University of Münster

Department of Information Systems

Analysing the Foundation Models for Time-Series Forecasting

Bachelor Thesis

submitted by

Ellen Alina Parthum

Chair of Data Science:

Machine Learning and Data Engineering

Principal Supervisor	Dr. Sugandha Arora Chair for Data Science: Machine Learning and Data Engineering
Student Candidate	Ellen Alina Parthum
Matriculation Number	527899
Field of Study	Information Systems
Contact Details	eparthum@uni-muenster.de
Submission Date	21.01.2024

Contents

Abbreviations

ACF Autocorrelation Function

AR Autoregressive

ARIMA Autoregressive Integrated Moving Average

ARMA Autoregressive Moving average

CNN Convolutional Neural Network

FNN Feedforward Neural Network

LLM Large Language Models

MA Moving Average

MAE Mean Absolute Error

MAPE Mean Absolute Percentage Error

MSE Mean Squared Error

NN Neural Network

PACF Partial Autocorrelation Function

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

RMSE Root Mean Squared Error

TTM Tiny Time Mixer

TimesFM Time-series Foundation Model

1 Introduction

Artificial intelligences, in particular Large Language Models (LLM), such as ChatGPT, have received considerable attention due to their capabilities. Generative Pretrained Transform models are not only used for the generation and rewriting of texts in a variety of styles, but also for the generation of images from user supplied text prompts, among other functions [amb24; Das+24a]. In addition to processing large quantities of text and image data, GPT models can also be applied to time for example. Time-series forecasting tools are increasingly adopting these types of techniques [Das+24a].
Time-series forecasting is a critical task in several industries, including retail, finance, manufacturing, healthcare, and the natural sciences [amb24]. These tasks are increasingly using deep learning models along with traditional statistical methods, such as Autoregressive Integrated Moving Average (ARIMA) [BJ68; Sal+20; Das+24a].
As mentioned above, LLM inspired time series foundation models, such as TimesFM and TimeGPT-1 have been developed to do time-series forecasting [Das+24a; GCM23; Woo+24]. These models are built using the transformer architecture [Woo+24], a model initially designed for text processing and sentence translation tasks [Vas+17]. The time series foundation models use the advantages of the transformer architecture to identify patterns in data and support advanced techniques like zero-shot learning and fine-tuning to forecast data [Das+24a; Woo+24]. This thesis present study examines the performance of TimesFM in comparison to TimeGPT-1 when executing zero-shot and fine-tuning forecasts. In addition, this thesis will also explore if a covariate impacts the foundation models. Covariates are a way of adding certain exogenous factors that provide additional information about the main variable and thereby influence the forecast [Das+24a; GCM23; Oli+23].

Working with foundation models is an exiting new topic with papers and investigations being published recently, as can be seen by Goel et al. [GPK68], whose paper was published on 15 October 2024. Also, the inclusion of covariates and its performance when incorporated into forecasting workflows is particularly highlighted by Das et al. [Das+23]. In order to gain some initial insights into these topics, this study investigates whether the inclusion of covariates - additional variables that provide contextual information, such as economic indicators, seasonal trends or other factors - improves or hinders the predictive ability of these models. Despite their potential to increase model accuracy, covariates and fine-tuning might lead to overfitting, multicollinearity or insufficient optimisation if not handled properly [Bos+24; Yin19; Yu+24].
Das et al.[Das+24a] emphasise the need for more research on handling covariates

in the foundation model Times FM. Furthermore, Das et al. [Das+24a] mentions that TimeGPT-1 is “the only other parallel work on a zero-shot foundation model for time-series forecasting”. Since the papers about TimesFM and TimeGPT-1 were published [Das+24a; GCM23], little research was conducted on the comparison of TimesFM and TimeGPT, apart from Puvvada and Chaudhuri [PC24] published on the 10th of October 2024. Their work does not include covariate handling.
This leads to the research questions:

RQ1. How do time series forecasting models like TimesFM and TimeGPT-1 compare in performance?

RQ2. How does TimesFM perform when covariates are incorporated, in comparison to other time-series forecasting models such as TimeGPT-1 and its performance without covariates?

The aim of this thesis is to use both TimesFM and TimeGPT-1 to perform zero-shot and fine-tuning forecasts on specified data and covariates. The intention is to evaluate which model performs better and to provide a rationale for this evaluation. Additionally, the traditional ARIMA model is used as a benchmark.

To tackle the research questions proposed in this thesis, a combination of data analysis and a semi-structured literature review is conducted. The literature review aims to provide context to the data analysis by identifying and synthesising relevant academic sources. The review process follows the methodological framework established by Webster and Watson [WW02], thereby, ensuring a systematic and transparent approach to identifying literature.
The data used in this study is quantitative in nature, focusing on evaluating the accuracy of time-series forecasting models. Although these techniques and analysis may be applied to any type of time series data, for this study electricity consumption and temperature data is used. The analysis investigates whether fine-tuning or incorporating covariates enhances the predictive accuracy of these models. To this end, a secondary data analysis is conducted, matching the guidelines outlined by Donnellan et al. [DL13]. This approach is appropriate because the primary interest lies in the accuracy of the forecasting models, rather than the specific nature of the data itself. Consequently, the data only needs to meet the criteria of being suitable for time-series forecasting, with further details specified in section 5.1. The use of secondary data analysis is both efficient and effective, allowing the research questions to be addressed without the need for extensive primary data collection [DL13].

1.1 Literature Review

This chapter outlines the methodology deployed for the literature review, including the processes for identifying, selecting, and analysing the literature. It then describes the execution of the literature analysis, thus providing a base for the empirical components of the thesis.

In their 2002 study, Webster and Watson [WW02] propose a three-step process for identifying relevant literature. The first step involves reviewing leading journals and journal databases, as well as examining conference proceedings. Within the context of this thesis, which is situated in the field of Information Systems, prioritizing Information Systems journals is a logical starting point. However, the focus should not be limited to this domain, as related disciplines may also offer valuable insights on the topic.
The search for additional literature is conducted using Google Scholar, which was selected for its accessibility and extensive database of recent academic work. This was particularly important given the inclusion of recent papers, such as "MOMENT: A Family of Open Time-series Foundation Models" by Goswami et al. [Gos+24], written in October 2024. Since such articles may not yet have undergone formal peer review or been published in traditional academic journals, the use of a search platform having access to all papers that can be found on Google is crucial [24a].
It was then based on the query: ("foundation model" OR "time-series foundation model" OR "time series foundation model") AND ("time series forecasting" OR "time-series forecasting") AND "transformer architecture". This search resulted in 219 initial results. To discover the relevant findings, a systematic screening process was implemented. That includes, articles that were too broad in scope or works that extended beyond the immediate focus of the research were disregarded. Outputs that are unavailable in English or German, as well as those lacking full-text access, were also excluded from the selection process.
The screening began with an evaluation of the titles, followed by an abstract review to determine alignment with the research objectives. Following this, the search results were narrowed to a number of probably relevant papers, which formed the basis for further analysis and discussion in this thesis.

The second step of the literature review involves performing a backward search, which is performed on the references within the selected studies to identify additional sources essential to the thesis. In particular, background information for explaining certain aspects of the time-series foundation model.
The majority of the literature ultimately incorporated into the thesis was discovered

using this method. The backward search proved particularly effective for refining the scope of the review and delving more deeply into aspects of the research topic.

The third step is to conduct a forward search by identifying articles that have cited the previously identified works. This approach determines the key articles that should be included in the literature review.
To find the papers that had worked with TimesFM and TimeGPT-1 and that may even include fine-tuning and covariate handling, the forward search focused on the citation pools by Garza et al. [GCM23] and Das et al. [Das+24a].
The three-step approach proposed by Webster and Watson [WW02] led to the identification of key sources, which are essential to the thesis. The main sources derived from this process are presented in appendix A.1. Furthermore, additional target searches on these sources are conducted to gain a more nuanced understanding of specific topics.

To structure the literature used in this thesis, Webster and Watson [WW02] propose applying a concept-centric framework. This method enables the identification of connections between sources and ensures their systematic organisation. Adopting this approach allows for a clear alignment of the literature with the discussion section of the thesis, ensuring that specific sources can effectively support key arguments. The concepts forming the basis of this framework were identified during the review of the literature, focusing on elements that are most relevant to the thesis. Following a thorough examination, the concepts were categorised, looking for the following themes: TimesFM, TimeGPT, Fine-tuning, Covariate handling, referenced in limitations and referenced in Future Research. Results for the main sources can be seen in appendix A.2.

2 Background

In order to comprehensively address the research questions and provide insights into the topics under discussion, it is essential to establish a clear understanding of the transformer architecture and, ultimately, the foundation models. This begins with an exploration of neural networks, which serves to set the scene.

2.1 Neural Networks

It is important to note that neural networks are inspired by the human brain and its remarkable ability to recognise images and detect patterns by integrating various contextual factors [Kro08]. In the early stages of neural network development, McCulloch and Pitts proposed a model in which a neuron was considered a switch that receives input from other neurons and becomes active only if the total weighted input exceeds a specific threshold [Agg18; Nie19; Kro08]. Building on this neural model, Rosenblatt introduced the concept of "perceptrons," which demonstrated that neural networks could learn from examples and identify patterns [Kro08].

The transformer model, a more advanced neural network architecture, uses feedforward neural networks, which necessitates an understanding of neural networks in general [Vas+17]. An approach to comprehending these networks is to begin with perceptrons. A perceptron takes multiple binary inputs $x = (x_1, x_2, \dots x_j)$ and generates a single binary output. Each input has an associated weight (w), and the output depends on the weighted sum $\sum_j w_j x_j$. If this sum surpasses a certain threshold, the output is 1; otherwise, it is 0 [Agg18; Nie19].

This threshold, which is related to the decision boundary of the data input, is also known as the bias (b) and can be added to the weighted sum to simplify the function [Agg18; Nie19]:

$$output = \begin{cases} 0 & \text{if } w \times x + b <= 0 \\ 1 & \text{if } w \times x + b > 0 \end{cases} \tag{2.1}$$

Function 2.1: w and x are vectors replacing the sum of $w \times x$.

Conceptually, bias serves as an indicator of the ease with which the perceptron outputs a 1. A high positive bias for the perceptron typically results in a 1 as an output, whereas for a high negative bias, it is quite challenging to output a 1 [Agg18; Nie19].

To achieve greater complexity and enhance accuracy in network outcomes, perceptrons can be extended by using sigmoid neurons. Similarly, perceptrons and sigmoid neurons accept inputs $x = (x_1, x_2, \dots, x_j)$ and have associated weights $w = (w_1, w_2, \dots, w_j)$ and a bias (b). However, unlike perceptrons, sigmoid neurons accept inputs that range continuously between 0 and 1. By summing the product of weights and neuron activations, a numeric output is produced; the sigmoid function is then applied to bring this output within the 0–1 spectrum, whereby it can either serve as a new input or as a percentage which indicates that this output is most likely the correct one [Agg18; Nie19], which is defined in Appendix A.7.

In order to implement the functionality of neurons within a network, the network is structured into an input layer, an output layer, and one or more hidden layers. The input layer consists of neurons that receive inputs, which are then conveyed to subsequent layers. The number of neurons in this layer is equal to the number of features present in the dataset. The output layer represents the model's predictions and varies depending on the task, while hidden layers are positioned between the input and output layers. These hidden layers apply transformations to the inputs before forwarding them, with the associated weights of these neurons being optimised throughout the training process to improve predictive performance. The strength or amplitude of the connections between neurons is represented by the neuron weights [Agg18; Nie19].

Feedforward Neural Network (NN) A Feedforward Neural Network (FNN) is a network in which the output of one layer serves as the input for the next. In this configuration, neurons of hidden layers are activated upon the recognition of specific patterns, which may be components of the final output. Activation signifies that a function, such as the sigmoid function or in case of the transformer architecture the Rectified Linear Unit (ReLU) function [Vas+17], calculates the product of the input and weight, adds the bias, and produces an output within the 0–1 range. Or in case of ReLU "which outputs the input directly if it is positive, otherwise, it will output zero" [Bro20; Vas+17]. The activated neuron subsequently triggers another pattern or component, with sub-components merging through successive layers [Agg18; Nie19].

Recurrent NN A Recurrent Neural Network (RNN) processes inputs sequentially, allowing it to handle variable-length inputs.

2.2 Transformer Architecture

The transformer architecture, initially developed for sentence translation [Vas+17] of languages, has significantly influenced the design of time-series foundation models, inspiring them to adopt a similar architectural framework [Das+24a; Ilb+24; Woo+24]. While the transformer processes input sequences through its layered structure, it is a distinct architecture from that of RNNs. Rather than being a recurrent network, the transformer is a feed-forward network that uses parallelisation across its components for enhanced computational efficiency. This architecture, along with the parallelizing hardware, is what enables a breakout computation of data [Vas+17].

To fully understand the transformer architecture, it is beneficial to undertake a detailed examination of its components. At its core, the architecture is built around an encoder-decoder structure [Vas+17]. Vaswani et al. [Vas+17] highlight that the transformer incorporates embedding layers and a pre-softmax linear transformation, employing methods similar to those described by Press et al. [PW16], who implemented the word2vec skip-gram model in their research. The skip-gram approach itself became widely recognized through the influential work of Mikolov et al. [Mik+13; ZJT16].

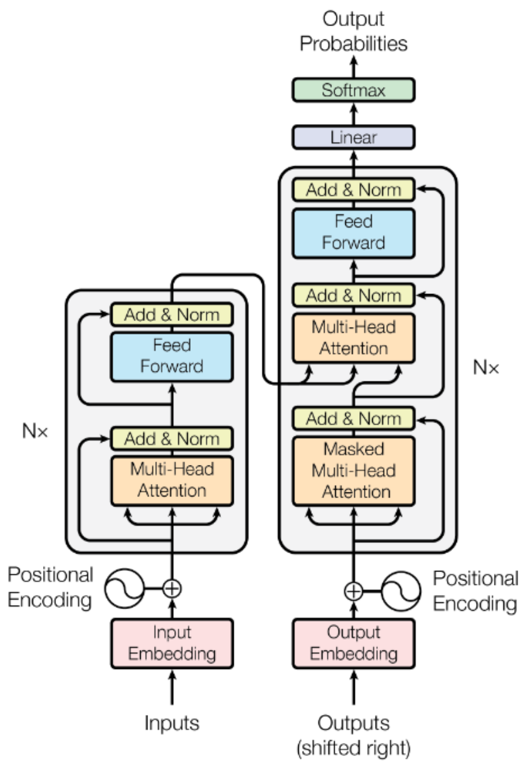

Figure 1 The Transformer - model architecture [Vas+17].

The architecture begins with an input embedding for the input sentence in the source language. This requires representing words or tokens as vectors with a dimensionality of d_{model} [Mik+13; Vas+17]. A token is a human-readable text that is translated via tokenization into a sequence of distinct tokens that can be subsequently used by statistical models [Sch+24]. This includes words, subwords, characters and special symbols.

Mikolov's [Mik+13] skip-gram model, which operates similarly to a feedforward neural network with a projection layer, is applied to identify the probability distribution over context words. The model makes predictions regarding tokens within a specified range both preceding and following the current token, assigning weights based on their contextual relevance. In the transformer architecture, these weights are further scaled by a factor of $\sqrt{d_{model}}$ to enhance stability during training. The same thing happens to the output embedding [Vas+17], where the sequence may vary in terms of both its input and output [Rad+19].
Once the tokens are converted into vectors, it is necessary to determine their sequential order. This is achieved through positional encoding, which encodes the positional information of each token. The positional encodings are added to the input embeddings, ensuring they share the same dimension, d_{model}, as the embeddings themselves. This is defined mathematically in Appendix A.8.

Encoder The encoder component of the model consists of a stack of $N = 6$ identical layers, each comprising two sublayers. The initial sublayer is a Multi-Head Attention mechanism. To understand Multi-Head Attention, it is essential to first grasp the concept of attention. The transformer architecture employs self-attention, which evaluates the similarity of each word in a sentence to every other word, including itself. Words that frequently co-occur or are strongly associated receive higher similarity scores. Technically, the attention mechanism is defined as a function that maps a query and a set of key-value pairs to an output, with the query, keys, values, and output all represented as vectors [Vas+17].

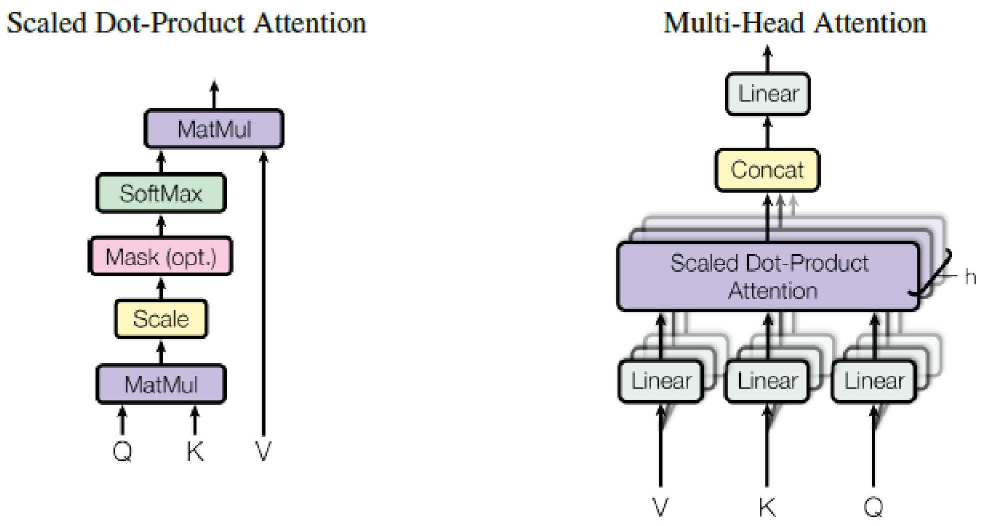

Figure 2 (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel [Vas+17].

The output of the attention mechanism is calculated as a weighted sum of the values, with weights determined by a compatibility function between the query and the corresponding key. The queries and keys are vectors of dimension d_k which represent the input tokens [Vas+17]. The compatibility is computed using the dot product, which involves multiplying each pair of queries and key elements and summing the results. Subsequently, the resulting product is then scaled by dividing it by $\sqrt{d_k}$ to maintain numerical stability, followed by the application of a softmax function [Vas+17]. This function transforms the scaled values into a probability distribution ranging from 0 to 1, with the total sum equal to 1 [RW23]. The resulting probabilities are then multiplied by another set of values, with dimension d_v, to produce the weighted outputs [Vas+17]. The overall function is expressed as:

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

The attention mechanism is applied simultaneously across a set of queries, forming a matrix Q, while K represents the keys and V the values [Vas+17]. The scaled dot-product attention serves as a fundamental component of multi-head attention. Prior to the application of this mechanism, the queries, keys, and values are linearly projected h times using distinct, learned linear projections. This approach allows the model to capture diverse aspects of relationships between words in a sentence [Vas+17].
In the architecture described by Vaswani et al. [Vas+17], there are h parallel attention layers, where each layer operates with dimensions $d_k = d_v = d_{model}/h = 64$. For each of these learning techniques, the attention function a is independently applied. The outputs, each with a dimension of d_v, are subsequently linked and projected through another learned linear transformation, effectively combining the results from all attention heads [Vas+17]. This process is mathematically represented as:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

A residual connection is applied to the matrix to preserve knowledge by adding the outputs from the positional encoding stage. This mechanism guarantees that information from earlier layers is directly passed to subsequent layers, helping to retain essential features and preventing information loss [AGK23]. Furthermore, a normalization layer [BKH16] is incorporated, standardizing the outputs of the residual connection to improve training stability. The "add and normalize" operation combines inputs from the positional encoding and multi-head attention layers, subsequently normalizing the combined output [AGK23]. Both the residual connection and normalization layer are integral to all sublayers within the transformer architecture, including both the encoder and decoder components.

The next layer of the encoder is the position-wise feed-forward network. As outlined in Section 2.1, the network performs the specified functionality, applying a ReLU activation function [Vas+17]. It processes each input vector independently, enabling the detection of patterns specific to individual inputs. Each position in the input sequence is treated separately but identically, ensuring uniformity across positions while preserving the learned relationships between them. This design allows the model to maintain consistency and effectively capture localised patterns.[Gev+20].

Decoder Similar to the encoder, the decoder comprises a stack of $N = 6$ identical layers. The input to the decoder combines word embeddings with positional encoding, incorporating a checkpoint mechanism to signal the end of the sentence. This results in the output embeddings being offset by one position for each translated word [Vas+17].

The first sublayer within the decoder is a multi-head attention layer, modified by a masking mechanism that prevents positions from attending to future positions in the sequence [Vas+17]. This masked multi-head attention layer operates similarly to the standard multi-head attention layer but applies a mask to ensure that each token at position i only attends to itself and preceding tokens, effectively ignoring future tokens [AGK23]. The output of this layer is passed through a residual connection and a layer normalisation before entering the "encoder-decoder attention" layer. In the “encoder-decoder-attention”-layer, the queries are derived from the output of the previous decoder layer, while the keys and values come from the encoder’s output. This setup allows each position in the decoder to attend to all positions in the input sequence, thereby capturing the significance of input words and their relationship with the output sequence [Vas+17].
The output of the “encoder-decoder-attention”-layer is then passed through a fully connected feed-forward network, which processes the values to generate the final scores. A softmax function is applied to these scores in order to select the most appropriate output word. The output tokens can be described as a vocabulary, with all possible outcome words or characters. The translated word is then fed back as an input to the output embedding layer, and the decoder repeats the process until it encounters the checkpoint indicating the end of the sentence [Vas+17].

2.3 Time Series Foundation Models

2.3.1 TimesFM

The research conducted by Google Research, as described by Das et al.[Das+24a], explores whether "large pretrained models trained on massive amounts of time-series

data learn temporal patterns that can be useful for time-series forecasting on previously unseen datasets?" [Das+24a]. Central to this analysis is whether a time-series foundation model can be designed to achieve a robust zero-shot out-of-the-box forecasting performance.
To address these questions, Google Research developed TimesFM (Time-series Foundation Model), which serves as a general-purpose zero-shot forecaster. As described by Das et al. [Das+24a], this model is trained on diverse time-series data, enabling it to have general knowledge and detect patterns in forecasting tasks, such as trend and seasonality patterns [Lia+24a]. According to Liang [Lia+24a], this diversity in training data provides TimesFM with a broad capability to handle a variety of forecasting scenarios. A key feature of the zero-shot forecasting capability is that the model can make predictions on new target datasets without any prior training on those specific datasets [Gru+24].

Architecture The architecture of TimesFM is designed to process a specified number of time points, denoted as C, from a given time series as context. This is represented mathematically as $y_{1:i} := y_1, ..., y_i$, and to predict the future H time points, denoted as $y_{i+1:i+H}$ [Das+24a]. This predictive mechanism is expressed formally as:

$$f : (y_{1:i}) \to \widehat{y}_{i+1:i+H}$$

where i refers to the index or length of the last time point of the time-series.

The design allows the model to adapt flexibly to varying lengths of context (C) and prediction horizons (H), while maintaining sufficient capacity to encode patterns learned from a large and diverse pretraining dataset [Das+24a]. Transformers have demonstrated adaptability to different context lengths in tasks within Natural Language Processing [Rad+19].

Still, the TimesFM architecture incorporates several modifications to the standard transformer framework to optimize its performance for time-series forecasting tasks. Time-series data are segmented into patches during training, which function similarly to tokens in Natural Language Processing models. This patch-based approach enhances both performance and inference speed by reducing the number of tokens fed into the transformer by a factor corresponding to the patch length. This can be seen in figure 3, which illustrates the importance of parallel hardware [Das+24a].
Another key distinction in TimesFM is, that it is a decoder only model. The model processes a sequence of input patches and is optimized to predict the next patch as a function of all preceding patches [Das+24a]. For instance, TimesFM predicts future electricity consumption patterns by taking a historical sequence of the data, splitting it into patches, and using its architecture to forecast subsequent patterns in

the sequence.
Unlike the original transformer architecture, which typically processes one token at a time, TimesFM permits the use of longer input patches. This design is especially advantageous for long-horizon forecasting tasks, where directly predicting the entire horizon has been observed to receive greater accuracy compared to multi-step auto-regressive decoding [Zen+23]. However, for scenarios involving unknown horizon lengths, TimesFM introduces additional flexibility. TimesFM welcomes unknown horizons by allowing the output patches to be longer than the input patches. The model predicts the future horizon in chunks (longer patches) and, when necessary, uses these predicted patches as the new context for subsequent predictions. This capability enables dynamic forecasting over variable and potentially unknown horizon lengths, demonstrating the adaptability and robustness of the TimesFM architecture in complex time-series forecasting scenarios [Das+24a].

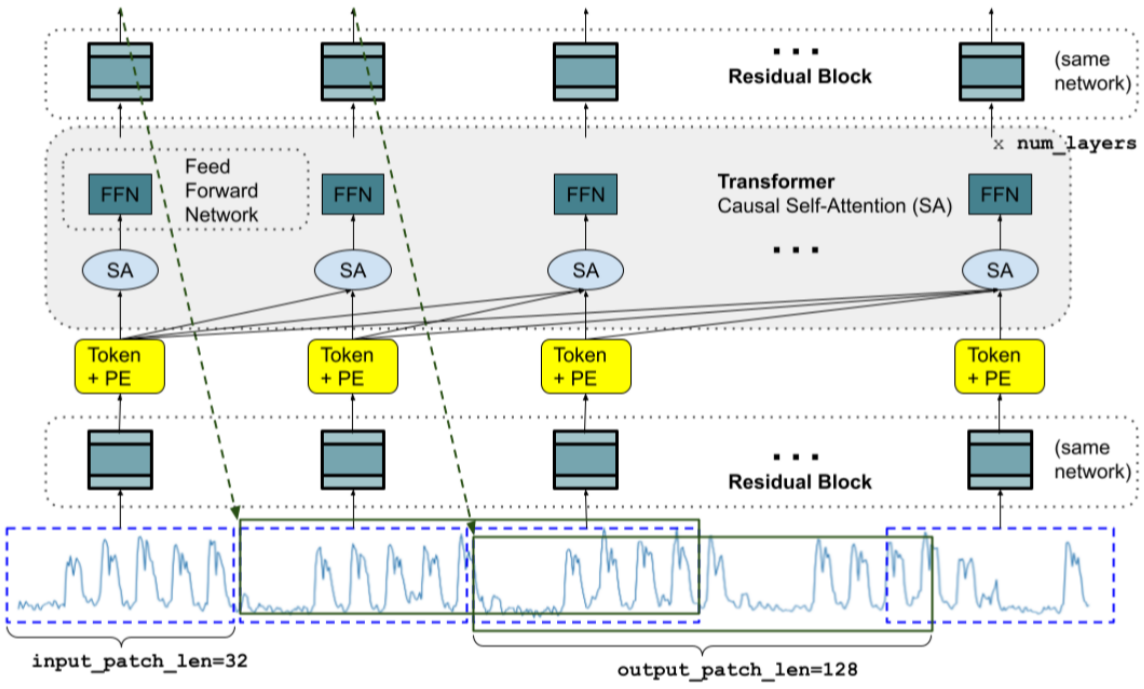

Figure 3 The TimesFM model architecture during training [Das+24a].

To gain a deeper understanding of the TimesFM architecture, it is essential to examine its input, transformer, and output layers.
The input layer is responsible for preprocessing the time-series data into input tokens. This is achieved by dividing the data into non-overlapping, bordering patches. These patches are then processed by a residual block, which transforms them into vectors with dimensions matching the model's predefined size, as defined by dimensions $model_{dim}$ [Das+24a].
The residual block in this preprocessing step is based on the time-series Dense Encoder introduced by Das et al. [Das+23]. This encoder incorporates an ReLU activa-

tion function, enabling the residual block to effectively capture patterns in the input patches while preserving the structural integrity of the time-series data.

In the TimesFM architecture, a binary padding mask, denoted as $m_{1:i}$, is provided alongside the input data. The mask assigns a value of 1 to indicate that the corresponding input in $y_{1:i}$ should be ignored, or a value of 0, which specifies that it should be used. The Input Residual Block is mathematically expressed in Appendix A.9. Following the preprocessing of input tokens, the architecture makes use of a stacked transformer module comprising N layers. Each layer includes a multi-head self-attention mechanism, which is succeeded by a feed-forward network. This configuration ensures that the tokens can only attend to input tokens that precede them in the sequence, preserving the temporal order of the time-series data [Das+24a]. The process is mathematically represented in Appendix A.10.

Last but not least, the output layer of the TimesFM architecture defines the predictions to be generated. Operating in decoder-only mode, each output token is trained to predict the portion of the time-series that immediately follows the last input patch associated with it. After processing through the transformer layers, the output tokens pass through a residual block that maps them to the corresponding predictions [Das+24a]. The process is mathematically represented in Appendix A.11.

Training TimesFM was trained on a large amount of time-series data, with the dimensionality of the dataset increasing to 200 million biases and weights, and the pre-training data containing around 100 billion timepoints. This data came from a variety of domains, including trend and seasonality patterns, but also time granularities, from sources such as google trends, wiki pageview statics and synthetic time-series. Additionally, data from publicly accessible sources such as electricity and traffic data [Das+24a] was used.
In order to train the model, each patch of time series data in a set was assigned a random number r and was sampled between 0 and the input patch length $p-1$. The first r elements of the mask $m_{1:r}$ are set to 1, and the rest are set to 0. This effectively masks out a fraction of the first input patch. Using this approach, it is ensured that all possible input context lengths, from 1 to the maximum training context length [Das+24a], are included.

2.3.2 TimeGPT-1

TimeGPT-1, developed by Nixtla, is the first pre-trained foundation model designed with a specific focus on time-series forecasting, capable of performing zero-shot predictions. This capability enables the model to make forecasts across diverse domains and applications without requiring explicit training on each specific dataset or task. It does this by using transfer learning, which underlying concept plays a crucial role in the design and functionality of TimeGPT-1. Transfer learning involves applying knowledge gained from one task to address a new one. In this context, the model is pre-trained on a large and diverse dataset, which allows the model to enhance its forecasting performance on datasets including characteristics such as frequency, sparsity, trend, seasonality, stationarity, and heteroscedasticity [GCM23].

The TimeGPT-1 model is structured to address time-series forecasting. The input features are defined as $X = \{y[0:t], x[0:t+h]\}$, where the historical time points are represented by $(y[0:t])$ and covariates by $(x[0:t+h])$. These covariates represent external variables that are available for either the historical period and the forecast horizon, thereby providing additional contextual information for the model's predictions. The target values for the specified forecasting horizon are defined by $Y = \{y[t+1:t+h]\}$ as the set of prediction time points [GCM23].
The model operates by estimating the conditional distribution necessary to forecast a specific period. This is mathematically expressed as:

$$P(y_{t+1:t+h} | y_{[0:t]}, x_{[0:t+h]}) = f_\theta(y_{[0:t]}, x_{[0:t+h]})$$

where h denotes the forecast horizon, and f_θ represents the function parameterized by θ, capturing the relationships between historical data, covariates, and the forecasted outcomes [GCM23].

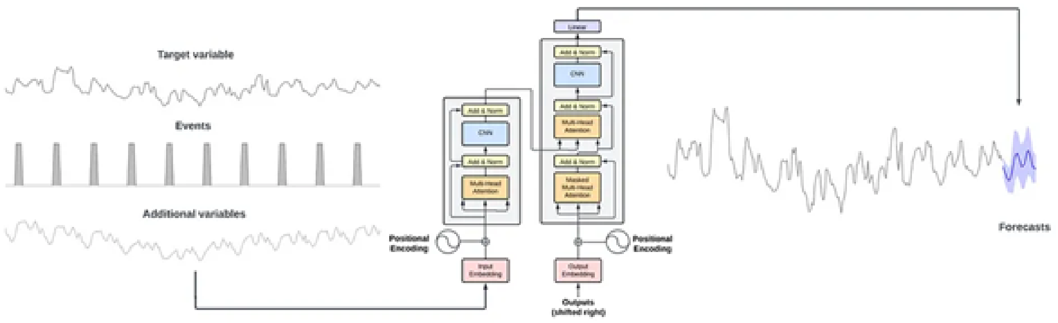

Figure 4 The TimeGPT-1 model architecture during training [GCM23].

Architecture TimeGPT-1 is built upon the transformer architecture and retains many of its core principles while being specifically adapted for time-series forecasting. Similarly to the original transformer, it processes input data in a tokenized format, where the input consists of a window of historical target values augmented with additional external variables (covariates). To improve the representation of sequential information, positional encoding is added to the input. The architecture adopts the encoder-decoder structure with multiple layers, each incorporating residual connections and layer normalization, mirroring the transformer design [GCM23]. In order to generate predictions, a linear layer maps the decoder's output to the dimensions of the forecasting window. This mapping captures the diversity of past events and enables the model to forecast future distributions accurately, relying on patterns identified in the training data. TimeGPT-1 is capable of processing time-series data with a range of frequencies, characteristics, input sizes, and forecasting horizons. Although inspired by the transformer model, TimeGPT-1 has been specifically tailored to handle the unique challenges of time-series forecasting. These include diverse frequencies, trends, seasonality, and non-stationary behaviours. Moreover, the model is trained to minimize forecasting error [GCM23].

Training As documented by Garza et al. [GCM23], TimeGPT-1 was trained on the largest time-series dataset available at the time of its development. The training process involved over 100 billion data points and required multiple days to complete. This was achieved through the integration of a cluster of NVIDIA A10G GPUs [GCM23]. Furthermore, the Adam Optimizer [KB14] was applied for the optimization process. The dataset spanned over a diverse range of domains, including finance, economics, demographics, healthcare, weather, IoT sensor data, energy, web traffic, sales, transport, and banking. This broad coverage provided TimeGPT-1 with exposure to a wide variety of time-series patterns across multiple fields, enabling it to develop a deep and versatile understanding of time-series data [GCM23].

2.3.3 Comparison of TimesFM and TimeGPT-1

	TimesFM	TimeGPT-1
Architecture		
Organisation	Google Research but not an officially supported Google product	Nixtla
Introduction	October 2023	February 2024
Model Architecture	Based on transformer, Decoder-only	Encoder-decoder transformer architecture
Self-Attention	Causal self-attention for sequence modeling – "each output token can only attend to input tokens that come before it in the sequence"	Self-attention in encoder and decoder layers, like in the transformer architecture
Residual Connections	Used in input, transformer, and output layers	On each layer of the encoder-decoder structure
Positional Encodings	Applied to input patches for temporal ordering	Applied to all input data

	TimesFM	TimeGPT
How they handle varying context lengths	Uses patching to break the input sequence into smaller, manageable chunks	The encoder processes the full historical sequence up to a specified maximum length
How they handle varying horizon lengths	Outputs are generated as larger chunks (longer output patches) rather than individual timesteps	The decoder generates each future timestep by using previously generated outputs as input for predictions
Testing		
Zero-shot support	Yes	Yes
Fine-tuning support	Yes, improves performance further on domain-specific tasks	Yes, can be fine-tuned for specific datasets
Covariate support	Yes	Yes
Training		
Parameter size for weights and biases	17M, 70M, and 200M models evaluated during training experiments	Not explicitly mentioned in the paper
Pre training data size	O(100 billion) time points	Over 100 billion data points
Domains	Google Trends, Wiki Pageviews, synthetic data, data from public sources such as electricity and traffic data	finance, economics, demographics, healthcare, weather, IoT sensor data, energy, web traffic, sales, transport, and banking
Optimizer	Adam	Adam
Computation Requirements	TPUv5e with 16 cores	NVIDIA A10G GPUs
Time used for Training	200M model took 2 days to complete 1.5M iterations	600 milliseconds per series for training and inference
Model Accessibility	TimesFM from previous installed environment, pre-trained model checkpoints available on Hugging Face	Importing Nixtla from previous installed environment, API for low-code time-series forecasting available
Context lengths	Takes historical data (≤ 512) as input, but can be specified within the parameters as well	Takes historical data used as input
Forecasting Horizon	Flexible, supports varying history and horizon lengths	Flexible, supports varying history and horizon lengths

Table 1 Comparison on Architecture, Training and Testing [Das+24a; GCM23]

3 Related Works

3.1 Traditional Methods

Several methods can be used for time series forecasting, traditional methods thereby include statistical methods, such as rolling averages, vector Autoregressive (AR) and ARIMA and exponential smoothing [Das+24a; McK84; SS17]. One particular example that arises from an interest in the correlation between past and present values at different time intervals, known as lags, is ARIMA. These relationships are represented by AR and Moving Average (MA) components, as well as their combination in Autoregressive Moving average (ARMA) models. To address non-stationary data, the ARIMA model was developed, which incorporates all three components: autoregressive, moving average, and differencing (integration) [SS17]. These traditional methods are inherently linear in nature and may consequently become outdated in the context of the progressive development of deep learning models, which possess the capacity to incorporate non-linear factors [Els+21; SS17].

3.2 Recent Methods

Garza et al. [GCM23] states the early successes of deep learning in time series forecasting to the adaptation of established architectures like RNNs and Convolutional Neural Network (CNN). RNNs were foundational to models such as DeepAR [Sal+20], while CNNs, shown to outperform RNNs in sequential data tasks [BKK18], became integral to modern models like DPMN [Oli+24] and TimesNet [Wu+22].
Transformer-based models [Vas+17] have recently gained popularity, demonstrating exceptional performance in large-scale and long-sequence forecasting tasks [Kun+23]. Das et al. [Das+24a] highlight recent advancements in adapting LLMs such as GPT-3 and LLaMA-2 for zero-shot forecasting [Gru+24], alongside fine-tuning methods for models such as GPT-2 [Zho+23; CPC23]. While most studies focus on fine-tuning pretrained models, TimesFM pioneered a zero-shot foundational model for diverse datasets, paralleled at that time only by TimeGPT-1 [GCM23].
Since the release of TimesFM in April 2024, transformer-based forecasters with zero-shot capabilities — such as MORAI [Woo+24], SAMformer [Ilb+24], and MOMENT [Gos+24] — have emerged. Additionally, the Tiny Time Mixer (TTM) [Eka+24] model demonstrates zero-shot learning using the TSMixer/MLP-Mixer architecture rather than a transformer-based approach.

4 Methodology

This study focuses on assessing the accuracy of time-series forecasting models for zero-shot forecasting and explores whether fine-tuning the models or incorporating covariates can enhance their predictive performance. To achieve this, a secondary data analysis is conducted in according to the guidelines provided by Donnellan et al. [DL13]. This process begins with the identification and preparation of the datasets, the main variable and the covariate used, ensuring they are appropriately formatted for the forecasting models and that the timeframe selected aligns with the research objectives. An initial analysis of the data should show if it is stationarity or non-stationarity and if there is a correlation before applying the forecasting models. The next step involves defining the evaluation parameters used in the data analysis. This metrics is influenced by the methodologies of Das et al. [Das+24a], Goswami et al. [Gos+24], Ekambaram et al. [Eka+24], Woo et al. [Woo+24], and Garza et al. [GCM23] who propose several evaluation parameters for zero-shot and fine-tuning. Further insights from Chiccio et al. [CWJ21] provide a selection of parameters for assessing the models' performance when covariates are included. Once the evaluation parameters are set, the settings for the foundational models are chosen and explained. These include various configurations such as context length, horizon length, and long-horizon forecasting. Fine-tuning settings, such as epochs, learning rate and patience, are also introduced.

The experimental phase of the study applies these settings systematically to compare model performance. Zero-shot forecasting is first conducted using a context length of 416 data points (days) and horizon lengths of 7, 30, (last) 60, and 128 days. Following this, the models are fine-tuned using the full dataset spanning from 2015 to 2019 and are evaluated across the same horizon lengths. This process is repeated with the addition of covariates to assess whether zero-shot forecasting and fine-tuning improves with their integration.

As a benchmark, the traditional time-series forecasting model ARIMA is implemented too. ARIMA is evaluated with the full dataset from 2015 to 2019 and the same horizon lengths as the foundational models. Additionally, it is tested with and without covariates.

5 Experimental Results

5.1 Dataset

This chapter presents the datasets selected for the experiment conducted in this thesis. It begins by outlining the preparation of the primary dataset used for time series forecasting. Subsequently, it details the identification and preparation of the covariate data, including an analysis of the data prior to the application of the foundation models, including an examination of trends, correlations, and other relevant characteristics. Furthermore, the evaluation parameters and experimental settings are defined, prior to presenting the results of the forecast.

5.1.1 Identifying and Preparing the Main Data

In their study, Donnellan et al. [DL13] provide a systematic approach to initiating a secondary data analysis. The first step involves identifying relevant existing datasets. Researchers may draw upon prior experience with datasets or use established archives to support this process. In their chapter, Donellan et al. [DL13] also suggest specific archives and resources to assist researchers in locating appropriate datasets.
The dataset presented in this paper was identified through comprehensive online and archival research, resulting in a small list of potential datasets. The final selection process was conducted in accordance with specific criteria essential for time-series forecasting [DL13]. Primarily, the data must be collected in a time-ordered sequence and associated with a defined time frame. Furthermore, the dataset must display consistency in its measurements, which includes the absence of gaps, uniformity in measurement units, stable frequency, and an unaltered data collection methodology [Wei13].

The second step is to undertake a detailed examination of the dataset descriptions. This typically involves an examination of the dataset's headings and variables, along with contextual information about the data, which is often provided in accompanying README documents [Che15; 20; DL13]. In some cases, prior studies that have used these datasets are referenced, thereby offering insights into their usage [Che15]. Furthermore, crucial information regarding the dataset's alignment criteria can often be obtained from these descriptions without direct access to the data themselves. This is particularly useful when access to the data are restricted or subject to approval processes [DL13].

The third step requires the selection of a dataset and the maintenance of a detailed record of all modifications made to it [DL13]. The dataset selected for this study covers electricity consumption, wind and solar power generation, and electricity prices, recorded at intervals of 15 minutes, 30 minutes, and 60 minutes. The most recent update was conducted in 2020, with annual updates provided since 2015. The dataset encompasses data from 32 European countries and is sourced from the ENTSO-E Transparency platform [20].
This dataset was selected due to its extensive coverage, spanning multiple years, making it suitable for long-term analysis [20]. This characteristic enables zero shot forecasting with 416 historic data points, as outlined by Das et al. [Das+24a], but also incorporates more historical data for fine-tuning. Additionally, the dataset exhibits minimal missing values, with only a few gaps at the beginning and end of certain fields. It also meets the required criteria of being organised in a time-ordered sequence and linked to precise timestamps.

To pre-process the data for use in the forecasting models, appropriate variables were selected. The first variable chosen was cet_cest_timestamp, representing the start of a time interval in Central European (Summer) Time. While the specific time variable was not critical, it was essential that it included a date. The second variable, which was the target for forecasting, was GB_GBN_load_actual_entsoe_transparency, representing total electricity consumption. This variable was selected because consumption is a summable value, an important characteristic when identifying a suitable covariate. This flexibility allows the data to align with various temporal resolutions, such as daily or monthly intervals.
The focus on Great Britain's energy consumption was driven by the availability of a corresponding covariate dataset, which was structured in a daily sequence. Consequently, the hourly electricity consumption data was aggregated into daily totals. This transformation was performed using Excel's Pivot Table tool, after the variables cet_cest_timestamp and GB_GBN_load_actual_entsoe_transparency were copied into a separate worksheet. Any missing data at the beginning of the dataset was removed during this process. A preliminary scan of the data indicated no further missing values or anomalies.
Afterwards, the data was filtered to include the years 2018 and 2019 for the purposes of zero-shot forecasting and the years 2015 to 2019 for fine-tuning. Which can be traced back to Das et al. [Das+24a] implying that the amount of context length for zero-shot forecasting, should not exceed 512 data points for the time-series forecasting models. Additionally, both models use long time-series data to fine-tune their models [Das+24a; GCM23].
The original dataset, initially in MS Excel format, was converted to a CSV file for

compatibility with Jupyter Notebook, thereby enabling more precise data handling and analysis.

5.1.2 Identifying and Preparing the Data Covariate

As the focus of this paper is on evaluating the performance of the foundation models with covariates, it is necessary to identify a suitable covariate for the primary forecasting variable. The selection process followed a methodology similar to that used for identifying the primary variable [DL13], with criteria including data collection in a time-ordered sequence, association with a defined time frame, consistency in measurements, and a stable data collection methodology [Wei13]. However, since this study aims to investigate whether a covariate related to the primary variable can improve the accuracy of time series forecasting, an additional criterion was required: the covariate must have an influence on electricity consumption.

In accordance with the methodology outlined by Donnellan et al. [DL13], an online search was conducted to identify potential datasets. However, before locating specific datasets, it was necessary to determine a relevant theme. This was achieved using the search string ("electricity consumption") AND (influence). A preliminary review of article titles quickly revealed a potential link between temperature and electricity consumption. This connection was further validated by a focused search using the string ("electricity consumption" AND temperature). The findings of studies such as those by Harish et al. [HST20], Zhang et al. [ZLM19], Amber et al. [AAH15], and Thornton et al. [THS16] have investigated this relationship and consistently concluded that temperature significantly influences electricity consumption.

Once the connection between temperature and electricity consumption had been established, it was necessary to identify a suitable dataset. An online search was conducted to locate a dataset providing daily temperature data for one of the 32 countries represented in the primary dataset, covering the period from the beginning of 2015 to the end of 2019. Additionally, the dataset needed to meet the established criteria outlined by Wei et al.[Wei13].
A suitable variable was identified in the CEDA Archive, a repository specialising in atmospheric and earth observation data. The "Met Office Hadley Centre Central England Temperature (HadCET) Series" [CM06] was selected, which provides daily mean temperature data for central England dating back to 1772. In accordance with the second step of Donnellan et al.'s [DL13] methodology, the dataset's description confirmed that the temperature values are recorded in Celsius. However, the data are not immediately accessible; users must first submit an application specifying the

intended purpose of the data. Following the submission of the required application, access rights were granted within one day.

In conclusion, for the third step of Donnellan et al.'s [DL13] framework, this dataset was selected because it aligns well with the primary dataset. Specifically, it provides daily temperature data for central England, corresponding to the electricity usage variable for Great Britain in the main dataset. Furthermore, the dataset contains no gaps or anomalies and satisfies the criteria of being a time-ordered sequence with precise timestamps.

In order to prepare the data for use in the forecasting models, an appropriate variable was selected. This was a relatively straightforward process, as it was necessary to represent the average daily temperature for the same years as the primary variable. The average daily temperature was chosen to provide a general representation of the climate on each day. Consequently, the variables dlycet_mean_2018.dat and dlycet_mean_2019.dat (and so forth to 2015 for fine-tuning) were selected. The data was available for download as a text file, which was subsequently converted into an MS Excel format for further processing.
In Excel, the data was arranged in chronological order according to the day on which they occurred. Entries corresponding to "missing" days, such as February 29th, 30th, and 31st, which were indicated by the placeholder value -32768, were removed. The resulting dataset consisted of two rows: one representing the date in a daily time order, and the other showing the daily average temperature for each corresponding date. Only the necessary number of decimal places has been added in order to obtain the corresponding values in the celsius format.
In the final stage of the process, the temperature data was integrated into the main dataset as a third row, alongside the existing variables for daily electricity consumption. The combined dataset was then converted into a CSV format for compatibility with the forecasting models, ensuring it contained both daily electricity consumption and temperature data.

5.1.3 Data analysis/description

In order to ascertain the nature of the data presented in the form of electricity consumption and temperature, Wei et al. [Wei13] set out a structured analytical approach. As the approach involves the analysis of a single variable at a time, such data may be categorised as stationary or non-stationary and seasonal or nonseasonal. Stationary time series models assume that statistical properties such as mean, variance, and autocovariance remain constant over time. This concept is divided into strict stationarity, where the joint probability distribution of variables does not change over

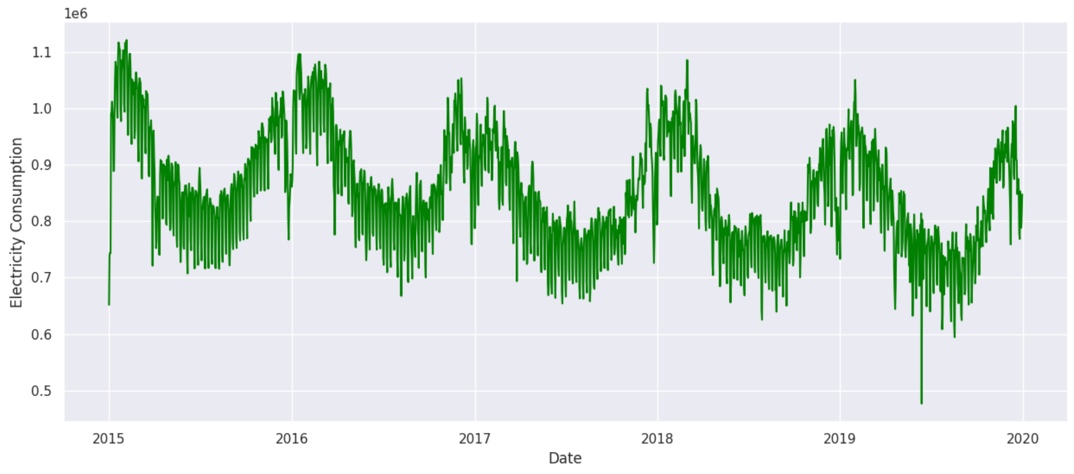

Figure 5 Plot of Electricity Consumption, y-axis in million. See appendix C.1.

time, and weak stationarity, which requires only that the mean, variance, and covariance remain constant. Non-stationary time series models differ by allowing changes in the mean and/or variance over time, reflecting a more dynamic process. Seasonal data can be described as patterns that recur at regular intervals, often linked to predictable temporal phenomena, such as high temperatures in summer [Wei13; HP22].

The visualisation in Figure 5 illustrates the existence of distinctive seasonal patterns, with higher electricity consumption observed from November to March and lower levels from May to September. Additionally, a weak negative trend can be detected. It is evident that from 2015 to 2020, there was a marginal decline in electricity consumption. These variations suggest that the data may exhibit non-stationary characteristics. To confirm this initial observation, further analysis is required to test the stationarity of the data formally [Wei13].

As the second step by Wei et al. [Wei13] suggests, a Dickey-Fuller test can be applied to determine whether a series is stationary or requires the removal of trend patterns. The p-value of this test is used to identify whether the data are stationary; a p-value below 0.05 indicates that the data are stationary. In the case of the electricity consumption data, the p-value is 0.0229, which is below the 0.05 threshold and therefore indicates that the data are stationary [Wei13]. Furthermore, the time series is used to compute and examine the sample Autocorrelation Function (ACF) and the sample Partial Autocorrelation Function (PACF) of the original data. These functions are applied to verify whether data are stationary and potentially seasonal as well. Additionally, they are used to identify the parameters for the baseline model ARIMA and therefore further analysed in section 5.3.

The ACF quantifies the correlation between a time series and its lagged versions at various time lags, representing the relationship between current values and their pre-

ceding values. These lags represent intervals within the data where comparisons are made, and the results are typically visualised in a plot [Wei13].
The ACF is derived by calculating the covariance between the data values at a given time point (Z_t) and the values at a subsequent lag (Z_{t+k}), normalised by the variance of both the current data values and their corresponding lagged values [Wei13].

$$p_k = \frac{Cov(Z_t, Z_{t+k})}{\sqrt{Vark(Z_t)}\sqrt{Vark(Z_{t+k})}}$$

In contrast, the PACF provides insights that are similar in nature, but which eliminate the influence of linear dependencies on intermediate variables between the current value and its lagged counterpart. Essentially, the PACF represents a conditional correlation, isolating the direct relationship between (Z_t) and (Z_{t+k}) by accounting for the effects of other intervening lags [Wei13].

$$p_k = Corr(Z_t, Z_{t+k}|Z_{t+k}, ..., Z_{t+k-1})$$

Wei et al. [Wei13] suggest using approximately n/4 lags, where n denotes the length of the dataset, for these observations. As demonstrated in Figure 6, the ACF highlights the pattern that can be detected, namely a regular variation in the shape of the curve. This provides more evidence that seasonal behaviour is a factor.

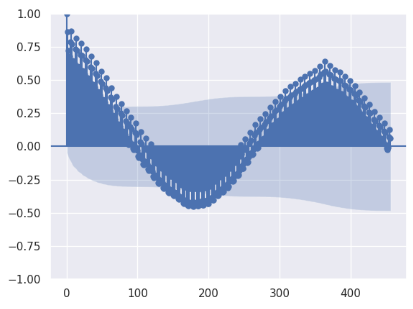

Figure 6 ACF of Electricity Consumption. See appendix C.1.

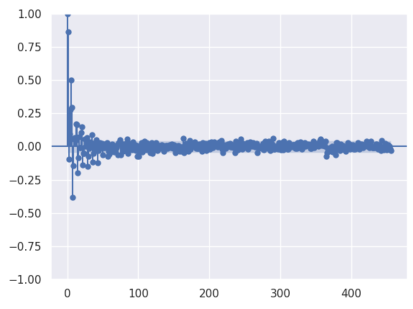

Figure 7 PACF of Electricity Consumption. See appendix C.1.

The methodology previously outlined for analysing electricity consumption is now applied to the second variable, temperature, serving as a covariate. Such as electricity consumption, the time series of temperature must be assessed for potential stationarity and seasonality to ensure a comprehensive understanding of its behaviour and its interplay with the primary variable [Wei13].
As seen above, the first step involves plotting the time series data for temperature [Wei13]. Figure 8 reveals clear seasonal patterns, with higher temperatures observed from May to October and lower temperatures from November to April. These fluctuations align with expected seasonal patterns, thereby reinforcing the assumption that temperature exhibits seasonality. Consequently, the data are also likely to be

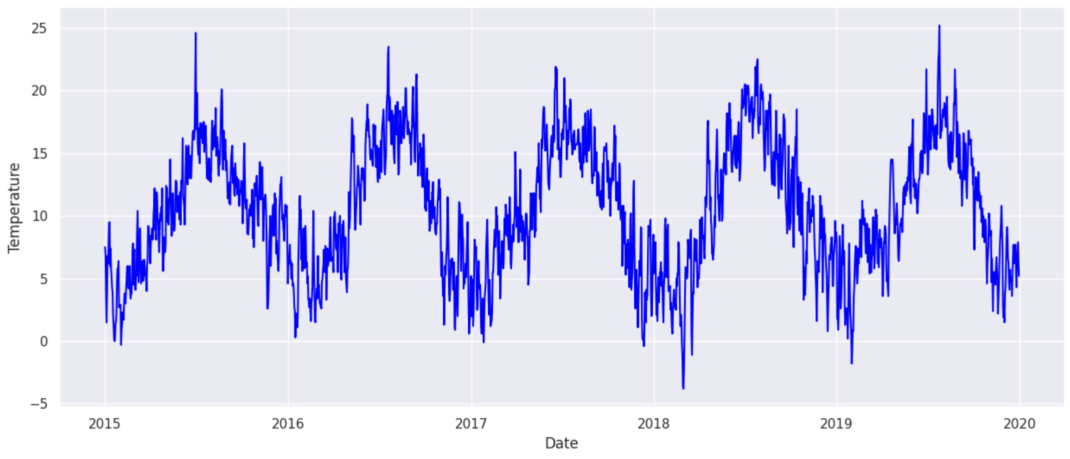

Figure 8 Plot of Temperature, y-axis in Celsius. See appendix C.1.

stationary, as it exhibits regular seasonal intervals [Wei13]. This preliminary observation sets the stage for the second step, during which the Dickey-Fuller test is calculated and ACF and PACF are computed and analysed [Wei13]. The Dickey-Fuller test exhibits a p-value of 0.0499, which is just barely below the threshold of 0.05. To confirm or disprove the results of the tests, the process follows the approach

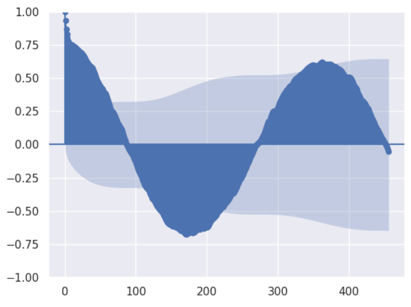

Figure 9 ACF of Temperature. See appendix C.1.

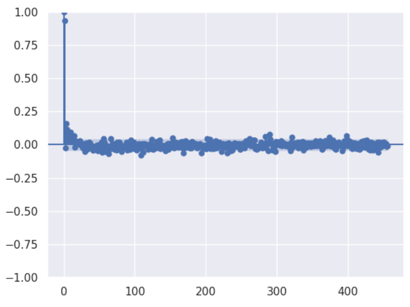

Figure 10 PACF of Temperature. See appendix C.1.

used for electricity consumption, aiming to identify seasonal influences within the temperature data. Again, the regular pattern, as indicated in the ACF in Figure 9 suggests stationarity [Wei13]. Additionally, the symmetrical pattern in the ACF values above and below zero further supports the possibility of seasonal behaviour. This characteristic implies that the temperature data exhibits recurring patterns over time, consistent with known seasonal fluctuations [Kis24].

In order to gain a deeper understanding of the relationship between electricity consumption and temperature, the analysis extends beyond the individual time series to consider their interaction. This is achieved by computing the Sample Correlation

Matrix Function to evaluate the strength and direction of the relationship between these two variables [HP22; Wei13]. Correlation measures the extent to which two variables move in relation to each other.
Before computing the correlation, it is essential to transform the seasonal time series into a non-seasonal ones. This is accomplished by differentiation, leaving the residuals, which represent the underlying data [HP22; Wei13]. These residuals are then used to calculate the correlation. The resulting value of -0.30 indicates a moderate negative correlation, see appendix C.1. The negative sign implies that as one variable increases, the other tends to decrease. This inverse relationship is further visualised in diagram 11, where opposing trends in the variables become clear [McL23].

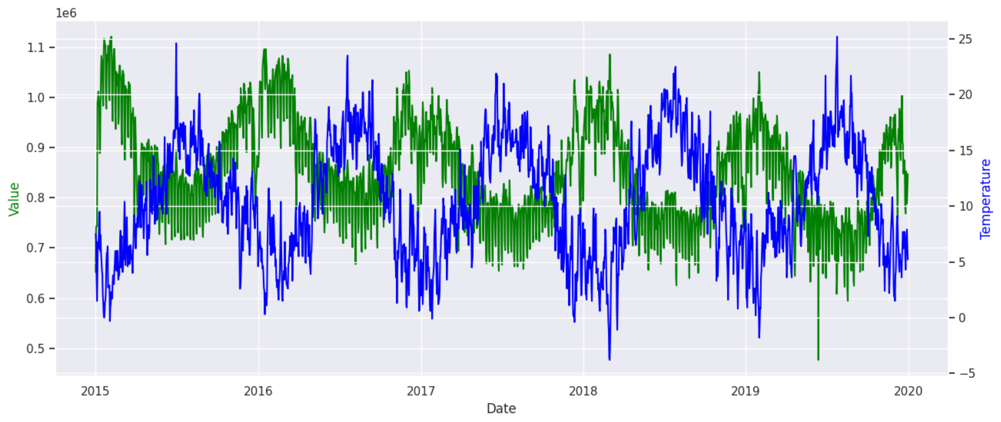

Figure 11 Plot of Electricity Consumption and Temperature for Comparison. See appendix C.1.

By applying this procedure to both electricity consumption and temperature, the analysis highlights their interaction within a stationary framework. This method allows for a clearer understanding of how these variables influence one another, laying the groundwork for more nuanced modelling and interpretation.

5.2 Evaluation Parameters

The metrics used to evaluate and compare the performance of the two foundational time series models, both with and without covariates, are based on the Mean Absolute Error (MAE), the Mean Squared Error (MSE), the Root Mean Squared Error (RMSE) and the Mean Absolute Percentage Error (MAPE). This metric should provide a comprehensive overview of the models accuracy in different aspects of error measurement. Even though, the same dataset is used for all tests, a discrepancy in scale arises due to the zero-shot accuracy assessment being performed with only 544 data points, whereas the fine-tuning process involves 1826 data points but also because the experiment is executed with different horizon lengths. To ensure the comparability of the results, the data are normalized for all error calculations, except for the MAPE, where normalisation is already part of the calculation [Rob23; CWJ21].

The MAE is a statistical measure that is also used by Das et al. [Das+24a]. It calculates the mean of the absolute differences between the predicted values (X_i) and actual values (Y_i) [CWJ21]. This metric is less sensitive to outliers, as it does not amplify errors caused by large deviations. Moreover, the MAE is expressed in the same unit as the forecasted values, making it intuitive for interpretation.

$$MAE = \frac{1}{m}\sum_{i=1}^{m}|X_i - Y_i|$$

In contrast, the MSE, used by Ekambaram et al. [Eka+24] and Goswami et al. [Gos+24], is calculated by squaring the differences between predicted and actual values [CWJ21].

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(X_i - Y_i)^2$$

This squaring process amplifies larger errors, making the metric more sensitive to outliers. However, the squaring process changes the scale of the error, meaning the resulting MSE value is in a different unit from the forecasted data. To address this issue and return to the same unit, the square root of the MSE is taken, resulting in the RMSE [CWJ21], as suggested by Garza et al. [GCM23].

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(X_i - Y_i)^2}$$

The RMSE combines the sensitivity to outliers of the MSE with the intuitive unit consistency of the MAE, providing an intermediate perspective on error magnitudes.

The MAPE is a bit different to other error-based metrics in that it is defined as the average absolute percentage difference between predicted values and actual values. It measures the extent to which predictions differ from actual outcomes, which indicates that a lower MAPE value indicates a more accurate prediction. For example a MAPE of 10% means that the prediction is off 10% to the actual values [Rob23]. It is calculated by subtracting the predicted values (Y_i) from the actual values (X_i) and dividing by the predicted values (Y_i). The total is summed and the mean is determined.

$$MAPE = \frac{1}{m}\sum_{i=1}^{m}\frac{(Y_i - X_i)}{Y_i}$$

To thoroughly evaluate the models, their performance is compared across multiple horizon lengths. The shortest horizon, spanning seven days, follows the suggestion by

Garza et al. [GCM23], offering a quick assessment of near-term predictive accuracy. Furthermore, a 30-day horizon is also analysed to evaluate the models' performance over a single month, this and the following forecasts are classified as long horizon forecasting by TimeGPT [GCM23]. Additionally, a 60-day horizon, representing the last two months of the data horizon, is included based on the first observations of the forecasting plots 16, which suggest that both TimesFM and TimeGPT may be less accurate over this period without covariates. Finally, the longest horizon, comprising 128 data points, is used to assess long-term predictive accuracy.

In addition to evaluating the accuracy of the models, TimesFM and TimeGPT-1 are assessed on their ease of use. This assessment is based on factors such as the flexibility of parameters and the simplicity of implementing the pre-built models.

5.3 Experimental settings

To evaluate the accuracy of foundational time series models, particularly in the context of their application with covariates, it is essential to establish a consistent set of parameters. These parameters consist of context length, horizon length, and patch size, which directly influence the predictive performance of the TimesFM and TimeGPT model [Das+24a].
The context length refers to the number of historical data points that the models consider when forecasting future values. Das et al. [Das+24a] standardise this parameter by using a context length of 512 data points across all of their zero-shot test datasets, but also make a point that it can vary. Similarly, models such as Tiny Time Mixers (TTMs) [Eka+24] and MOMENT [Gos+24] have used context lengths of 512 or less. TimeGPT-1, in contrast, uses a context length equal to the total number of data points provided as input, aligning it directly with the length of the dataset [GCM23]. With fine-tuning the dataset is split into three sections, first the training section, second the validation section and third the testing section. To ensure that all three sections have enough data points (so at least 544 for the validation and testing sets, leaving 738 for training), this thesis adopts a context length of 416 for all zero-shot forecast tests, including those involving covariates.
The horizon length defines the number of future data points that the models aim to predict. In previous studies, TimesFM used horizon lengths between 96 and 192 data points [Das+24a], while TimeGPT-1 limited its horizon length to seven data points for daily data [GCM23]. For this analysis, the longest horizon length tested is 128 data points. Additionally, shorter horizons, such as the 7-day forecast period recommended by Garza et al. [GCM23], a monthly horizon length of 30 days and the last 60 days of the forecast, which are day 68 to day 128, are included to evaluate performance under different scenarios.

Another parameter is the patch size, which applies specifically to TimesFM. While TimeGPT-1 does not reveal a corresponding patch size in its paper or code, which is not publicly available [GCM23], Das et al. [Das+24a] demonstrated that TimesFM achieves optimal performance with a patch size of 32. This configuration was found to minimise the MAE, thereby enhancing overall accuracy [Das+24a]. Therefore, a patch size of 32 is used consistently for TimesFM across all predictions, including those involving covariates. In addition, the min-max scaler is applied to both models for the purpose of normalisation, thus facilitating the comparison of values derived from different horizon and context lengths. This means that the deviation amount is applied to a scale between 1 and 0, the minimum values being 0 and the maximum 1 [24c]. By maintaining these parameter settings, this analysis tries to ensure consistency and comparability with previous studies, enabling a robust evaluation of model performance both with and without the inclusion of covariates.
Due to the 14-day horizon established within the TimeGPT-1 model, the long-horizon forecasting setting is employed for TimeGPT-1. This implies that the seven-day forecast relies on the "normal" TimeGPT-1, whereas the remaining forecasts apply the "timeGPT-1-long-horizon" setting [24b].

Additionally, to the AI models, the baseline model ARIMA is implemented. It operates with distinct parameters, namely p, d and q. The value of p represents the number of observations that are separated by a specific time interval. Thus, the value at which the PACF in Figure 12 exhibits a decline is used to define p. The value at which the ACF in Figure 13 shows a decline is used to determine q. Finally, d represents the number of differencing steps needed to detrend the time-series by the model. In order to identify these parameters, it is necessary to differentiate the data values as their seasonality need to be removed [Wei13], as determined in section 5.1.3. In order to eliminate the impact of seasonal fluctuations in daily data, it is essential to ascertain the appropriate seasonal factor. As illustrated in Figure 5, the data reaches its maximum height and minimum point on an annual basis at approximately the same time, indicating that the seasonal repetition occurs every 365 days, meaning year by year. This necessitates the differentiation of 365 data points, resulting in 1826 minus 365 resulting in 1461 data points. Consequently, the model predicts on fewer data points than TimesFM and TimeGPT. The process of differentiation is done twice, as ARIMA is tested with covariates and without covariates. Given that the differentiated values are negative, the calculation of the MAPE values relies on the ARIMA forecast being conducted on the denormalised values.

After the differentiation of the electricity consumption data and temperature data the Dickey-Fuller test for Electricity consumption and temperature now return a p-value of 0.0043 and $6x10^{-14}$, which is considerably smaller than 0.05, see appendix

B.3. Next, the ACF and PACF are examined to identify the p and q values. Both

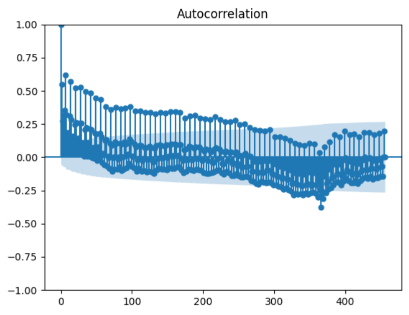

Figure 12 ACF of Electricity Consumption. See appendix B.3.

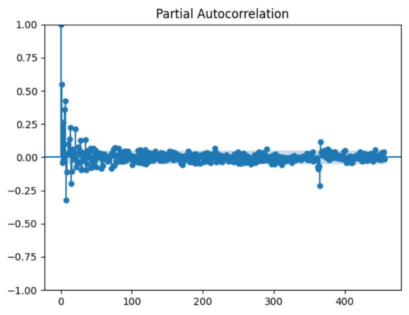

Figure 13 PACF of Electricity Consumption. See appendix B.3.

figures, 12 and 13, exhibit a cut off after one time point, determining the parameters p and q as 1. The d parameter is 0, as no trend needs differentiation. Similarly, differencing is applied to the covariate in light of the detection of a seasonal pattern in section 5.1.3. In order to predict future values with covariates in ARIMA, the same parameters (p,q,d) are used as previously, since they were identified as being related to the main variable.

5.4 Fine-tuning

A further comparison of the models is conducted, with particular attention paid to their fine-tuning capabilities and the accuracy of their forecasts. The process of fine-tuning involves adjusting the weights of specific layers in a pre-trained model using a task-specific dataset [Das+24a; GCM23; Lia+24b]. In the case of TimesFM, the focus is on non-transformer layers, such as the residual block, as the transformer layers are fixed during fine-tuning [Das+24a]. The particulars of the layers that are trained or whether all layers are updated for TimeGPT-1 are not described by Garza et al. [GCM23]. A more detailed examination of the fine-tuning process is described by Liao et al. [Lia+24b], which starts with pre-trained model weights as a baseline, which are then adjusted with a lower learning rate than in the pre-training process to prevent substantial deviation from the original weights. The Adam optimiser is applied in this process, to minimize the loss function, for both models [Das+24a; GCM23; Lia+24b].

The fine-tuning process involves a target dataset for forecasting purposes, which is trained using a source dataset [Das+24a; GCM23]. In this study, the target dataset remains consistent with the one used for zero-shot forecasting, with a forecasting context length of 416 data points and a horizon of up to 128 points. The source dataset is an extended electricity consumption and temperature dataset, beginning in 2015 with 1826 data points - including the 416 data points of the target dataset.

In contrast to TimeGPT-1, which processes the entire dataset (source and target) without explicitly separating it into training and testing sets [GCM23], TimesFM allows the user to manually divide the dataset into three portions: training, validation, and testing [Das+24a]. These intervals are defined as follows: the first interval is for testing from data point 1280 to data point 1824, 1824 instead of 1826 because it is a multiple of 32 and appeared to function more efficiently in the code, the second interval is for validation 736 to and 1280 the third interval is for training at 0 to 736. Both models allow the customization of fine-tuning parameters. For instance, the horizon lengths are identical to those employed in zero-shot forecasting, consisting of 7, 30, the last 60, and 128 data points. TimeGPT-1 maintains the distinction between the long-horizon model and the standard model for the seven-day forecast [GCM23]. Training steps, referred to as epochs in TimesFM [Das+24a], are set to 100 for TimeGPT-1, in accordance with the findings of Garza et al. [GCM23] which demonstrate a significant improvement in rMAE (relative Mean Average Error) up to this threshold, with the improvement becoming less pronounced thereafter. For TimesFM, the epoch varies, it is set to a value of 500, but with earlier cut-offs if required, for example at 209 or 417. The cut-off is determined by the patience setting in the code, which is not customisable in TimeGPT-1. The patience setting is fixed at 5 for all model variants of TimesFM, and this is the underlying cause of the early cut-off if the model does not improve with subsequent training steps. It is important to note that increasing the value of patience will increase the limit value at which the model will terminate. See Appendix B.1, fine-tuned code, e.g. set the patience to 10 manually.

The loss functions are further customizable parameters for both models. TimeGPT-1 includes a range of prebuilt loss functions, such as MAE, MSE, MAPE, and RMSE. The MSE is implemented in this study, because MAPE can't be considered (is not used on normalised data), and a short run of the other MAE and RMSE shows that there is almost no difference in the outcome A.5. Similarly, TimesFM implements loss functions via prebuilt libraries or custom methods, in this thesis the loss function from the given code is implemented [Das+24a; SZ24]. Furthermore, additional parameters include the patch size for TimesFM, which remains at 32 for the purpose of fine-tuning, as is the case in zero-shot forecasting. Additionally, the learning rate can be modified in TimesFM, with the possible values of 1e-2 or 5e-2 being selected based on their optimal alignment with the specified model, considering the horizon length. This approach draws inspiration from the learning rate defined within the provided code of TimesFM [SZ24]. In this study, the other settings provided in the TimesFM prebuilt fine-tuning example code were adopted.

Both fine-tuning implementations have the capacity to incorporate covariates, although the way in which they are used differs [Das+24a; GCM23]. In contrast to

TimeGPT-1, which employs historical or future covariate data for forecasting (Garza, 2023), TimesFM uses the covariate in the data loader which needs the length of the entire fine-tuning process (1824)[Das+24a]. In order to maintain consistency with the zero-shot forecast comparison, the future covariate values are used in this analysis for TimeGPT-1.

5.5 Results

In order to analyse the predicted data and conduct the experiment, it is necessary to make certain assumptions and to provide a degree of concretisation. First, the complete raw dataset was used for predictions, following the approach suggested by Das et al. [Das+23] and Garza et al. [GCM23] for their foundation models TimesFM and TimeGPT-1. These researchers argue that their models are specifically designed to detect and incorporate seasonal and trend patterns directly from the data, distinguishing them from models such as ARIMA, which require such patterns to be removed prior to analysis [Wei13]. This capability allows for a more nuanced use of the dataset without the need for preprocessing to filter out underlying trends or seasonality.
Second, in order to forecast the time series with a covariate, the actual temperature values corresponding to the 128 future data points of electricity consumption were used for the TimeGPT-1 model with zero-shot and fine-tuning, because TimeGPT-1 can only use future or historic exogenous variables [24b]. These values reflect the actual observed temperatures during that period, thereby enabling the model to use the accurate covariate information for improved prediction of the primary variable. However, in scenarios where the prediction is for a truly future time frame, actual temperature data would not yet exist. In such cases, temperature forecasts could be applied instead, providing a predictive input for the covariate to influence the main variable's forecast.

5.5.1 Comparison of the Performances

Metric		Zero-Shot		Finetuning		ARIMA
		TimesFM	TimeGPT	TimesFM	TimeGPT	
MAE	7	0.0768	0.0884	**0.0459**	0.0820	0.0879
	30	0.0394	0.0402	0.0512	**0.0388**	0.0724
	-60	0.2487	0.2645	0.1096	0.1639	**0.0683**
	128	0.1490	0.1631	0.0842	0.1019	**0.0656**
MSE	7	0.0074	0.0094	**0.0039**	0.0085	0.0133
	30	0.0026	0.0023	0.0041	**0.0023**	0.0094
	-60	0.0670	0.0755	0.0195	0.0310	**0.0070**
	128	0.0348	0.0401	0.0127	0.0164	**0.0069**
RMSE	7	0.0859	0.0971	**0.0626**	0.0920	0.1153
	30	0.0512	0.0483	0.0637	**0.0481**	0.0971
	-60	0.2588	0.2748	0.1275	0.1759	**0.0837**
	128	0.1865	0.2002	0.1065	0.1280	**0.0829**
MAPE	7	0.0742	0.0842	**0.0408**	0.0519	0.0808
	30	0.0364	0.0372	0.0418	**0.0358**	0.0644
	-60	0.1773	0.1885	0.0869	0.1161	**0.0489**
	128	0.1100	0.1206	0.0677	0.0755	**0.0516**

Table 2 Results of the TimesFM, TimeGPT-1 and ARIMA model, separated into zero-zhot forecasting and fine-tuning results. ARIMA is applied to a context length equal to the training dataset used for fine-tuning. -60 represents the last 60 days of the forecast. The best results are **bold**.
As the fine-tuning of TimesFM only improves the model, if an improvement is found and MAPE is calculated with unscaled values, the TimesFM model could not be improved and therefore the MAPE values for TimesFM are calculated with the normal (unmodified) model checkpoints.

An examination of the forecasted data, as seen in figure 14 and 15, produced by the two foundational models reveals that both models accurately predict the initial segment of the forecast period. The TimesFM model closely follows the actual data from September to October. However, it fails to fully capture the seasonal dynamics, as evidenced by its inability to replicate the pronounced upward movement in the data from October to December. Similarly, TimeGPT-1 aligns with the actual data between September and October, but also fails to account for the seasonal increase in the later months, instead showing a slight downward trend until December. This seasonal increase is better modelled in the fine-tuned version, where a seasonal increase is shown, as can be seen in figure 22.

The error metrics, seen in table 2 confirm these observations. For both TimesFM and TimeGPT-1, the MAE and RMSE remain relatively low for forecasting horizons of seven and 30 days, generally below or equal to 0.1. However, during the final 60 days of the forecast, the MAE and RMSE exceed 0.25 for both models, indicating

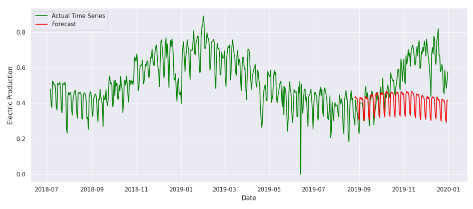

Figure 14 TimesFM forecast. See Appendix 18

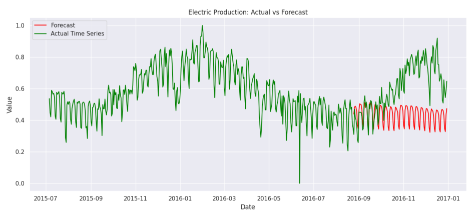

Figure 15 TimesGPT-1 forecast. See Appendix 20

at least a doubling to tripling of the deviation. In contrast, the benchmark ARIMA model does not exhibit this deterioration in forecasting accuracy, maintaining MAE and RMSE values below 0.1 across all forecast horizons (7, 30, and 60 days). Also for all models the value of deviation for the 30 day forecast is lower than for the 7 day forcast.

In the case of the fine-tuned version of TimesFM, the forecast displays a lower deviation from the original data compared to the zero-shot prediction. The 30-day forecast demonstrates marginal deficiencies, exhibiting a minimal decline of approximately 0.01 in the MAE, RMASE and MAPE metrics. In contrast, the forecast for the previous 60 days exhibits a substantial enhancement of approximately 50%. Fine-tuning TimeGPT-1 results in improved MAE and RMSE values across all horizons and the trend. The observed increase in deviation for the last 60 days of the forecast, in the zero-shot model, seems to have weakened. Additionally, the deviation nearly doubles from MAE to RMSE for the first 7 and 30 days.

Other key results indicate that TimesFM achieves lower MAE, MSE, RMSE, and MAPE values for zero-shot forecasts over 7 and 30 days compared to TimeGPT-1 and ARIMA. Nevertheless, ARIMA demonstrates the lowest errors for the final 60 days and for the overall forecast period of 128 days. In fine-tuning TimesFM achieves the best forecasting results for the 7 day forecast, TimeGPT-1 for the 30 day forecast and ARIMA still exhibits the best values for the long forecasting lengths. Both TimesFM and TimeGPT-1 experience greater errors in zero-shot forecasts and fine-tuning over extended horizons (from 7 to 128 days), as indicated by worsening error metrics, although both models get better for the 30 day forecast (with exception of the MAE and MAPE for TimesFM in fine-tuning).

5.5.2 Impact of Covariates on the Forecasting Models

Incorporating covariates into the forecast models produces notably different effects, as shown in figure 16 and 17. In the case of zero-shot forecasting, TimesFM improves significantly, capturing more fluctuations and effectively representing seasonal dynamics. In contrast, the performance of TimeGPT-1 suffers a substantial decline, displaying increased fluctuations but failing to align with the actual data. The forecast shows only minimal resemblance to the actual data, with failure to capture the upward trend in electricity consumption and instead exhibiting a decline before showing a small upward curve. The same can be detected for the fine-tuned version, which is just slitly closer to the actual values, as can be seen in Figure 23.

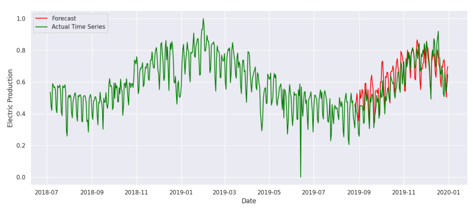

Figure 16 TimesFM forecast with covariates. See Appendix 19

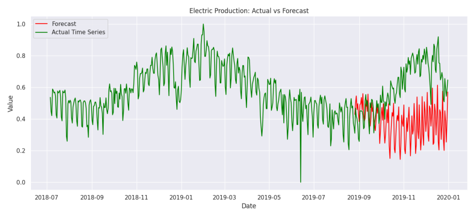

Figure 17 TimesGPT-1 forecast with covariates. See Appendix 21

The error metrics reflect these findings, as seen in table 3. With TimeGPT-1, the MAE, MSE, RMSE, and MAPE worsen significantly over longer forecasting horizons, whereas TimesFM maintains mostly consistent error values across different horizons. There is even a trend in decreasing deviation for zero-shot forecasts.
Fine-tuning slightly improves TimeGPT-1's performance, except for the forecast of the first 7 days, resulting in a increase in the observed deviations for fine-tuning for the MSE and MAPE. Notably, TimeGPT-1 achieves greater accuracy than TimesFM for the first 30 days in terms of MAE and MSE, but not for RMSE and MAPE any more. These differences are minimal. TimesFM's predictions also improve with fine-tuning, except for the forecast for the last 60 days, which are slightly worse too, again showing a slight increase of deviation with longer forecasting horizons. It even outperforms ARIMA on the MAE and MAPE for all horizon lengths, and on the MSE and RMSE for the 7 and 30 day horizon lengths. TimeGPT-1 only outperforms ARIMA for the 30 day horizon length. This goes for all error based matrics.

When comparing the models with and without covariates, TimesFM shows a significant enhancement of performance in zero-shot forecasting when including covariates. Particularly for longer horizons, as it eliminates the trend of increasing deviation. The fine-tuned values for TimesFM confirm this, but again show a slight decline in accuracy for the last 60 values of the forecast. In contrast, TimeGPT-1's forecasts degrade considerably when covariates are included, although fine-tuning improves its accuracy. The ARIMA model with a longer context length also benefits from the inclusion of covariates for the 30, last 60 and 128 days, though to a lesser extent and not including the MAPE values, which worsened.

Metric		Zero-Shot		Finetuning		ARIMA
		TimesFM	TimeGPT	TimesFM	TimeGPT	
MAE	7	0.0982	0.1175	**0.0513**	0.1154	0.0998
	30	0.0791	0.0661	0.0509	**0.0494**	0.0707
	-60	0.0578	0.3265	**0.0574**	0.3250	0.0583
	128	0.0675	0.2207	**0.0547**	0.2131	0.0587
MSE	7	0.0110	0.0147	**0.0056**	0.0157	0.0149
	30	0.0076	0.0058	0.0040	**0.0039**	0.0090
	-60	0.0053	0.1213	0.0061	0.1165	**0.0050**
	128	0.0067	0.0704	0.0059	0.0667	**0.0056**
RMSE	7	0.1050	0.1214	**0.0687**	0.1252	0.1221
	30	0.0873	0.0759	**0.0622**	0.0626	0.0946
	-60	0.0729	0.3483	0.0742	0.3413	**0.0710**
	128	0.0818	0.2653	0.0747	0.2583	**0.0746**
MAPE	7	0.0843	0.1123	**0.0444**	0.1088	0.0925
	30	0.0639	0.0603	**0.0420**	0.0445	0.0632
	-60	**0.0376**	0.2267	0.0475	0.2324	0.0419
	128	0.0484	0.1600	**0.0442**	0.1584	0.0466

Table 3 Results of the TimesFM, TimeGPT-1 and ARIMA model with covariates, separated into zero-shot forecasting and fine-tuning results. ARIMA is applied to a context length equal to the training dataset used for fine-tuning. -60 represents the last 60 days of the forecast. The best results are **bold**.

6 Discussion

6.1 Analysis of the Results

The findings of TimesFM and TimeGPT-1 for time series forecasting should provide an understanding of their performance characteristics, particularly when examined under zero-shot and fine-tuning conditions, across various forecast horizons. These findings are contextualised using existing literature to evaluate the challenges and strengths of foundational models relative to traditional methods as ARIMA.
The first research question answered is as follows: "How do time series forecasting models like TimesFM and TimeGPT-1 compare in performance?"

In zero-shot scenarios, TimesFM demonstrates its strength in capturing short-term trends, as evidenced by its low MAE and RMSE values (<0.09) for 7- and 30-day horizons. However, this accuracy decreases as the forecast horizon increases, with MAE and RMSE values rising above 0.24 for the last 60 days. This trend reflects the findings of several studies, such as those utilizing the ETTh1, ETTm1, and ETTh2 datasets, which consistently report declining accuracy with rising horizon lengths [Das+24a; Liu+24; Shi+24; Yan+24]. The underlying cause is the difficulty that foundational models face in capturing long-term dependencies. As observed by Bose et al. [Bos+24], foundational models such as TimesFM struggle with seasonality and heterogeneity, including variations in scale, variance, and trends. This limitation could be further reinforced when only one or two seasonal cycles are present in the training data, limiting the ability of the model to generalise beyond its learned patterns.
Similarly, TimeGPT-1 initially aligns well with actual data for shorter horizons, but its performance degrades significantly for longer horizons, as its MAE and RMSE exceed 0.2 for the last 60 days. While TimeGPT-1 is pre-trained on a massive dataset of 100 billion data points [GCM23], the model's generalizability to load forecasting appears limited prior to fine-tuning. The Nixtla documentation on long-horizon forecasting highlights the challenges associated with extending forecasts over extended periods, noting that unknown factors and increasing uncertainty undermine accuracy [24b]. This suggests that TimeGPT-1's pre-training may not sufficiently capture the specific seasonal dependencies required for accurate long-term load forecasting. A backward search revealed no further research on TimeGPT-1, that was accessible, – or experiments with TimeGPT-1, and the only one that does only keeps in the forecasting horizon range of the original paper [Lia+24b]. Apart from the long-horizon instructions on the Nixtla, there has been no further research on this model with longer horizon lengths. In addition, the long horizon forecast for TimeGPT-1 also

starts at two weeks and does not specify for how long the forecast is accurate [24b]. As can be seen in Fig. 17 that the decline in accuracy begins around one month after the start of the forecast.
Comparing the foundational models with ARIMA reveals the strengths and weaknesses of each approach. While TimesFM and TimeGPT-1 outperform ARIMA for short-term forecasts (7 and 30 days, except for TimeGPT-1 for the 7 day forecast in MAE and MAPE), ARIMA consistently exhibits lower MAE and RMSE values over longer horizons, maintaining accuracy even for 60-day and 128-day periods. The higher vales of TimeGPT-1 for the MAE and MAPE could be due to the reduced sensibility to outliers [CWJ21]. The better accuracy for longer forecast horizons of ARIMA may be attributable to the explicit integration of seasonal patterns and its statistical nature, which avoids the generalisation pitfalls, like overfitting, data heterogeneity [Yin19; Bos+24] or limited representation of seasonal periods, of deep learning models in the absence of sufficient seasonal data [PC24; Kon+23; SR20].

Fine-tuning introduces notable improvements for both foundational models. TimesFM exhibits reduced deviations for longer horizons, with MAE and RMSE stabilising between 0.04 and 0.065, although minimal deficiencies are observed for the initial 30 days. Das et al. [Das+24b] demonstrate how fine-tuning allows TimesFM to adapt to dataset-specific distributions, thereby reducing errors caused by untrained dependencies. Although, it should be noted that Das et al. [Das+24b] uses in-context fine-tuning with TimesFM and not traditional methods like used in this thesis. However, fine-tuning is not without its risks, such as overfitting or catastrophic forgetting, where task-specific updates overwrite pre-trained knowledge or forgetting of learned patterns [Das+24b; Yin19]. This phenomenon emphasises the balance between preserving pre-trained patterns and acquiring new task-specific knowledge through fine-tuning, effectively almost eliminating the zero-shot trend of increasing deviations over time. The improvement is linked to the fine-tuning process, where the model's weights to better adapt to the data distribution and characteristics of load forecasting [Lia+24b]. As shown in Liao et al. [Lia+24b] study on load forecasting with TimeGPT-1, fine-tuning optimises the model for practical applications, enhancing accuracy across all metrics. TimesFM, which occasionally exhibits worsened results for certain datasets and horizons after fine-tuning, like seen for the 30 day forecast, suggesting potential overfitting or inadequate parameter adjustments for those cases. As Das et al. [Das+24a] have demonstrated, inferior fine-tuning outcomes were obtained for the ETTm1 dataset for horizon lengths of 96 and 192 data points in comparison to the zero-shot results outlined in the original TimesFM paper.

In both zero-shot and fine-tuning scenarios of all three models, the forecasting for 30 days is almost always superior to the 7-day forecasts. This can be attributed to the reduced percentage of outliers in the 30-day forecasts relative to the 7-day forecasts. Consequently, outliers in the 7-day forecasts exert a more significant influence on the error metric. This phenomenon could be attributed to the dataset itself, as evidenced by the ETTm1 dataset in the study by Liu et al.[Liu+24], which demonstrates a decline in deviation. However, this decline appears atypical when compared to the findings of the three other papers [Das+24a; Shi+24; Yan+24] that also utilise the ETTm1 dataset, which do not exhibit this decline for a more extended forecast horizon. Only fine-tuning for TimesFM falls into the pattern of worsening horizon length over time for the 30 day forecast.

The comparative analysis reveals that TimesFM excels in shorter horizons, achieving better results than TimeGPT-1 and ARIMA for 7-day forecasts. TimeGPT-1, however, outperforms for intermediate (30-day) horizons when it is fine-tuned, while ARIMA remains the most reliable for long-term forecasts.

To answer the second research question: "How does TimesFM perform when covariates are incorporated in comparison to other time-series forecasting models such as TimeGPT-1 and its performance without covariates?" covariates are included and the results are analysed. Firstly, the correlation between electricity consumption and temperature is examined, as calculated by the residuals, which represent the deviations between the predicted and actual values. This reflects the extent to which the models capture the underlying patterns in the data. The correlation of residuals for these two variables is -0.3, indicating a moderate negative correlation. This suggests that while the models may incorporate covariates effectively, there may be additional unexplained factors influencing the predictions, indicating that this covariate alone is not sufficient to fully model the dynamics of electricity consumption [Che+24].

In zero-shot forecasting, the inclusion of covariates significantly improves TimesFM's performance, enabling it to better capture fluctuations and seasonal factors. The stability of its error metrics, including MAE, MSE, RMSE, and MAPE, across different horizons highlights its robustness. Now, in zero-shot forecasting there is a downward trend in deviation recognisable. TimesFM's ability to outperform ARIMA in most instances, particularly for longer horizons. This is supported by findings from Massa et al. [MF24], where domain-specific covariates such as sales promotions significantly enhanced temporal modeling accuracy and reduced bias for the model Lag-Llama. Meaning that it is possible for a foundation model to outperform traditional models, such as ARIMA. Furthermore, Das et al. [Das+24b] emphasizes that covariates play a critical role in identifying relationships between variables, which helps to maintain consistent accuracy across of various horizons lengths. On the other hand, Das et al. [Das+24a] state, that TimesFM is trained without dataset specific dynamic or static

covariates.
TimeGPT-1, however, demonstrates significant performance degradation with covariates in zero-shot forecasting, as its MAE, MSE, RMSE, and MAPE values worsen substantially over longer horizons. This aligns with insights from Liao et al. [Lia+24b], which attributes the decline to the model's reliance on pre-trained datasets. These datasets may disrupt learned patterns when covariates are introduced, particularly if the covariates introduce noise or lack meaningful alignment with the target dataset [Lia+24b; MF24]. This phenomenon could be explained by the low correlation value or by not having the ability to recognise this inverse correlation. Williams et al. [Wil+24] further support this, emphasizing the need for models to align their capacity for multidimensional inputs with the complexities of covariates. In a related study by Cedeno Jimenez and Brovelli [CB24] TimeGPT-1 was used to estimate ground-level NO_2 concentrations. Initially, the model was trained without any additional variates, and thereafter also with a covariate. However, the study found that the inclusion of a covariate did not improve the model, even though it had a high correlation with the first variable [CB24].
ARIMA, in contrast, shows modest improvements with covariates, particularly for long-term horizons such as 30, 60, and 128 days. However, the enhancement in performance is not consistent across all metrics, with MAPE values occasionally worsening.

Fine-tuning further reveals the strengths and limitations of these models. TimesFM demonstrates improved accuracy with fine-tuning, particularly for shorter horizons, while maintaining a stable performance. Only its accuracy declines minimally for the last 60 days of the forecasts again. The paper by Ekambaram et al. [Eka+24] provides an explanation for this phenomenon, suggesting that foundational models adapt effectively to extended temporal dependencies during fine-tuning, mitigating risks of catastrophic forgetting. TimesFM's lower deviation values in both zero-shot and fine-tuned scenarios underscore its capacity to represent actual data trends, aligning with findings of Bose et al. [Bos+24] which highlight foundational models' strength in learning generalized patterns while requiring covariates to refine specific dataset representations.
As previously outlined, Ekambaram et al. [Eka+24] have explained that fine-tuning foundational models frequently encounters challenges such as catastrophic forgetting, which may limit TimeGPT-1's capacity to fully integrate covariates. While TimeGPT-1's fine-tuned performance does improve slightly, particularly for shorter horizons, deviations remain high for longer horizons, such as the last 60 days.
ARIMA benefits moderately from covariates in fine-tuned forecasts, with improvements in some metrics, particularly for long-term horizons. TimesFM consistently

outperforms ARIMA in MAE and MAPE across all horizon lengths and in MSE and RMSE for shorter horizons, such as 7 and 30 days. These results emphasise TimesFM's superior adaptability and robustness in leveraging covariates, as demonstrated in both zero-shot and fine-tuned scenarios. It is possible that this is the case because TimesFM uses both the historical and future covariate values for zero-shot and fine-tuning. In comparison, TimeGPT-1 only outperforms ARIMA in for the 30 day forecast with fine-tuning across all error-based metricises.

Comparing just the code of TimesFM B.1 and TimeGPT-1 B.2, it can be seen that TimeGPT code is smaller and more comprehensible. The forecast can be done with only one small, line of code. Even for including fine-tuning and covariates. TimesFM code is longer and especially the fine-tuning code is a more complex. On the other hand, TimesFM has way more customisable parameters, which are the cause for the longer code.

A comparison of the code for TimesFM in Appendix B.1 and TimeGPT-1 in Appendix B.2 reveal that the TimeGPT-1 code is more compact and easily comprehensible. Remarkably, the forecast can be executed with a mere line of code, a feat that remains unparalleled even when accounting for fine-tuning and covariates. In contrast, the TimesFM code is more extensive, particularly the fine-tuning code. However, TimesFM offers a greater number of customisable parameters, which contributes to the more extensive nature of its code.

In conclusion, TimesFM is shown to be the most effective model for integrating covariates, maintaining consistent performance across horizons and outperforming TimeGPT-1 and ARIMA in most scenarios. ARIMA is still a close second-best model for longer horizons, and it forecasts the electricity consumption with slightly fewer outliers than TimesFM, as indicated by the comparison of MAE and RMSE [CWJ21]. The challenges faced by TimeGPT-1 in handling covariates highlight the importance of aligning model architecture and training strategies with the specific demands of multi-dimensional inputs.

6.2 Limitations

The analysis of TimesFM and TimeGPT-1 reveals several limitations that impact the models' forecasting performance and their ability to generalize across diverse datasets and tasks. These limitations stem from the architectural constraints, data representation issues and methodological challenges inherent in the models.
One notable limitation of TimesFM is its maximum context length for zero-shot forecasts, which restricts the amount of historical data the model can process during

training and inference. While this constraint can ensure computational efficiency, it may limit the model's ability to learn long-term temporal dependencies, especially for datasets with multi-year trends or strong seasonal components [Das+24a; NS24]. It remains unclear whether TimeGPT-1 also operates with a maximum context length, as its proprietary code has not been published [GCM23].

The choice and integration of covariates present another challenge. While TimesFM demonstrated substantial performance improvements when covariates were included, the analysis was constrained by the limited number of exogenous variables used. Electricity consumption, for example, is influenced by multiple factors which could be e.g. economic activities, income and hours of sunlight, which were not captured in the analysis [Che+24; NS24]. Furthermore, the mismatch between the geographic scope of the electricity consumption data (UK-wide) and the temperature data (limited to Central England) introduces inconsistencies [20; CM06]. This misalignment may reduce the meaningfulness of covariates and introduce noise into the models' predictions. Aligning the geographic scope of these datasets would likely improve the correlation.

The inherent black-box nature of foundational models like TimesFM and TimeGPT-1 also poses a significant limitation. This opacity complicates the traceability of predictions and hinders reproducibility. Even with identical parameters, fine-tuning can lead to slightly different outcomes due to the stochastic nature of training. Additionally, this black-box characteristic obscures the interpretability of how covariates are weighted and integrated into the model's decision-making processes, a crucial factor for evaluating residual variability and correlations [Das+24a].

The fine-tuning process itself introduces challenges, particularly in the choice of loss functions. Both TimesFM and TimeGPT-1 allow for different loss functions during fine-tuning, yet the optimal selection remains unclear without extensive experimentation. To achieve the best results, multiple loss functions would be needed to be tested and compared, adding computational overhead and complexity to the training process [Das+24a; GCM23]. Moreover, TimesFM's lack of pretraining with covariates highlights another constraint, as the model's baseline performance could potentially be enhanced with a more robust pretraining strategy that includes appropriate variables [Das+24a].

For TimeGPT-1, the inability to simultaneously use historic and future covariates from the same data source represents a significant limitation [GCM23]. This constraint reduces the model's capability to leverage temporal relationships effectively. Additionally, while TimeGPT-1 displayed a reduction in performance with covariates during zero-shot forecasting, its improvement through fine-tuning was modest, with high deviations when longer horizons are to be predicted.

6.3 Further Research

Future research into foundational models like TimesFM and TimeGPT-1 is essential to address the existing limitations and unlock their full potential in time series forecasting. These models have demonstrated strong performance for short and intermediate forecasting horizons, yet their struggles with long-term dependencies and seasonal dynamics highlight their need for improvement. In contrast, ARIMA remains a reliable benchmark for long-term forecasting due to its statistical simplicity and explicit handling of seasonal patterns. To bridge the gap between these approaches, future studies could explore hybrid models that integrate the adaptability of foundational models with the robustness of classical statistical methods. Such approaches could ensure more consistent performance across all forecast horizons.

The critical role of covariates in enhancing forecasting accuracy, as evidenced in this study, emphasizes the need for advanced architectures and fine-tuning strategies to optimize their integration. Covariates have shown their potential to improve model predictions, yet foundational models like TimesFM are not pre-trained with covariates, limiting their efficiency in capturing complex dynamics. Future work could focus on extending pretraining strategies to incorporate covariates meaningfully, addressing a key gap noted in Das et al.'s [Das+24a] exploration of "Covariate handling." While this thesis contributes to the zero-shot and fine-tuning use of covariates, the inclusion of covariates during the training phase remains an open area for investigation. Similarly, this study aligns with Garza et al.'s [GCM23] suggestion "Comparison with recent Foundation Models" by comparing foundational models like TimesFM and TimeGPT-1 to the statistical model ARIMA. Extending the methodology used in this thesis to other datasets could further assess the ability to generalize the findings presented here.

The results obtained in this thesis are drawn from a single dataset, which provides an initial framework for evaluating the models but does not yet allow for broad generalizations. Both TimesFM and TimeGPT-1 use multiple datasets in their original analyses to validate their performance, underscoring the importance of expanding the scope of comparisons in future research.

Additionally, since identical context lengths and horizon lengths were used for both models during fine-tuning in this study, cross-validation was not employed. While this approach ensured consistency in the evaluation process, incorporating cross-validation in future work would provide a more robust validation of the models' performance [GCM23].

7 Conclusion

In conclusion, the thesis highlights the comparative analysis of foundation time series forecasting models like TimesFM and TimeGPT-1, focusing on their performance across different forecast horizons and scenarios, including zero-shot and fine-tuning conditions. The results underscore the advancements and limitations of these models, particularly their ability to handle long-term dependencies and integrate covariates effectively.

TimesFM and TimeGPT-1 demonstrate strong performances in short-term horizons, and benefit significantly from fine-tuning, which enhances their adaptability and accuracy across longer horizons. However, both models exhibit limitations in their capacity to accommodate specific seasonal dependencies without fine-tuning, particularly in the context of long-term forecasts. The comparative analysis reveals that TimesFM exhibits marginal superiority over TimeGPT-1 in zero-shot and fine-tuning scenarios for forecasts without covariates. A comparison with traditional models such as ARIMA reveals that foundational models excel in adaptability and handling short-term forecasts but still fall short in capturing long-term seasonality and trends, a domain where statistical models seem to have an advantage.

The findings emphasise the critical role of fine-tuning, yet also demonstrate that the integration of covariates can enhance the applicability of foundational models in real-world scenarios, at least for TimesFM. When using covariates, TimesFM outperforms TimeGPT-1 significantly. It even demonstrates superior performance in terms of seasonal patterns when compared to ARIMA. However, TimeGPT-1 exhibits a substantial decline in accuracy when applied with covariates, and even fine-tuning does not reverse this trend. Other challenges like overfitting, catastrophic forgetting, and the black-box nature of foundational models were also discussed as areas requiring further innovation.

Overall, this thesis contributes to the growing understanding of foundational models' potential and limitations, paving the way for future research focused on hybrid approaches, advanced fine-tuning techniques, and better integration of domain-specific features to enhance the robustness and generalizability of time series forecasting models. These advancements could bridge the performance gap between foundational and traditional methods, particularly for complex, long-term forecasting tasks.

A Appendix

A.1 Table of Sources

Source	Title	Content
Bose et al. 2024	From RNNs to Foundation Models: An Empirical Study on Commercial Building Energy Consumption	TimesFM experiment on ComStock dataset
Cedeno Jimenez and Brovelli 2024	Estimating Ground-Level NO_2 Concentrations Using Machine Learning Exclusively with Remote Sensing and ERA5 Data: The Mexico City Case Study	TimeGPT experiment on ground-level NO_2 concentrations in Mexico City
Chen et al. 2024	Econometric analysis of factors influencing electricity consumption in Spain: Implications for policy and pricing strategies	Variables of factors that influence electricity consumption
Das et al. 2024 (a)	A decoder-only Foundation Model for Time-Series Forecasting	Introduction of TimesFM and comparison with other time-series forecasting models
Das et al. 2024 (b)	In-Context Fine-Tuning for Time-Series Foundation Models	Introducing another way on fine-tuning TimesFM
Ekambaram et al. 2024	Tiny Time Mixers (TTMs): Fast Pre-trained Models for Enhanced Zero/Few-Shot Forecasting of Multivariate Time Series	Introducing the Tiny Time Mixers model and comparing it to other forecasting models
Garza et al. 2023	TimeGPT-1	Introduces TimeGPT-1 and compares it to other time-series models
Goela et al. 2024	Time-Series Foundation Model for Value-at-Risk	Examines if TimesFM is suitable for the estimation of value-at-risk (VaR).
Ilbert et al. 2024	SAMformer:Unlocking the Potential of Transformers in Time Series Forecasting with Sharpness-Aware Minimization and Channel-Wise Attention	Introducing the SAMFormer model and comparing it to other forecasting models

Source	Title	Content
Kontopoulou et al. 2023	A Review of ARIMA vs. Machine Learning Approaches for Time Series Forecasting in Data Driven Networks	Literature review regarding the comparison of ARIMA and machine learning algorithms applied to time series forecasting problems
Liao et al. 2024	TimeGPT in load forecasting: A large time series model perspective	Comparison of TimeGPT with other forecasting models
Liu et al. 2024	LSTPrompt: Large Language Models as Zero-Shot Time Series Forecasters by Long-Short-Term Prompting	Introduces LSTPrompt and compares it to other time-series models
Massa and Fanucchi 2024	Domain-adapted Lag-Llama for Time Series Forecasting in the African Retail Sector	Introduces LSTPrompt and compares it to other time-series models
Nasim and Almeida 2024	Fine-Tuned MLP-Mixer Foundation Models as data-driven Numerical Surrogates?	Introduces MLP-Mixer Foundation Models and compares it to other time-series models
Puvvada and Chaudhuri 2024	Critical Evaluation of Time Series Foundation Models in Demand Forecasting	Comparison of TimesFM and TimeGPT
Saa and Ranathunga 2020	Comparison between ARIMA and Deep Learning Models for Temperature Forecasting	Explaining ARIMA and concluding that Deep Learning Models can outperfore ARIMA on temperature forecasting
Shi et al. 2024	Time-MoE: Billion-Scale Time Series Foundation Models with Mixture of Experts	Introduces Time-MoE Foundation Models and compares it to other time-series models
Vaswani et al. 2023	Attention Is All You Need	Introduction and explanation of the Transformer Architecture
Williams et al. 2024	Context is Key: A Benchmark for Forecasting with Essential Textual Information	Introduces benchmarks for integrating numerical and textual data to improve forecasting accuracy across various domains
Woo et al. 2024	Unified Training of Universal Time Series Forecasting Transformers	Introduces MOIRAI and LOTSA and compares MOIRAI to other forecasting models
Yang et al. 2024	ViTime: A Visual Intelligence-Based Foundation Model for Time Series Forecasting	Introduces ViTime Foundation Models and compares it to other time-series models
Ying 2019	An Overview of Overfitting and its Solutions	Explains issues in machine learning

Table 4 Sources used in the discussion

A.2 Table of Categorised Sources

Articles	Concepts					
	TimesFM	TimeGPT	Fine-tuning	Covariates	Limitations	Future Research
Bose et al. 2024	x		x			
C. J. and B. 2024		x	x	x		
Chen et al. 2024					x	
Das et al. 2024 (a)	x		x	x	x	x
Das et al. 2024 (b)	x		x			
Ekambaram et al. 2024	x	x				
Garza et al. 2023		x	x	x	x	x
Goela et al. 2024	x	x	x			
Ilbert et al. 2024						
Kontopoulou et al. 2023						
Liao et al. 2024		x	x			
Liu et al. 2024	x					
M. and F. 2024						
N. and A. 2024	x	x			x	
P. and C. 2024	x	x	x	x	x	x
Saa and R. 2020						
Shi et al. 2024	x	x				
Vaswani et al. 2023	x	x	x		x	x

Articles	Concepts					
Williams et al. 2024						
Woo et al. 2024	x	x	x	x	x	x
Yang et al. 2024	x					
Ying 2019						

Table 5 Sources categorised from above

A.3 Graphs for zero-shot

TimesFM without covariates

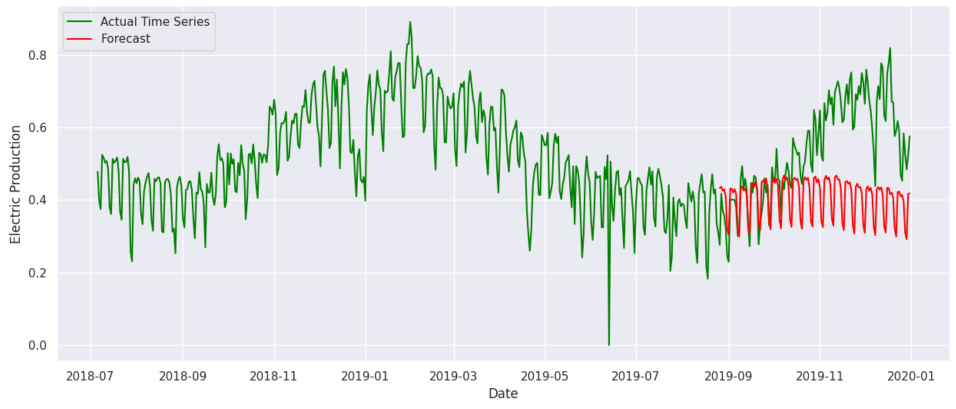

Figure 18 TimesFM without covariates. See appendix B.1

TimesFM with covariates

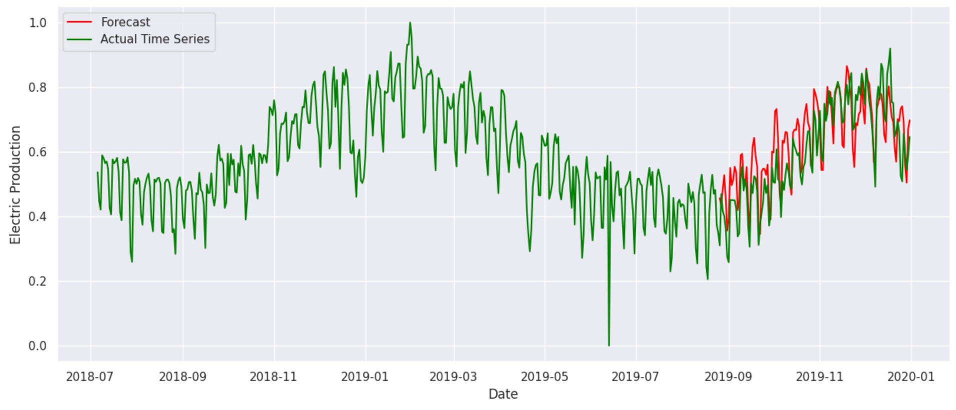

Figure 19 TimesFM with covariates. See appendix B.1

TimeGPT-1 without covariates

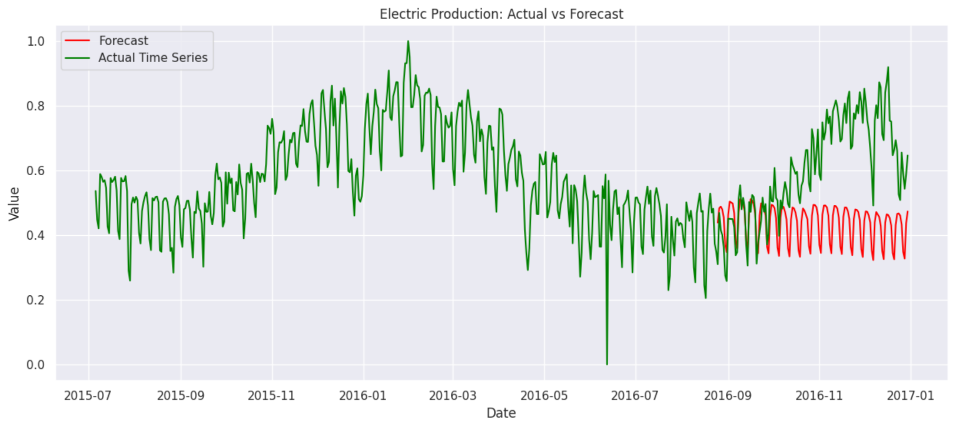

Figure 20 TimeGPT-1 forecast without covariates. See appendix B.2

TimeGPT-1 with covariates

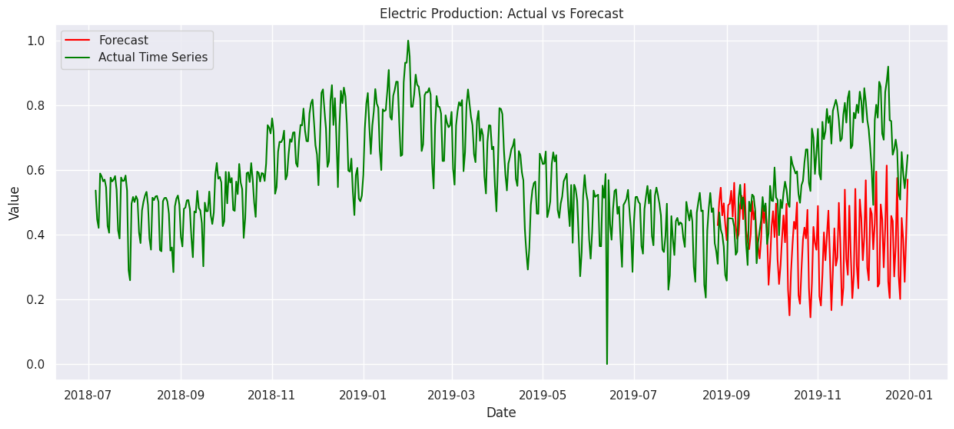

Figure 21 TimeGPT-1 with covariates. See appendix B.2

A.4 Graphs for fine-tuning

TimesFM

I regret to inform that in order to view the graph of the fine-tuned TimesFM model, both with and without covariates, I must refer to the code. It can be said, that it doesn't look that different to the zero-shot TimesFM graph, only exhibiting a greater degree of fluctuation. See appendix B.1

TimeGPT-1

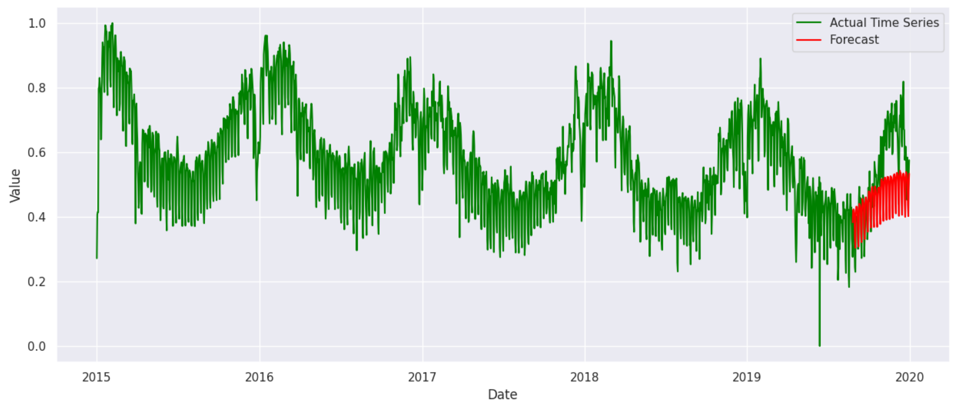

Figure 22 Fine-tuned TimesGPT-1 plot. Red is the Forecast, green is the actual Time Series.See appendix B.2

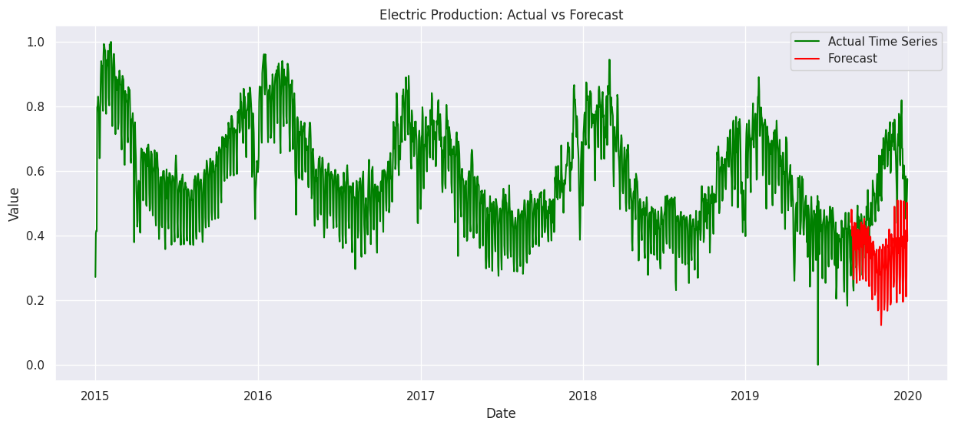

Figure 23 Fine-tuned TimesGPT-1 plot with covariates, with MSE loss function. Red is the Forecast, green is the actual Time Series. See appendix B.2

A.5 Graphs for fine-tuning TimesGPT to show difference in MSE, MAE and RMSE loss functions

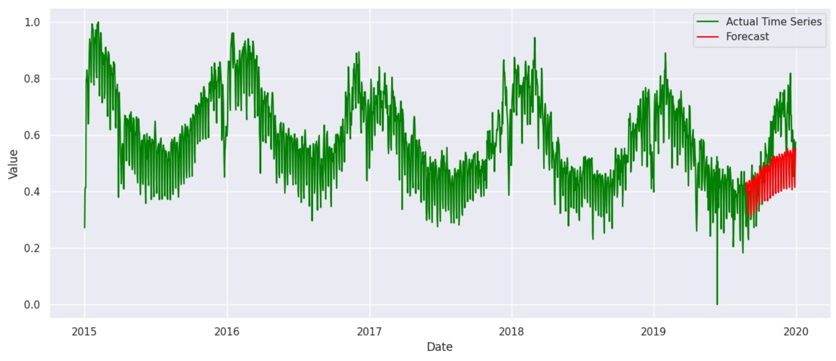

Figure 24 Fine-tuned TimesGPT-1 plot, with MAE loss function. Red is the Forecast, green is the actual Time Series. See appendix B.2

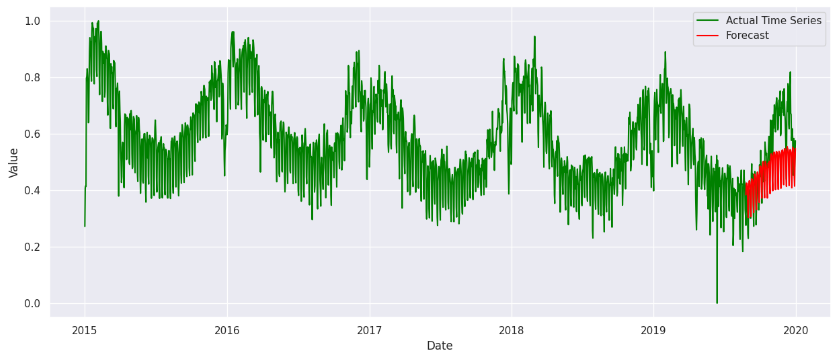

Figure 25 Fine-tuned TimesGPT-1 plot, with RMSE loss function. Red is the Forecast, green is the actual Time Series. See appendix B.2

A.6 Graphs of ARIMA

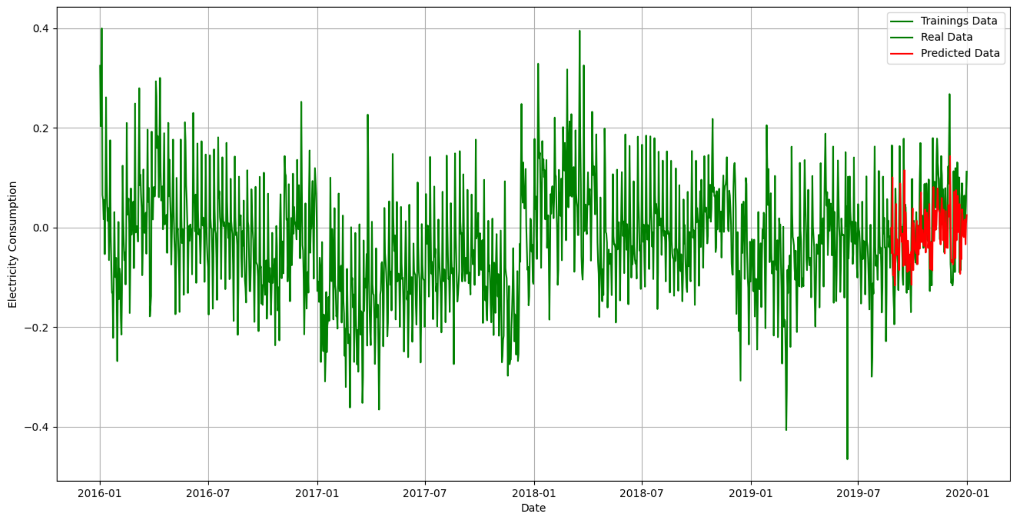

Figure 26 ARIMA plot. See appendix B.3

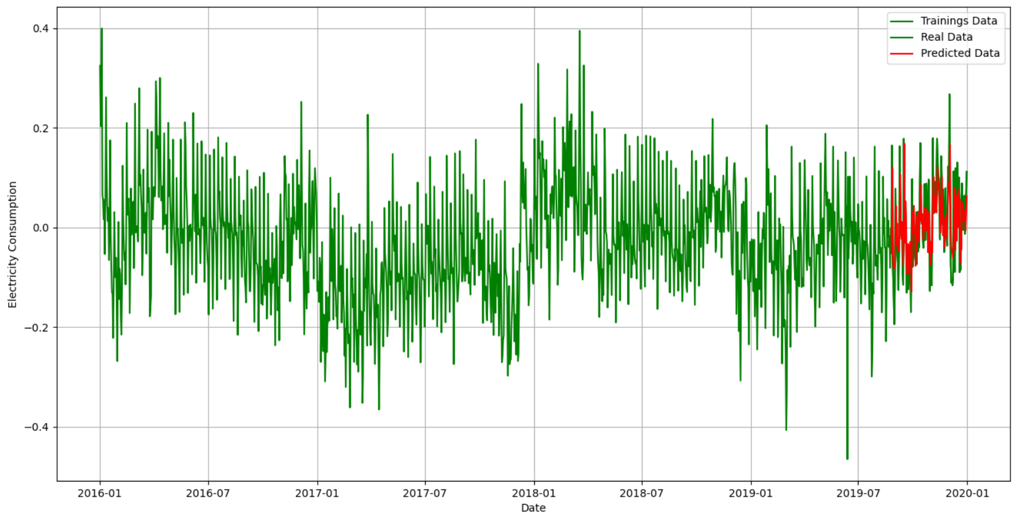

Figure 27 ARIMA plot with covariates. See appendix B.3

A.7 Sigmoid Function

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

The function in its entirety is as follows:

$$\frac{1}{1 + exp(-\sum_j w_j x_j - b)}$$

Mathematically, this function can be expressed as follows: in cases where $z \equiv w \times x + b$ yields a large positive value, $e^{-z} \approx 0$, resulting in $\sigma(z) \approx 1$. Therefore, when $z = w \times x + b$ is large and positive, the sigmoid neuron's behaviour mirrors that of a perceptron, with an output approaching 1. In contrast, when $z = w \times x + b$ is significantly negative, $e^{-z} \to \infty$, resulting in $\sigma(z) \approx 0$, which again resembles perceptron behaviour. When $z = w \times x + b$ is neither close to 1 nor 0, there is a notable variation in the sigmoid neuron's output in comparison to the perceptron model [Agg18; Nie19].

A.8 Positional Encoding

This encoding relies on sine and cosine functions of varying frequencies to represent positional information [Vas+17]:

$$PE_{(pos,2i)} = \sin(pos/1000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/1000^{2i/d_{model}})$$

pos = position and i is dimension [Vas+17]

A.9 Input Residual Block

The inputs $y_{1:i}$ are divided into patches of length $input\_patch\_len(p)$, where the j-th patch is represented as $\tilde{y}_j = y_{p(j-1)+1:pj}$. Similarly, the binary mask is divided into patches corresponding to the input data, represented as $\tilde{m}_j = m_{p(j-1)+1:pj}$ [Das+24a]. The j-th input token for the subsequent transformer layers is then expressed as:

$$t_j = InputResidualBlock(\tilde{y}_j \odot (1 - \tilde{m}_j)) + PE_j$$

where PE_j denotes the j-th positional encoding. This formulation ensures that only the relevant portions of the input data, as defined by the mask, are processed, while incorporating positional information to maintain the temporal structure of the time-series data [Das+24a].

A.10 Stacked Transformer

$$o_j = StackedTransformer((t_1, \dot{m}_1), ..., (t_j, \dot{m}_j))$$

where $\dot{m}_j$ represents the masking indicator for the j-th token. This indicator is defined as $min\{m_{p(j-1)+1:pj}\}$, which determines whether the j-th token is considered valid based on the corresponding patch's mask values [Das+24a].

A.11 Output Residual Block

$$\tilde{y}_{pj+1:pj+h} = OutputResidualBlock(o_j)$$

where $\tilde{y}_{pj+1:pj+h}$ represents the predicted time-series segment for the j-th patch, and o_j is the output token corresponding to that patch [Das+24a].

B Code

B.1 Code TimesFM

TimesFM code without covariates:

```
#!pip install timesfm[pax]

import timesfm

import pandas as pd
data=pd.read_csv('/content/total_consumption.csv')
data

data['Date']=pd.to_datetime(data['Date']) #convert to year-month-day
data.head()

data = data.set_index('Date')
from sklearn.preprocessing import MinMaxScaler
# Normalize data, not needed for MAPE, and change all scaled_df to data
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
scaled_df = pd.DataFrame(scaled_data, columns=['Value', 'temp'])
scaled_df['Date'] = pd.date_range(start='2015-01-01', periods=len(scaled_
columns_order = ['Date'] + [col for col in scaled_df.columns if col != 'D
scaled_df = scaled_df[columns_order]

df = pd.DataFrame({'unique_id':[1]*len(scaled_df),'ds': scaled_df["Date"]
df = df[-544:] #because i only use 416+128 data points

# Spliting data frame into train and test set
split_idx = int(len(df) * 0.766)
# Split the dataframe into train = historical data and test sets
train_df = df[:split_idx]
test_df = df[split_idx:]
print(train_df.shape, test_df.shape)
```

```
# Initialize the TimesFM model with specified parameters
tfm = timesfm.TimesFm(
    hparams=timesfm.TimesFmHparams(
        backend="cpu",
        per_core_batch_size=32,
        context_len=416,
        horizon_len=128,
        input_patch_len=32,
        output_patch_len=128,
        num_layers=20,
        model_dims=1280,
      ),
    # Load the pretrained model checkpoint
      checkpoint=timesfm.TimesFmCheckpoint(
          huggingface_repo_id="google/timesfm-1.0-200m"),
)
# Forecasting the values using the TimesFM model
timesfm_forecast = tfm.forecast_on_df(
   inputs=train_df,          # Input training data for training
   freq="D",                 # Frequency of the time-series data (daily)
   value_name="y",           # Name of the column containing the values to b
   num_jobs=-1,              # Set to -1 to use all available cores
)
timesfm_forecast = timesfm_forecast[["ds","timesfm"]]


# Let's Visualise the Data
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')  # Setting the warnings to be ignored

scaled_df = scaled_df[-512:]
# Set the style for seaborn
sns.set(style="darkgrid")
# Plot size
plt.figure(figsize=(15, 6))
# Plot actual timeseries data
sns.lineplot(x="ds", y='y', data=df, color='green', label='Actual␣Time␣Se
# Plot forecasted values
sns.lineplot(x="ds", y='timesfm', data=timesfm_forecast, color='red', lab
plt.xlabel('Date')
plt.ylabel('Electric␣Production')
# Show the legend
plt.legend()
# Display the plot
plt.savefig('TimesFM_Electricity_Plot.png')
plt.show()

#calculationg the errors
import numpy as np
actuals = test_df['y'] # add .iloc[-60:] for the last 60 datapoints, and
predicted_values = timesfm_forecast['timesfm'] # same as above and always
# Convert to numpy arrays
actual_values = np.array(actuals)
predicted_values = np.array(predicted_values)
# Calculate error metrics
MAE = np.mean(np.abs(actual_values - predicted_values))  # Mean Absolute
MSE = np.mean((actual_values - predicted_values)**2)      # Mean Squared E
RMSE = np.sqrt(np.mean((actual_values - predicted_values)**2))
# Root Mean Squared Error
MAPE = np.mean(np.abs((actual_values - predicted_values) / actual_values)
# MAPE

# Print the error metrics
print(f"Mean␣Absolute␣Error␣(MAE):␣{MAE}")
print(f"Mean␣Squared␣Error␣(MSE):␣{MSE}")
print(f"Root␣Mean␣Squared␣Error␣(RMSE):␣{RMSE}")
print(f"Mean␣Absolute␣Percentage␣Error␣(MAPE):␣{MAPE}")
```

TimesFM code with covariates: The code uses a wrong scale with only 730 data points of the time series not 1826 data points as all other models, but as the results will only change slightly to better the results will stay the same. Only the -60 day zero-shot forecast will have a better MAE result as the fine-tuning result aligning with the MSE and RMSE.

```
# I would recommend to use jupyter notebook

#!pip install timesfm[pax]
```

```
# Import timesfm library
import timesfm

# Initialize the TimesFM model with specified parameters
tfm = timesfm.TimesFm(
    hparams=timesfm.TimesFmHparams(
        backend="cpu",
        per_core_batch_size=32,
        context_len=416,
        horizon_len=128,
        input_patch_len=32,
        output_patch_len=128,
        num_layers=20,
        model_dims=1280,
      ),
    # Load the pretrained model checkpoint
      checkpoint=timesfm.TimesFmCheckpoint(
          huggingface_repo_id="google/timesfm-1.0-200m"),
)

import pandas as pd
import numpy as np
from collections import defaultdict
data = pd.read_csv('Electricity.csv')
data['Date'] = pd.to_datetime(data['Date'])
data = data[-544:] #because i only use 416+128 data points

#data = data.set_index('Date')
from sklearn.preprocessing import MinMaxScaler
# Normalize data, not needed for MAPE, and change all scaled_df to data
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
scaled_df = pd.DataFrame(scaled_data, columns=['Value', 'temp'])
scaled_df['Date'] = pd.date_range(start='2018-07-06', periods=len(scaled_
columns_order = ['Date'] + [col for col in scaled_df.columns if col != 'D
scaled_df = scaled_df[columns_order]

df = pd.DataFrame({'unique_id':[1]*len(data),'ds': data["Date"], "y":data
# Spliting data frame into train and test set
split_idx = int(len(df) * 0.766)
# Split the dataframe into train = historical data and test sets
train_df = df[:split_idx]
test_df = df[split_idx:]
print(train_df.shape, test_df.shape)

def get_full_data(context_len: int, horizon_len: int):
    # Ensure enough data is present for context and horizon
    if len(train_df) < context_len:
        raise ValueError(f"Dataset␣must␣have␣at␣least␣{context_len}␣rows␣

    if len(train_df) < context_len + horizon_len:
        print("Not␣enough␣data␣for␣the␣specified␣horizon␣length.␣Padding␣

    # Extract or pad context
    inputs = train_df["y"][:context_len].tolist()
    if len(inputs) < context_len:
        inputs += [0] * (context_len - len(inputs))  # Pad with zeros if
    print(inputs)

    # Extract or pad covariates
    temp = data["temp"][:context_len + horizon_len].tolist()
    if len(temp) < context_len + horizon_len:
        temp += [0] * ((context_len + horizon_len) - len(temp))
# Pad with zeros
    print(temp)

    # Return data in the correct structure
    return {
        "inputs": [inputs],  # Single example with context length
        "temp": [temp],      # Covariate length includes horizon
    }

# Define context and horizon lengths
context_len = 416
horizon_len = 128
```

```
# Get prepared data
full_data = get_full_data(context_len=context_len, horizon_len=horizon_le

# Debugging: Verify lengths
print(f"Inputs␣length:␣{len(full_data['inputs'][0])}")  # Should be 416
print(f"Temp␣length:␣{len(full_data['temp'][0])}")      # Should be (416

# Run forecast with covariates
try:
    cov_forecast, ols_forecast = tfm.forecast_with_covariates(
        inputs=full_data["inputs"],  # Context data
        dynamic_numerical_covariates={
            "temp": full_data["temp"],  # Covariates (includes horizon)
        },
        freq=[0] * len(full_data["inputs"]),  # Frequency mask
        xreg_mode="xreg␣+␣timesfm",           # Mode for external regress
        ridge=0.0,                            # Regularization
        force_on_cpu=False,                   # Use GPU if available
        normalize_xreg_target_per_input=True, # Normalize covariates
    )
    print("Forecast␣completed␣successfully!")
    print("Covariate␣Forecast:", cov_forecast)
    print("OLS␣Forecast:", ols_forecast)
except ValueError as e:
    print(f"Error␣during␣forecasting:␣{e}")

#set up graph
# Flatten all arrays inside cov_forecast into a single list
flattened_values = [item for sublist in cov_forecast for item in sublist]
# Convert the flattened list into a pandas Series or DataFrame
flattened_df = pd.DataFrame(flattened_values, columns=["Value"])
flattened_df['Date'] = pd.date_range(start='2019-08-26', periods=len(flatt

# Let's Visualise the Data
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')  # Setting the warnings to be ignored
# Set the style for seaborn
```

```
sns.set(style="darkgrid")
# Plot size
plt.figure(figsize=(15, 6))
# Plot forecasted values
sns.lineplot(x="Date", y='Value', data=flattened_df, color='red', label='
# Plot Actual Time Series
sns.lineplot(x="ds", y='y', data=df, color='green', label='Actual␣Time␣Se
plt.xlabel('Date')
plt.ylabel('Electric␣Production')
# Show the legend
plt.legend()
# Display the plot
plt.savefig('For_Cov_Electricity_plot.png')
plt.show()

actuals = test_df['y'] # add .iloc[-60:] for the last 60 datapoints, and
predicted_values = flattened_df['Value'] # same as above and always para
# Convert to numpy arrays
actual_values = np.array(actuals)
predicted_values = np.array(predicted_values)
# Calculate error metrics
MAE = np.mean(np.abs(actual_values - predicted_values))  # Mean Absolute
MSE = np.mean((actual_values - predicted_values)**2)      # Mean Squared E
RMSE = np.sqrt(np.mean((actual_values - predicted_values)**2))
# Root Mean Squared Error
MAPE = np.mean(np.abs((actual_values - predicted_values) / actual_values)
# MAPE

# Print the error metrics
print(f"Mean␣Absolute␣Error␣(MAE):␣{MAE}")
print(f"Mean␣Squared␣Error␣(MSE):␣{MSE}")
print(f"Root␣Mean␣Squared␣Error␣(RMSE):␣{RMSE}")
print(f"Mean␣Absolute␣Percentage␣Error␣(MAPE):␣{MAPE}")
```

TimesFM fine-tuned code with and without covariates:

```
import os
os.environ['XLA_PYTHON_CLIENT_PREALLOCATE'] = 'false'
os.environ['JAX_PMAP_USE_TENSORSTORE'] = 'false'
```

```
!pip install timesfm[pax] # only istall pax not torch

import timesfm

# set up model
tfm = timesfm.TimesFm(
      hparams=timesfm.TimesFmHparams(
          backend="gpu",
          per_core_batch_size=32,
          horizon_len=128,
      ),
      checkpoint=timesfm.TimesFmCheckpoint(
          huggingface_repo_id="google/timesfm-1.0-200m"),
  )

import gc
import numpy as np
import pandas as pd
from timesfm import patched_decoder
from timesfm import data_loader

from tqdm import tqdm
import dataclasses
import IPython
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt

#read data
data = pd.read_csv('/content/total_consumption.csv')

#not needed if normalisation is not wanted, like for MAPE
data = data.set_index('Date')
from sklearn.preprocessing import MinMaxScaler
# Normalize data
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
scaled_df = pd.DataFrame(scaled_data, columns=['Value', 'temp'])
scaled_df['Date'] = pd.date_range(start='2015-01-01', periods=len(scaled_
columns_order = ['Date'] + [col for col in scaled_df.columns if col != 'D
scaled_df = scaled_df[columns_order]
scaled_df.to_csv('modified_table.csv', index=False)  # index=False to exc

# set up the dataframe for the training, validation and testing sets
DATA_DICT = {
    "consumption": {
        "boundaries": [736, 1280, 1824],
        "data_path": "/content/total_consumption.csv", # for normalised d
        "freq": "D",
    },
}

#set up the dataset
dataset = "consumption"
data_path = DATA_DICT[dataset]["data_path"]
freq = DATA_DICT[dataset]["freq"]
int_freq = timesfm.freq_map(freq)
boundaries = DATA_DICT[dataset]["boundaries"]

data_df = pd.read_csv(open(data_path, "r"))
data_df

ts_cols = [col for col in data_df.columns if col not in ["Date", "temp"]]
num_cov_cols = None # for covariate: ['temp']
cat_cov_cols = None

context_len = 416
pred_len = 128

num_ts = len(ts_cols)
batch_size = 32

# set up the data_loader for the model
dtl = data_loader.TimeSeriesdata(
      data_path=data_path,
      datetime_col="Date",
      num_cov_cols=num_cov_cols,
      cat_cov_cols=cat_cov_cols,
        ts_cols=np.array(ts_cols),
        train_range=[0, boundaries[0]],
        val_range=[boundaries[0], boundaries[1]],
        test_range=[boundaries[1], boundaries[2]],
        hist_len=context_len,
        pred_len=pred_len,
        batch_size=num_ts,
        freq=freq,
        normalize=False,
        epoch_len=None,
        holiday=False,
        permute=False,
    )

#sort batches
train_batches = dtl.tf_dataset(mode="train", shift=1).batch(batch_size)
val_batches = dtl.tf_dataset(mode="val", shift=pred_len)
test_batches = dtl.tf_dataset(mode="test", shift=pred_len)

for tbatch in tqdm(train_batches.as_numpy_iterator()):
    pass
print(tbatch[0].shape)

# shows prior mae loss before training, not necessary
mae_losses = []
for batch in tqdm(test_batches.as_numpy_iterator()):
    past = batch[0]
    actuals = batch[3]
    forecasts, _ = tfm.forecast(list(past), [0] * past.shape[0], normaliz
    forecasts = forecasts[:, 0 : actuals.shape[1]]
    mae_losses.append(np.abs(forecasts - actuals).mean())

print(f"MAE:␣{np.mean(mae_losses)}")

# set up everything for training
import jax
from jax import numpy as jnp
from praxis import pax_fiddle
from praxis import py_utils
from praxis import pytypes
from praxis import base_model
from praxis import optimizers
from praxis import schedules
from praxis import base_hyperparams
from praxis import base_layer
from paxml import tasks_lib
from paxml import trainer_lib
from paxml import checkpoints
from paxml import learners
from paxml import partitioning
from paxml import checkpoint_types

# PAX shortcuts
NestedMap = py_utils.NestedMap
WeightInit = base_layer.WeightInit
WeightHParams = base_layer.WeightHParams
InstantiableParams = py_utils.InstantiableParams
JTensor = pytypes.JTensor
NpTensor = pytypes.NpTensor
WeightedScalars = pytypes.WeightedScalars
instantiate = base_hyperparams.instantiate
LayerTpl = pax_fiddle.Config[base_layer.BaseLayer]
AuxLossStruct = base_layer.AuxLossStruct

AUX_LOSS = base_layer.AUX_LOSS
template_field = base_layer.template_field

# Standard prng key names
PARAMS = base_layer.PARAMS
RANDOM = base_layer.RANDOM

key = jax.random.PRNGKey(seed=1234)

model = pax_fiddle.Config(
    patched_decoder.PatchedDecoderFinetuneModel,
    name='patched_decoder_finetune',
    core_layer_tpl=tfm.model_p,
)
```

```
@pax_fiddle.auto_config
def build_learner() -> learners.Learner:
  return pax_fiddle.Config(
      learners.Learner,
      name='learner',
      loss_name='avg_qloss',
      optimizer=optimizers.Adam(
          epsilon=1e-7,
          clip_threshold=1e2,
          learning_rate=1e-2, # change learning rate to 1e-2 to 5e-2 for
          lr_schedule=pax_fiddle.Config(
              schedules.Cosine,
              initial_value=1e-3,
              final_value=1e-4,
              total_steps=3100,
          ),
          ema_decay=0.9999,
      ),
      # Linear probing i.e we hold the transformer layers fixed.
      bprop_variable_exclusion=['.*/stacked_transformer_layer/.*'],
  )

task_p = tasks_lib.SingleTask(
    name='ts-learn',
    model=model,
    train=tasks_lib.SingleTask.Train(
        learner=build_learner(),
    ),
)

task_p.model.ici_mesh_shape = [1, 1, 1]
task_p.model.mesh_axis_names = ['replica', 'data', 'mdl']

DEVICES = np.array(jax.devices()).reshape([1, 1, 1])
MESH = jax.sharding.Mesh(DEVICES, ['replica', 'data', 'mdl'])

num_devices = jax.local_device_count()
print(f'num_devices:␣{num_devices}')
print(f'device␣kind:␣{jax.local_devices()[0].device_kind}')

jax_task = task_p
key, init_key = jax.random.split(key)

# To correctly prepare a batch of data for model initialization (now that
# inference is merged), we take one devices*batch_size tensor tuple of da
# slice out just one batch, then run the prepare_input_batch function ove


def process_train_batch(batch):
    past_ts = batch[0].reshape(batch_size * num_ts, -1)
    actual_ts = batch[3].reshape(batch_size * num_ts, -1)
    return NestedMap(input_ts=past_ts, actual_ts=actual_ts)


def process_eval_batch(batch):
    past_ts = batch[0]
    actual_ts = batch[3]
    return NestedMap(input_ts=past_ts, actual_ts=actual_ts)


jax_model_states, _ = trainer_lib.initialize_model_state(
    jax_task,
    init_key,
    process_train_batch(tbatch),
    checkpoint_type=checkpoint_types.CheckpointType.GDA,
)

jax_model_states.mdl_vars['params']['core_layer'] = tfm._train_state.mdl_
jax_vars = jax_model_states.mdl_vars
gc.collect()

jax_task = task_p


def train_step(states, prng_key, inputs):
  return trainer_lib.train_step_single_learner(
      jax_task, states, prng_key, inputs
  )


def eval_step(states, prng_key, inputs):
  states = states.to_eval_state()
  return trainer_lib.eval_step_single_learner(
      jax_task, states, prng_key, inputs
  )

key, train_key, eval_key = jax.random.split(key, 3)
train_prng_seed = jax.random.split(train_key, num=jax.local_device_count(
eval_prng_seed = jax.random.split(eval_key, num=jax.local_device_count())

p_train_step = jax.pmap(train_step, axis_name='batch')
p_eval_step = jax.pmap(eval_step, axis_name='batch')

replicated_jax_states = trainer_lib.replicate_model_state(jax_model_state
replicated_jax_vars = replicated_jax_states.mdl_vars

best_eval_loss = 1e7
step_count = 0
patience = 0
NUM_EPOCHS = 500
PATIENCE = 5
TRAIN_STEPS_PER_EVAL = 1000
#train_state_unpadded_shape_dtype_struct
CHECKPOINT_DIR='./checkpoints'

def reshape_batch_for_pmap(batch, num_devices):
  def _reshape(input_tensor):
    bsize = input_tensor.shape[0]
    residual_shape = list(input_tensor.shape[1:])
    nbsize = bsize // num_devices
    return jnp.reshape(input_tensor, [num_devices, nbsize] + residual_sha

  return jax.tree.map(_reshape, batch)

#start training loop
for epoch in range(NUM_EPOCHS):
```

```
print(f"______________________Epoch:␣{epoch}______________________", flush=Tr
train_its = train_batches.as_numpy_iterator()
if patience >= PATIENCE:
    print("Early␣stopping.", flush=True)
    break
for batch in tqdm(train_its):
    train_losses = []
    if patience >= PATIENCE:
        print("Early␣stopping.", flush=True)
        break
    tbatch = process_train_batch(batch)
    tbatch = reshape_batch_for_pmap(tbatch, num_devices)
    replicated_jax_states, step_fun_out = p_train_step(
        replicated_jax_states, train_prng_seed, tbatch
    )
    train_losses.append(step_fun_out.loss[0])
    if step_count % TRAIN_STEPS_PER_EVAL == 0:
        print(
            f"Train␣loss␣at␣step␣{step_count}:␣{np.mean(train_losses)
            flush=True,
        )
        train_losses = []
        print("Starting␣eval.", flush=True)
        val_its = val_batches.as_numpy_iterator()
        eval_losses = []
        for ev_batch in tqdm(val_its):
            ebatch = process_eval_batch(ev_batch)
            ebatch = reshape_batch_for_pmap(ebatch, num_devices)
            _, step_fun_out = p_eval_step(
                replicated_jax_states, eval_prng_seed, ebatch
            )
            eval_losses.append(step_fun_out.loss[0])
        mean_loss = np.mean(eval_losses)
        print(f"Eval␣loss␣at␣step␣{step_count}:␣{mean_loss}", flush=T
        if mean_loss < best_eval_loss or np.isnan(mean_loss):
            best_eval_loss = mean_loss
            print("Saving␣checkpoint.")
            jax_state_for_saving = py_utils.maybe_unreplicate_for_fu
                replicated_jax_states
                )
                checkpoints.save_checkpoint(
                    jax_state_for_saving, CHECKPOINT_DIR, overwrite=True
                )
                patience = 0
                del jax_state_for_saving
                gc.collect()
            else:
                patience += 1
                print(f"patience:␣{patience}")
        step_count += 1

train_state = checkpoints.restore_checkpoint(jax_model_states, CHECKPOINT

print(train_state.step)
tfm._train_state.mdl_vars['params'] = train_state.mdl_vars['params']['cor
tfm.jit_decode()

#mae loss after training, not necessary for results
mae_losses = []
for batch in tqdm(test_batches.as_numpy_iterator()):
    past = batch[0]
    actuals = batch[3]
    _, forecasts = tfm.forecast(list(past), [0] * past.shape[0])
    forecasts = forecasts[:, 0 : actuals.shape[1], 5]
    mae_losses.append(np.abs(forecasts - actuals).mean())

print(f"MAE:␣{np.mean(mae_losses)}")

#setup for graph of the actual values and forecasted values
# Flatten all arrays inside cov_forecast into a single list
flattened_past = [item for sublist in past for item in sublist]
# Convert the flattened list into a pandas Series or DataFrame
flattened_train_df = pd.DataFrame(flattened_past, columns=["Value"])
flattened_train_df['Date'] = pd.date_range(start='2018-07-04', periods=le

# Flatten all arrays inside cov_forecast into a single list
flattened_actuals = [item for sublist in actuals for item in sublist]
# Convert the flattened list into a pandas Series or DataFrame
```

```
flattened_test_df = pd.DataFrame(flattened_actuals, columns=["Value"])
flattened_test_df['Date'] = pd.date_range(start='2019-08-24', periods=len

# Flatten all arrays inside cov_forecast into a single list
flattened_values = [item for sublist in forecasts for item in sublist]
# Convert the flattened list into a pandas Series or DataFrame
flattened_df = pd.DataFrame(flattened_values, columns=["Value"])
flattened_df['Date'] = pd.date_range(start='2019-08-24', periods=len(flatt

# Let's Visualise the Data
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')  # Setting the warnings to be ignored
# Set the style for seaborn
sns.set(style="darkgrid")
# Plot size
plt.figure(figsize=(15, 6))
# Plot Actual Time Series
sns.lineplot(x="Date", y='Value', data=flattened_train_df, color='green',
sns.lineplot(x="Date", y='Value', data=flattened_test_df, color='green',
# Plot forecasted values
sns.lineplot(x="Date", y='Value', data=flattened_df, color='red', label='
plt.xlabel('Date')
plt.ylabel('Electric␣Production')
# Show the legend
plt.legend()
# Display the plot
plt.savefig('FineT_For_Cov_Electricity_plot.png')
plt.show()

#calcualting the losses
mse_losses = []
rmse_losses = []
mape_losses = []
r2_losses = []
mae_losses = []

for batch in tqdm(test_batches.as_numpy_iterator()): #in each batch is in
    past = batch[0]
    actuals = batch[3]

    # Forecast using the model
    _, forecasts = tfm.forecast(list(past), [0] * past.shape[0])
    forecasts = forecasts[:, 0 : actuals.shape[1], 5]  # Select appropria

    #forecasts = forecasts[:, :30] # needs to be changed for each forecas
    #actuals = actuals[:, :30] # the same as above, no need to run the en

    # Calculate MAE
    mae_losses.append(np.abs(forecasts - actuals).mean())

    # Calculate MSE
    mse = ((forecasts - actuals) ** 2).mean()
    mse_losses.append(mse)

    # Calculate RMSE
    rmse = mse ** 0.5
    rmse_losses.append(rmse)

    # Calculate MAPE
    mape = (np.abs((actuals - forecasts) / actuals)).mean()
    mape_losses.append(mape)


# Print the aggregated metrics
print(f"MAE: {np.mean(mae_losses)}")
print(f"MSE: {np.mean(mse_losses)}")
print(f"RMSE: {np.mean(rmse_losses)}")
print(f"MAPE: {np.mean(mape_losses)}")
```

B.2 Code TimeGPT-1

TimeGPT-1 code without covariates:

```
import pandas as pd
from nixtla import NixtlaClient

# Get your API Key at dashboard.nixtla.io
# 1. Instantiate the NixtlaClient
```

```
nixtla_client = NixtlaClient(api_key = 'nixtla-tok-Xfsa5rQzdeWsiq3PICaZgJoF

# 2. Read historic electricity demand data
data = pd.read_csv('total_consumption.csv')

data['Date']=pd.to_datetime(data['Date']) #convert to year-month-day

data = data.set_index('Date')

from sklearn.preprocessing import MinMaxScaler
# Normalize data, not needed for MAPE, and change all scaled_df to data
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
scaled_df = pd.DataFrame(scaled_data, columns=['Value', 'temp'])
# Mape: change "scaled_df" to "data"
scaled_df['Date'] = pd.date_range(start='2015-07-06', periods=len(scaled_
columns_order = ['Date'] + [col for col in scaled_df.columns if col != 'D
scaled_df = scaled_df[columns_order]
scaled_df = scaled_df[-544:]

#Let's Visualise the Datas
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore') # Settings the warnings to be ignored
sns.set(style="darkgrid")
plt.figure(figsize=(15, 6))
sns.lineplot(x="Date", y='Value', data=scaled_df, color='green')
plt.xlabel('Date')
plt.ylabel('Electricity␣Consumption')
plt.show()

# Mape: change "scaled_df" to "data"
df = pd.DataFrame({'unique_id':[1]*len(scaled_df),'ds': scaled_df["Date"]
"y":scaled_df['Value']})

# Spliting training and test set
split_idx = int(len(df) * 0.766)
# Split the dataframe into train = historical data and test sets
```

```
train_df = df[:split_idx]
test_df = df[split_idx:]
print(train_df.shape, test_df.shape)

# Forecast with TimeGPT
fcst_df = nixtla_client.forecast(train_df, h=128, model='timegpt-1-long-h

# Let's Visualise the Data
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')  # Setting the warnings to be ignored
# Set the style for seaborn
sns.set(style="darkgrid")
# Plot size
plt.figure(figsize=(15, 6))
# Plot actual timeseries data
sns.lineplot(x="ds", y='TimeGPT', data=fcst_df, color='red', label='Forec
# Plot forecasted values
sns.lineplot(x="Date", y='Value', data=scaled_df, color='green', label='A
# Set plot title and labels
plt.title('Electric␣Production:␣Actual␣vs␣Forecast')
plt.xlabel('Date')
plt.ylabel('Value')
# Show the legend
plt.legend()
# Display the plot
plt.savefig('For_TimeGPT_plot.png')
plt.show()

import numpy as np
actuals = test_df['y'] # add .iloc[-60:] for the last 60 datapoints, and
predicted_values = fcst_df['TimeGPT'] # same as above and always parallel
# Convert to numpy arrays
actual_values = np.array(actuals)
predicted_values = np.array(predicted_values)
# Calculate error metrics
MAE = np.mean(np.abs(actual_values - predicted_values))  # Mean Absolute
MSE = np.mean((actual_values - predicted_values)**2)     # Mean Squared E
```

```
RMSE = np.sqrt(np.mean((actual_values - predicted_values)**2))
# Root Mean Squared Error
MAPE = np.mean(np.abs((actual_values - predicted_values) / actual_values)
# MAPE

# Print the error metrics
print(f"Mean␣Absolute␣Error␣(MAE):␣{MAE}")
print(f"Mean␣Squared␣Error␣(MSE):␣{MSE}")
print(f"Root␣Mean␣Squared␣Error␣(RMSE):␣{RMSE}")
print(f"Mean␣Absolute␣Percentage␣Error␣(MAPE):␣{MAPE}")
```

TimeGPT-1 code with covariates:

```
import pandas as pd
from nixtla import NixtlaClient

# Get your API Key at dashboard.nixtla.io
# 1. Instantiate the NixtlaClient
nixtla_client = NixtlaClient(api_key = 'nixtla-tok-Xfsa5rQzdeWsiq3PICaZgJoF

# 2. Read historic electricity demand data
data = pd.read_csv('total_consumption.csv')

data['Date']=pd.to_datetime(data['Date']) #convert to year-month-day

data = data.set_index('Date')

from sklearn.preprocessing import MinMaxScaler
# Normalize data, not needed for MAPE, and change all "scaled_df" to "dat
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
scaled_df = pd.DataFrame(scaled_data, columns=['Value', 'temp'])
# Mape: change "scaled_df" to "data"
scaled_df['Date'] = pd.date_range(start='2018-07-06', periods=len(scaled_
columns_order = ['Date'] + [col for col in scaled_df.columns if col != 'D
scaled_df = scaled_df[columns_order]
# Mape: change "scaled_df" to "data"
scaled_df = scaled_df[-544:]
# Mape: change "scaled_df" to "data"
```

```
df = pd.DataFrame({'unique_id':[1]*len(scaled_df),'ds': scaled_df["Date"]

# Spliting training and test set
split_idx = int(len(df) * 0.766)
# Split the dataframe into train = historical data and test sets
train_df = df[:split_idx]
test_df = df[split_idx:]
print(train_df.shape, test_df.shape)

#set up covariate
future_ex_vars_df = test_df[['unique_id','ds','temp']]
future_ex_vars_df

timegpt_fcst_ex_vars_df = nixtla_client.forecast(df=train_df, X_df=future_


# Let's Visualise the Data
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')  # Setting the warnings to be ignored
# Set the style for seaborn
sns.set(style="darkgrid")
# Plot size
plt.figure(figsize=(15, 6))
# Plot forecasted values
sns.lineplot(x="ds", y='TimeGPT', data=timegpt_fcst_ex_vars_df, color='re
# Plot Actual Time Series
sns.lineplot(x="Date", y='Value', data=scaled_df, color='green', label='A
# Set plot title and labels
plt.title('Electric␣Production:␣Actual␣vs␣Forecast')
plt.xlabel('Date')
plt.ylabel('Value')
# Show the legend
plt.legend()
# Display the plot
plt.savefig('For_Cov_TimeGPT_plot.png')
plt.show()
```

```
import numpy as np
actuals = test_df['y'] # add .iloc[-60:] for the last 60 datapoints, and
predicted_values = timegpt_fcst_ex_vars_df['TimeGPT'] # same as above and

# Convert to numpy arrays
actual_values = np.array(actuals)
predicted_values = np.array(predicted_values)
# Calculate error metrics
MAE = np.mean(np.abs(actual_values - predicted_values))  # Mean Absolute
MSE = np.mean((actual_values - predicted_values)**2)     # Mean Squared E
RMSE = np.sqrt(np.mean((actual_values - predicted_values)**2))
# Root Mean Squared Error
MAPE = np.mean(np.abs((actual_values - predicted_values) / actual_values)
# MAPE

# Print the error metrics
print(f"Mean␣Absolute␣Error␣(MAE):␣{MAE}")
print(f"Mean␣Squared␣Error␣(MSE):␣{MSE}")
print(f"Root␣Mean␣Squared␣Error␣(RMSE):␣{RMSE}")
print(f"Mean␣Absolute␣Percentage␣Error␣(MAPE):␣{MAPE}")
```

TimeGPT-1 fine-tuned code without covariates:

```
import pandas as pd
from nixtla import NixtlaClient
from datasetsforecast.long_horizon import LongHorizon
from utilsforecast.losses import mae
from utilsforecast.losses import mae, mse, rmse, mape, smape

# 1. Instantiate the NixtlaClient
nixtla_client = NixtlaClient(api_key = 'nixtla-tok-Xfsa5rQzdeWsiq3PICaZgJoF

data = pd.read_csv('total_consumption.csv')
data['Date']=pd.to_datetime(data['Date']) #convert to year-month-day

data = data.set_index('Date')

from sklearn.preprocessing import MinMaxScaler
# Normalize data, not needed for MAPE, and change all scaled_df to data
```

```
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
scaled_df = pd.DataFrame(scaled_data, columns=['Value', 'temp'])
# Mape: change "scaled_df" to "data"
scaled_df['Date'] = pd.date_range(start='2015-01-01', periods=len(scaled_
columns_order = ['Date'] + [col for col in scaled_df.columns if col != 'D
scaled_df = scaled_df[columns_order]

scaled_df.insert(loc=0, column='unique_id', value=1)

df = pd.DataFrame({'unique_id':[1]*len(scaled_df),'ds': scaled_df["Date"]
"y":scaled_df['Value']})

# Spliting into training and test set
split_idx = int(len(df) * 0.93)
# Split the dataframe into train = historical data and test sets
train_df = df[:split_idx]
test_df = df[split_idx:]
print(train_df.shape, test_df.shape)
train_df.head()

# 7 day forecast
timegpt_fcst_finetune_mae_df = nixtla_client.forecast(
    df=train_df,
    h=7,
    finetune_steps=100,
    finetune_loss='mse',   # Set your desired loss function
    time_col='ds',
    target_col='y',
)

import numpy as np
actuals = test_df['y'].iloc[:7]
predicted_values = timegpt_fcst_finetune_mae_df['TimeGPT']
# Convert to numpy arrays
actual_values = np.array(actuals)
predicted_values = np.array(predicted_values)
# Calculate error metrics
MAE = np.mean(np.abs(actual_values - predicted_values))  # Mean Absolute
```

```
MSE = np.mean((actual_values - predicted_values)**2)      # Mean Squared E
RMSE = np.sqrt(np.mean((actual_values - predicted_values)**2))
# Root Mean Squared Error
MAPE = np.mean(np.abs((actual_values - predicted_values) / actual_values)
# MAPE

# Print the error metrics
print(f"Mean␣Absolute␣Error␣(MAE):␣{MAE}")
print(f"Mean␣Squared␣Error␣(MSE):␣{MSE}")
print(f"Root␣Mean␣Squared␣Error␣(RMSE):␣{RMSE}")
print(f"Mean␣Absolute␣Percentage␣Error␣(MAPE):␣{MAPE}")

# 30, -60 and 128 day forecast
fcst_df = nixtla_client.forecast(
    df=train_df,
    h=128,
    finetune_steps=100,
    finetune_loss='mse',
    model='timegpt-1-long-horizon',
    time_col='ds',
    target_col='y'
)

# Let's Visualise the Data
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')  # Setting the warnings to be ignored
# Set the style for seaborn
sns.set(style="darkgrid")
# Plot size
plt.figure(figsize=(15, 6))
# Plot forecasted values
sns.lineplot(x="Date", y='Value', data=scaled_df, color='green', label='A
# Plot actual timeseries data
sns.lineplot(x="ds", y='TimeGPT', data=fcst_df, color='red', label='Forec
# Set plot title and labels
plt.xlabel('Date')
plt.ylabel('Value')
```

```
# Show the legend
plt.legend()
# Display the plot
plt.savefig('FineT_For_TimeGPT_plot.png')
plt.show()


actuals = test_df['y']# add .iloc[-60:] for the last 60 datapoints, and c
predicted_values = fcst_df['TimeGPT'] # same as above and always parallel
# Convert to numpy arrays
actual_values = np.array(actuals)
predicted_values = np.array(predicted_values)
# Calculate error metrics
MAE = np.mean(np.abs(actual_values - predicted_values))  # Mean Absolute
MSE = np.mean((actual_values - predicted_values)**2)      # Mean Squared E
RMSE = np.sqrt(np.mean((actual_values - predicted_values)**2))
# Root Mean Squared Error
MAPE = np.mean(np.abs((actual_values - predicted_values) / actual_values)
# MAPE

# Print the error metrics
print(f"Mean␣Absolute␣Error␣(MAE):␣{MAE}")
print(f"Mean␣Squared␣Error␣(MSE):␣{MSE}")
print(f"Root␣Mean␣Squared␣Error␣(RMSE):␣{RMSE}")
print(f"Mean␣Absolute␣Percentage␣Error␣(MAPE):␣{MAPE}")
```

TimeGPT-1 fine-tuned code with covariates:

```
import pandas as pd
from nixtla import NixtlaClient

# Get your API Key at dashboard.nixtla.io
# 1. Instantiate the NixtlaClient
nixtla_client = NixtlaClient(api_key = 'nixtla-tok-Xfsa5rQzdeWsiq3PICaZgJoF

# 2. Read historic electricity demand data
data = pd.read_csv('total_consumption.csv')

data['Date']=pd.to_datetime(data['Date']) #convert to year-month-day
```

```
data = data.set_index('Date')

from sklearn.preprocessing import MinMaxScaler
# Normalize data, not needed for MAPE, and change all scaled_df to data
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
scaled_df = pd.DataFrame(scaled_data, columns=['Value', 'temp'])
# Mape: change "scaled_df" to "data"
scaled_df['Date'] = pd.date_range(start='2015-01-01', periods=len(scaled_
columns_order = ['Date'] + [col for col in scaled_df.columns if col != 'D
scaled_df = scaled_df[columns_order]
scaled_df

df = pd.DataFrame({'unique_id':[1]*len(scaled_df),'ds': scaled_df["Date"]

# Spliting into 218 foercasting values
split_idx = int(len(df) * 0.93)
# Split the dataframe into train = historical data and test sets
train_df = df[:split_idx]
test_df = df[split_idx:]
print(train_df.shape, test_df.shape)

future_ex_vars_df = test_df[['unique_id','ds','temp']]



# 7 day forecast
#all 1698 time points plus the 7 time points that will be forecasted
dataShort = data.iloc[:1705]
temp_future = future_ex_vars_df.iloc[:7]



timegpt_fcst_finetune_mae_df = nixtla_client.forecast(
    df=train_df,
    X_df = temp_future,
    h=7,
    finetune_steps=100,
    finetune_loss='mse',
    time_col='ds',
    target_col='y',
    feature_contributions=True,
)

import numpy as np
actuals = test_df['y'].iloc[:7]
predicted_values = timegpt_fcst_finetune_mae_df['TimeGPT']
# Convert to numpy arrays
actual_values = np.array(actuals)
predicted_values = np.array(predicted_values)
# Calculate error metrics
MAE = np.mean(np.abs(actual_values - predicted_values))  # Mean Absolute
MSE = np.mean((actual_values - predicted_values)**2)     # Mean Squared E
RMSE = np.sqrt(np.mean((actual_values - predicted_values)**2))
# Root Mean Squared Error
MAPE = np.mean(np.abs((actual_values - predicted_values) / actual_values)
# MAPE

# Print the error metrics
print(f"Mean␣Absolute␣Error␣(MAE):␣{MAE}")
print(f"Mean␣Squared␣Error␣(MSE):␣{MSE}")
print(f"Root␣Mean␣Squared␣Error␣(RMSE):␣{RMSE}")
print(f"Mean␣Absolute␣Percentage␣Error␣(MAPE):␣{MAPE}")

# 30, -60 and 128 day forecast
fcst_df = nixtla_client.forecast(
    df=train_df,
    X_df = future_ex_vars_df,
    h=128,
    finetune_steps=100,
    finetune_loss='mse', #lossfunctions = 'default', 'mae', 'mse', 'rmse'
    model='timegpt-1-long-horizon',
    time_col='ds',
    target_col='y',
    feature_contributions=True
)

# Let's Visualise the Data
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')  # Setting the warnings to be ignored
# Set the style for seaborn
sns.set(style="darkgrid")
# Plot size
plt.figure(figsize=(15, 6))
# Plot forecasted values
sns.lineplot(x="Date", y='Value', data=scaled_df, color='green', label='A
# Plot actual timeseries data
sns.lineplot(x="ds", y='TimeGPT', data=fcst_df, color='red', label='Forec
# Set plot title and labels
plt.title('Electric␣Production:␣Actual␣vs␣Forecast')
plt.xlabel('Date')
plt.ylabel('Value')
# Show the legend
plt.legend()
# Display the plot
plt.savefig('FineT_For_Cov_TimeGPT_plot.png')
plt.show()


actuals = test_df['y']# add .iloc[-60:] for the last 60 datapoints, and c
predicted_values = fcst_df['TimeGPT'] # same as above and always parallel
# Convert to numpy arrays
actual_values = np.array(actuals)
predicted_values = np.array(predicted_values)
# Calculate error metrics
MAE = np.mean(np.abs(actual_values - predicted_values))  # Mean Absolute
MSE = np.mean((actual_values - predicted_values)**2)     # Mean Squared E
RMSE = np.sqrt(np.mean((actual_values - predicted_values)**2))
# Root Mean Squared Error
MAPE = np.mean(np.abs((actual_values - predicted_values) / actual_values)
# MAPE

# Print the error metrics
print(f"Mean␣Absolute␣Error␣(MAE):␣{MAE}")
print(f"Mean␣Squared␣Error␣(MSE):␣{MSE}")
print(f"Root␣Mean␣Squared␣Error␣(RMSE):␣{RMSE}")
print(f"Mean␣Absolute␣Percentage␣Error␣(MAPE):␣{MAPE}")
```

B.3 Code ARIMA

ARIMA Code with and without covariates on the full dataset:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error, mean_absolute_error
import math

net_df = pd.read_csv("total_consumption.csv", index_col="Date", parse_dat

from sklearn.preprocessing import MinMaxScaler
# Normalize data
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(net_df)
scaled_df = pd.DataFrame(scaled_data, columns=net_df.columns)
scaled_df['Date'] = pd.date_range(start='2015-01-01', periods=len(scaled_
columns_order = ['Date'] + [col for col in scaled_df.columns if col != 'D
scaled_df = scaled_df[columns_order]

scaled_df = scaled_df.set_index('Date')

scaled_df2 = scaled_df.diff(365)
scaled_df2 = scaled_df2[365:]

from statsmodels.tsa.stattools import adfuller

#calculate the p values with the dicky-fuller test
result = adfuller(scaled_df2["Value"])
print(f"ADF␣Statistic:␣{result[0]}")
print(f"p-value:␣{result[1]}")

result = adfuller(scaled_df2["temp"])
```

```
print(f"ADF␣Statistic:␣{result[0]}")
print(f"p—value:␣{result[1]}")

#Print the ACF and PCF with the deseasonalised values
plot_acf(scaled_df2["Value"], lags=1826/4) # interpretation of q = 1, 182
plt.savefig('PACF_long.png')
plot_pacf(scaled_df2["Value"], lags=1826/4) # interpretation of p = 1
plt.savefig('ACF_long.png')

#splitting the data into training and test set
train_data, test_data = scaled_df2[0:int(len(scaled_df2)*0.913)], scaled_
train_arima = train_data['Value']
test_arima = test_data['Value']

#Settting up the forecast
history = [x for x in train_arima]
y = test_arima
# make first prediction
predictions = list()
model = ARIMA(history, order=(1,0,1))
model_fit = model.fit()
yhat = model_fit.forecast()[0]
predictions.append(yhat)
history.append(y[0])

# rolling forecasts
for i in range(1, len(y)):
    # predict
    model = ARIMA(history, order=(1,0,1))
    model_fit = model.fit()
    yhat = model_fit.forecast()[0]
    # invert transformed prediction
    predictions.append(yhat)
    # observation
    obs = y[i]
    history.append(obs)

#visualise data
import matplotlib.pyplot as plt
```

```
plt.figure(figsize=(16,8))
plt.plot(scaled_df2.index, scaled_df2['Value'], color='green', label = 'T
plt.plot(test_data.index, y, color = 'green', label = 'Real␣Data')
plt.plot(test_data.index, predictions, color = 'red', label = 'Predicted␣
plt.xlabel('Date')
plt.ylabel('Electricity␣Consumption')
plt.legend()
plt.grid(True)
plt.savefig('arima_model_norm.png')
plt.show()



# back differentiation inot data with seasonality
past_values = scaled_df['Value'].iloc[-len(predictions) - 365:-365].value
reverted_predictions = np.array(predictions) + past_values

# additional values for the second dimension of the scaler
reverted_predictions_full = np.zeros((len(reverted_predictions), scaled_d
reverted_predictions_full[:, 0] = reverted_predictions  # Annahme: 'Value

# Denormalise
original_scale_predictions_full = scaler.inverse_transform(reverted_predi

# defining a dataframe with new values
original_scale_predictions = original_scale_predictions_full[:, 0]
predictions_df = pd.DataFrame(original_scale_predictions, columns=['Value
dates = test_data.index
predictions_df['Date'] = dates[-len(original_scale_predictions):]
predictions_df = predictions_df.set_index('Date')



# report performance
import numpy as np

#calculating error metrics
y_subset = y[:7]# change to [:30] for 30 day forecast, [-60:] for 60 days
predictions_subset = predictions[:7]# change to [:30] for 30 day forecast

y_dNorm = net_df['Value'][-128:]
y_dNorm = y_dNorm[:7] # change to [:30] for 30 day forecast, [-60:] for 6
predictions_dNorm = predictions_df['Value']
predictions_dNorm = predictions_dNorm[:7]# change to [:30] for 30 day for

mae = mean_absolute_error(y_subset, predictions_subset)
print('MAE:␣' + str(mae))

mse = mean_squared_error(y_subset, predictions_subset)
print('MSE:␣' + str(mse))

rmse = math.sqrt(mse)
print('RMSE:␣' + str(rmse))

mape = np.mean(np.abs((y_dNorm - predictions_dNorm) / y_dNorm))
print('MAPE:␣' + str(mape))



# Do the Forecast with covariates
from statsmodels.tsa.statespace.sarimax import SARIMAX

# Assume 'train_data' and 'test_data' include both the target variable ('
train_values = train_data['Value']
test_values = test_data['Value']

# Prepare covariates for training and testing
train_covariates = train_data[['temp']]  # Select relevant covariates
test_covariates = test_data[['temp']]

# Initialize history and predictions
history_values = [x for x in train_values]
history_covariates = train_covariates.values.tolist()

predictions2 = []

# Rolling Forecast
for t in range(len(test_values)):
    # Train the SARIMAX model with the covariates
    model = SARIMAX(
        endog=history_values,
        exog=history_covariates,
        order=(1, 0, 1)  # Adjust ARIMA parameters as needed
    )
    model_fit = model.fit(disp=False)

    # Forecast the next value using the next covariate values
    next_covariates = test_covariates.iloc[t].values.reshape(1, −1)
# Ensure 2D shape for prediction
    yhat = model_fit.forecast(steps=1, exog=next_covariates)[0]
    predictions2.append(yhat)

    # Update history with the actual observed value and covariates
    history_values.append(test_values.iloc[t])
    history_covariates.append(next_covariates[0])

#visualise data
plt.figure(figsize=(16,8))
plt.plot(scaled_df2.index, scaled_df2['Value'], color='green', label = 'T
plt.plot(test_data.index, y, color = 'green', label = 'Real␣Data')
plt.plot(test_data.index, predictions2, color = 'red', label = 'Predicted
plt.xlabel('Date')
plt.ylabel('Electricity␣Consumption')
plt.legend()
plt.grid(True)
plt.savefig('arima_model_cov_norm.png')
plt.show()

# back differentiation to forecast with seasonality
past_values = scaled_df['Value'].iloc[−len(predictions2) − 365:−365].valu
reverted_predictions = np.array(predictions2) + past_values

# additional values for the second dimension of the scaler
reverted_predictions_full = np.zeros((len(reverted_predictions), scaled_d
reverted_predictions_full[:, 0] = reverted_predictions  # Annahme: 'Value

# Denormalise
original_scale_predictions_full = scaler.inverse_transform(reverted_predi
# defining a dataframe with new values
original_scale_predictions = original_scale_predictions_full[:, 0]
predictions_df = pd.DataFrame(original_scale_predictions, columns=['Value
dates = test_data.index
predictions_df['Date'] = dates[-len(original_scale_predictions):]
predictions_df = predictions_df.set_index('Date')

#visualise data
plt.figure(figsize=(16,8))
plt.plot(net_df.index, net_df['Value'], color='green', label = 'Trainings
plt.plot(predictions_df.index, predictions_df['Value'], color = 'red', la
plt.xlabel('Date')
plt.ylabel('Electricity␣Consumption')
plt.legend()
plt.grid(True)
plt.savefig('Cov_arima_model.png')
plt.show()

#calculationg error metrics
y_subset = y[:7]# change to [:30] for 30 day forecast, [-60:] for 60 days
predictions_subset = predictions2[:7]# change to [:30] for 30 day forecas

y_dNorm = net_df['Value'][-128:]
y_dNorm = y_dNorm[:7]# change to [:30] for 30 day forecast, [-60:] for 60
predictions_dNorm = predictions_df['Value']
predictions_dNorm = predictions_dNorm[:7]# change to [:30] for 30 day for

mae = mean_absolute_error(y_subset, predictions_subset)
print('MAE:␣' + str(mae))

mse = mean_squared_error(y_subset, predictions_subset)
print('MSE:␣' + str(mse))

rmse = math.sqrt(mse)
print('RMSE:␣' + str(rmse))

mape = np.mean(np.abs((y_dNorm - predictions_dNorm) / y_dNorm))
print('MAPE:␣' + str(mape))
```

C Datasets

C.1 Dataset code

Code for analysis of data without model influence

```
# %%
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA

# %%
net_df = pd.read_csv("total_consumption.csv", index_col="Date", parse_dat
net_df

# %%
plt.figure(figsize=(16,8))
plt.plot(net_df.index, net_df['Value'], color='green', label = 'Electrici
plt.xlabel('Date')
plt.ylabel('Electricity␣Consumption')
plt.legend()
plt.grid(True)
#plt.savefig('Electricity_Con.png')
plt.show()

# %%
plot_acf(net_df["Value"], lags=1826/4) # interpretation of q = 1, 1826 =
plt.savefig('PACF_long.png')
plot_pacf(net_df["Value"], lags=1826/4) # interpretation of p = 1
plt.savefig('ACF_long.png')

# %%
plt.figure(figsize=(16,8))
plt.plot(net_df.index, net_df['temp'], color='blue', label = 'Electricity
plt.xlabel('Date')
plt.ylabel('Electricity␣Consumption')
plt.legend()
plt.grid(True)
#plt.savefig('Temp.png')
plt.show()

# %%
plot_acf(net_df["temp"], lags=1826/4) # interpretation of q = 1, 1826 = #
plt.savefig('PACF_long.png')
plot_pacf(net_df["temp"], lags=1826/4) # interpretation of p = 1
plt.savefig('ACF_long.png')

# %%
fig, ax1 = plt.subplots(figsize=(16, 8))

# Plot the first dataset on the primary y-axis
ax1.plot(net_df.index, net_df['Value'], color='green', label='Electricity
ax1.set_xlabel('Date')
ax1.set_ylabel('Electricity␣Consumption', color='green')
ax1.tick_params(axis='y', labelcolor='green')
ax1.legend(loc='upper␣left')
ax1.grid(True)

# Create a secondary y-axis
ax2 = ax1.twinx()
ax2.plot(net_df.index, net_df['temp'], color='blue', label='Temperature')
ax2.set_ylabel('Temperature', color='blue')
ax2.tick_params(axis='y', labelcolor='blue')
ax2.legend(loc='upper␣right')

# Show the plot
plt.tight_layout()
plt.show()

# %%
from statsmodels.tsa.seasonal import seasonal_decompose

# Decompose the first variable ('Electricity_Consumption')
decompose_temp = seasonal_decompose(net_df['Value'], period=365)
residuals_temp = decompose_temp.resid.dropna()
```

```
# Decompose the second variable ('Temperature')
decompose_metric = seasonal_decompose(net_df['temp'], period=365)
residuals_metric = decompose_metric.resid.dropna()

# Ensure both residuals are aligned by index
residuals_temp, residuals_metric = residuals_temp.align(residuals_metric,

# Calculate correlation between residuals
correlation = residuals_temp.corr(residuals_metric)
print(f"The␣correlation␣between␣residuals␣is:␣{correlation:.2f}")
```

C.2 Dataset zero-shot

Only for TimesFM with covariates and only the first 25 values of the dataset.

```
Date,Value,temp
1/1/2018,803223,5.4
1/2/2018,938756,6.8
1/3/2018,954048,6.6
1/4/2018,975697,7.9
1/5/2018,980272,4.5
1/6/2018,932129,2.1
1/7/2018,915470,2.5
1/8/2018,1040300,1.6
1/9/2018,1038575,4
1/10/2018,1004965,5.1
1/11/2018,1009012,3.1
1/12/2018,1012875,4.5
1/13/2018,918965,5.5
1/14/2018,908515,5.6
1/15/2018,974099,6.3
1/16/2018,1007289,4.4
1/17/2018,1022838,5.5
1/18/2018,1018884,4.5
1/19/2018,1018352,2.7
1/20/2018,949654,1.5
1/21/2018,955158,4.4
1/22/2018,977383,5.1
1/23/2018,958615,9.1
```

```
1/24/2018,959914,9.8
```

...

C.3 Dataset fine-tuning

The first 25 values of the total consumption dataset.

```
Date,Value,temp
01/01/15,651715,7.5
01/02/15,738854,6.9
01/03/15,744368,3.7
01/04/15,743527,1.5
01/05/15,990116,3.9
01/06/15,994139,6.8
01/07/15,1011706,6.2
01/08/15,995232,7.5
01/09/15,978970,9.4
01/10/15,888688,9.5
01/11/15,889877,6
01/12/15,989587,7.4
01/13/15,1051137,5.8
01/14/15,1082612,5.2
01/15/15,1071192,4.3
01/16/15,1073613,3.9
01/17/15,1002053,2.1
01/18/15,983273,1.4
01/19/15,1090923,0.3
01/20/15,1116823,0
01/21/15,1113467,0.3
01/22/15,1104323,1.4
01/23/15,1103904,1.6
01/24/15,998265,2.5
```

...

Bibliography

[20] *Data Package Time series*. Open Power System Data. Version 2020-10-06. 2020. URL: https://doi.org/10.25832/time_series/2020-10-06.

[24a] *How to publish your research*. 2024. URL: https://authorservices.taylorandfrancis.com/publishing-your-research/ (visited on 11/24/2024).

[24b] *Long-horizon forecasting*. 2024. URL: https://docs.nixtla.io/docs.

[24c] *MinMaxScaler*. 2024. URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html.

[AAH15] K.P. Amber, M.W. Aslam, and S.K. Hussain. "Electricity consumption forecasting models for administration buildings of the UK higher education sector". In: *Energy and Buildings* 90 (2015), pp. 127–136. DOI: 10.1016/j.enbuild.2015.01.008.

[Agg18] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer, 2018. DOI: 10.1007/978-3-319-94463-0.

[AGK23] K. Al-hammuri, F. Gebali, and A. Kanan. "Vision transformer architecture and applications in digital health: a tutorial and survey". In: *Visual Computing for Industry, Biomedicine, and Art* 6.14 (2023). DOI: 10.1186/s42492-023-00140-9.

[amb24] amberblog. *20 Fun Things To Do With Chatgpt*. 2024. URL: https://amberstudent.com/blog/post/creative-and-cool-things-to-do-with-chat-gpt.

[BJ68] George Box and Gwilym M Jenkins. "Some recent advances in forecasting and control". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 17 (2 1968), pp. 91–109. DOI: doi.org/10.2307/2985674.

[BKH16] Jimmy Ba, Jamie Kiros, and Geoffrey Hinton. "Layer Normalization". In: (2016). DOI: 10.48550/arXiv.1607.06450.

[BKK18] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling". In: *ArXiv* abs/1803.01271 (2018). DOI: CorpusID:4747877.

[Bos+24] Shourya Bose et al. "From RNNs to Foundation Models: An Empirical Study on Commercial Building Energy Consumption". In: (2024). DOI: doi.org/10.48550/arXiv.2411.14421.

[Bro20] Jason Brownlee. *A Gentle Introduction to the Rectified Linear Unit (ReLU)*. 2020. URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

[CB24] Jesus Rodrigo Cedeno Jimenez and Maria Antonia Brovelli. "Estimating Ground-Level NO2 Concentrations Using Machine Learning Exclusively with Remote Sensing and ERA5 Data: The Mexico City Case Study". In: *Remote Sensing* 16.17 (2024). DOI: 10.3390/rs16173320.

[Che+24] Yueyan Chen et al. "Econometric analysis of factors influencing electricity consumption in Spain: Implications for policy and pricing strategies". In: *Heliyon* 10.17 (2024). DOI: https://doi.org/10.1016/j.heliyon.2024.e36217.

[Che15] Daqing Chen. *Online Retail*. UCI Machine Learning Repository. 2015. URL: https://doi.org/10.24432/C5BW33.

[CM06] Hadley Centre for Climate Prediction and Research (MOHC). *Met Office Hadley Centre Central England Temperature (HadCET) Series*. 2006. URL: https://catalogue.ceda.ac.uk/uuid/a946415f9345f6da9bf4c475c19477b6/?jump=related-docs-anchor.

[CPC23] Ching Chang, Wenjie Peng, and Tien-Fu Chen. "LLM4TS: Two-Stage Fine-Tuning for Time-Series Forecasting with Pre-Trained LLMs". In: *ArXiv* abs/2308.08469 (2023). DOI: 10.48550/arXiv.2308.08469.

[CWJ21] D Chicco, MJ Warrens, and G. Jurman. "The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation". In: *PeerJ Computer Science* (2021). DOI: 10.7717/peerj-cs.623. URL: https://peerj.com/articles/cs-623/#p-12.

[Das+23] Abhimanyu Das et al. "Long-term Forecasting with TiDE: Time-series Dense Encoder". In: *ArXiv* abs/2304.08424 (2023). DOI: doi.org/10.48550/arXiv.2304.08424.

[Das+24a] Abhimanyu Das et al. "A DECODER-ONLY FOUNDATION MODEL FOR TIME-SERIES FORECASTING". In: (2024). DOI: doi.org/10.48550/arXiv.2310.10688.

[Das+24b] Abhimanyu Das et al. "In-Context Fine-Tuning for Time-Series Foundation Models". In: (2024). DOI: doi.org/10.48550/arXiv.2410.24087. arXiv: 2410.24087.

[DL13] M. Brent Donnellan and Richard E. Lucas. *Chapter 28 - Secondary Data Analysis*. Oxford University Press, 2013. DOI: 10.1093/oxfordhb/9780199934898.001.0001.

[Eka+24] Vijay Ekambaram et al. “Tiny Time Mixers (TTMs): Fast Pre-trained Models for Enhanced Zero/Few-Shot Forecasting of Multivariate Time Series”. In: *ArXiv* abs/2401.03955 (2024). DOI: CorpusID:266844130.

[Els+21] Shereen Elsayed et al. “Do We Really Need Deep Learning Models for Time Series Forecasting?” In: *ArXiv* abs/2101.02118 (2021). DOI: CorpusID:230770122.

[GCM23] Azul Garza, Cristian Challu, and Max Mergenthaler-Canseco. “TimeGPT-1”. In: 2023. DOI: doi.org/10.48550/arXiv.2310.03589.

[Gev+20] Mor Geva et al. “Transformer Feed-Forward Layers Are Key-Value Memories”. In: (2020). DOI: 10.48550/arXiv.2012.14913.

[Gos+24] Mononito Goswami et al. “MOMENT: A Family of Open Time-series Foundation Models”. In: *International Conference on Machine Learning.* 2024.

[GPK68] Anubha Goel, Puneet Pasricha, and Juho Kanniainen. “Time-Series Foundation Model for Value-at-Risk”. In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 17 (2 1968), pp. 91–109. DOI: doi.org/10.48550/arXiv.2410.11773.

[Gru+24] Nate Gruver et al. “Large language models are zero-shot time series forecasters”. In: *Proceedings of the 37th International Conference on Neural Information Processing Systems.* Curran Associates Inc., 2024. DOI: doi.org/10.48550/arXiv.2310.07820.

[HP22] Changquan Huang and Alla Petukhina. *Nonstationary Time Series Models.* Springer International Publishing, 2022. DOI: 10.1007/978-3-031-13584-2_5.

[HST20] Santosh Harish, Nishmeet Singh, and Rahul Tongia. “Impact of temperature on electricity demand: Evidence from Delhi and Indian states”. In: *Energy Policy* 140 (2020). DOI: 10.1016/j.enpol.2020.111445.

[Ilb+24] Romain Ilbert et al. “SAMformer:Unlocking the Potential of Transformers in Time Series Forecasting with Sharpness-Aware Minimization and Channel-Wise Attention”. In: *Proceedings of the 41 st International Conference on Machine Learning* (2024). DOI: doi/abs/10.5555/3692070.3692911.

[KB14] Diederik Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *International Conference on Learning Representations* (2014). DOI: doi.org/10.48550/arXiv.1412.6980.

[Kis24] András Kis. *Understanding Autocorrelation and Partial Autocorrelation Functions (ACF and PACF).* 2024. URL: `https://medium.com/@kis.andras.nandor/understanding-autocorrelation-and-partial-autocorrelation-functions-acf-and-pacf-2998e7e1bcb5`.

[Kon+23] Vaia I. Kontopoulou et al. "A Review of ARIMA vs. Machine Learning Approaches for Time Series Forecasting in Data Driven Networks". In: *Future Internet* 15.8 (2023). DOI: `10.3390/fi15080255`.

[Kro08] Anders Krogh. "What are artificial neural networks?" In: *Nat Biotechnol* 26 (2008), pp. 195–197. DOI: `10.1038/nbt1386`.

[Kun+23] Manuel Kunz et al. "Deep Learning Based Forecasting: A Case Study from the Online Fashion Industry". In: *Forecasting with Artificial Intelligence: Theory and Applications.* Ed. by Mohsen Hamoudia, Spyros Makridakis, and Evangelos Spiliotis. Springer Nature Switzerland, 2023, pp. 279–311. DOI: `10.1007/978-3-031-35879-1_11`.

[Lia+24a] Yuxuan Liang et al. "Foundation Models for Time Series Analysis: A Tutorial and Survey". In: *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining.* Association for Computing Machinery, 2024, pp. 6555–6565. DOI: `10.1145/3637528.3671451`.

[Lia+24b] Wenlong Liao et al. "TimeGPT in load forecasting: A large time series model perspective". In: *Applied Energy* 379 (2024). DOI: `https://doi.org/10.1016/j.apenergy.2024.124973`.

[Liu+24] Haoxin Liu et al. "LSTPrompt: Large Language Models as Zero-Shot Time Series Forecasters by Long-Short-Term Prompting". In: (2024). DOI: `doi.org/10.48550/arXiv.2402.16132`.

[McK84] Ed. McKenzie. "General exponential smoothing and the equivalent arma process". In: *Journal of Forecasting* 3.3 (1984), pp. 333–344. DOI: `https://doi.org/10.1002/for.3980030312`.

[McL23] Saul McLeod. *Correlation in Psychology: Meaning, Types, Examples coefficient.* 2023. URL: `https://www.simplypsychology.org/correlation.html`.

[MF24] Kelian Massa and Dario Fanucchi. "Domain-adapted Lag-Llama for Time Series Forecasting in the African Retail Sector." In: *NeurIPS Workshop on Time Series in the Age of Large Models.* 2024. URL: `https://openreview.net/forum?id=mTCMfeK4jR`.

[Mik+13] Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *1st International Conference on Learning Representations, ICLR2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings* (2013). DOI: doi.org/10.48550/arXiv.1301.3781.

[Nie19] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2019. URL: http://neuralnetworksanddeeplearning.com/index.html.

[NS24] Imran Nasim and João Lucas de Sousa Almeida. “Fine-Tuned MLP-Mixer Foundation Models as data-driven Numerical Surrogates?” In: *NeurIPS 2024 Workshop on Data-driven and Differentiable Simulations, Surrogates, and Solvers*. 2024. URL: https://openreview.net/forum?id=dUg1dihFDT.

[Oli+23] Kin G. Olivares et al. “Neural basis expansion analysis with exogenous variables: Forecasting electricity prices with NBEATSx”. In: *International Journal of Forecasting* 39.2 (2023), pp. 884–900. DOI: doi.org/10.1016/j.ijforecast.2022.03.001.

[Oli+24] Kin G. Olivares et al. “Probabilistic hierarchical forecasting with deep Poisson mixtures”. In: *International Journal of Forecasting* 40.2 (2024), pp. 470–489. DOI: https://doi.org/10.1016/j.ijforecast.2023.04.007.

[PC24] Santosh Kumar Puvvada and Satyajit Chaudhuri. “Critical Evaluation of Time Series Foundation Models in Demand Forecasting”. In: *NeurIPS Workshop on Time Series in the Age of Large Models*. 2024. URL: https://openreview.net/forum?id=TS42sRKINd.

[PW16] Ofir Press and Lior Wolf. “Using the Output Embedding to Improve Language Models”. In: (2016). DOI: doi.org/10.48550/arXiv.1608.05859.

[Rad+19] Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. In: 2019. URL: CorpusID:160025533.

[Rob23] Amber Roberts. *Mean Absolute Percentage Error (MAPE): What You Need To Know*. 2023. URL: https://arize.com/blog-course/mean-absolute-percentage-error-mape-what-you-need-to-know/.

[RW23] Jingli Ren and Haiyan Wang, eds. *Mathematical Methods in Data Science*. MIT Press, 2023. Chap. Chapter 3 - Calculus and optimization, pp. 51–89. URL: https://www.sciencedirect.com/science/article/pii/B9780443186790000090.

[Sal+20] David Salinas et al. “DeepAR: Probabilistic forecasting with autoregressive recurrent networks”. In: *International Journal of Forecasting* 36 (3 2020), pp. 1181–1191. DOI: doi.org/10.1016/j.ijforecast.2019.07.001.

[Sch+24] Craig W. Schmidt et al. “Tokenization Is More Than Compression”. In: *International Conference on Learning Representations* (2024). DOI: 10.48550/arXiv.2402.18376.

[Shi+24] Xiaoming Shi et al. “Time-MoE: Billion-Scale Time Series Foundation Models with Mixture of Experts”. In: (2024). DOI: doi.org/10.48550/arXiv.2409.16040.

[SR20] Eranga De Saa and Lochandaka Ranathunga. “Comparison between ARIMA and Deep Learning Models for Temperature Forecasting”. In: *ArXiv* (2020). DOI: CorpusID:226282476.

[SS17] Robert H. Shumway and David S. Stoffer. “ARIMA Models”. In: *Time Series Analysis and Its Applications: With R Examples.* Springer International Publishing, 2017, pp. 75–163. DOI: 10.1007/978-3-319-52452-8_3.

[SZ24] Rajat Sen and Yichen Zhou. *google-research/timesfm.* 2024. URL: https://github.com/google-research/timesfm.

[THS16] H.E Thornton, B.J. Hoskins Hoskins, and A.A Scaife. “The role of temperature in the variability and extremes of electricity and gas demand in Great Britain”. In: vol. 11. Environmental Research Letters, 2016. DOI: 10.1088/1748-9326/11/11/114015.

[Vas+17] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems.* Vol. 30. Curran Associates, Inc., 2017.

[Wei13] William W.S. Wei. “Time Series Analysis”. In: *The Oxford Handbook of Quantitative Methods in Psychology: Vol. 2: Statistical Analysis.* Oxford University Press, 2013. DOI: 10.1093/oxfordhb/9780199934898.013.0022.

[Wil+24] Andrew Williams et al. “Context is Key: A Benchmark for Forecasting with Essential Textual Information”. In: 2024. DOI: 10.48550/arXiv.2410.18959.

[Woo+24] Gerald Woo et al. “Unified Training of Universal Time Series Forecasting Transformers”. In: *Proceedings of the 41 st International Conference on Machine Learning* (2024). DOI: doi/10.5555/3692070.3694248.

[Wu+22] Haixu Wu et al. “TimesNet: Temporal 2D-Variation Modeling for General Time Series Analysis”. In: *ArXiv* abs/2210.02186 (2022). DOI: 10.48550/arXiv.2210.02186.

[WW02] Jane Webster and Richard T. Watson. “Analyzing the Past to Prepare for the Future: Writing a Literature Review”. In: *MIS Quarterly* 26 (2002). DOI: 10.2307/4132319.

[Yan+24] Luoxiao Yang et al. “ViTime: A Visual Intelligence-Based Foundation Model for Time Series Forecasting”. In: (2024). DOI: doi.org/10.48550/arXiv.2407.07311.

[Yin19] Xue Ying. “An Overview of Overfitting and its Solutions”. In: *Journal of Physics: Conference Series* 1168.2 (2019). DOI: 10.1088/1742-6596/1168/2/022022.

[Yu+24] Xiaoxin Yu et al. “Misstatements, misperceptions, and mistakes in controlling for covariates in observational research”. In: *Epidemiology and Global Health* (2024). DOI: 10.7554/eLife.82268.

[Zen+23] Ailing Zeng et al. “Are Transformers Effective for Time Series Forecasting?” In: *Proceedings of the AAAI Conference on Artificial Intelligence* 37.9 (2023), pp. 11121–11128. DOI: 10.1609/aaai.v37i9.26317.

[Zho+23] Tian Zhou et al. “One Fits All: Power General Time Series Analysis by Pretrained LM”. In: *Neural Information Processing Systems*. 2023. DOI: CorpusID:258741419.

[ZJT16] Yating Zhang, Adam Jatowt, and Katsumi Tanaka. “Towards understanding word embeddings: Automatically explaining similarity of terms”. In: *2016 IEEE International Conference on Big Data (Big Data)* (2016), pp. 823–832. DOI: 10.1109/BigData.2016.7840675.

[ZLM19] Chen Zhang, Hua Liao, and Zhifu Mi. “Climate impacts: temperature and electricity consumption”. In: *Natural Hazards* 99 (3 2019). DOI: 10.1007/s11069-019-03653-w.

Declaration of Authorship

I hereby declare that, to the best of my knowledge and belief, this thesis titled *Analysing the Foundation Models for Time-Series Forecasting* is my own, independent work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citation and references; this also holds for tables and graphical works.

Münster, 21.01.2024

[illegible]

Ellen Alina Parthum

Consent Form

Name: Ellen Alina Parthum
Title of Thesis: Analysing the Foundation Models for Time-Series Forecasting

What is plagiarism? Plagiarism is defined as submitting someone else's work or ideas as your own without a complete indication of the source. It is hereby irrelevant whether the work of others is copied word by word without acknowledgment of the source, text structures (e.g. line of argumentation or outline) are borrowed or texts are translated from a foreign language.

Use of plagiarism detection software. The examination office uses plagiarism software to check each submitted bachelor and master thesis for plagiarism. For that purpose the thesis is electronically forwarded to a software service provider where the software checks for potential matches between the submitted work and work from other sources. For future comparisons with other theses, your thesis will be permanently stored in a database. Only the School of Business and Economics of the University of Münster is allowed to access your stored thesis. The student agrees that his or her thesis may be stored and reproduced only for the purpose of plagiarism assessment. The first examiner of the thesis will be advised on the outcome of the plagiarism assessment.

Sanctions Each case of plagiarism constitutes an attempt to deceive in terms of the examination regulations and will lead to the thesis being graded as "failed". This will be communicated to the examination office where your case will be documented. In the event of a serious case of deception the examinee can be generally excluded from any further examination. This can lead to the exmatriculation of the student. Even after completion of the examination procedure and graduation from university, plagiarism can result in a withdrawal of the awarded academic degree.

I confirm that I have read and understood the information in this document. I agree to the outlined procedure for plagiarism assessment and potential sanctioning.

Münster, 21.01.2024

[illegible]

Ellen Alina Parthum