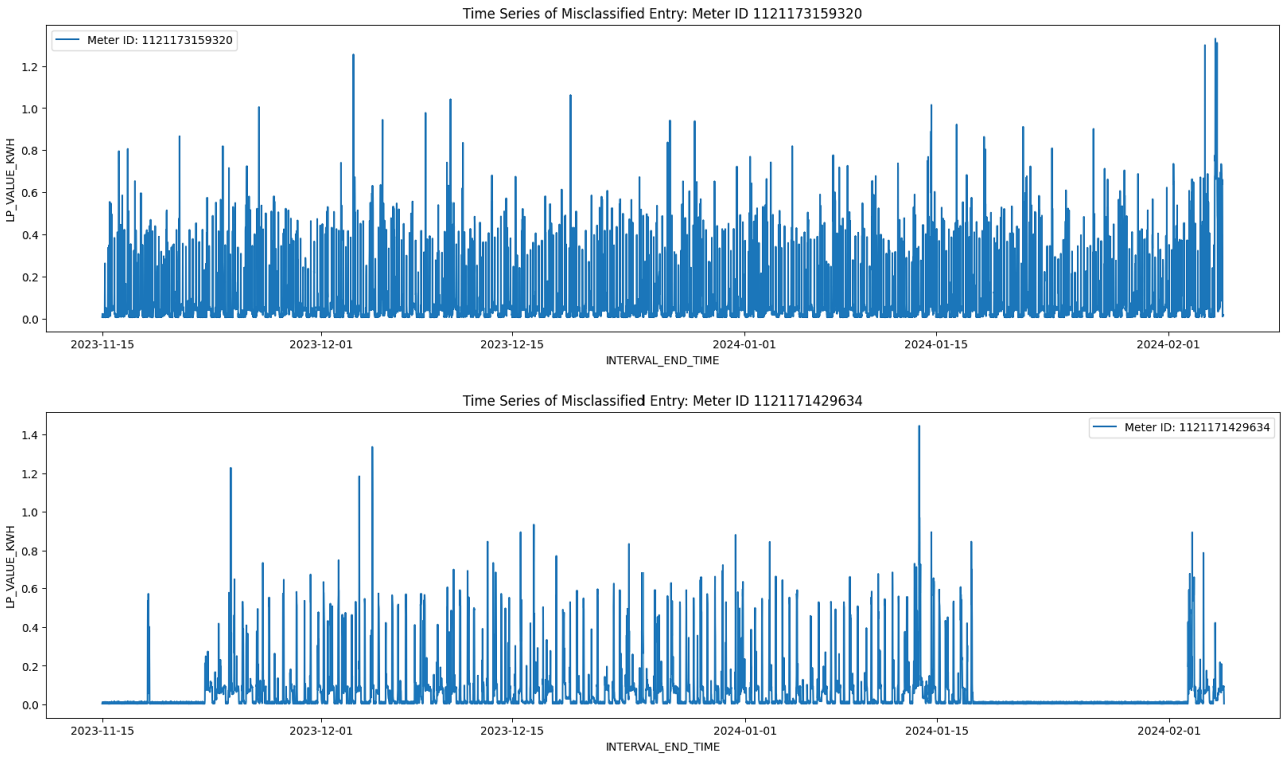

Figure 104 False negatives for the Random Forest Classifier

See Figure 104 for the illustration of two misclassified examples. No significant peaks are present in the observed timeframe. In some instances, including the second case of Figure 104, the building seemed unoccupied, resulting in no relevant power consumption. Another explanation for the lack of visible charging cycles might be the presence of batteries and solar devices on the customer's site. Those appliances will flatten the overall consumption and feed peaks using the available power from the batteries. For consumers like that, an accurate classification of a charging station may not be possible with the available data.

Another important aspect is the explainability of the model's results. Feature importance gives us a clear view of the most important features in retrieving the prediction results.

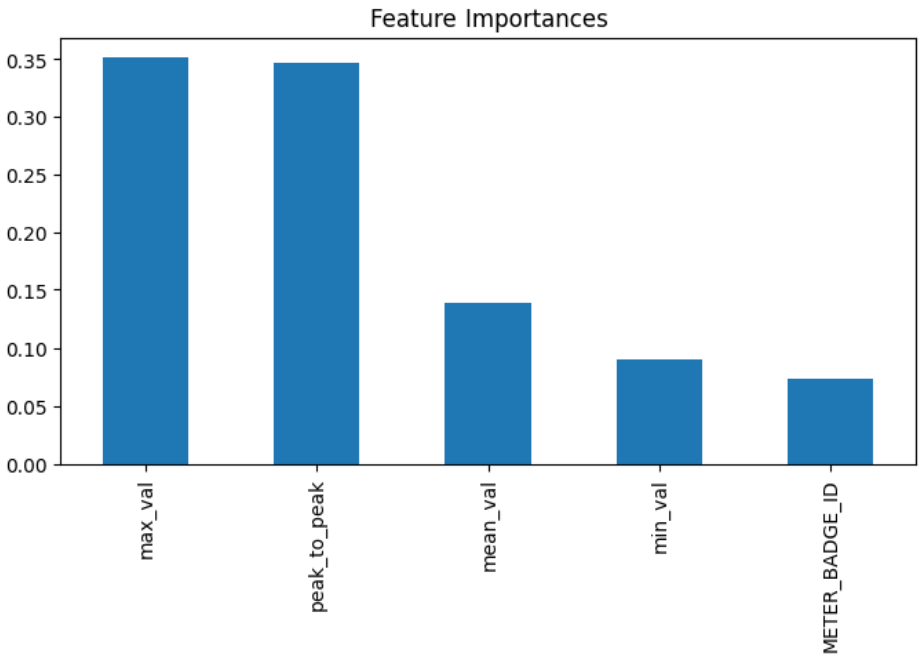

Figure 105 Relative feature importance for the Random Forest Classifier

Figure 105 depicts the relative feature importance of the Random Forest Classifier. The maximum peak and the maximum to minimum peak ratio are the most important features in the classifier's decision-making. Fortunately, the identifier, also part of the dataset, was irrelevant. It is a highly desirable result, as it carries no valuable information besides making the time series identifiable. Given the accurate results and the straightforward approach, it is reasonable to use feature extraction and a simple classifier such as the Random Forest Model for future implementations.

8.3.1.2 Tslearn Shapelet Model

A second, more sophisticated approach we tried on the source data is based on the *Tslearn* Shapelet model [136]. Shapelets are highly discriminative subsequences of time series that enable a classifier to distinguish between classes. They are defined as being maximally representative of a class [137]. Shapelets are extracted from the training data. Their distance to a specific time series is then used for classification [138]. The best possible matching of a Shapelet to a time series is depicted in Figure 106.

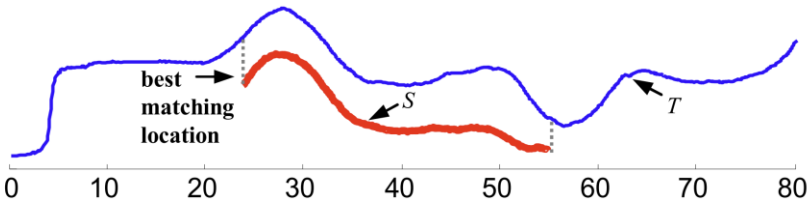

Figure 106 Matching of Shapelet *S* in time series *T* [137]

```
# Define the shapelet model parameters
shapelet_sizes = grabocka_params_to_shapelet_size_dict(
    n_ts=X_train.shape[0], # number of timeseries to be searched
    ts_sz=max_ts_length,  # length of the longest time series
    n_classes=2, # number of classes
    l=0.000001,  # percentage of the length of time series to be used for shapelets
    r=40  # sampling rate - number of times the shapelet is searched for in the data
)

# Initialize ShapeletModel
shp_clf = ShapeletModel(n_shapelets_per_size=shapelet_sizes, optimizer="sgd", weight_regularizer=0.01, max_iter=200, verbose=1)
```

Figure 107 Tslearn Shapelet model parameters

Figure 107 depicts the mandatory parameters of the Tslearn Shapelet model. We used a classical *GridSearchCV* model to find the best parameter settings for our classifier. The parameter *l* represents the percentage of the time series length used for Shapelets. That way, it controls the length of the Shapelets relative to the length of the time series. This parameter impacts the model's ability to capture meaningful patterns. Too small a value might make the Shapelets too specific, whereas too large a value may make them too general. The parameter *r* defines how often the Shapelets should be sampled in the time series. A higher sampling rate means more Shapelets are considered. This may lead to better generalization but also increased computational complexity. The best *r* and *l* parameter settings were 60 and 0.000001, respectively. The overall performance of the best Shapelet Model was inferior to that of the Random Forest Classifier. See Table 8.2 for the detailed overall metric results. We achieved an accuracy of 65% and an F1 score of 46%. The model was computationally more expensive than the Random Forest Model, and the overall metrics were worse (see Figure 108). The high number of false negatives was especially concerning, and we believe that the Shapelet model failed to capture real, meaningful patterns. Therefore, we decided to use the Random Forest Model we built in the first step.

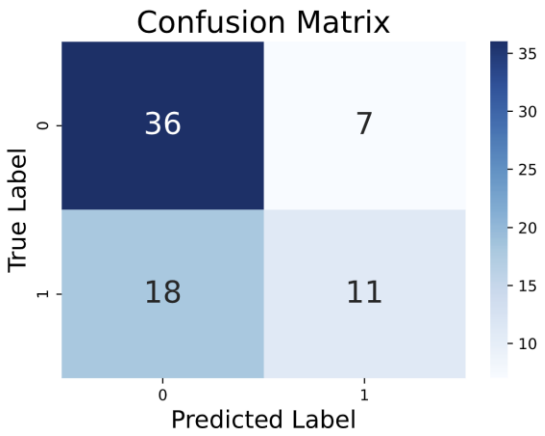

Figure 108 Confusion Matrix for the Shapelet Model

Table 8.2 Evaluation metrics for the *Tslearn* Shapelet model

	Precision	Recall	F1-Score	Accuracy
Overall	0.61	0.37	0.46	0.65

8.3.2 Heat Pump Classification

Due to the different types of heat pumps, we need to consider this use case as a multiclassification with two distinct positive classes and one negative class. Due to the lack of labels and the numerous unlabeled heat pump consumption time series, we decided to use a semi-supervised time series classification [139] in addition to the purely supervised approach. We used two additional regressors: the air temperature and the global radiation. The time frame used for classification covered two weeks, from 1.12.2023 to 15.12.2023.

8.3.2.1 Supervised Heat Pump Classification

We created a supervised model using the *RocketClassifier* from the *sktime* library [140]. The *RocketClassifier* wraps the Rocket transformer architecture using the *RidgeClassifierCV*. One of the main problems with our labeled dataset was the imbalance between negative and positive labels. Only 38 type-specific and 213 negative labels were available. We decided to undersample the negative label set to 50 time series. See Figure 109 for the confusion matrix and Table 8.3 for the overall results of the supervised classification. The confusion matrix indicates that the number

of labels for ground-source heat pumps was insufficient for the model to learn the characteristics of that type. The classifier never predicted the respective label 1 for ground source heat pumps. The classification performance for the air-source heat pumps and the negative labels was sufficient, leading to an overall F1 score of 72%.

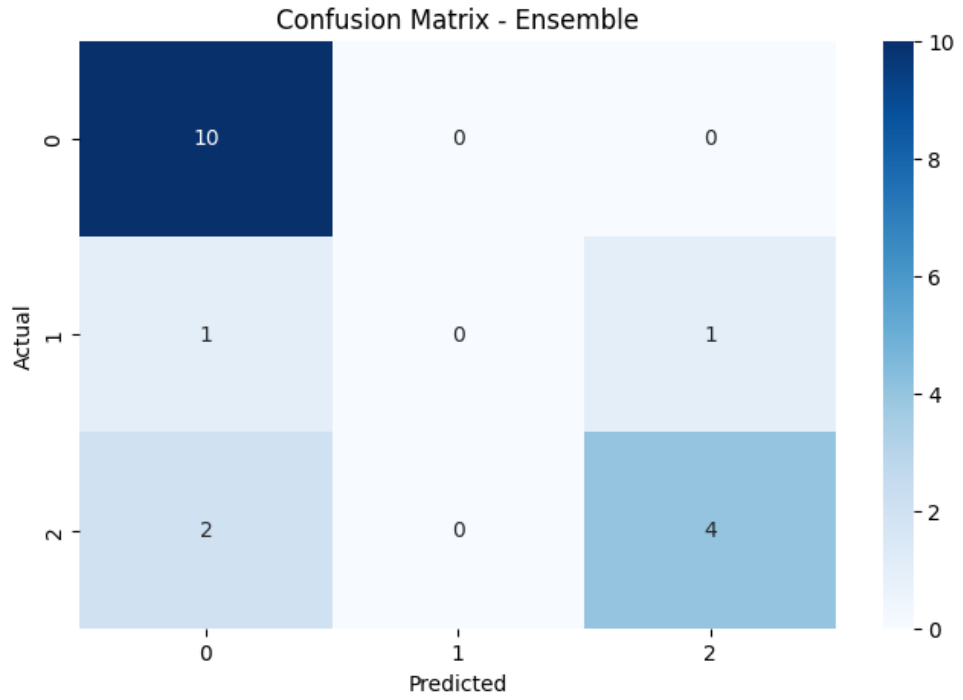

Figure 109 Confusion matrix supervised classification (0: no heat pump, 1: Ground, 2: Air)

Table 8.3 Evaluation metrics for the supervised heat pump classification

	Precision	Recall	F1-Score	Accuracy
Overall	0.69	0.77	0.72	0.77

8.3.2.2 Semi-Supervised Heat Pump Classification

Many time series are generally labeled as heat pump-specific positive labels, but only a few are labeled as heat pump-type-specific. In our use case, a semi-supervised approach that created a model based on the labeled subset and then propagated the labels to the unlabeled set would be beneficial.

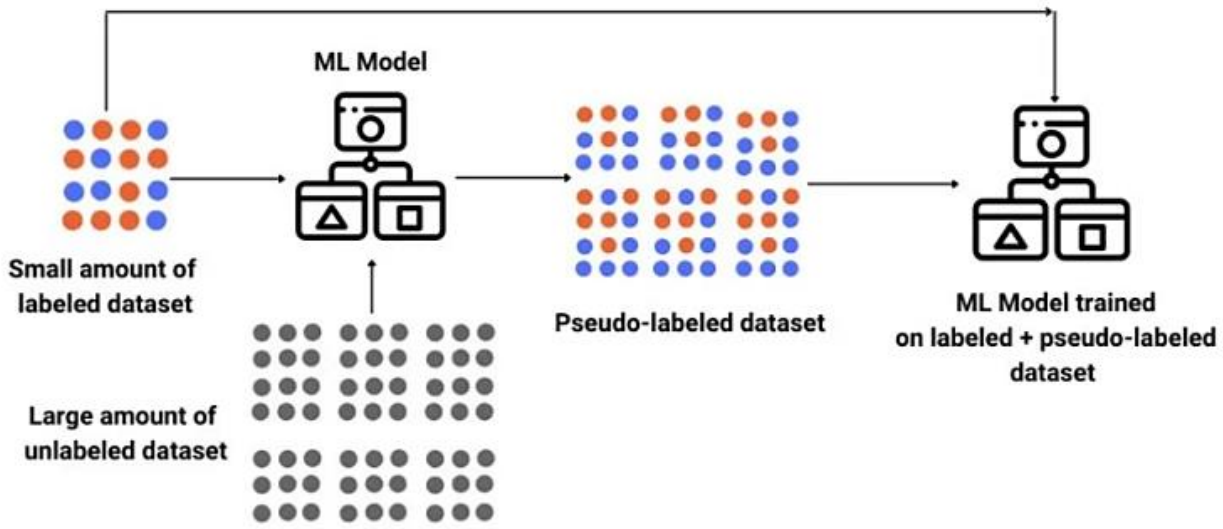

Figure 110 Semi-supervised classification approach [141]

In semi-supervised classifications, the usually limited set of labeled data is used to build an initial classifier model, which is then used to pseudo-label the unlabeled dataset. In the final step, the classification is done on the labeled and the pseudo-labeled datasets (Figure 110). We used an Autoencoder model to extract features from the data (Figure 111). The use of an Autoencoder to extract features aimed to eliminate irrelevant dimensions in the data and condense it to the necessary features for meaningful classification. The basic Autoencoder model architecture included an encoder and decoder layer of 128 neurons with ReLu and Sigmoid activation, respectively. In the next step of the semi-supervised architecture, a self-training model was built to learn the characteristics of the labeled dataset. This model was then used to pseudo-label all unlabeled or heat-pump unspecific time series. Random Forest Classifiers, including 100 subtrees, were used as a base and final classifier model.

```
# Autoencoder model
input_dim = X_combined.shape[1]
encoding_dim = 64  # latent space dimension

# Encoder
input_layer = layers.Input(shape=(input_dim,))
encoded = layers.Dense(128, activation='relu')(input_layer)
encoded = layers.Dense(encoding_dim, activation='relu')(encoded)

# Decoder
decoded = layers.Dense(128, activation='relu')(encoded)
decoded = layers.Dense(input_dim, activation='sigmoid')(decoded)

# Autoencoder
autoencoder = models.Model(input_layer, decoded)

# Compile the autoencoder
autoencoder.compile(optimizer='adam', loss='mse')
```

Figure 111 Autoencoder architecture for feature extraction

Table 8.4 Evaluation metrics for the supervised heat pump classification

	Precision	Recall	F1-Score	Accuracy
Base Model (Self-trainer)				
Class 0	0.81	0.83	0.82	
Class 1	0.29	0.17	0.21	
Class 2	0.72	0.73	0.73	
Overall Base Model				0.75
Final Predictor				
Class 0	0.95	0.97	0.96	
Class 1	0.92	0.87	0.9	
Class 2	0.96	0.93	0.94	
Overall Final Predictor			0.95	0.95

For the overall results of the base model used for pseudo-labeling and the final predictor, see Table 8.4 and Figure 112.

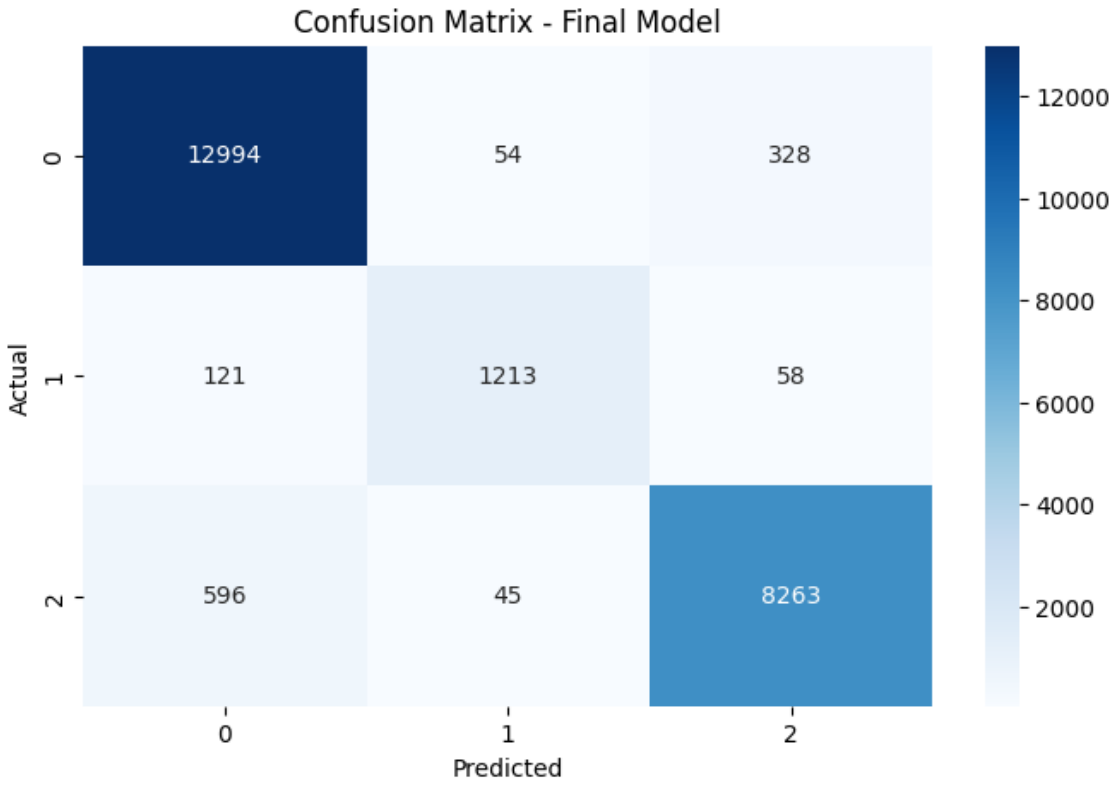

Figure 112 Confusion matrix for semi-supervised classification

The feasibility of evaluating the final prediction using the semi-supervised approach highly depends on the quality of the label propagation. It is important to consider the high overall performance of the final predictor. The results of the self-trained base model are quite similar to the ones of the pure supervised approach, which is unsurprising.

9. Project Evaluation and Discussion

This master thesis addresses two crucial questions in the field of smart meter data analytics for low-voltage energy network management: the feasibility of creating accurate energy consumption forecasts based on 15-minute interval data and the identification of electric vehicle charging stations and heat pumps using individual consumption profiles. We could prove that both topics can be addressed accurately using state-of-the-art time series forecasting models and supervised and semi-supervised learning methods for the classification tasks.

Various forecasting models, including traditional statistical methods, filtering methods, and advanced deep-learning techniques, were employed. We believe the large volume of data significantly contributed to the success of deep-learning models, especially Transformer-based architectures, compared to classical statistical models like SARIMA. The complexity of multi-seasonal consumption patterns poses a challenge for traditional statistical models. Meanwhile, Transformer-based architectures like Temporal Fusion Transformers can easily handle them using their attention mechanisms. The Temporal Fusion Transformer model exhibited excellent performance, and its ability to forecast multivariate time series and provide probabilistic forecasts was particularly advantageous. The TFT models achieved an overall MAPE of 30.35% and SMAPE of 30.03% on individual networks for our evaluation scenarios, demonstrating their robustness. The training time for Neural Prophet was also notably short, with an overall MAPE of 38.25% and SMAPE of 33.87%, making it an excellent choice for quick and efficient forecasts. The performance of the ensemble method AutoGluon was disappointing, with a MAPE of 16.37% and SMAPE of 16.19%, highlighting that larger datasets and more complex models do not always guarantee better results. Lag Llama's zero-shot performance was remarkable, making it a viable alternative for forecasting without any training effort. Because LLMs are a relatively new technology in the field of time series forecasting, we expect major improvements in these methods. Notably, only about a quarter of the meters have the fine-granular resolution of 15-minute intervals, limiting the data basis for more precise forecasts.

The second research question addressed in this thesis explores the identification of electric vehicle charging stations and heat pumps within KNG's distribution network. Supervised learning methods were applied to classify the consumption profiles using label data derived from external sources, such as customer contracts and ERP systems. We first used a Shapelet extraction method for the classification tasks. Configuring the Shapelets was particularly difficult and time-consuming, and the overall results were disappointing, with an accuracy of 65%. The extraction of features such as peak values and regularity of consumption patterns resulted in high precision and recall, with an F1 score of 91% and an accuracy of 92%. This demonstrated that good results could be achieved even without complex models. The identification of heat pumps presented greater challenges due to the different types of heat pumps and a lack of appropriate labels. The purely supervised approach was performed on the limited label set and achieved an F1 score of 71%, but it completely failed on the identification of ground source heat pumps, making it unusable for real-world use cases. The semi-supervised classification, which included label propagation for a large amount of unlabeled data, achieved satisfactory results, although further improvements are possible. Automated data extraction from customer documents using AI could enhance the reliability of labels and improve classification performance. The reliability of labels also affects the evaluation possibilities of the whole approach, as the assigned pseudo-labels of the semi-supervised approach are not reliable.

This master thesis highlights the potential of smart meter data to enhance grid planning and operation, ensuring a more efficient and reliable power grid. Future work will focus on refining these models and exploring additional data sources to improve accuracy and applicability.

Consumption forecasting enhancement will include further separation of larger clusters to better manage outliers in the individual network results. Additionally, we will develop specialized models for networks with unique consumption patterns that do not fit into existing clusters. These models will be tested in consecutive forecasts over several months to determine optimal retraining intervals and establish best practices for model retraining. A major upcoming change is the legal requirement of 15-minute intervals for all Smart Meters in KNG's network, which will enable the inclusion of feeder information and reevaluate our current approaches. The role of weather data as a regressor will also be reassessed once feeder information is integrated.

For the consumer profile classification, we plan to use sophisticated AI-based methods to extract label information from unstructured customer documents, increasing the number of available labels.

These advancements will enable better management of energy resources and contribute to the sustainability of the energy grid.

10. Bibliography

[1] *Intelligente Messgeräte-Anforderungs-Verordnung (IMA-VO)*. [Online]. Available: https://www.e-control.at/bereich-recht/verordnungen-zu-strom/-/asset_publisher/tiRyh5zzUOU7/content/intelligente-messger%25C3%25A4te-anforderungs-verordnung-ima-vo-

[2] *Intelligente Messgeräte-Einführungsverordnung (IME-VO)*. [Online]. Available: https://www.e-control.at/bereich-recht/verordnungen-zu-strom/-/asset_publisher/tiRyh5zzUOU7/content/intelligente-messger%25C3%25A4te-einf%25C3%25BChrungsverordnung-ime-vo-

[3] F. Dewangan, A. Y. Abdelaziz, and M. Biswal, 'Load forecasting models in smart grid using smart meter information: a review', *Energies*, vol. 16, no. 3, p. 1404, 2023.

[4] G. Hafeez, K. S. Alimgeer, and I. Khan, 'Electric load forecasting based on deep learning and optimized by heuristic algorithm in smart grid', *Applied Energy*, vol. 269, p. 114915, 2020.

[5] *Elektrizitätswirtschafts- und -organisationsgesetz*. [Online]. Available: https://www.ris.bka.gv.at/NormDokument.wxe?Abfrage=Bundesnormen&Gesetzesnummer=20007045&FassungVom=2023-11-16&Artikel=&Paragraf=0&Anlage=&Uebergangsrecht=

[6] M. H. Rashid, 'AMI smart meter big data analytics for time series of electricity consumption', in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, IEEE, 2018, pp. 1771–1776.

[7] B. Yildiz, J. I. Bilbao, J. Dore, and A. Sproul, 'Household electricity load forecasting using historical smart meter data with clustering and classification techniques', in *2018 IEEE Innovative Smart Grid Technologies-Asia (ISGT Asia)*, IEEE, 2018, pp. 873–879.

[8] H. Jiang, Y. Zhang, E. Muljadi, J. J. Zhang, and D. W. Gao, 'A short-term and high-resolution distribution system load forecasting approach using support vector regression with hybrid parameters optimization', *IEEE Transactions on Smart Grid*, vol. 9, no. 4, pp. 3341–3350, 2016.

[9] B. P. Hayes, J. K. Gruber, and M. Prodanovic, 'Multi-nodal short-term energy forecasting using smart meter data', *IET Generation, Transmission & Distribution*, vol. 12, no. 12, pp. 2988–2994, 2018.

[10] Y. Goude, R. Nedellec, and N. Kong, 'Local short and middle term electricity load forecasting with semi-parametric additive models', *IEEE transactions on smart grid*, vol. 5, no. 1, pp. 440–446, 2013.

[11] A. M. Pirbazari, A. Chakravorty, and C. Rong, 'Evaluating feature selection methods for short-term load forecasting', in *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*, IEEE, 2019, pp. 1–8.

[12] X. Yu *et al.*, 'Load forecasting based on smart meter data and gradient boosting decision tree', in *2019 Chinese Automation Congress (CAC)*, IEEE, 2019, pp. 4438–4442.

[13] B. Goehry, Y. Goude, P. Massart, and J.-M. Poggi, 'Aggregation of multi-scale experts for bottom-up load forecasting', *IEEE Transactions on Smart Grid*, vol. 11, no. 3, pp. 1895–1904, 2019.
[14] D. Alberg and M. Last, 'Short-term load forecasting in smart meters with sliding window-based ARIMA algorithms', *Vietnam Journal of Computer Science*, vol. 5, pp. 241–249, 2018.
[15] S. R. Twanabasu and B. A. Bremdal, 'Load forecasting in a smart grid oriented building', 2013.
[16] M. Ghofrani, M. Hassanzadeh, M. Etezadi-Amoli, and M. S. Fadali, 'Smart meter based short-term load forecasting for residential customers', in *2011 North American Power Symposium*, IEEE, 2011, pp. 1–5.
[17] J. Roth, J. Chadalawada, R. K. Jain, and C. Miller, 'Uncertainty matters: Bayesian probabilistic forecasting for residential smart meter prediction, segmentation, and behavioral measurement and verification', *Energies*, vol. 14, no. 5, p. 1481, 2021.
[18] Y. Yang, W. Li, T. A. Gulliver, and S. Li, 'Bayesian deep learning-based probabilistic load forecasting in smart grids', *IEEE Transactions on Industrial Informatics*, vol. 16, no. 7, pp. 4703–4713, 2019.
[19] K. Gajowniczek and T. Ząbkowski, 'Short term electricity forecasting using individual smart meter data', *Procedia Computer Science*, vol. 35, pp. 589–597, 2014.
[20] Z. Aung, M. Toukhy, J. Williams, A. Sanchez, and S. Herrero, 'Towards accurate electricity load forecasting in smart grids', *Proceedings of DBKDA*, pp. 51–57, 2012.
[21] S. Humeau, T. K. Wijaya, M. Vasirani, and K. Aberer, 'Electricity load forecasting for residential customers: Exploiting aggregation and correlation between households', in *2013 Sustainable internet and ICT for sustainability (SustainIT)*, IEEE, 2013, pp. 1–6.
[22] P. Vrablecová, A. B. Ezzeddine, V. Rozinajová, S. Šárik, and A. K. Sangaiah, 'Smart grid load forecasting using online support vector regression', *Computers & Electrical Engineering*, vol. 65, pp. 102–117, 2018.
[23] Y.-H. Hsiao, 'Household electricity demand forecast based on context information and user daily schedule analysis from meter data', *IEEE Transactions on Industrial Informatics*, vol. 11, no. 1, pp. 33–43, 2014.
[24] B. Asare-Bediako, W. Kling, and P. Ribeiro, 'Day-ahead residential load forecasting with artificial neural networks using smart meter data', in *2013 IEEE Grenoble conference*, IEEE, 2013, pp. 1–6.
[25] S. Sulaiman, P. A. Jeyanthy, and D. Devaraj, 'Artificial neural network based day ahead load forecasting using Smart Meter data', in *2016 Biennial International Conference on Power and Energy Systems: Towards Sustainable Energy (PESTSE)*, IEEE, 2016, pp. 1–6.
[26] A. Shahzadeh, A. Khosravi, and S. Nahavandi, 'Improving load forecast accuracy by clustering consumers using smart meter data', in *2015 international joint conference on neural networks (IJCNN)*, IEEE, 2015, pp. 1–7.
[27] Y. Wang, Q. Chen, M. Sun, C. Kang, and Q. Xia, 'An ensemble forecasting method for the aggregated load with subprofiles', *IEEE Transactions on Smart Grid*, vol. 9, no. 4, pp. 3906–3908, 2018.

[28] S. Hosein and P. Hosein, 'Load forecasting using deep neural networks', in *2017 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, IEEE, 2017, pp. 1–5.
[29] H. Shi, M. Xu, and R. Li, 'Deep learning for household load forecasting—A novel pooling deep RNN', *IEEE Transactions on Smart Grid*, vol. 9, no. 5, pp. 5271–5280, 2017.
[30] L. Li, K. Ota, and M. Dong, 'When weather matters: IoT-based electrical load forecasting for smart grid', *IEEE Communications Magazine*, vol. 55, no. 10, pp. 46–51, 2017.
[31] Y. Liang and P. K. Saha, 'Energy Consumption Forecasting Based on Long Short-term Memory Neural Network with Realistic Smart Meter Data', in *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, 2022, pp. 1374–1379.
[32] J. Ponoćko and J. V. Milanović, 'Forecasting demand flexibility of aggregated residential load using smart meter data', *IEEE Transactions on Power Systems*, vol. 33, no. 5, pp. 5446–5455, 2018.
[33] M. N. Fekri, H. Patel, K. Grolinger, and V. Sharma, 'Deep learning for load forecasting with smart meter data: Online Adaptive Recurrent Neural Network', *Applied Energy*, vol. 282, p. 116177, 2021.
[34] L. Sehovac and K. Grolinger, 'Deep learning for load forecasting: Sequence to sequence recurrent neural networks with attention', *Ieee Access*, vol. 8, pp. 36411–36426, 2020.
[35] R. Jiao, T. Zhang, Y. Jiang, and H. He, 'Short-term non-residential load forecasting based on multiple sequences LSTM recurrent neural network', *IEEE Access*, vol. 6, pp. 59438–59448, 2018.
[36] W. Kong, Z. Y. Dong, Y. Jia, D. J. Hill, Y. Xu, and Y. Zhang, 'Short-term residential load forecasting based on LSTM recurrent neural network', *IEEE transactions on smart grid*, vol. 10, no. 1, pp. 841–851, 2017.
[37] S. E. Razavi, A. Arefi, G. Ledwich, G. Nourbakhsh, D. B. Smith, and M. Minakshi, 'From load to net energy forecasting: Short-term residential forecasting for the blend of load and PV behind the meter', *IEEE Access*, vol. 8, pp. 224343–224353, 2020.
[38] M. Imani and H. Ghassemian, 'Residential load forecasting using wavelet and collaborative representation transforms', *Applied Energy*, vol. 253, p. 113505, 2019.
[39] Y.-S. Jeong, M. K. Jeong, and O. A. Omitaomu, 'Weighted dynamic time warping for time series classification', *Pattern recognition*, vol. 44, no. 9, pp. 2231–2240, 2011.
[40] C. Holder, M. Middlehurst, and A. Bagnall, 'A review and evaluation of elastic distance functions for time series clustering', *Knowledge and Information Systems*, vol. 66, no. 2, pp. 765–809, 2024.
[41] *tslearn.clustering.TimeSeriesKMeans*. Accessed: Jul. 25, 2024. [Online]. Available: https://tslearn.readthedocs.io/en/stable/gen_modules/clustering/tslearn.clustering.TimeSeriesKMeans.html
[42] *statsmodels*. Accessed: Jul. 25, 2024. [Online]. Available: https://www.statsmodels.org/stable/index.html
[43] A. Nielsen, *Practical time series analysis: Prediction with statistics and machine learning*. O'Reilly Media, 2019.
[44] M. Peixeiro, *Time series forecasting in python*. Simon and Schuster, 2022.
[45] W. A. Fuller, *Introduction to statistical time series*. John Wiley & Sons, 2009.

[46] B. Lim, S. Ö. Arık, N. Loeff, and T. Pfister, 'Temporal fusion transformers for interpretable multi-horizon time series forecasting', *International Journal of Forecasting*, vol. 37, no. 4, pp. 1748–1764, 2021.
[47] Kafritsas, Nikos, 'Temporal Fusion Transformer: Time Series Forecasting with Interpretability'. Accessed: Jan. 10, 2024. [Online]. Available: https://towardsdatascience.com/temporal-fusion-transformer-googles-model-for-interpretable-time-series-forecasting-5aa17beb621
[48] M. Hohberg, P. Pütz, and T. Kneib, 'Generalized additive models for location, scale and shape for program evaluation', 2018.
[49] T. Kneib, A. Silbersdorff, and B. Säfken, 'Rage against the mean–a review of distributional regression approaches', *Econometrics and Statistics*, vol. 26, pp. 99–123, 2023.
[50] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.
[51] R. E. Kalman, 'A New Approach to Linear Filtering and Prediction Problems', *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.
[52] M. Athans, R. Wishner, and A. Bertolini, 'Suboptimal state estimation for continuous-time nonlinear systems from discrete noisy measurements', *IEEE Transactions on Automatic Control*, vol. 13, no. 5, pp. 504–514, 1968.
[53] S. J. Julier and J. K. Uhlmann, 'New extension of the Kalman filter to nonlinear systems', in *Signal processing, sensor fusion, and target recognition VI*, Spie, 1997, pp. 182–193.
[54] G. Evensen, 'The ensemble Kalman filter: Theoretical formulation and practical implementation', *Ocean dynamics*, vol. 53, pp. 343–367, 2003.
[55] R. Labbe, 'Kalman and bayesian filters in python', *Chap*, vol. 7, no. 246, p. 4, 2014.
[56] S. J. Taylor and B. Letham, 'Forecasting at scale', *The American Statistician*, vol. 72, no. 1, Art. no. 1, 2018.
[57] S. Hochreiter and J. Schmidhuber, 'Long short-term memory', *Neural computation*, vol. 9, no. 8, Art. no. 8, 1997.
[58] O. Triebe, H. Hewamalage, P. Pilyugina, N. Laptev, C. Bergmeir, and R. Rajagopal, 'Neuralprophet: Explainable forecasting at scale', *arXiv preprint arXiv:2111.15397*, 2021.
[59] A. Zeng, M. Chen, L. Zhang, and Q. Xu, 'Are transformers effective for time series forecasting?', in *Proceedings of the AAAI conference on artificial intelligence*, 2023, pp. 11121–11128.
[60] Y. Nie, N. H. Nguyen, P. Sinthong, and J. Kalagnanam, 'A time series is worth 64 words: Long-term forecasting with transformers', *arXiv preprint arXiv:2211.14730*, 2022.
[61] K. Rasul *et al.*, 'Lag-llama: Towards foundation models for probabilistic time series forecasting', *Preprint*, 2024.
[62] G. Woo, C. Liu, A. Kumar, C. Xiong, S. Savarese, and D. Sahoo, 'Unified training of universal time series forecasting transformers', *arXiv preprint arXiv:2402.02592*, 2024.
[63] C. Catlin, 'AutoTS: Automated time series forecasting', *PyPI. February*, vol. 6, p. 2022, 2022.
[64] S. Wang *et al.*, 'Timemixer: Decomposable multiscale mixing for time series forecasting', *arXiv preprint arXiv:2405.14616*, 2024.

[65] J. G. De Gooijer and R. J. Hyndman, '25 years of time series forecasting', *International journal of forecasting*, vol. 22, no. 3, pp. 443–473, 2006.
[66] R. J. Hyndman and G. Athanasopoulos, *Forecasting: principles and practice*. OTexts, 2018.
[67] L.-R. Forecasting, 'From Crystal Ball to Computer', *Scott Armstrong Robert J. Genetski*, 1978.
[68] R. J. Hyndman and A. B. Koehler, 'Another look at measures of forecast accuracy', *International journal of forecasting*, vol. 22, no. 4, pp. 679–688, 2006.
[69] R. G. Brown, 'Statistical forecasting for inventory control', *(No Title)*, 1959.
[70] R. G. Brown, *Smoothing, forecasting and prediction of discrete time series*. Courier Corporation, 2004.
[71] C. C. Holt, 'Forecasting seasonals and trends by exponentially weighted moving averages', *International journal of forecasting*, vol. 20, no. 1, pp. 5–10, 2004.
[72] P. R. Winters, 'Forecasting sales by exponentially weighted moving averages', *Management science*, vol. 6, no. 3, pp. 324–342, 1960.
[73] *PMDARIMA*. [Online]. Available: https://alkaline-ml.com/pmdarima
[74] A. C. Harvey and S. Peters, 'Estimation procedures for structural time series models', *Journal of forecasting*, vol. 9, no. 2, pp. 89–108, 1990.
[75] T. Hastie and R. Tibshirani, 'Generalized additive models: some applications', *Journal of the American Statistical Association*, vol. 82, no. 398, pp. 371–386, 1987.
[76] G. Welch and G. Bishop, 'Welch & Bishop , An Introduction to the Kalman Filter 2 1 The Discrete Kalman Filter In 1960', 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:9209711
[77] B. Lim and S. Zohren, 'Time-series forecasting with deep learning: a survey', *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 379, no. 2194, p. 20200209, Feb. 2021, doi: 10.1098/rsta.2020.0209.
[78] R. Pascanu, T. Mikolov, and Y. Bengio, 'On the difficulty of training recurrent neural networks', in *International conference on machine learning*, Pmlr, 2013, pp. 1310–1318.
[79] Leo, Cristian, 'The Math behind LSTM'. Accessed: Aug. 14, 2024. [Online]. Available: https://towardsdatascience.com/the-math-behind-lstm-9069b835289d
[80] B. Carpenter *et al.*, 'Stan: A probabilistic programming language', *Journal of statistical software*, vol. 76, 2017.
[81] Q. Wen *et al.*, 'Transformers in time series: A survey', *arXiv preprint arXiv:2202.07125*, 2022.
[82] V. Ashish, 'Attention is all you need', *Advances in neural information processing systems*, vol. 30, p. I, 2017.
[83] S. Li *et al.*, 'Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting', *Advances in neural information processing systems*, vol. 32, 2019.
[84] J. Devlin, 'Bert: Pre-training of deep bidirectional transformers for language understanding', *arXiv preprint arXiv:1810.04805*, 2018.
[85] T. B. Brown, 'Language models are few-shot learners', *arXiv preprint ArXiv:2005.14165*, 2020.

[86] H. Zhou *et al.*, 'Informer: Beyond efficient transformer for long sequence time-series forecasting', in *Proceedings of the AAAI conference on artificial intelligence*, 2021, pp. 11106–11115.
[87] H. Wu, J. Xu, J. Wang, and M. Long, 'Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting', *Advances in neural information processing systems*, vol. 34, pp. 22419–22430, 2021.
[88] T. Zhou, Z. Ma, Q. Wen, X. Wang, L. Sun, and R. Jin, 'Fedformer: Frequency enhanced decomposed transformer for long-term series forecasting', in *International conference on machine learning*, PMLR, 2022, pp. 27268–27286.
[89] Leo, Cristian, 'The Math Behind Transformers'. Accessed: Aug. 14, 2024. [Online]. Available: https://medium.com/@cristianleo120/the-math-behind-transformers-6d7710682a1f
[90] Y. Liu, H. Wu, J. Wang, and M. Long, 'Non-stationary transformers: Exploring the stationarity in time series forecasting', *Advances in Neural Information Processing Systems*, vol. 35, pp. 9881–9893, 2022.
[91] Y. Zhang and J. Yan, 'Crossformer: Transformer utilizing cross-dimension dependency for multivariate time series forecasting', in *The eleventh international conference on learning representations*, 2023.
[92] Y. Liu *et al.*, 'itransformer: Inverted transformers are effective for time series forecasting', *arXiv preprint arXiv:2310.06625*, 2023.
[93] V. Ekambaram, A. Jati, N. Nguyen, P. Sinthong, and J. Kalagnanam, 'Tsmixer: Lightweight mlp-mixer model for multivariate time series forecasting', in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 459–469.
[94] S. Elsayed, D. Thyssens, A. Rashed, H. S. Jomaa, and L. Schmidt-Thieme, 'Do we really need deep learning models for time series forecasting?', *arXiv preprint arXiv:2101.02118*, 2021.
[95] Nikos, Kafritas, 'Will Transformers Revolutionize Time-Series Forecasting?' Accessed: Aug. 14, 2024. [Online]. Available: https://towardsdatascience.com/will-transformers-revolutionize-time-series-forecasting-1ac0eb61ecf3
[96] A. Garza and M. Mergenthaler-Canseco, 'TimeGPT-1', *arXiv preprint arXiv:2310.03589*, 2023.
[97] S. Dooley, G. S. Khurana, C. Mohapatra, S. V. Naidu, and C. White, 'Forecastpfn: Synthetically-trained zero-shot forecasting', *Advances in Neural Information Processing Systems*, vol. 36, 2024.
[98] A. Das, W. Kong, R. Sen, and Y. Zhou, 'A decoder-only foundation model for time-series forecasting', *arXiv preprint arXiv:2310.10688*, 2023.
[99] H. Touvron *et al.*, 'Llama: Open and efficient foundation language models', *arXiv preprint arXiv:2302.13971*, 2023.
[100] N. Erickson *et al.*, 'Autogluon-tabular: Robust and accurate automl for structured data', *arXiv preprint arXiv:2003.06505*, 2020.
[101] D. Deng, F. Karl, F. Hutter, B. Bischl, and M. Lindauer, 'Efficient automated deep learning for time series forecasting', in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2022, pp. 664–680.

[102] O. Shchur *et al.*, 'AutoGluon–TimeSeries: AutoML for probabilistic time series forecasting', in *International Conference on Automated Machine Learning*, PMLR, 2023, pp. 9–1.
[103] *NIXTLA*. Accessed: Jul. 25, 2024. [Online]. Available: https://www.nixtla.io/
[104] *StatsForecast*. Accessed: Jul. 25, 2024. [Online]. Available: https://nixtlaverse.nixtla.io/statsforecast/docs/getting-started/getting_started_complete.html
[105] *MLForecast*. Accessed: Jul. 25, 2024. [Online]. Available: https://nixtlaverse.nixtla.io/mlforecast/forecast.html
[106] T. Gneiting and M. Katzfuss, 'Probabilistic forecasting', *Annual Review of Statistics and Its Application*, vol. 1, no. 1, pp. 125–151, 2014.
[107] F. Garza, M. M. Canseco, C. Challú, and K. G. Olivares, 'StatsForecast: Lightning fast forecasting with statistical and econometric models', *PyCon: Salt Lake City, UT, USA*, 2022.
[108] D. Salinas, V. Flunkert, J. Gasthaus, and T. Januschowski, 'DeepAR: Probabilistic forecasting with autoregressive recurrent networks', *International journal of forecasting*, vol. 36, no. 3, pp. 1181–1191, 2020.
[109] G. Ke *et al.*, 'Lightgbm: A highly efficient gradient boosting decision tree', *Advances in neural information processing systems*, vol. 30, 2017.
[110] R. Ratcliff, 'Group reaction time distributions and an analysis of distribution statistics.', *Psychological bulletin*, vol. 86, no. 3, p. 446, 1979.
[111] S. Salvador and P. Chan, 'Toward accurate dynamic time warping in linear time and space', *Intelligent Data Analysis*, vol. 11, no. 5, pp. 561–580, 2007.
[112] *Unit8 DARTS*. Accessed: Jul. 25, 2024. [Online]. Available: https://unit8.com/resources/darts-time-series-made-easy-in-python/
[113] *Tensorflow Keras LSTM*. Accessed: Jul. 25, 2024. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM
[114] *Optuna*. Accessed: Jul. 25, 2024. [Online]. Available: https://optuna.org/
[115] *LagLama Model*. Accessed: Jul. 25, 2024. [Online]. Available: https://github.com/time-series-foundation-models/lag-llama
[116] *Pretrained Lag Llama Model*. Accessed: Jul. 25, 2024. [Online]. Available: https://huggingface.co/time-series-foundation-models/Lag-Llama
[117] *AutoGluon*. Accessed: Jul. 25, 2024. [Online]. Available: auto.gluon.ai
[118] *UNI2TS*. [Online]. Available: https://github.com/SalesforceAIResearch/uni2ts
[119] M. Zeifman and K. Roth, 'Nonintrusive appliance load monitoring: Review and outlook', *IEEE transactions on Consumer Electronics*, vol. 57, no. 1, pp. 76–84, 2011.
[120] M. Martinez-Pabon, T. Eveleigh, and B. Tanju, 'Smart meter data analytics for optimal customer selection in demand response programs', *Energy Procedia*, vol. 107, pp. 49–59, 2017.
[121] P. Carroll, T. Murphy, M. Hanley, D. Dempsey, and J. Dunne, 'Household classification using smart meter data', *Journal of official statistics*, vol. 34, no. 1, pp. 1–25, 2018.
[122] S. Haben, C. Singleton, and P. Grindrod, 'Analysis and clustering of residential customers energy behavioral demand using smart meter data', *IEEE transactions on smart grid*, vol. 7, no. 1, pp. 136–144, 2015.

[123] H. Kim *et al.*, 'An ica-based hvac load disaggregation method using smart meter data', in *2023 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, IEEE, 2023, pp. 1–5.
[124] K. Gajowniczek and T. Ząbkowski, 'Data mining techniques for detecting household characteristics based on smart meter data', *Energies*, vol. 8, no. 7, pp. 7407–7427, 2015.
[125] M. A. Devlin and B. P. Hayes, 'Non-intrusive load monitoring and classification of activities of daily living using residential smart meter data', *IEEE transactions on consumer electronics*, vol. 65, no. 3, pp. 339–348, 2019.
[126] R. Al Talib, S. M. Qaisar, H. Fatayerji, and A. Waqar, 'Features Mining and Machine Learning for Home Appliance Identification by Processing Smart meter Data', in *2023 1st International Conference on Advanced Innovations in Smart Cities (ICAISC)*, IEEE, 2023, pp. 1–6.
[127] A. Ridi, C. Gisler, and J. Hennebert, 'ACS-F2—A new database of appliance consumption signatures', in *2014 6th international conference of soft computing and pattern recognition (SoCPaR)*, IEEE, 2014, pp. 145–150.
[128] Z. Luo, R. Qi, Q. Li, J. Zheng, and S. Shao, 'ABODE-Net: an attention-based deep learning model for non-intrusive building occupancy detection using smart meter data', in *International Conference on Smart Computing and Communication*, Springer, 2022, pp. 152–164.
[129] C. Feng, A. Mehmani, and J. Zhang, 'Deep learning-based real-time building occupancy detection using AMI data', *IEEE Transactions on Smart Grid*, vol. 11, no. 5, pp. 4490–4501, 2020.
[130] C. Beckel, L. Sadamori, and S. Santini, 'Automatic socio-economic classification of households using electricity consumption data', in *Proceedings of the fourth international conference on Future energy systems*, 2013, pp. 75–86.
[131] G. Sun, Y. Cong, D. Hou, H. Fan, X. Xu, and H. Yu, 'Joint household characteristic prediction via smart meter data', *IEEE Transactions on Smart Grid*, vol. 10, no. 2, pp. 1834–1844, 2017.
[132] C. Beckel, L. Sadamori, T. Staake, and S. Santini, 'Revealing household characteristics from smart meter data', *Energy*, vol. 78, pp. 397–410, 2014.
[133] O. Y. Al-Jarrah, Y. Al-Hammadi, P. D. Yoo, and S. Muhaidat, 'Multi-layered clustering for power consumption profiling in smart grids', *IEEE Access*, vol. 5, pp. 18459–18468, 2017.
[134] G. Chicco, 'Overview and performance assessment of the clustering methods for electrical load pattern grouping', *Energy*, vol. 42, no. 1, pp. 68–80, 2012.
[135] *Isolation Forest*. Accessed: Jul. 25, 2024. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html
[136] *Tslearn ShapeletModel*. Accessed: Jul. 25, 2024. [Online]. Available: https://tslearn.readthedocs.io/en/stable/gen_modules/tslearn.shapelets.html
[137] L. Ye and E. Keogh, 'Time series shapelets: a new primitive for data mining', in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2009, pp. 947–956.

[138] A. Mueen, E. Keogh, and N. Young, ‘Logical-shapelets: an expressive primitive for time series classification’, in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011, pp. 1154–1162.
[139] L. Wei and E. Keogh, ‘Semi-supervised time series classification’, in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 748–753.
[140] *SKTime*. Accessed: Jul. 25, 2024. [Online]. Available: www.sktime.net
[141] Gayatri Sharma, ‘A Gentle Introduction to Semi Supervised Learning’. Accessed: Jul. 25, 2024. [Online]. Available: https://medium.com/@gayatri_sharma/a-gentle-introduction-to-semi-supervised-learning-7afa5539beea

11. List of Abbreviations

ACF	Autocorrelation Function
ADF	Augmented Dickey-Fuller Test
AIC	Akaike Information Criterion
ANN	Artificial Neural Networks
CNN	Convolutional Neural Networks
DNN	Deep neural networks
EVCS	Electric Vehicle Charging Stations
GAMLSS	Generalized Additive Models for Location, Scale and Shape
GRU	Gated Recurrent Unit
HVAC	Heat, Ventilation, and Air-Conditioning
ICA	Independent Component Analysis
IME	Intelligent Metering Device Extended
IMS	Intelligent Metering Device Standard
KNG	Kärnten Netz GmbH
kWh	Kilo Watt hours
LOTSA	Large-scale Open Time Series Archive
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MASE	Mean Absolute Scaled Error
MSE	Mean Squared Error
MDMS	Meter Data Management System
MOIRAI	Masked Encoder-based Universal Time Series Forecasting Transformer
NILM	Non-Intrusive Load Monitoring
PACF	Partial Autocorrelation Function
PLC	Power Line Carrier
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Networks

SARIMA	Seasonal Auto Regressive Integrated Moving Average
SMAPE	Symmetric Mean Absolute Percentage
SVM	Support Vector Machine
TFT	Temporal Fusion Transformers
VAR	Vector Auto Regression
WCSS	Within Cluster Sum of Squares

Appendix A: Source Code

A.1 Consumption Forecast Models

A.1.1 Neural Prophet Models on Final Cluster Application

```
from neuralprophet import NeuralProphet
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive
from datetime import datetime
import kaleido

# cluster_1_full contains complete data for cluster 1

# Forecasting time frames
train_start_date = '2023-10-16'
train_end_date = '2023-11-12'
test_end_date = '2023-11-14'

# Transform date field to pandas datetime format
cluster_1_full['READ_TIME'] = pd.to_datetime(cluster_1_full['READ_TIME'])

# restrict time frames
cluster_1 = cluster_1_full[(cluster_1_full['READ_TIME'] >= train_start_date)
  & (cluster_1_full['READ_TIME'] < train_end_date)]
cluster_1_test = cluster_1_full[(cluster_1_full['READ_TIME'] >= train_end_date)
  & (cluster_1_full['READ_TIME'] < test_end_date)]

# rename according to Neural Prophet requirements
cluster_1.rename(columns={'READ_TIME':'ds',
                          'CONSUMPTION':'y',
                          'TRAFO':'ID'},
                 inplace=True)
cluster_1_test.rename(columns={'READ_TIME':'ds',
                               'CONSUMPTION':'y',
                               'TRAFO':'ID'},
                      inplace=True)

# Create Neural Propeht model - Batch size 128, epochs 8,
# standrad normalization and Huber loss function
npmodel_cluster_1 = NeuralProphet(batch_size=128,
                                  epochs=8,
                                  normalize="standardize",
                                  loss_func="Huber")

npmodel_cluster_1.set_plotting_backend("plotly-static")

# fite the trainin data
metrics_cluster_1 = npmodel_cluster_1.fit(cluster_1)

# predict the results for the test data
forecast_cluster_1 = npmodel_cluster_1.predict(cluster_1_test)
```

A.1.2 Temporal Fusion Transformer Models on Final Cluster Application

Environment Setup and Data Preparation

```
!pip install ipython-autotime
!pip install torch==2.1.0
!pip install git+https://github.com/jdb78/pytorch-forecasting
import pytorch_lightning as pl
import numpy as np
import pandas as pd
from pytorch_forecasting import TimeSeriesDataSet
from google.colab import drive
from pytorch_forecasting import Baseline, TemporalFusionTransformer
from pytorch_forecasting.data import GroupNormalizer
import matplotlib.pyplot as plt
import lightning.pytorch as pl
from lightning.pytorch.callbacks import EarlyStopping, LearningRateMonitor
from lightning.pytorch.loggers import TensorBoardLogger
import torch
from pytorch_forecasting.metrics import MAE, SMAPE, PoissonLoss, QuantileLoss
from sklearn import metrics

# forecasting time frames
train_start_date = '2023-10-16'
train_end_date = '2023-11-14'

# cluster_0_full contains the complete data for cluster 0

# Transform the date column to pandas datetime
cluster_0_full['READ_TIME'] = pd.to_datetime(cluster_0_full['READ_TIME'])

# restrict time frames
cluster_0 = cluster_0_full[(cluster_0_full['READ_TIME'] >= train_start_date)
  & (cluster_0_full['READ_TIME'] < train_end_date)]
```

TFT Parameter Setup

```
####
# TFT setting up parameters

# set training timeframes and batch size
max_prediction_length = 192 # 2 days on quarter-hourly data
max_encoder_length = 7*96 # week
batch_size = 64
training_cutoff = cluster_0["TIME_IDX"].max() - max_prediction_length*2
test_cutoff = cluster_0["TIME_IDX"].max() - max_prediction_length

# create TimeSeriesDataSet
training_0 = TimeSeriesDataSet(
    cluster_0[lambda x: x.TIME_IDX <= training_cutoff],
    time_idx="TIME_IDX",
    target="CONSUMPTION",
    group_ids=["TRAFO"],
    min_encoder_length=max_encoder_length//2,
    max_encoder_length=max_encoder_length,
    min_prediction_length=1,
    max_prediction_length=max_prediction_length,
    static_categoricals=["TRAFO"],
    time_varying_known_reals=["TIME_IDX","DAY","DAY_OF_WEEK", "MONTH", 'HOUR'],
    time_varying_unknown_reals=['CONSUMPTION'],
    target_normalizer=GroupNormalizer(groups=["TRAFO"], transformation="softplus"),  # normalization
    add_relative_time_idx=True,
    add_target_scales=True,
    add_encoder_length=True,
    allow_missing_timesteps=True
)

# create train, validation, test datasets
validation_0 = TimeSeriesDataSet.from_dataset(training_0,
                                              cluster_0[lambda x: x.TIME_IDX <= training_cutoff],
                                              predict=True,
                                              stop_randomization=True)
test_0 = TimeSeriesDataSet.from_dataset(training_0,
                                        cluster_0[lambda x: x.TIME_IDX <= test_cutoff],
                                        predict=True,
                                        stop_randomization=True)
train_dataloader_0 = training_0.to_dataloader(train=True, batch_size=batch_size, num_workers=4)
val_dataloader_0 = validation_0.to_dataloader(train=False, batch_size=batch_size*10, num_workers=4)
test_dataloader_0 = test_0.to_dataloader(train=False, batch_size=batch_size*10, num_workers=4)

# set early stopping parameters
early_stop_callback = EarlyStopping(monitor="val_loss", min_delta=1e-4, patience=3, verbose=True, mode="min")
lr_logger = LearningRateMonitor()

# create Tensorborad logger if required
logger = TensorBoardLogger("lightning_logs")
```

Model Training

```
####
# TFT Training for cluster 0
# use the pytorch lightning Trainer
trainer_0 = pl.Trainer(
    max_epochs=10,
    accelerator='gpu',
    devices=1,
    enable_model_summary=True,
    gradient_clip_val=0.1,
    callbacks=[lr_logger, early_stop_callback],
    logger=logger)

# Build Temporal Fusion Transformer model
tft_0 = TemporalFusionTransformer.from_dataset(
    training_0,
    learning_rate=0.001,
    hidden_size=160,
    attention_head_size=4,
    dropout=0.1,
    hidden_continuous_size=160,
    output_size=7,  # [0.02, 0.1, 0.25, 0.5, 0.75, 0.9, 0.98]
    loss=QuantileLoss(), # probabilistic forecasts with loss per quantile
    log_interval=10,
    reduce_on_plateau_patience=4)

# Fit the Training data
trainer_0.fit(
    tft_0,
    train_dataloaders=train_dataloader_0,
    val_dataloaders=val_dataloader_0)

# Save model checkpoints for later retrieval
model_0 = trainer_0.checkpoint_callback.best_model_path
best_model_0 = TemporalFusionTransformer.load_from_checkpoint(model_0)
torch.save(best_model_0.state_dict(),
           "../TFT/Models/cluster_0.ckpt")
```

Evaluation of Results

```python
# get trafos
trafos = cluster_0['TRAFO'].unique()

# save results to local CPU from GPU
actuals_0 = torch.cat([y[0] for x, y in iter(test_dataloader_0)]).to('cuda')

# predict test data
predictions_0 = best_model.predict(test_dataloader_0)
predictions_mode_raw_0 = best_model.predict(test_dataloader_0, mode="raw",
                                            return_x=True)

# display result plots
for idx in range(len(trafos)):
    fig, ax = plt.subplots(figsize=(10, 4))
    best_model.plot_prediction(predictions_mode_raw_0.x,
                               predictions_mode_raw_0.output,
                               idx=idx,
                               add_loss_to_title=QuantileLoss(),ax=ax)

# calc MAPE SMAPE
def calculate_errors(trafos,predictions, actuals, path):
    num_series = len(predictions)
    results = {
        "ID": [],
        "MAPE": [],
        "SMAPE": []}

    overall_mape = calculate_mape(predictions, actuals)
    overall_smape = calculate_smape(predictions, actuals)

    print("Overall MAPE:", overall_mape)
    print("Overall SMAPE:", overall_smape)

    for i in range(num_series):
        series_mape = calculate_mape(predictions[i], actuals[i])
        series_smape = calculate_smape(predictions[i], actuals[i])

        results["ID"].append(trafos[i])
        results["MAPE"].append(series_mape)
        results["SMAPE"].append(series_smape)

    df = pd.DataFrame(results)
    df.to_csv(path, index=False)
    return df

# Calc Results
predictions = predictions_2.cpu().numpy()
actuals = actuals_2.cpu().numpy()
results_df = calculate_errors(trafos, predictions, actuals,
                              '../TFT/metrics/cluster_2_metrics.csv')
```

A.2 NILM Profile Identification

A.2.1 Supervised Learning Classification EVC

```
import pandas as pd
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split

# read data
df_emob = pd.read_csv('FINAL_COMPLETE_LABELS_EMOB\emob_labelled.csv', sep=';', decimal=".")
# transform date column
df_emob['INTERVAL_END_TIME'] = pd.to_datetime(df_emob['INTERVAL_END_TIME'])

# reduce the time frame
start_date = '2023-11-15'
end_date = '2024-02-05'
mask = (df_emob['INTERVAL_END_TIME'] >= start_date) & (df_emob['INTERVAL_END_TIME'] <= end_date)
df_emob_filtered = df_emob.loc[mask]

meters_of_interest = df_emob_filtered['METER_BADGE_ID'].unique()

# extract features
features = []
for meter_id in meters_of_interest:
    meter_data = df_emob_filtered[df_emob_filtered['METER_BADGE_ID'] == meter_id]['LP_VALUE_KWH']
    min_val = meter_data.min()
    max_val = meter_data.max()
    peak_to_peak_val = meter_data.max() - meter_data.min()
    mean_val = meter_data.mean()
    features.append([meter_id, min_val, max_val, peak_to_peak_val, mean_val])

features_df = pd.DataFrame(features, columns=['METER_BADGE_ID',
                                              'min_val',
                                              'max_val',
                                              'peak_to_peak',
                                              'mean_val'])
features = np.nan_to_num(features)
labels = df_emob_filtered.groupby('METER_BADGE_ID')['Class'].first().values

# Splitting the dataset into training and testing sets
X_train_feature, X_test_feature, y_train_feature, y_test_feature = train_test_split(features,
                                                                                    labels,
                                                                                    test_size=0.2,
                                                                                    random_state=42)

# Model Training
clf = RandomForestClassifier(n_estimators=500, random_state=42)
clf.fit(X_train_feature, y_train_feature)

# Model Evaluation
y_pred_feature = clf.predict(X_test_feature)
print(classification_report(y_test_feature, y_pred_feature))
```

A.2.2 Supervised Learning Classification Heat Pumps

```python
import pandas as pd
import numpy as np
from sktime.classification.kernel_based import RocketClassifier
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sktime.classification.compose import ColumnEnsembleClassifier

# wp_labelled_reduced contains the complete label data without the UE (=feeder)
df = pd.DataFrame(wp_labelled_reduced)

df = pd.DataFrame(wp_labelled_reduced)

def create_nested_df(df, group_col, time_col, feature_cols):
    df_grouped = df.groupby(group_col)
    X = pd.DataFrame()
    max_length = df_grouped.size().max()
    for feature in feature_cols:
        X[feature] = [pd.Series(group[feature].values).reindex(
            range(max_length), fill_value=np.nan) for _, group in df_grouped]
    y = df_grouped['class'].first().values
    return X, y

X, y = create_nested_df(df, 'METER_BADGE_ID',
                        'TIME_IDX',
                        ['LP_VALUE_KWH',
                         'Villach_TL_Lufttemp_MWm',
                         'Villach_Gl_Globalstrahlung'])

# Fill any missing values resulting from reindexing
X = X.applymap(lambda col: col.fillna(method='ffill').fillna(method='bfill'))

# Perform train-test split
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)

pipeline = make_pipeline(
    RocketClassifier())

# Fit the training data
pipeline.fit(X_train, y_train)

# Predict the class labels on the test data
y_pred = pipeline.predict(X_test)

print("Predictions:", y_pred)

# Evaluate
accuracy = np.mean(y_pred == y_test)
print("Accuracy:", accuracy)
```

A.2.3 Semi-Supervised Learning Classification Heat Pumps

Data Preparation

```
### Data Preparation
import pandas as pd
from datetime import datetime
from sklearn.preprocessing import StandardScaler

# wp_unlabelled and wp_labelled contain the lablelled and unlabelled datasets
# remove rows where UE=1 from unlabelled
wp_unlabelled = wp_unlabelled[wp_unlabelled['UE'] != 1]

# Time frame from 1.12.2023 - 31.12.2023
start_date = pd.to_datetime('2023-12-01')
end_date = pd.to_datetime('2023-12-15')

wp_labelled_filtered['INTERVAL_END_TIME'] =
  pd.to_datetime(wp_labelled_filtered['INTERVAL_END_TIME'])
wp_unlabelled_filtered['INTERVAL_END_TIME'] =
  pd.to_datetime(wp_unlabelled_filtered['INTERVAL_END_TIME'])

wp_labelled_reduced = wp_labelled_filtered[(
    wp_labelled_filtered['INTERVAL_END_TIME'] >= start_date)
    & (wp_labelled_filtered['INTERVAL_END_TIME'] <= end_date)]
wp_unlabelled_reduced = wp_unlabelled_filtered[(
    wp_unlabelled_filtered['INTERVAL_END_TIME'] >= start_date)
    & (wp_unlabelled_filtered['INTERVAL_END_TIME'] <= end_date)]

# Remove unused columns
wp_labelled_reduced = wp_labelled_reduced.drop(columns=['Type','UE'])
wp_labelled_reduced['TIME_IDX'] = wp_labelled_reduced.groupby('METER_BADGE_ID').cumcount()
wp_labelled_reduced = wp_labelled_reduced.drop(columns=['INTERVAL_END_TIME'])
wp_unlabelled_reduced = wp_unlabelled_reduced.drop(columns=['UE'])
wp_unlabelled_reduced['TIME_IDX'] = wp_unlabelled_reduced.groupby('METER_BADGE_ID').cumcount()
wp_unlabelled_reduced = wp_unlabelled_reduced.drop(columns=['INTERVAL_END_TIME'])

wp_unlabelled_reduced = wp_unlabelled_reduced.drop(columns=['CHANNEL_ID','INTERVAL_LEN_SEC',
                                    'DATA_SCOURCE_ID','time',
                                    'Villach_WG_Windgeschw',
                                    'Villach_P_Luftdruck_MWm',
                                    'Villach_RR_Niederschlag',
                                    'Villach_RF_RelFeuchte'])

# Extract features and labels from the labeled dataset
X_labeled = wp_labelled_reduced.drop(columns=['class']).values
y_labeled = wp_labelled_reduced['class'].values
# Extract features from the unlabeled dataset
X_unlabeled = wp_unlabelled_reduced.values
# Normalize the data
scaler = StandardScaler()
X_labeled = scaler.fit_transform(X_labeled)
X_unlabeled = scaler.transform(X_unlabeled)
# Combine the labeled and unlabeled data for autoencoder training
X_combined = np.vstack((X_labeled, X_unlabeled))
```

Autoencoder Feature Extraction

```
### Autoencoder feature extraction
import tensorflow as tf
from tensorflow.keras import layers, models

# Autoencoder model
input_dim = X_combined.shape[1]
encoding_dim = 64  # latent space dimension

# Encoder
input_layer = layers.Input(shape=(input_dim,))
encoded = layers.Dense(128, activation='relu')(input_layer)
encoded = layers.Dense(encoding_dim, activation='relu')(encoded)

# Decoder
decoded = layers.Dense(128, activation='relu')(encoded)
decoded = layers.Dense(input_dim, activation='sigmoid')(decoded)

# Autoencoder
autoencoder = models.Model(input_layer, decoded)

# Compile the autoencoder
autoencoder.compile(optimizer='adam', loss='mse')

# Train the autoencoder
autoencoder.fit(X_combined,
                epochs=3,
                batch_size=64,
                shuffle=True,
                validation_split=0.2)

# Extract the encoder part of the autoencoder
encoder = models.Model(inputs=input_layer, outputs=encoded)

# Use the encoder to transform both labeled and unlabeled data
X_labeled_encoded = encoder.predict(X_labeled)
X_unlabeled_encoded = encoder.predict(X_unlabeled)
```

Semi-supervised training

```
### Semisupervised training process
from sklearn.model_selection import train_test_split
from sklearn.semi_supervised import SelfTrainingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

X_train, X_test, y_train, y_test = train_test_split(X_labeled_encoded,
                                                    y_labeled,
                                                    test_size=0.2,
                                                    random_state=42)

# Train a classifier on the encoded features of the labeled data
base_classifier = RandomForestClassifier(n_estimators=100)
self_training_model = SelfTrainingClassifier(base_classifier)

# Fit the model on the training data
self_training_model.fit(X_train, y_train)

# Evaluate the model on the test data
y_pred = self_training_model.predict(X_test)
print(classification_report(y_test, y_pred))

# Predict pseudo-labels for the unlabeled data
pseudo_labels = self_training_model.predict(X_unlabeled_encoded)

# Combine the labeled data with the pseudo-labeled data
X_combined_encoded = np.vstack((X_labeled_encoded, X_unlabeled_encoded))
y_combined = np.concatenate((y_labeled, pseudo_labels))

# Retrain the classifier on the combined dataset
final_classifier = RandomForestClassifier(n_estimators=100)
final_classifier.fit(X_combined_encoded, y_combined)

# Evaluate the model on the test set again after retraining
y_pred_final = final_classifier.predict(X_test)
print(classification_report(y_test, y_pred_final))
```