UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

eXascale Infolab

UNIVERSITY OF FRIBOURG

MASTER THESIS

Unrolling Time

Author:
Albin Aliu

Supervisors:
Dr. Giuseppe Cuccu
Prof. Dr. Philippe Cudré-Mauroux

December 30, 2024

A thesis submitted in fulfillment of the requirements for the degree of Master in the

eXascale Infolab
Department of Informatics

Boulevard de Pérolles 90 • 1700 Fribourg • Switzerland
phone +41 (26) 300 84 65 • diuf-secr@unifr.ch • www3.unifr.ch/inf

Abstract

Albin Aliu

Unrolling Time

Time series data is of great interest across various domains, such as finance, natural sciences and engineering applications. It is thus to no surprise that quite some research has been done in providing models which can capture the complexity of time series data. In recent years, with the hype around generative AI, transformer-based models such as Autoformer, PatchTST, and iTransformer have dominated the benchmarks alongside neural network models such as N-BEATS, DLiner, and TimeMixer. These models offer impressive performance compared to classical time series models like ARIMA or those based on RNNs. However, the main drawback of these large and complex models is that they usually require large amounts of data to perform well and also require long training times.

In this work, we present a framework of how patterns can be learned from time series data even with models like linear regression, which usually do not work well on time series data out of the box. Our approach is a two-layered stacking approach, where on both layers linear regression is used. The first layer generates a set of predictions, which is the input for the second layer, similar to how layers are stacked in regular feedforward neural networks. The second layer is trained on the data generated by the first layer and then used to make a final prediction. The framework is implemented in Python and provides an API to further extend it with other models.

We evaluated the model using linear regression on both layers on various common datasets for one-step ahead predictions. Compared to classical time series models like SARIMA, we saw performance gains of up to 93% on complex time series that do not exhibit regularity.

We conclude that even simpler, non-traditional time series models can be made to capture the complexity of time series using our framework.

Keywords: forecasting, time series, machine learning, linear regression, model stacking, lagging

Contents

Chapter 1

Introduction

Historically, mathematicians Carl Friedrich Gauss and Adrien-Marie Legendre developed the first time series models. Both used the least squares models to compute orbits of celestial bodies (Stigler, 1986) around 1800. Around 150 years later, the method of exponential smoothing was known. In the 1970s, ARIMA (Box and Jenkins, 1976) was introduced and laid the foundation for many more models and variations, including the introduction of mathematical rigor to time series analysis. In recent years, very complex neural networks and transforms dominated the field (Wang et al., 2024b), where big players like Google (Scott and Varian, 2014), Meta (Taylor and Letham, 2017), Microsoft (Ke et al., 2017) and Amazon (Salinas et al., 2020) each investigate heavily in time series research and provide a model of their own. A big drawback of these recent models is that they require large amounts of data and long training times, which in some cases is not an option.

Our proposed work is a two layered approach with a specific preprocessing and batching of the data, which enables to learn different patterns. Before introducing the method of *unrolling time*, we quickly review the basics of time series analysis to establish a common language.

1.1 Time Series Data

1.1.1 Time Series Data Characteristics

Time series data offer certain characteristics which can render some traditional statistical models unapplicable. In this section, we will briefly discuss certain characteristics which need to be taken into account when developing a new model. This list is surely non-exhaustive but should provide a solid foundation to treat this type of data.

Temporal Ordering. Standard statistical models often assume that observations are independent and identically distributed. This assumption is not usually applicable to time series data, where observations are time-specific and exhibit temporal ordering. In time series, each observation is part of a sequence, and its position in time contains information. This means the current observation might be influenced by previous ones, which differs from the assumed independence in many traditional statistical methods.

Trends. In time series data, trends indicate long-term upward or downward movements. They are different from short-term variations or seasonal patterns and represent consistent changes over time. Recognizing trends is necessary for model accuracy, as overlooking them can lead to skewed predictions by failing to account for the data's directional movement.

Cyclic Behavior. Cyclic behavior in time series data refers to patterns that occur at irregular intervals, often influenced by economic, political, or environmental factors. These patterns differ from seasonal trends due to their irregular nature in terms of duration and magnitude. Modeling such behavior can be challenging due to this irregularity, but it is a necessary aspect of understanding time series, especially in areas like economics where business cycles have significant effects.

Stationarity. Stationarity in time series data refers to the concept where statistical properties of the series, such as mean, variance, and autocorrelation, are constant over time (Shumway and Stoffer, 2000). This is fundamental in time series analysis because many models are developed on the assumption of stationarity.

There are two main types of stationarity:

- *Strong or Strict Stationarity:* This implies that the joint distribution of any subset of time series observations remains the same regardless of shifts in time. It's a stringent form, seldom met in practical scenarios due to its rigorous requirements.
- *Weak or Second-Order Stationarity:* This is a less strict form, requiring only the mean and variance to be constant over time and the autocovariance to depend only on the lag between observations and not on time itself. Many time series models assume at least weak stationarity.

Non-stationary data, where these statistical properties change over time, often require transformation to make them stationary. Such transformations might include differencing the series, applying logarithmic or square root transformations, or using other models designed to handle non-stationarity.

Noise or Random Variation. Noise in time series data represents the unpredictable and random elements that are inherent in real-world data. This noise can arise from different sources, such as measurement errors or unforeseen events. Separating actual patterns from this noise is a key aspect of time series analysis.

Univariate, Multivariate When considering *univariate data* in the context of time series, we are typically referring to *one variable*, often also called a *signal*. For example, this might include a series of temperature measurements in degrees Celsius. However, when thinking about models, the terms *univariate* and *multivariate* become a bit more ambiguous. If a model solely considers one stream of input, e.g., 10 previous measurements, then it is clearly a *univariate* model. If the model, however, accepts two streams of inputs, e.g., the previous 10 measurements from two different locations, and it uses both data streams interdependently, then it is considered a *multivariate* model. The proposed framework in this work will use an univariate treatment of the data, but it is possible, depending on the choice of model on each layer, to treat the data as multivariate and use multivariate models for the intermediate predictions of layer 1, as seen later in section 2.2.1.

1.1.2 Time Series Models

Given the definitions from above, we can now introduce some common models used in the context of *time series forecasting*. Among other properties, time series models can be divided into two different categories: *univariate models* and *multivariate*

models. We further try to categorize the models into fields by taking their historical background into account.

Classical models

Even though all kinds of models are in the end by definition statistical models, we are focusing here on models that arose in the time before abundant computing power and data were available for training models like eural networks and transformers.

Autoregressive Models (AR) are linear *regression models* (Box and Jenkins, 1976) defined as a linear combination of a finite number of past values of a time series. Given p past values z_i at equally spaced time intervals $i = t-1, t-2, \ldots, t-p$ and an *error term* a_t (often referred to as *white noise*), the *autoregressive model of order* p is denoted as $AR(p)$ and defined as

$$z_t = \phi_1 z_{t-1} + \phi_2 z_{t-2} + \ldots + \phi_p z_{t-p} + a_t \tag{1.1}$$

This model contains $p+1$ unknown parameters: $\phi_1, \ldots, \phi_p$ and σ_a^2. In practice, a_t is typically assumed to be normnally distributed with zero mean and constant variance.

Note 0.1. It is common to either treat the data before the parameter learning or do mean centering on the fly by substituting z_i with $z_i' = z_i - \mu$, where μ is the mean of the series. This is an additional parameter to be learned from the data.

Moving Average Models (MA) are also a form of linear *regression models*, focusing on the error component of time series data. In a moving average model of order q, denoted as $MA(q)$, the current value of the series is defined as a linear combination of the past q error terms. Suppose $a_{t-1}, a_{t-2}, \ldots, a_{t-q}$ represent the error terms at time intervals $t-1, t-2, \ldots, t-q$. Then, the moving average model is defined as

$$z_t = \mu + a_t + \theta_1 a_{t-1} + \theta_2 a_{t-2} + \ldots + \theta_q a_{t-q} \tag{1.2}$$

Analogously, the $q+2$ parameters θ_i, σ_a^2 and μ (mean) are learned from the training data.

Autoregressive Integrated Moving Average Models (ARIMA) combine the ideas of autoregressive models (AR) and moving average models (MA), and also incorporate the concept of integration. This model is particularly useful for non-stationary time series data, a common occurrence in many practical applications. The ARIMA model is denoted as $ARIMA(p, d, q)$ where p is the order of the autoregressive part, d is the degree of differencing (the number of times the data have had past values subtracted), and q is the order of the moving average part.

The ARIMA model is given by the equation:

$$\left(1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p\right)(1-B)^d z_t = \left(1 + \theta_1 B + \theta_2 B^2 + \ldots + \theta_q B^q\right) a_t \tag{1.3}$$

In this equation, B is the *backshift operator*, defined by $Bz_t = z_{t-1}$. The term $(1-B)^d z_t$ represents the differencing operation applied d times to achieve stationarity in the time series data. The parameters ϕ_i and θ_j are the coefficients of the autoregressive and moving average parts of the model, respectively, which are learned

from the data (Kotu and Deshpande, 2019).

Seasonal ARIMA (SARIMA) extends the ARIMA framework to handle seasonal patterns by incorporating additional seasonal autoregressive, integrated, and moving average terms. This is denoted as $ARIMA(p,d,q) \times (P,D,Q)_s$, where (P,D,Q) are the seasonal orders and s is the length of the seasonal cycle. The seasonal differencing term $(1-B^s)^D$ is applied alongside the non-seasonal differencing to ensure stationarity, and the seasonal AR and MA components capture seasonal dependencies that occur at regular intervals. A common extension of SARIMA is *SARIMAX*, which introduces external (also denoted exogenous) variables as additional predictors, allowing the model to incorporate broader contextual factors and potentially improve forecast accuracy (Kotu and Deshpande, 2019). Note that these exogenous variables must be known or estimated at each prediction step.

Recent Models

After (S)ARIMA, RNNs dominated the field for quite some time before transformers and even more complex neural networks were introduced. We quickly go over some of the popular models from the past years.

N-BEATS (Oreshkin et al., 2020) is a deep neural architecture that relies solely on fully connected layers with backward and forward residual links. Its design allows the model to capture both trend and seasonality components effectively without the need for recurrent or convolutional layers, offering inherent interpretability due to its basis expansion approach.

The **Autoformer** (Wu et al., 2022) builds upon the transformer architecture by introducing the Auto-Correlation Mechanism to replace self-attention. This modification enables the model to capture long-term dependencies more efficiently, addressing the computational inefficiencies of traditional transformers in time series forecasting.

DLinear (Zeng et al., 2022) challenges the necessity of complex deep learning models by demonstrating that simple linear models can outperform transformers, especially for long-term forecasting tasks. By decomposing the time series into trend and seasonal components, DLinear simplifies the modeling process while maintaining high performance.

Inspired by advancements in computer vision, models like **TimeMixer** (Wang et al., 2024a) and **PatchTST** (Nie et al., 2023) adopt architectures originally designed for image processing. TimeMixer replaces convolutional layers with MLP layers to capture both temporal and feature-wise dependencies efficiently. PatchTST processes time series data as patches, similar to Vision Transformers, enabling the model to capture both local and global temporal patterns.

TimesNet (Wu et al., 2023) introduces a novel neural network architecture specifically designed for time series analysis. It utilizes time-domain operations to model temporal variations effectively, capturing intricate patterns that general models might overlook.

Lastly, **SCINet** (Liu et al., 2022) employs a Sample Convolution and Interaction mechanism to capture complex temporal dependencies. By modeling both short-term and long-term patterns, SCINet enhances forecasting performance without relying on recurrent structures.

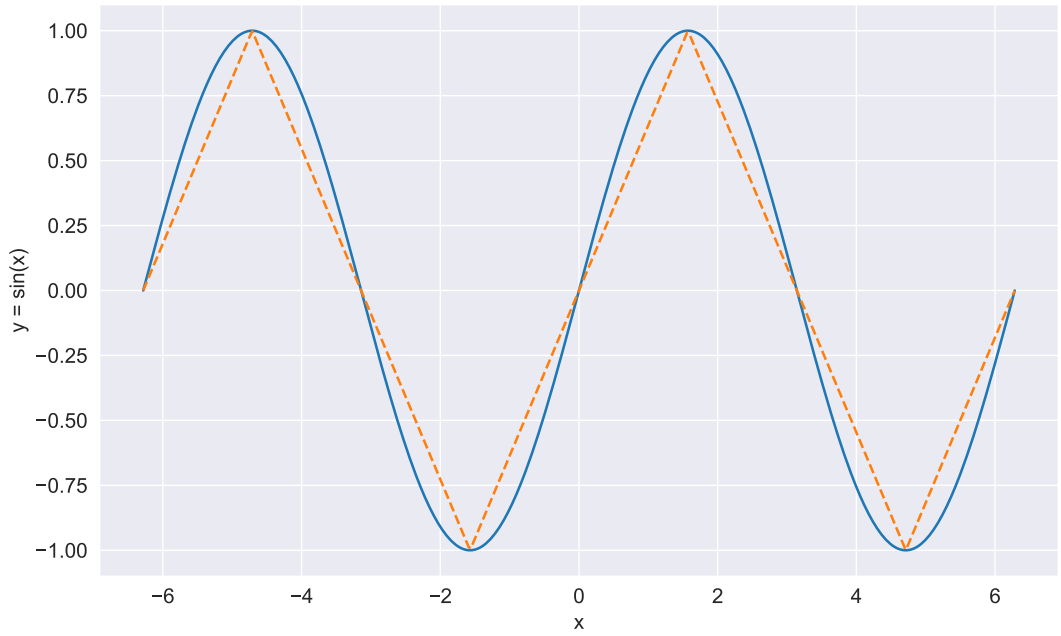


FIGURE 1.1: Graph of the original function $\sin(x)$ and its piecewise linear segment approximation using eight segments.

One major drawback of these recent models is that they typically need large datasets for training and also demand significant computational resources, compared to classical models like ARIMA and SARIMA.

1.2 Linear Approximation

1.2.1 Segmentation

Our approach is based on the fundamental mathematical principle that any differentiable function can be locally approximated to arbitrary precision by a linear function, as formalized by Taylor's theorem (see Theorem 1.2.1). This concept is foundational in mathematical analysis and is particularly relevant in time series analysis, where data inherently focus on specific time points, making local linear approximations especially powerful.

Figure 1.1 shows an example of a piecewise linear approximation, approximating the function $\sin(x)$. The linear functions chosen here are just the segments for each interval at each $\frac{\pi}{2}$ step. And there are many ways to determine each linear function component, as well as how to choose the convenient interval steps.

For a continuous function f, we get the slope of the function at point x_0 by:

$$\lim_{x \to x_0} \frac{f(x_0) - f(x)}{x_0 - x} = f'(x) \tag{1.4}$$

This describes the process of the segment windows in Figure 1.1 becoming arbitrarily small and we can approximate the function at point x_0 to arbitrary precision. Using this idea to model a function at a given point, we come across *Taylor's Theorem*, in our case restricted to $n = 1$ and we only introduce the first order of the Taylor polynomial, that is a linear function.

Theorem (First Order Taylor Polynomial)**.** *Given a function $f : \mathbb{R} \to \mathbb{R}$ that is differentiable at $x = a$, the first order Taylor polynomial of f at a is defined as*

$$P(x) = f(a) + f'(a)(x - a)$$

This polynomial $P(x)$ is the linear approximation of the function f around the point $x = a$. In other words, $P(x)$ is the tangent to the curve $y = f(x)$ at the point $(a, f(a))$.

Some limitations with segmentation.
In practice, we tend to work with data that is discretely sampled, but, in theory, the functions underlying these data are often continuous. However, continuous function modeling over a complete domain is computationally costly, especially when high accuracy is required. This approach looks only at the local information around a single point; it might not capture the global behavior of the function. Besides, limited computational resources prohibit the use of an infinite number of segments to approximate a function. Yet, we obtained a good approximation to $\sin(x)$ using only eight segments. Determining the optimal breakpoint numbers and individual segment lengths with regard to a required approximated resolution is actually an optimization problem of model parameters. One classic work concerning such topics dates back to 1961 in (Stone, 1961); for an overview regarding today's research in this direction, consider (Warwicker and Rebennack, 2021).
Finally, it should be noted that if we do the naive approach by only using the first and last value of an interval to compute the segment, we effectively discard all information the intermediate points might have.

1.2.2 Linear Regression

To address the limitations inherent in the naive segmentation technique, we employ the principle of *least squares fitting*. In the context of a one-dimensional data set, this approach involves fitting a straight line that minimizes the cumulative error across all data points within a specified segment window. For our purposes, we assume $\dim(\mathcal{F}) = 1$ and $\mathcal{T} \subset \mathbb{Z}$. Formally, given a time series T and a non-empty subset $S \subseteq T$, the residual error for each point in S can be described as follows:

$$\forall (t, x) \in S : f(t) + e_t = \hat{y} + e_t = x \tag{1.5}$$

The objective is to minimize the total least squares error, which is represented by:

$$\sum_{i=1}^{|S|} e_i^2 = \sum_{i=1}^{|S|} (x_i - \hat{y}_i)^2 \tag{1.6}$$

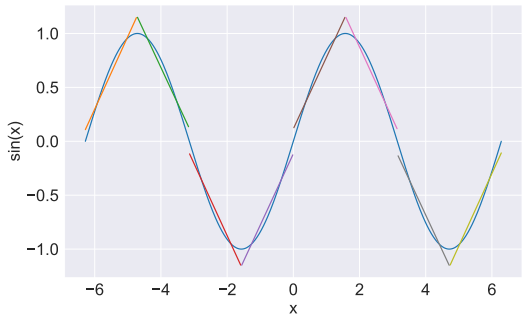
This minimization problem can be formalized using the argmin operator:

$$f^* = \underset{f}{\operatorname{argmin}} \sum_{i=1}^{|S|} (x_i - f(t_i))^2 \tag{1.7}$$

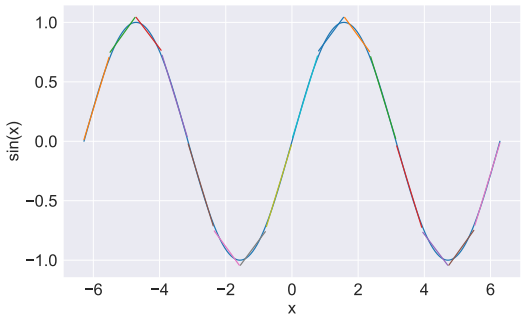
Here, f^* represents the optimal linear function that minimizes the total least squares error for the subset S. This is just a fancy way to say: Assume $f(x) = mx + q$, then we want to solve

$$\underset{m,q}{\operatorname{argmin}} \sum_{i=1}^{|S|} (x_i - (mt + q))^2 \tag{1.8}$$

In fact, what we are doing here, is *Linear Regression* (Barr and Çetinkaya-Rundel, 2022). Taking our previous example $\sin(x)$ results then in these graphs, depending on the number of segments:



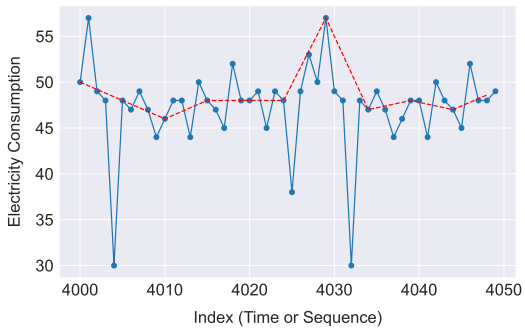
(A) Graph of the original function $\sin(x)$ and its piecewise linear regression approximation using 8 segments.



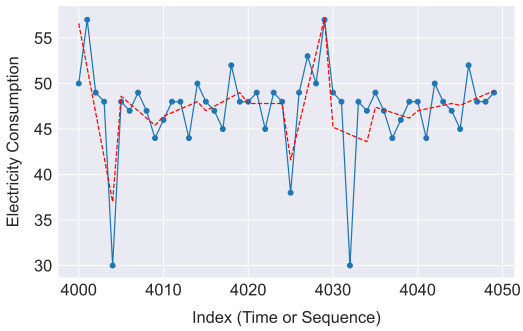
(B) Graph of the original function $\sin(x)$ and its piecewise linear regression approximation using 16 segments.

FIGURE 1.2: Comparative graphs of piecewise linear regression approximations.

To illustrate the power of this method, Figure 1.3 shows the fitting on a real time series data set using both techniques shown so far.



(A) Graph of an electricity data set and its piecewise segmentation approximation using 10 segments.



(B) Graph of an electricity data set and its piecewise linear regression approximation using 10 segments.

FIGURE 1.3: Comparative graphs of piecewise linear regression approximations and segmentation approximations.

1.2.3 Higher dimensional data

So far, all examples have been one-dimensional, primarily because they are simpler to illustrate and conceptualize. However, it is important to note that when dealing with datasets of multiple features, we are in the realm of *multivariate models*. In such instances and in the context of linear models, particularly when applying Linear Regression, we are essentially fitting a hyperplane to the data, rather than a line.

Additionally, there exists a case in which the univariate data comprises vectorial data points, rather than scalars. Despite this distinction, the computational and fitting processes remain the same; it is merely the interpretation of the data that differs. Refer to Figure 1.4 for further illustrations related to a two-dimensional dataset.

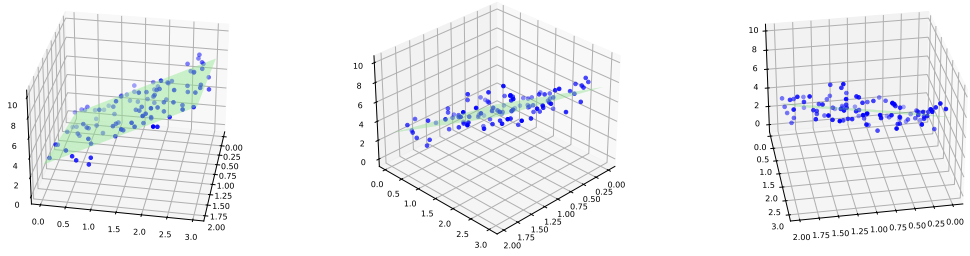


FIGURE 1.4: Three 3D plots showing a plane fit on two-dimensional data, viewed from different angles.

1.3 Lagging features

A feature engineering technique commonly used in various models in the context of time series data is *lagging features*. In this technique, new features are created from previous past values, hence the name lagging. This technique is fundamental to models like ARIMA or SARIMA, which rely on past values to forecast future trends. Our proposed framework employs this strategy, too. See section 1.5.3 for our detailed method. In general, lagging features involve shifting the time series data by one or more time steps, to align past values with current outcomes. This alignment allows models to learn from previous cycles and patterns, improving their accuracy. This is particularly crucial in time series forecasts, where past trends influence future trends (Shumway and Stoffer, 2000, Hyndman et al., 2008).

Below is an example illustrating how lagging features are created from a given dataset wth a two day lag:

Day	Sales
1	100
2	120
3	115
4	130
5	125

TABLE 1.1: Original Sales Data

Day	Sales	Sales 1-day Lag	Sales 2-day Lag
1	100	-	-
2	120	100	-
3	115	120	100
4	130	115	120
5	125	130	115

TABLE 1.2: Sales Data with Lagged Features

1.4 Ensemble Models

An ensemble model combines multiple individual models, often referred to as *base models*, into a single model. The intention behind this architecture is to improve prediction accuracy and robustness beyond what any single base model could achieve. For a given set of models $m_1, \ldots, m_n$, and an input x, each m_i makes a prediction $m_i(x) = \hat{y}_i$. These predictions are then aggregated into a single prediction $m(x) = \hat{y}$, where m refers to our ensemble model. The aggregation method can vary; for regression tasks, it might incorporate a weighted average of $\hat{y}_i$, while for classification tasks, a majority vote among the $\hat{y}_i$ might be used (Oza, 2005).

Classical ensemble techniques include *bagging* and *boosting*, which aim to reduce variance and bias, respectively (Breiman, 1996) (Freund and Schapire, 1997). Bagging, or *Bootstrap Aggregating*, involves training multiple instances of the same model on different subsets of the training data and then aggregating their predictions. Boosting involves sequentially training models where each model tries to correct the errors made by the previous models, resulting in a focus on difficult to predict instances.

1.4.1 Stacking

A more advanced technique in the context of ensemble learning is the method of *stacking* as introduced in (Wolpert, 1992), predating boosting and bagging. The concept behind stacking is that on top of the base models, a second model is trained on the predictions of each base model, often referred to as the *meta-model*. This meta-model leverages the strengths of the base models and corrects their weaknesses by combining their predictions.

Given a (x, y) pair from our training data, where x is the input and y the real target, and given a set of models $m_1, \ldots, m_n$, the stacking approach creates synthetic training data for the meta-model in the form of $((m_1(x), \ldots, m_n(x)), y) = ((\hat{y}_1, \ldots, \hat{y}_n), y)$. The meta-model is then trained to combine these predictions $\hat{y}_1, \ldots, \hat{y}_n$ into a final prediction $\hat{y}$. This approach has been shown to improve model accuracy and robustness (Wolpert, 1992), and it offers more flexibility in model architecture, as it allows for combining a variety of different base models like decision trees, support vector machines, or neural networks and thus allows the model to handle more complex data distributions. However, it must be noted that stacking requires an additional layer of computation for training the meta-model, making the overall system more computationally expensive and harder to maintain. Depending on the choice of base models, this can be a limitation when training resources are constrained and the base models are computationally intensive to train.

1.5 Mathematical Definitions, Notations and Constructions

In the following subchapters, we present a set of mathematical definitions, notations and constructions, that will serve as the foundation for describing the method described in section 2.

1.5.1 Foundations

Definition 1 (Time Series)**.** *Given a set $\mathcal{F}$, referred to as the feature space, and a totally ordered set $(\mathcal{T}, \leq)$, called the time domain, a time series T is a function from $\mathcal{T}$ to $\mathcal{F}$. The function T assigns to each element t in $\mathcal{T}$ exactly one element x in $\mathcal{F}$, potentially allowing the same value of x to be associated with multiple values of t. The function can be represented by the set of pairs:*

$$T = \{(t, T(t)) \mid t \in \mathcal{T}\} \tag{1.9}$$

Here, $\mathcal{T}$ provides the temporal context for each observation x, ensuring that for every $t \in \mathcal{T}$, there exists a $x \in \mathcal{F}$.

Note 1.1. In most cases, we simply use $\mathcal{T} = \mathbb{Z}$ and $\mathcal{F} = \mathbb{R}^n$.

It is often assumed and preferred that the time domain $\mathcal{T}$ is equi-spaced. This assumption allows for the omission of the time parameter in the model, treating time as an implicit variable (Guthrie, 2020). Dealing with time series data that is not equally spaced presents additional challenges. These challenges necessitate certain assumptions and require additional preprocessing steps such as sampling.

In this work, we proceed under the assumption that $\mathcal{T}$ is equi-spaced, formally stated as:

$$\forall t_j, t_{j+1} \in \mathcal{T} : t_j - t_{j+1} = c, \tag{1.10}$$

where c is a constant, and the subtraction operation is appropriately defined for the set T. Without loss of generality, we proceed with simplification of our notation by assuming $c = 1$,

Note 1.2. It is important to note that if we wish to be rigorous, $\mathcal{T}$ cannot be just any totally ordered set; it needs to be a subset of some *ordered abelian group*. This subset however does not need to be a (sub)group itsself, as we can do the operations in the realm of the group. For simplicity, however, further specifications will be omitted from this discussion.

An element $x \in \mathcal{F}$ can be denoted as a vector $x = (f_1, \ldots, f_k)$. Each set of feature values F_i is defined as:

$$F_i := \pi_i(\mathcal{F}) = \{f_i \mid (f_1, \ldots, f_k) \in \mathcal{F}\} \tag{1.11}$$

where π_i is the projection function mapping to the i-th feature.

For notation simplicity, a time series T can also be represented as a (finite) sequence:

$$(x_t)_{t=1}^{|T|} = (x_1, \ldots, x_{|T|}) \tag{1.12}$$

where each $x_t = T(t)$ is a vector $x_t = (f_{1_t}, \ldots, f_{k_t})$, with f_{i_t} being the i-th feature at time t.

To further simplify the set notation used in representations of time series data while still keeping in mind the ordered nature of the data, we use the following representation:

$$\{x_t\}_T = \{x_t \mid x_t \in T\}. \tag{1.13}$$

Note here that, even though we are using curly brackets similar to set notations, the (temporal) order is still maintained and implicitly assumed.

Proposition 2. *A time series T inherits naturally a total ordering from $\mathcal{T}$.*

Proof. Given a time series T as defined in 1.9, we have that $\mathcal{T}$ is totally ordered by definition. Therefore, for any $x, y \in T$, it holds that $x = (t_x, x')$ and $y = (t_y, y')$, where the conditions $t_x < t_y$, $t_x = t_y$, or $t_y < t_x$ are true. We naturally extend this relation $\mathcal{R}$ to elements of $\mathcal{T}$. Thus, for $x = (t_x, x')$ and $y = (t_y, y')$, we define $\mathcal{R}'(x, y) := R(t_x, t_y)$. By using this *natural mapping* on T, we inherit the total ordering from $\mathcal{T}$ to T.

Definition 3 (Subseries)**.** *Given a time series T defined as a function from $\mathcal{T}$ to $\mathcal{F}$, and a non-empty subset $\mathcal{U} \subset \mathcal{T}$, we define a subseries U of T as a function from $\mathcal{U}$ to $\mathcal{F}$ that restricts T to $\mathcal{U}$. The subseries U can be represented by the set of pairs:*

$$U = \{(t, T(t)) \mid t \in \mathcal{U}\} \tag{1.14}$$

where U is a subset of the set representation of T.
We also refer to the set of all possible subseries of T as $\mathcal{S}(T)$.

Note 3.1. We further assume that $\mathcal{U} \subset \mathcal{T}$ is equi-spaced. If not, the data requires sampling.

Definition 4 (Preprocessing Function)**.** *A preprocessing function p is a map $p : \mathcal{S}(T) \to X$.*

Definition 5 (Model)**.** *A model is a map $m : X \to Y$, where X is the input space and Y the target space. This model is learned from the preprocessed time series data.*

Definition 6 (Model Spaces)**.** *The input space X consists of all possible inputs that a model m can accept. The target space Y includes all potential outputs that the model can produce.*

Note 6.1. With p, X can be constructed as desired. In the case where we want to predict some feature f_i at time t using all other features f_j at the same time t, we choose $X := \mathcal{F} \setminus F_i$ and $Y := F_i = \pi_i(\mathcal{F})$.

It is also in the preprocessing function, where we usually drop the t, as we assume $\mathcal{T}$ is equi-spaced and thus implicitly given.

Definition 7 (Univariate Model)**.** *An **univariate model** is a map $m : X \to Y$, where X and Y are sets of scalars. This model only has one independent variable in X and one dependent variable in Y.*

Definition 8 (Multivariate Model)**.** *A **multivariate model** is a map $m : X \to Y$, where X is a set of vectors and Y is a set of vectors or scalars. This model involves multiple independent variables in X and predicts one or more dependent variables in Y.*

Definition 9 (Time Series Forecasting Model, Prediction)**.** *If m represents a time series forecasting model — a model learned from time series data and used to predict future values, which is our primary focus in this work — then $m(x) = \hat{y}$ is commonly referred to as the prediction, and y is known as the target or actual value for the input x.*

Definition 10 (Horizon)**.** *The horizon in the context of time series forecasting refers to the length of the future time interval that a model attempts to predict. Formally, for a time series T and a prediction model m, if t_T is the latest available timestep in T, the horizon h is a positive integer such that the model m provides forecasts for the time series values at $t_T + 1, t_T + 2, \ldots, t_T + h$. This interval represents the set of future time points $\{t_T + 1 \leq t \leq t_T + h\}$ where predictions are made.*

Definition 11 (One-step ahead forecast/prediction)**.** *A one-step ahead forecast or prediction refers to the estimate provided by a forecasting model for the next immediate time point in a time series. For a given current time point t, the one-step ahead prediction is the forecast for time $t + 1$. This type of prediction focuses on the shortest possible horizon, which is $h = 1$, using the information available up to time t to predict the value at $t + 1$.*

Note 11.1. For simplicity in the notations moving forward, we will refer to the next immediate time step of a series T as the time step $t + 1$ and the latest available in the series as at step t. When we reference the time series explicitly, if it is not obvious in the context, we subscript additionally the series we are referring to, e.g. t_T or $(t + 1)_T$. Finally, we assume we start with $t = 1$ as the first index.

1.5.2 Slicing

We are going to introduce some notations for *slicing* T into a subseries as it is commonly used with lists in Python. Given two timesteps $a, b \in \mathcal{T}$, we define the subseries

$$T_{[a:b]} = \{ (t, x) \mid t \in [a, b) \ \subset \mathcal{T}, x \in \mathcal{F} \} \tag{1.15}$$

Note here that a is included and b is not included.

Additionally, we are going to introduce a notation $(\ \cdot\ )_{-1}$ for *removing the latest data point of a time series* T (*at time* $t = t_T$).

$$T_{-1} = T \setminus \{(t_T, x)\} \tag{1.16}$$

and recursively define for $r \geq 2$:

$$T_{-r} = T_{-r+1} \setminus \{(t_{T_{-r+1}}, x)\} \tag{1.17}$$

These notations allow for the construction of any *connected* subseries $U \subset T$, where *connected* means that we do not discard any datapoints in between.

Note here that 1.17 is just a special case of $T_{[a:b]}$ for $a = 1$ and $b = t_T - r - 1$.

1.5.3 Lagging

The lagging, as conceptually described in section 1.3, can be expressed as a preprocesing function $\ell : \mathcal{S}(T) \to X$. Given an a number of desired lags $\lambda \in \mathbb{N}$, we define:

$$\ell_\delta(U) : \mathcal{S}(T) \to X \tag{1.18}$$

$$\{x_t\}_U \mapsto \{x_t, x_{t-1}, \ldots, x_{t-\lambda}\}_U \tag{1.19}$$

for some subseries $U \subset T$. Note here that if U contains data points at $t = 1, \ldots, \lambda$, we silently discard them, since they would require features at $t = 0, -1, \ldots, \lambda - 1$. Additionally, we will not add any additional rows after t_T, that could be constructed for the lagging features, due to the uknown values for $f_{i_{t_T+j}} \forall i$ and $j > 0$. These values are unknown because they are not part of our dataset for T.

1.5.4 Chain of Subseries Construction

Definition 12 (Chain of Subseries)**.** *Given a time series T, a chain of suberies of T is defined as a subset $C \subset \mathcal{S}(T)$, where for all $U, V \in C : U \subset V$ or $V \subset U$.*

Let $\omega, \eta \in \mathbb{N}$. We will denote ω as the *window size* or *window length* and η as the *number of windows*. We construct a chain of subseries $C = \{T_i \mid 1 \leq i \leq \eta\}$. Each T_i is a subseries of T defined as:

$$T_i = T_{[t-\omega i\,:\,t]} \tag{1.20}$$

where t represents the latest available time step of the series T.

It is easy to see that $\forall (1 \leq i \leq \eta - 1) : T_i \subset T_{i+1}$ and thus $T_1 \subset T_2 \subset \ldots \subset T_\eta$.

This construction essentially dissects the time domain of the time series into a chain. Figure 1.5 depicts this time dissection.

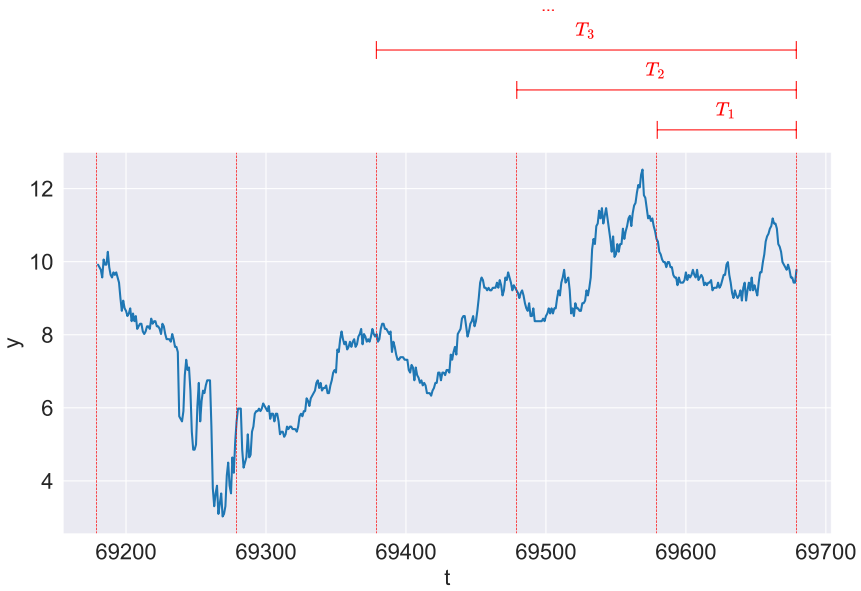


FIGURE 1.5: Image of graph with markings of T1, T2, etc.

Chapter 2

Method

In this chapter, we introduce the architecture of our proposed approach. The initial section provides an overview of the model's architecture. Subsequent sections will detail the specifics of the preprocessing, as well as the two layers a time series T goes through before arriving at the final prediction $\hat{y}$, which represents the next immediate time step in T.

2.1 Overview of Model Architecture

The model takes as input a time series T. This time series undergoes a preprocessing step in which the data is enriched with lagged features and dissected into smaller subseries. These subseries are then processed through *Layer 1* (denoted as L_1), where each subseries yields one prediction. The collection of all these predictions forms the output of L_1 and serves as training data for the meta-model, *Layer 2* (denoted as L_2). L_2 then uses this data to make a final prediction $\hat{y}$, which corresponds to a single value at the same sampling interval as T. Figure 2.1 depicts a simplified pipeline of the described model.

By the nature of this model, the training data to the model is at the same time the input to the model. Also, we are going to focus for now on doing a one-step ahead forecast. However, all of these mechanics can be adapted to increase the horizon of the model as we will show later.

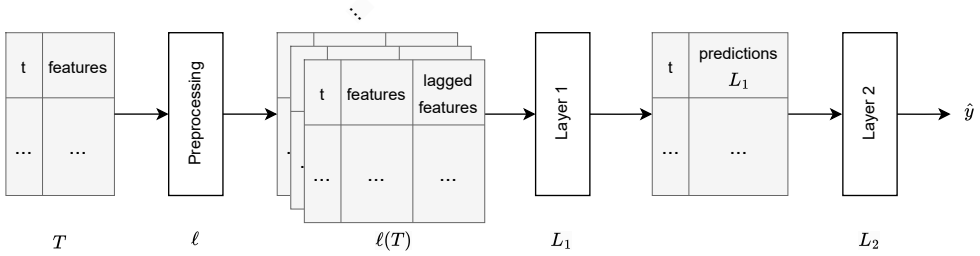


FIGURE 2.1: The pipeline from a time series T to the prediction $\hat{y}$.

2.1.1 Final formalities and time assumptions

In the case of one-step ahead forecast, we are interested in predicting one or multiple features f_i of $x = (f_1, \ldots, f_k) \in \mathcal{F}$ for the next immediate time step in a given time series T (or any of its subseries U). That is, we want to predict f_i at $t = \max(\mathcal{T}) + 1$ (under the assumption that $\mathcal{T}$ is equi-spaced).

We further proceed with the case where we restrict our method to predicting *one* feature f_j, given *all other* features f_i, $j \neq i$. The restriction on *using all other features*

during inference is artificially set, to simplify the mathematical description of the model. Indeed, it is possible to work with a *configurable list of features available at time t* during training and inference, such that unknown features at time t can be discarded.

For the further sections, let $T = \{(t, T(t)) \mid t \in \mathcal{T}\}$ be a time series as in 1.9.

2.2 L_1 Construction

2.2.1 Base Model

We start by introducing the following hyperparameters:

- Let $\lambda \in \mathbb{N}$. This will be the number of lags that we apply to our time series T. To do so, we choose a preprocessing (lagging) function ℓ_λ as defined in section 1.5.3.
- Let $\omega, \eta \in \mathbb{N}$. These parameters will be used to construct a chain of subseries, as defined in section 1.5.4. ω denotes the *window size* and η the *number of windows* in a subseries chain.

Given a chain of subseries $T_1 \subset T_2 \subset \ldots \subset T_\eta$, we transform each T_i into a series with λ lagging features by applying ℓ_λ on it:

$$T_i' = \ell_\lambda(T_i) \tag{2.1}$$

This applies the lagging operation on all features. If the series T and hence its subseries T_i so far only had *one feature*, i.e. $\dim \mathcal{F} = 1$, and thus consisted of a signal only, then this preprocessing will transform each subseries T_i into a series with $\lambda + 1$ features for any $t \in \mathcal{T}$:

$$(f_t, f_{t-1}, \ldots, f_{t-\lambda}) \in \mathcal{F}^{\lambda+1},\ t \in \mathcal{T} \tag{2.2}$$

If the series already had $k \in \mathbb{N}_{>1}$ features, then T_i' will have $k(\lambda + 1)$ features:

$$(f_{1_t}, f_{1_{t-1}}, \ldots, f_{1_{t-\lambda}}, \ldots, f_{k_t}, f_{k_{t-1}}, \ldots, f_{k_{t-\lambda}}) \in F_1^{\lambda+1} \times F_2^{\lambda+1} \times \ldots \times F_k^{\lambda+1} \tag{2.3}$$

This construction still maintains a chain:

$$\forall\ 1 \leq i \leq \eta - 1 :\ T_i' \subset T_{i+1}' \tag{2.4}$$

and thus

$$T_1' \subset T_2' \subset \ldots \subset T_\eta'. \tag{2.5}$$

Note. In the framework we have developed, we do not give restrictions about the choice of model for either layer. For our experiments, we introduce later the `alpha` model, which uses on both layers linear regression models. For this reason, we will use linear regression as a base model for the subsequent sections to describe our method. However, we want to stress that any other regression model could replace these base models.

We fit per preprocessed subseries T'_i a linear regression model m_i on the series T'_i, yielding η linear regression models $m_1, \dots m_\eta$ trained up to but without $t_{T'_i} = t_{T_i} = t_T$[1] as follows:

Before fitting, we drop any non-lagged feature, i.e. we drop the f_{i_t} columns for all features i. One of these features is at the same time the target feature f_{j_t}. Thus our input space is

$$x_t = (f_{1_{t-1}}, f_{1_{t-2}}, \dots, f_{1_{t-\lambda}}, f_{2_{t-1}}, \dots, f_{2_{t-\lambda}}, \dots, f_{k_{t-\lambda}}) \in \prod_{i=1}^{k} F_i^\lambda \tag{2.6}$$

and the target space

$$y_t = (f_{j_t}) \in F_j.$$

Note here that our input x_t (at time t) is all features at time $t-1, \dots, t-\lambda$.

We fit each model m_i on each T'_i up to but without x at $t_{T'_i}$. Finally, to put the models m_i into our mathematical context, we note that:

$$m_i : F_1^\lambda \times F_2^\lambda \times \dots \times F_k^\lambda \to F_j$$

This might differ if the models m_i are fitted differently, see section 2.2.4.

2.2.2 Base Model Prediction

Using the above described method, we construct now the training data for the L_2 layer, the meta-model, by repeating the process δ times with a different chain of subseries of T. For this, we introduce two more hyperparameters:

- Let $\sigma \in \mathbb{N}$. This represents the *step size*, which is used to slice off the last σ data entries from our original time series T (cf. section 1.5.2).
- Let $\delta \in \mathbb{N}$. This determines the size of the synthetic dataset to train L_2.

Given a time series T, a number of windows η, a window size ω and a lagging function ℓ_λ, we construct a lagged chain of suberies $T'_1 \subset T'_2 \subset \dots \subset T'_\eta$ as described in the previous section, starting with T. For each T'_i, a model m_i is trained up to but without $t_{T'_i}$.

We construct x_t as defined in 2.9 for the next immediate timestep $t = t_{T'_i} + 1$:

$$x_{t_{T'_i}+1} = (f_{1_{t_{T'_i}}}, f_{1_{t_{T'_i}-1}}, \dots, f_{1_{t_{T'_i}-\lambda}}, f_{2_{t_{T'_i}}}, \dots, f_{2_{t_{T'_i}-\lambda}}, \dots, f_{k_{t_{T'_i}-\lambda}}) \in \prod_{i=1}^{k} F_i^\lambda \tag{2.7}$$

.

We then compute η predictions, i.e. for each T'_i:

$$m_i(x_{t_{T'_i}+1}) = \hat{y}_{i_{L_1}}.$$

From these predictions, we will build a new row for our new dataset:

[1] This equality is only true given our strategy to devise the time as described in section 1.5.4. In case of a different strategy, where the latest available timestep t_T of the original time series T is not present in all elements of the subseries chain, i.e. $\exists\, T_i : x_{t_T} \notin T_i$, then this equality does not hold.

$$(\hat{y}_{1_{L_1}}, \ldots, \hat{y}_{\eta_{L_1}})$$

This is our first data point of our synthetic dataset for training L_2. We will further mark it with a superscript to be able to distinguish it later:

$$(\hat{y}^0_{1_{L_1}}, \ldots, \hat{y}^0_{\eta_{L_1}})$$

We start with the superscript index of 0 because there was no slicing done on our origin series T, which we will do in the iteration process of L_1. We will additionally denote the models m_i yielding the first data points as m_i^0.

2.2.3 Iteration process of L_1

We repeat the previously described process with the same configuration in terms of hyperparameters but with a cut off time series: We will reduce the data points in the original time series T by cutting off the last σ entries, i.e. we will construct $T_{-\sigma} \subset T$ (cf. 1.5.2). This will be our new base series. Given this series, we will repeat the same process to compute m_i for $(\ell_\lambda(T_{-\sigma}))_i = (T'_{-\sigma})_i$. This will yield again η trained models m_i based on $(T'_{-\sigma})_i$. We will denote them as m_i^1, as they are the first iteration after our base iteration. In the same way as before, we are going to construct $x_{t_{T_{-\sigma}+1}}$. Note here the following equality:

$$x_{t_{T_{-\sigma+1}}} = x_{t_T-\sigma+1} \tag{2.8}$$

Finally, by applying the new learned models m_i^1, we compute:

$$m_i^1(x_{t_{T_{-\sigma}+1}}) = \hat{y}^1_{i_{L_1}}$$

This gives us our second row in our new data set for L_2:

$$\hat{y}^1_{L_1} = (\hat{y}^1_{1_{L_1}}, \ldots, \hat{y}^1_{\eta_{L_1}})$$

We repeat this δ times, to get δ rows:

$$m_i^j(x_{t_{T_{-\sigma}+1}}) = \hat{y}^j_{L_1} = (\hat{y}^j_{1_{L_1}}, \ldots, \hat{y}^j_{\eta_{L_1}}), \;\; 0 \leq j \leq \delta - 1.$$

Finally, each $\hat{y}^j_{L_1}$ represents a row of η predictions for $t = t_T - j\sigma + 1$. More specifically, $\hat{y}^0_{L_1}$ represents the prediction at $t_T + 1$, i.e. the next step in our time series.

2.2.4 Features available at t

The data construction described above essentially fits a model m to predict the feature target f_j at time t using all features f_i at time $t-1, \ldots, t-\lambda$. We do this because when we use m to predict f_j at $t+1$, the features f_i at time $t+1$ are usually not available for any i. However, if for any feature f_i, the value is known at time $t+1$, then it is possible not to drop f_i at t during fitting. Later, the value for the feature f_i can be injected during inference of $t+1$. This is similar to how SARIMA models require additional features as input during prediction when the model is trained using an exogenous variable.

2.3 L_2 Construction and Training

2.3.1 L_2 Training

Once the data for the L_2 layer has been constructed, we can proceed with the training of the second layer. The second layer resembles very much the convential stacking technique, where we train a *decision maker* on top of our L_1 data. The choice of model is here again up to the user, but once again Linear Regression yields reliable results while keeping the overall system complexity low and does not require too much data to be fitted.

Given a model of choice, referred to as the *decision maker* and further down as DM, we want (and have) that

$$\text{DM} : F_j^{\eta} \rightarrow F_j.$$

The DM model is responsible for making decisions based on the η predictions provided by the models in L_1.

To train the DM model, we fit the model on the rows $\hat{y}_{L_1}^j$ for $j = 1, ..., \delta - 1$ as input features, and the real values y as target feature, which we still have from the original data set T. We intentionally do not include $\hat{y}_{L_1}^j$ at $j = 0$, i.e. $\hat{y}_{L_1}^0$. This data point represents the predictions for $t = t_T + 1$ as noted above. We also do not have an actual value y for $\hat{y}_{L_1}^0$, as this represents a prediction in the future, not available to our data set.

2.3.2 L_2 Prediction

After training the DM as outlined previously, we proceed to the final prediction generated by this combination of models. The DM was trained on $\hat{y}_{L_1}^j$ for $j = 1, ..., \delta - 1$, excluding $j = 0$. Now, we use this last data point to infer a value and obtain our final prediction:

$$\text{DM}(\hat{y}_{L_1}^0) = \hat{y}.$$

Here, $\hat{y}$ represents the final prediction from our model, specifically predicting the feature f_j at time $t_T + 1$ for a given series T.

2.4 Extending the prediction horizon

So far, the method presented focuses on generating a prediction for the next immediate timestep in a time series T, i.e., at $t = t_T + 1$. We show now some possible methods for extending this to a larger prediction horizon.

2.4.1 Fit L_1 to predict h days ahead

The first and most straightforward option is to fit the m_i models to predict h days ahead, instead of $h = 1$, as we previously did. The rest of the construction and method stays the same. This means, given a $h \in \mathbb{N}_{>1}$, we build our input space for m_i to be:

$$x_t = (f_{1_{t-h}}, f_{1_{t-h-1}}, \ldots, f_{1_{t-h-\lambda}}, f_{2_{t-h}}, \ldots, f_{2_{t-\lambda}}, \ldots, f_{k_{t-\lambda}}) \in \prod_{i=1}^{k} F_i^{\lambda} \tag{2.9}$$

and the target space stays the same:

$$y_t = (f_{j_t}) \in F_j.$$

Similarly, as before, our input x_t (at time t) is all features at time $t-h, \ldots, t-h-\lambda$.

Finally, if the model is trained in this manner, then the model is trained to always predict h steps ahead, similarly to how before it was only able to predict one value one-step ahead.

2.4.2 Data Sampling

Another direct approach is to change the frequency at which the data is sampled. For instance, if the original time series T consists of one measurement every 5 minutes, we can resample it at an hourly frequency. Training the model as before but on the hourly-sampled data, a one-step ahead prediction now corresponds to a full hour's prediction horizon. This allows the horizon of the model to be extended solely by preprocessing the data differently.

2.4.3 Autoregression

While the strategies discussed above effectively shift the horizon by adjusting the meaning of a single-step prediction, they do not inherently produce multiple successive predictions. To obtain a forecast for multiple future steps, we can apply the model autoregressively. In other words, once a prediction is made, this prediction can be appended to the original time series T, thus providing updated input data for predicting the subsequent time step.

This approach is straightforward when dealing with a univariate time series, as the model directly uses its own previous prediction as an additional data point. However, if the dataset is multivariate, the model is trained to predict only the designated target feature f_j and not the entire feature set $(f_1, \ldots, f_{j-1}, f_{j+1}, \ldots, f_k)$. In this case, simply lagging the other features is not possible without additional input data for a subsequent prediction. If external sources are available to inject these complementary features at each new time point, the same process can be extended as described in section 2.2.4. Another alternative is to train a separate model for each feature; the predictions from these additional models can then be combined to form the complete dataset suitable for autoregressive forecasting of the target variable.

Chapter 3

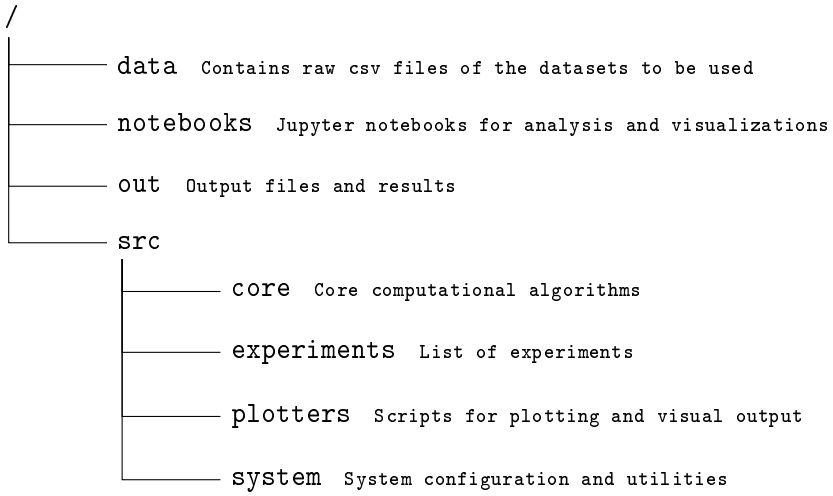
The Framework

In this chapter we describe the implementation of the framework in Python. Our implementation includes an API designed to help with the creation of custom *models* and *experiments* using the previously discussed techniques. The framework offers high configurability for various parameters and supports concurrent execution of multiple time series and experiments. The code can be found on GitHub.

We start by discussing the structure and general runtime flow of the framework, followed by a detailed examination of the API and its configurations.

3.1 Structure

The framework is structured as follows:



The framework's codebase is organized within the `src` directory. The `src/system` directory contains scripts that orchestrate the concurrent execution of selected experiments, located at `src/experiments`. The system configuration, managed through the file located at `src/system/config.py`, specifies which experiments to execute, supporting the following parameters:

3.2 Configurations

The framework is configured on two levels as follows:

- *System configuration* file located at `src/system/config.py`
- *Experiment configuration* file, usually located at `src/experiments/your_experiment/config.py`

The system configuration is mainly concerned with configuring parameters of the system such as concurrent execution, storing system outputs and also defining which experiments to run, while experiment configuration is specific to the experiment and model used. Parameters and Hyperparameters such as window sizes, step size and others are configured here.

3.2.1 System Configuration

The system configuration file is a simple python dictionary with the following keys:

- **parallel_execution**: *Boolean*. Specifies whether experiments should be executed in parallel using Ray[1].
- **ray_cores**: *Integer*. Specifies the number of cores to be utilized with Ray.
- **experiments_to_launch**: *Array of configurations*. Specifies configuration details for experiments to be launched.
  - **Experiment configuration format**:
    - **module**: *String*. Specifies the module to load.
    - **config**: *String*. Specifies the configuration file to load.
    - **exclude_configs**: *Path (optional)*. Specifies the directory of JSON files to exclude certain configurations.
- **out**: *Dictionary*. Specifies what outputs should be generated by the system.
  - **logs**: *Boolean*. Specifies whether logs should be outputted.
  - **l1_plots**: *Boolean*. Specifies whether base model plots should be generated.
  - **metrics_plots**: *Boolean*. Specifies whether plots comparing actual values to predicted values should be generated.
  - **profiling**: *Boolean*. Specifies whether profiling information should be collected.
  - **config**: *Boolean*. Specifies whether configuration details should be outputted.
  - **experiment**: *Boolean*. Specifies whether experiment details should be outputted.
- **OUT_FOLDER**: *Callback function*. Specifies a function configured to accept parameters `uid`, `module_name`, and `time`, returning a valid `Path` object. This callback defines the path for storing output files. The function `utils.create_dir()` can be used within this callback to create and ensure the existence of the directory structure if necessary.

An example configuration for parallel execution could look like this:

```
system_config = {
    'parallel_execution'           : True,
    'ray_cores'                    : 240,
    'experiments_to_launch'        : [
        {
```

[1]Ray is an open-source library for Python designed to help scale Python applications by parallelizing tasks to run efficiently on multiple cores and machines. https://github.com/ray-project/ray

```
            'module'                : 'experiments.ETT.experiment',
            'config'                : 'experiments.ETT.configs.explore'
        }
    ],

    'out' : {
        'logs'                      : True,
        'll_plots'                  : False,
        'metrics_plots'             : False, #L2 plots
        'profiling'                 : False,
        'config'                    : True,
        'experiment'                : True,
    },

    # Where to store
    OUT_FOLDER: lambda uid, module_name, time: utils.create_dir(
        Path('../out/runs') / time / module_name / uid
    )
}
```

3.2.2 Experiment Configuration

To launch an experiment, two things are required: A module with a `run` function and an accompanying configuration dictionary which are then referenced in the system configuration. These files are then stored in `src/experiments/your_experiment` as `experiment.py` and `config.py`. Storing the experiment code in a separate file enables the use of different model configurations for the same experiment.

The experiment configuration file

An example of a configuration file for one experiment should adhere to the following format:

```
  explore = {
      # Data processing configuration
      "cube": {
          "shifts": [-2, -4, -6, -8, -10, -12, -14, -16],
          "cut_off_fn": lambda df: df,
          "dropna": True,
      },
      # Model configuration
      "model": {
          "layer_1": layer_1_curry,
          "layer_1_config": {
              "y_column": "y_t",
          },
          "layer_2": layer_2_curry,
          "layer_2_config": {
              "y_column": "y_t",
          },
          "dm_training_set_size": [30, 100],
          "windows": [4, 8, 16],
          "direction": -1,
          "step_size": [16, 24, 32],
          "window_size": [64, 128, 256],
      },
      "experiment": {
          "experiment_range": range(-50, -1, 1),
          "dataset": Path("../data/electricity_consumption.csv"),
      },
```

```
28 }
```

The configuration is split up into three parts: The *cube*, *model*, and *experiment* configurations.

Hyperparameter Optimization

Each key's value in the configuration dictionary can either be a single value of the appropriate type or a list of those values. When a list is provided, the framework will run experiments in parallel using the cross product of all the lists in the dictionar to find the optimal parameter values using grid search. For example, the following configuration:

```
dict = {
  "param1": ["a", "b"],
  "param2": ["c", "d"]
}

```

The function call `configurator.cross_product(dict)` will produce four configurations (as a generator) corresponding to the combinations:

```
{ "param1": "a", "param2": "c" }
{ "param1": "a", "param2": "d" }
{ "param1": "b", "param2": "c" }
{ "param1": "b", "param2": "d" }
```

The system will then run the specified experiment for each configuration. If enabled, each configuration will be stored as a `config.json` file in the corresponding output folder.

Additionally, to save on computing, it is possible to exclude a list of configurations, by specifying in the system configuration a path to a directory containing configurations to exlcude. Ideally, use the outputted `config.json` files. The system will then exclude all configurations found in the specified directory.

Cube

A *cube* is a class that models a structured data container. Technically, it is a dictionary mapping time-series names to their corresponding pandas[2] DataFrame representations. The cube supports three parameters:

- **shifts** (λ): *Integer*. Specifies the number of lags to apply during preprocessing. (cf. 1.5.3).
- **cut_off_fn**: *Callback function*. A function that accepts a pandas DataFrame as its parameter `df`. This function allows for additional cutoffs or modifications to be applied to the series.
- **dropna**: *Boolean*. Specifies whether to remove rows that contain missing values, which may result from the creation of lagged features.

[2]Pandas is an open-source Python library that provides flexible and expressive data structures and data analysis tools. https://github.com/pandas-dev/pandas

Model

The **model** section of the configuration file is used to define the parameters and functions necessary for training and making predictions. The *layer_1* is responsible for generating a pandas DataFrame with the training data that is passed to *layer_2* to train the decision maker. Additionally, other hyperparameters are configured here:

- **layer_1**: *Callable function*. This function will receive two arguments when called: `chains_dfs` and `source_cube`. The `chains_dfs` is a list of tuples of the format: (`series` (string), list of pandas DataFrames corresponding to $T'_1 \subset T'_2 \subset \ldots \subset T'_\eta$). The `source_cube` is a `Cube` object that can be used to access the entire scope of the time series during training and predictions. Finally, the `layer_1` function will be provided all arguments defined in the `layer_1_config` of the model configuration.
- **layer_1_config**: *Dictionary*. This dictionary contains additional (custom) configuration options that are passed directly to the `layer_1` function. In most examples in the source code, the desired target feature name is often passed as an additional parameter.
- **layer_2**: *Callable function*. This function will be called with two arguments: `training_data` and `source_cube`. `training_data` is the output generated by `layer_1`.
- **layer_2_config**: *Dictionary*. Similar to `layer_1_config`, this dictionary provides additional (custom) configuration options for the `layer_2` function.
- **dm_training_set_size** (δ): *Integer*. Specifies size for the training dataset that `layer_1` will generate for `layer_2`.
- **windows** (η): *Integer*. The number of windows to use per chain.
- **step_size** (σ): *Integer*. The step size between each subsequent subset chain start.
- **window_size** (ω): *Integer*. The window size. (cf. section 1.5.4).

Experiment

The experiment config is technically specific to the `experiment.py` file. In the provided examples, we use the following parameters:

- **experiment_range**: *Range*. Specifies the range of the predictions to be made in the time series. The framework will cut off this range. 0 represents the the next immediate time step outside of the available data for the time series.
- **dataset**: *Path*. Specifies the path to the dataset file that will be used in the experiment. This path should point to a CSV file containing the data for the experiment.

3.2.3 The experiment module

Once a proper configuration has been set up, the specified module in the system configuration should contain a function called `run`, accepting a dictionary as input. This dictionary is the experiment's configuration that it is run with. The framework ships

with a basic experiment setup that accepts as described above a `experiment_range` and `dataset` argument, found in `system/experiments_defaults.py`. This experiment setup uses also the built-in model called `alpha`, which uses on both layers L_1 and L_2 linear regression.

An example of how the default shipped `default_run` function can be used to load one time series:

```
def run(config: Dict[str, Any]) -> None:
  dataset_path = get_dataset_path(config)
  df = pd.read_csv(file_path, header=0, parse_dates=[0], index_col=0)
  dfs = {'series' : df}
  default_run(config, dfs)
```

3.3 Models

The framework ships with the `alpha` model, which processes the data as described in Section 2. The `alpha` model does not make any assumptions about the models used in L_1 and L_2, as it is only concerned with partitioning, processing and feeding the data into each layer. That is why the default configuration also requires a `layer_1` and `layer_2` function. The framework also ships with built-in implementations using linear regression for both layers, which can be found in `core/layer_1` and `core/layer_2` respectively.

3.3.1 The `alpha` Model

The `alpha` model dissects time as described in Section 1.5.4. To develop an alternative model that segments time differently within this framework, one would need to createa a new model. The model's core functionality is to dissect the time and provide the data frames to the first and then the second layer (or any other layer if more layers are added). The module `cube_to_training_data` is specific to the `alpha` model, as it is the one responsible for the the segmentation of the data and then calling the `layer_1` on each window T_i'.

The implementation of the `alpha` model is otherwise fairly straightforward:

```
def predict(
    layer_1: Callable,
    layer_1_config: Dict[str, Any],
    layer_2: Callable,
    layer_2_config: Dict[str, Any],
    subcube: Cube,
    source_cube: Cube,
    dm_training_set_size: int,
    windows: int,
    window_size_fn: Callable[[int], int],
    cut_off_step,
    direction: int = -1,
    ):

    training_data = cube_to_training_data.process(
        generator_fn = layer_1,
        generator_args = layer_1_config,
        cube = subcube,
        dm_training_set_size = dm_training_set_size,
        windows = windows,
        window_size_fn = window_size_fn,
        cut_off_step = cut_off_step,
        direction = direction,
```

```
            source_cube = source_cube
        )

        predictions = layer_2(
            training_data=training_data,
            source_cube=source_cube,
            **layer_2_config
        )

        return predictions
```

3.4 The Framework's Analysis Tool

The framework provides an analysis tool to facilitate the evaluation of a run experiment. The function `analyze_runs`, found in `analysis/analysis_tools.py`, accepts two arguments: A Path object for the `path` argument and a list of strings for the `config_fields` argument. The function will traverse the given path recursively and pick up all `predictions.json` and `config.json` files. It returns a pandas DataFrame with columns for the metrics mentioned above and the configuration values for the requested dictionary paths in `config_fields`.

An example call that requests the most relevant hyperparameters could look something like this:

```
analyze_runs(
    Path('runs/2024-01-11'),
    [
        'cube.shifts',
        'model.windows',
        'model.dm_training_set_size',
        'model.window_size',
        'model.step_size',
        'experiment.dataset'
    ]
)
```

Chapter 4

Evaluation

To evaluate our framework, we used multiple datasets commonly used in benchmarks of time series models. Additionally, we ran some baseline experiments using ARIMA, SARIMA or SARIMAX models, depending on the data characteristics, and regular Linear Regression. We then compared these results against the results of our `alpha` model on the same datasets, which uses on both layers linear regression. In this chapter, we detail our experiments and their results.

4.1 Datasets

4.1.1 Air Passengers

The Air Passengers dataset is a classic time series dataset from Box and Jenkins, 1976. It contains monthly totals of international airline passengers (in thousands) from 1949 to 1960. The dataset exhibits trends and seasonality but is non-stationary. It is a univariate dataset containing only the date and passenger count features. Figure 4.1a visualizes this dataset.

4.1.2 Electricity Transformer Temperature

The widely known dataset *Electricity Transformer dataset*, often abbreviated as *ETT* (Zhou et al., 2021), is a multivariate dataset and often used in research in the context of time series forecasting[1]. It consits of eight features:

- **date**: The recorded date (ISO 8601)
- **HUFL**: **High UseFul Load**
- **HULL**: **High UseLess Load**
- **MUFL**: **Middle UseFul Load**
- **MULL**: **Middle UseLess Load**
- **LUFL**: **Low UseFul Load**
- **LULL**: **Low UseLess Load**
- **OT**: **Oil Temperature** (target feature)

[1] Papers with Code - ETT Dataset — https://paperswithcode.com/dataset/ett, accessed 28-10-2024.

The *ETT-small* version consists of four CSV files of equi-spaced measurements from two different provinces in China: *ETTh1*, *ETTh2*, *ETTm1* and *ETTm2*. The files with *h* in the name are hourly measurements and the files with *m* in the name have frequency of 4 records per hour (every 15 minutes).

The dataset can be used for both short-term and long-term predictions, as it provides a robust combination of short-term and long-term periodic patterns, as well as irregular ones. Additionally, the data also exhibits long-term trends. Figures 4.1b, 4.1c, 4.1d, and 4.1e visualize the target feature for each dataset.

4.1.3 Weather

The Weather dataset, sourced from the Max-Planck-Institut für Biogeochemie[2], is a multivariate time series dataset designed for long-term forecasting tasks. It provides meteorological data recorded every 10 minutes from 2020-01-01 to 2021-01-01. The data is non-stationary and does not exhibit any trend. The dataset includes the following 21 features:

- **date**: The recorded date (ISO 8601)
- **p (mbar)**: Atmospheric pressure in millibars
- **T (°C)**: Temperature in degrees Celsius
- **Tpot (K)**: Potential temperature in Kelvin
- **Tdew (°C)**: Dew point temperature in degrees Celsius
- **rh (%)**: Relative humidity in percent
- **VPmax (mbar)**: Maximum vapor pressure in millibars
- **VPact (mbar)**: Actual vapor pressure in millibars
- **VPdef (mbar)**: Vapor pressure deficit in millibars
- **sh (g/kg)**: Specific humidity in grams per kilogram
- **H2OC (mmol/mol)**: Water vapor concentration in millimoles per mole
- **rho (g/m^3)**: Air density in grams per cubic meter
- **wv (m/s)**: Wind speed in meters per second
- **max. wv (m/s)**: Maximum wind speed in meters per second
- **wd (°)**: Wind direction in degrees
- **rain (mm)**: Rainfall in millimeters
- **raining (s)**: Duration of rain in seconds
- **SWDR (W/m^2)**: Shortwave downward radiation in watts per square meter
- **PAR (μmol/m^2/s)**: Photosynthetically active radiation in micromoles per square meter per second

[2]Weather Station Beutenberg from 2020-01-01 to 2021-01-01 — https://www.bgc-jena.mpg.de/wetter/, accessed 21-12-2024.

- **max. PAR (μmol/m^2/s)**: Maximum photosynthetically active radiation in micromoles per square meter per second
- **Tlog (°C)**: Log temperature in degrees Celsius
- **OT (ppm)**: CO_2 concentration in parts per million (target feature)

Figure 4.1f visualizes the target feature of this dataset.

4.2 Environment Specifications

4.2.1 Hardware

Architecture

- Type: x86_64, 64-bit
- Byte Order: Little Endian

CPU

- Model: AMD EPYC 7742 64-Core Processor
- Cores: 128 cores (64 cores per socket, 2 sockets)
- Threads per Core: 2
- Frequency: 1500 MHz to 2250 MHz

Memory

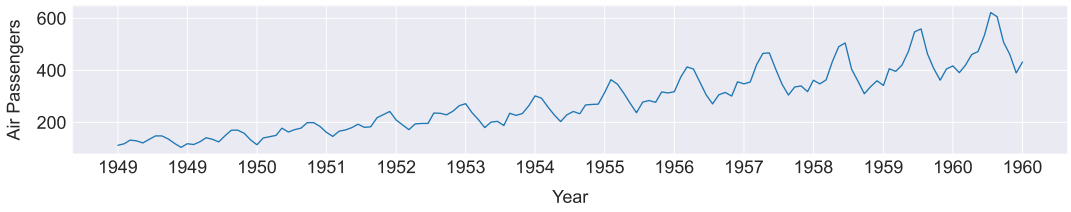
- L1 Cache: 4 MiB (data + instructions)
- L2 Cache: 64 MiB
- L3 Cache: 512 MiB

4.2.2 Software

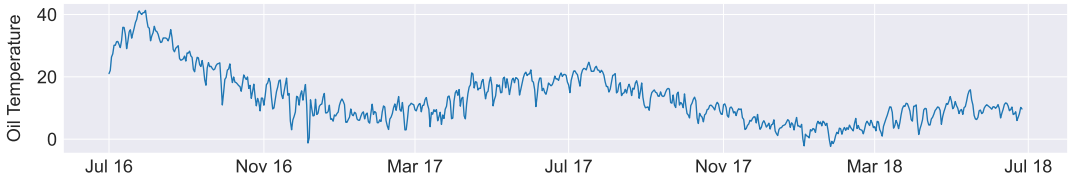
The framework is written in Python 3.11 using `pandas`[3] 2.1.4, `scikit-learn`[4] 1.3.2 and `ray`[5] 2.9.0. For the baseline experiments, we used `pmdarima`[6] 2.0.4.

The repository with all the source code and experiments for reproduction can be found on GitHub[7].

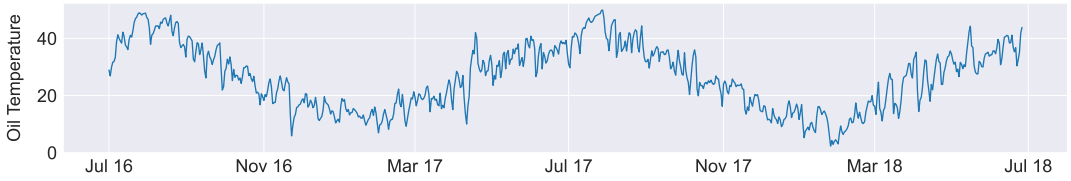
[3] https://pandas.pydata.org/
[4] https://scikit-learn.org/
[5] https://www.ray.io/
[6] https://pypi.org/project/pmdarima/
[7] https://github.com/albxncom/thesis-final



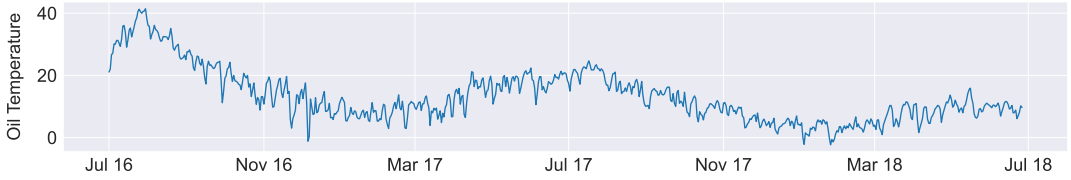
(A) Air Passengers dataset



(B) ETTm1 dataset

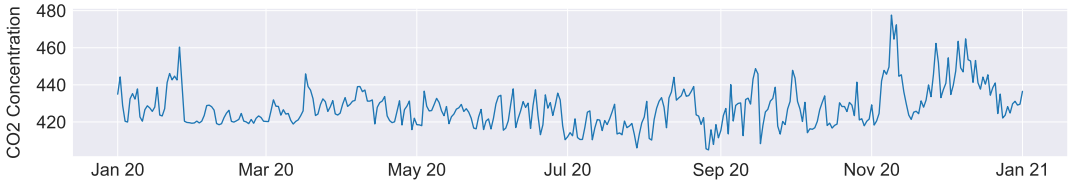


(C) ETTm2 dataset



(D) ETTh1 dataset

(E) ETTh2 dataset



(F) Weather dataset

FIGURE 4.1: Plots of the target feature for each used dataset

4.3 Experimental Setup

We conducted multiple experiments on different datasets using the `alpha` model. It works on univariate as well as multivariate data. The code can be found in `src/core/layer_1/lin_reg.py` and in `src/core/layer_2/lin_reg_one_step.py` respectively.

4.3.1 Experiment Categorization

We categorize the experiments into two categories based on whether the data is univariate or multivariate. Additionally, we list which datasets and models were used for comparison.

Univariate Experiments

The Air Passengers dataset is originally univariate. We created additional univariate time series datasets of the target feature of ETTh1 and Weather. Additionally, both datasets were downsampled to one measurement per day to accelerate computations for the baseline models. We used the same sampling to evaluate the `alpha` model.

We trained ARIMA models for the ETTh1 and Weather dataset and a SARIMA model for the Air Passengers. While the Air Passengers dataset exhibits seasonality, the ETTh1 dataset also contains a two-year seasonal pattern. However, this seasonality is insufficient for reliably training a SARIMA model due to shifts in the data. The Weather dataset does not display seasonality. Finally, we trained our `alpha` model for each dataset to compare.

Multivariate Experiments

For the multivariate experiments, we evaluated our framework on the complete ETT (ETTh1, ETTh2, ETTm1, and ETTm2) and the Weather dataset. We then compared our results with trained ARIMAX models on the same datasets. For these experiments, we did not resample the data. The SARIMA models utilized all features as exogenous variables and were provided these features at the time of prediction. Conversely, the alpha model was kept simpler, and thus these features were not available when the prediction was made (see Section 2.2.4).

4.3.2 Experiment Procedure

The Baseline Models

The `alpha` model is a one-step ahead model. To ensure a fair comparison with the baseline models, each baseline model was also utilized to make only one-step ahead predictions. Specifically, for each time series, we initially excluded the last 30 data points and trained the respective model on the remaining dataset using `pmdarima`'s `auto_arima` function to determine the optimal order. The initially determined optimal order was then fixed and used for all 30 predictions to maintain consistency across the prediction steps. The trained model was then used to generate a prediction for the immediate next timestep. Subsequently, we incrementally expanded the training window by one data point — excluding only the last 29 data points — and

retrained the model to make the subsequent prediction using the fixed order. This iterative process was continued, progressively increasing the training set by one data point each time until all desired predictions were obtained.

The `alpha` Model

Given an experiment configuration, the framework will launch a *run* for each configuration of the experiment configuration's cross product as described in Section 3.2.2. The `alpha` model was used for all runs. The framework then collects and stores the specific configuration and predictions made into a `predictions.json` and `config.json` file in the output folder.

The framework abstracts this procedure in the function `run` found in `src/system/experiment_defaults.py`. For this experiment, the experiment section in the configuration file accepts a `experiment_range` parameter. We opted for a range of `range(-30,0,1)`. For each run, the model is trained iteratively by incrementally expanding the training window one data point at a time, starting with a cut-off at the 30th last data point and progressing to the last data point (i.e., from -30 to -1). After each incremental training, the model generates a one-step ahead prediction for the subsequent timestamp. This process results in the model being trained and making a single prediction 30 times per run, thereby mirroring the iterative methodology used for the baseline experiments.

4.4 Evaluation Metrics

To assess the performance of the `alpha` model on the ETT dataset, several standard evaluation metrics commonly used in time series forecasting were computed.

For each run, the following metrics were computed:

- **Mean Squared Error (MSE)**
- **Mean Absolute Error (MAE)**
- **Root Mean Squared Error (RMSE)**
- **Root Relative Squared Error (RRSE)**

These metrics were calculated over 30 predictions for each configuration, ranging from -30 to -1, as detailed in Section 4.3.2.

All metrics discussed below are outlined in Witten, Frank, and Hall, 2011.

4.4.1 Root Mean Squared Error (RMSE)

The **Root Mean Squared Error (RMSE)** is the square root of the Mean Squared Error (MSE). It provides an error metric in the same units as the target variable, making it easily interpretable.

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

4.4.2 Mean Absolute Error (MAE)

The **Mean Absolute Error (MAE)** measures the average magnitude of the errors made:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

4.4.3 Root Relative Squared Error (RRSE)

The **Root Relative Squared Error (RRSE)** is a normalized version of the RMSE.

$$\text{RRSE} = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where:

- $\bar{y}$ is the mean of the actual values.

An RRSE value of less than 1 is an indication that the model performs better than using the mean as prediction.

4.5 Results

For the results section, we will elaborate on the *ETTh1.csv* dataset specifically before summarizing the overall results over all datasets. All these results can be reproduced with the experiments in the `src/experiments` folder. Finally, the provided Jupyter Notebook in `src/analysis/` can be used to visualize and conveniently display the results.

4.5.1 Hyperparameter Optimization Results

With the *ETTh1.csv* dataset, we experimented with multiple ranges for the hyperparameters. In most cases, we started with 1 or 2 and continued with the 2^n sequence. Table 4.1 shows the ten best (out of 1680) configurations sorted by MSE.

TABLE 4.1: Ten best runs on *ETTh1.csv* sorted by MSE (rounded to three decimals)

MSE	MAE	RMSE	RRSE	η	δ	ω	σ	λ
0.073	0.228	0.270	0.392	16	256	256	1	16
0.100	0.292	0.317	0.460	16	256	8	8	32
0.110	0.269	0.332	0.482	16	128	256	2	16
0.125	0.314	0.353	0.512	64	128	8	2	128
0.145	0.313	0.381	0.553	4	128	4	8	64
0.146	0.316	0.381	0.554	4	128	4	8	128
0.147	0.276	0.383	0.556	8	128	8	32	64
0.150	0.292	0.387	0.562	32	256	4	8	32
0.157	0.326	0.396	0.575	4	128	4	4	128
0.160	0.336	0.399	0.580	4	128	4	16	32

Configuration distribution

Figure 4.2 shows box plots of the distribution of MSE vs. hyperparameter configurations across all 1680 runs. The white dots represent outliers. Not all runs succeeded, which is also noticeable when looking, for example, at the plot for the η parameter (number of windows/chain length) for value 64. These failures occur because the product of window length and the number of windows exceeds the available time series data.

Some values produce better results than others, e.g. the average MSE increases up to 1.8 times for the window length between $\omega \leqslant 64$ and $\omega = 256$. Also to note: for the number of shifts $\lambda = -16$ shows best results for this dataset.

4.5.2 Univariate Results

The Air Passengers dataset exhibits strong seasonality (with 144 data points spanning 10 years and a 12-point seasonal cycle), enabling SARIMA to excel with a 42% performance improvement over `alpha`. In contrast, the Weather dataset does not follow a clear seasonal pattern, and the ETTh1 dataset only shows weak weekly seasonality (period of 7), which provides only a negligible advantage. Consequently, SARIMA and `alpha` achieve comparable results on ETTh1, while in the Weather dataset, predicting CO_2 — an irregular time series lacking strong seasonality and patterns — allows the `alpha` model to outperform SARIMA by approximately 30% in terms of MSE and 20% in MAE. Table 4.2 shows the results for the univariate datasets, with bold values indicating the best performance on the specific error measurement.

TABLE 4.2: Experiment Results for Univariate Datasets

	`alpha`							SARIMA								
Dataset	MSE	MAE	η	δ	ω	σ	λ	MSE	MAE	p	d	q	P	D	Q	s
Air Passengers	467.41	17.98	4	64	16	1	16	**268.8**	**12.9**	1	1	0	2	1	1	12
Weather	**52.45**	**5.55**	8	64	32	4	2	75.3	7	4	0	0	0	0	0	-
ETTh1	**1.07**	0.74	8	64	4	4	2	1.1	**0.73**	1	1	2	1	0	1	7

We conclude from these results that SARIMA excels when the dataset encompasses multiple seasonal cycles like the Air Passengers dataset does. On the other hand, `alpha` demonstrates superior performance on irregular datasets such as the Weather dataset. Additionally, when the data contains some regular patterns but no seasonality, like the ETTh1 dataset, both models perform comparably.

4.5.3 Multivariate Results

We begin by comparing a simple Linear Regression model, trained on the raw data using all features, which makes a single prediction after each training phase. This process is repeated 30 times with different cutoff points, following our methodology. The results, presented in Table 4.3, demonstrate that Linear Regression fails to capture the complexity of the data. In contrast, both `alpha` and SARIMA, despite relying solely on the target feature and disregarding all other features in our univariate experiments, perform more effectively. However, by wrapping our framework around Linear Regression models on both layers and *unrolling* the time series in a particular way, we make Linear Regression work in this setting.

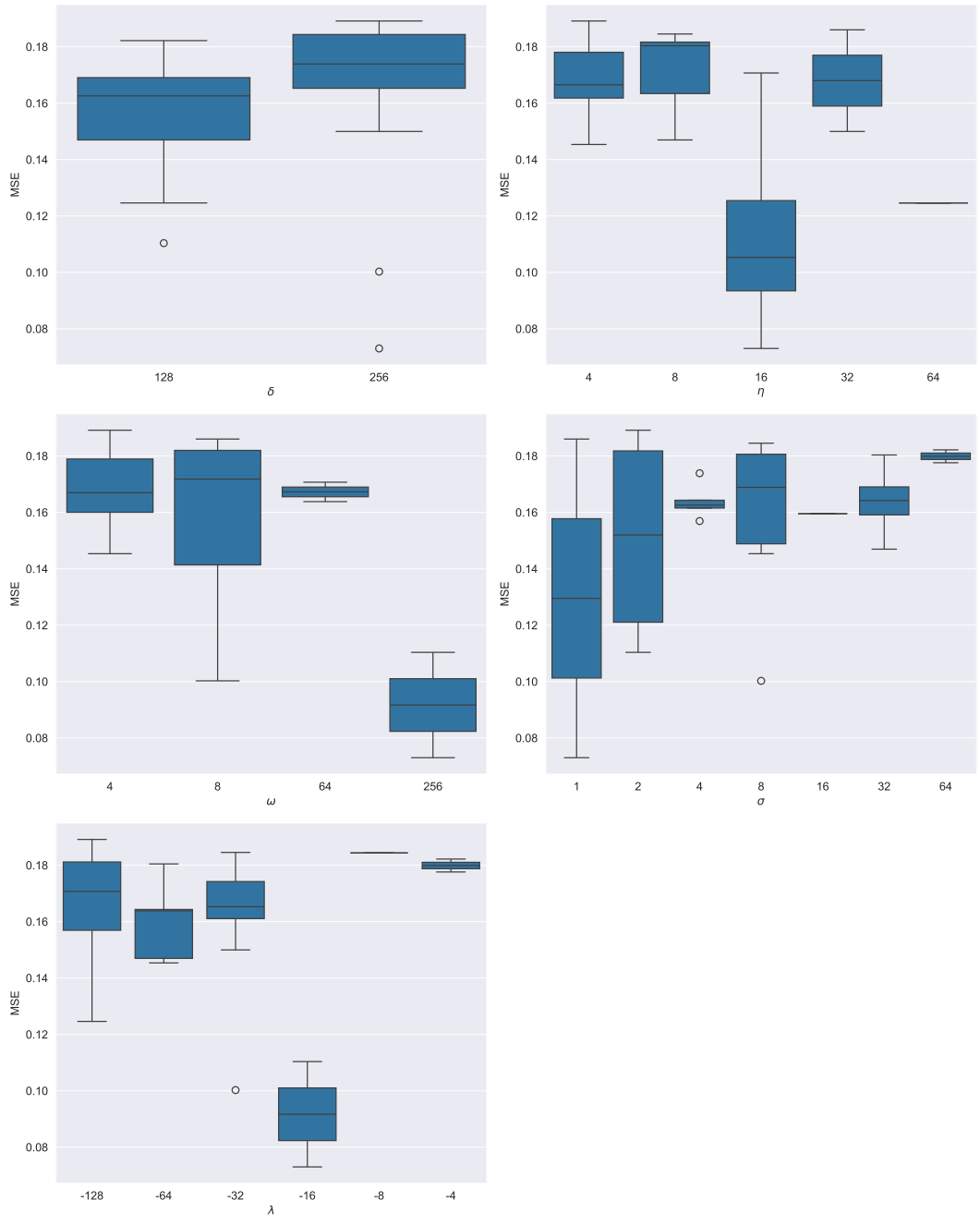


FIGURE 4.2: Box plot of hyperparameter distribution.

TABLE 4.3: Performance of a Linear Regression model on ETT

Dataset	MSE	MAE	RMSE	RRSE
ETTh1	34.592	5.556	5.882	6.828
ETTh2	102.323	9.545	10.116	2.962
ETTm1	20.284	4.458	4.504	7.499
ETTm2	99.117	9.825	9.956	11.768

We present the results of our multivariate experiments in Table 4.4, which lists the optimal hyperparameter configurations for the `alpha` and SARIMAX models identified for each dataset, along with the corresponding MSE, MAE, RMSE, and RRSE values for each configuration. The SARIMAX model achieves, on average, a 69% improvement in MSE and a 50% improvement in MAE over `alpha` on the ETT dataset without ETTh1. However, on ETTh1, `alpha` outperforms SARIMAX by 59% in MSE and 28% in MAE. Interestingly, `alpha` faces considerable challenges in modeling the complexity of ETTh2. Nevertheless, SARIMAX also shows relatively weak performance on ETTh2 compared to its results on the other ETT datasets. On the Weather dataset, the `alpha` model substantially outperforms SARIMAX by a wide margin in both error metrics. It attains an MSE of 1.83, whereas SARIMAX's MSE is 26.13—an improvement of approximately 93%. Similarly, `alpha` achieves roughly a 77% improvement in MAE over SARIMAX. These results further strengthen our assumption that `alpha` can capture the complexity of irregular, non-cyclic datasets that do not exhibit regular patterns.

TABLE 4.4: Experiment results for multivariate datasets

	`alpha`									SARIMAX						
Dataset	MSE	MAE	RMSE	RRSE	η	δ	ω	σ	λ	MSE	MAE	RMSE	RRSE	p	d	q
ETTh1	**0.07**	**0.23**	0.27	0.39	16	256	256	1	16	0.17	0.32	0.41	0.53	4	1	1
ETTh2	2.76	1.42	1.66	0.49	16	64	128	1	8	**0.56**	**0.55**	0.75	0.22	2	1	3
ETTm1	0.09	0.25	0.30	0.50	32	64	16	1	64	**0.04**	**0.16**	0.19	0.31	1	1	2
ETTm2	0.22	0.38	0.47	0.56	32	64	8	1	64	**0.06**	**0.18**	0.24	0.27	1	1	4
Weather	**1.83**	**1**	1.35	0.75	8	128	64	4	4	26.13	4.30	5.11	1.02	3	0	2

Chapter 5

Conclusion

This thesis introduces a novel two-layered framework for time series forecasting, achieving three primary milestones. First, we formulate the framework in a precise, rigorous, and abstract manner, ensuring its applicability to any models designated for the first and second layers (L_1 and L_2). Second, we implemented this framework in Python, providing a convenient API that facilitates the development, execution, and evaluation of various models within this two-layered approach. Third, our experimental results, as presented in Table 4.3, demonstrate that models such as Linear Regression can experience substantial performance improvements when applied to time series data using our framework. Specifically, the initial model, `alpha`, which leverages Linear Regression on both layers, outperforms traditional Linear Regression and the ARIMA model on irregular, non-cyclic datasets for both univariate and multivariate time series. This significant enhancement underscores the effectiveness of our two-layered approach in handling diverse and complex time series forecasting tasks. Our introduced framework and method proposes a new way of decomposing time series data in so-called *chains of subseries*, similar to patching, which shows promising results in recent models. By adjusting the hyperparameters, we explored different settings via a grid search approach.

We evaluated the `alpha` model on the Air Passenger, Weather and Electricity Transformer Temperature (ETT) datasets, some widely recognized standard datasets in time series analysis, to assess the performance in short-term predictions for both univariate and multivariate time series. The experimental results demonstrate that our framework achieves performance comparable to SARIMA on cyclical time series and even outperforms SARIMA when handling irregular time series. The experimental results also showed that only scaling the parameters to high values does not always result in better performance.

The proposed `alpha` model within this framework currently only supports a one-step ahead prediction, which makes training and predicting very costly. Future work could focus on developing a more elaborate model capable of predicting a larger horizon. The way that we patch the time series into windows $T'_i \subset T'_{i+1}$ could further be explored by creating patches that only intersect to some extent but not as a whole. The framework showed drastic improvements for linear regression on time series data. Another study on this topic could be using a classical time series model like ARIMA on L_1 or a classical neural network for L_2. Furthermore, the hyperparameter search process could be made more efficient. Currently, the grid search approach is computationally costly and limited to predefined values (e.g., $a = 1, 2, 4, 8$), potentially missing optimal parameters like $a = 3, 5, 6, 7$. Future work could explore more effective optimization techniques, such as randomized search or Bayesian optimization, to enhance performance and reduce computational overhead.

Bibliography

Barr David Diez, Christopher and Mine Çetinkaya-Rundel (2022). *Fitting a Line by Least Squares Regression*. https://stats.libretexts.org/@go/page/314. [Online; accessed 2024-01-12].

Box, George.E.P. and Gwilym M. Jenkins (1976). *Time Series Analysis: Forecasting and Control*. Holden-Day.

Breiman, Leo (1996). "Bagging Predictors". In: *Machine Learning* 24.2, pp. 123–140.

Freund, Yoav and Robert E Schapire (1997). "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences* 55.1, pp. 119–139. ISSN: 0022-0000. DOI: https://doi.org/10.1006/jcss.1997.1504. URL: https://www.sciencedirect.com/science/article/pii/S002200009791504X.

Guthrie, William F. (2020). *NIST/SEMATECH e-Handbook of Statistical Methods (NIST Handbook 151)*. en. DOI: 10.18434/M32189. URL: https://www.itl.nist.gov/div898/handbook/.

Hyndman, R. et al. (2008). *Forecasting with Exponential Smoothing: The State Space Approach*. Springer Series in Statistics. Springer Berlin Heidelberg. ISBN: 9783540719182. URL: https://books.google.ch/books?id=GSyzox8Lu9YC.

Ke, Guolin et al. (2017). "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

Kotu, Vijay and Bala Deshpande (2019). "Chapter 12 - Time Series Forecasting". In: *Data Science (Second Edition)*. Ed. by Vijay Kotu and Bala Deshpande. Second Edition. Morgan Kaufmann, pp. 395–445. ISBN: 978-0-12-814761-0. DOI: https://doi.org/10.1016/B978-0-12-814761-0.00012-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128147610000125.

Liu, Minhao et al. (2022). *SCINet: Time Series Modeling and Forecasting with Sample Convolution and Interaction*. arXiv: 2106.09305 [cs.LG]. URL: https://arxiv.org/abs/2106.09305.

Nie, Yuqi et al. (2023). *A Time Series is Worth 64 Words: Long-term Forecasting with Transformers*. arXiv: 2211.14730 [cs.LG]. URL: https://arxiv.org/abs/2211.14730.

Oreshkin, Boris N. et al. (2020). *N-BEATS: Neural basis expansion analysis for interpretable time series forecasting*. arXiv: 1905.10437 [cs.LG]. URL: https://arxiv.org/abs/1905.10437.

Oza, Nikunj C. (2005). "Ensemble Data Mining Methods". In: *Encyclopedia of Data Warehousing and Mining*. Ed. by John Wang. IGI Global, pp. 448–453. DOI: 10.4018/978-1-59140-557-3.ch085. URL: https://doi.org/10.4018/978-1-59140-557-3.ch085.

Salinas, David et al. (2020). "DeepAR: Probabilistic forecasting with autoregressive recurrent networks". In: *International Journal of Forecasting* 36.3, pp. 1181–1191. ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2019.07.001. URL: https://www.sciencedirect.com/science/article/pii/S0169207019301888.

Scott, Steven and Hal Varian (Jan. 2014). "Predicting the Present with Bayesian Structural Time Series". In: *Int. J. of Mathematical Modelling and Numerical Optimisation* 5, pp. 4 –23. DOI: 10.1504/IJMMNO.2014.059942.

Shumway, Robert H. and David S. Stoffer (2000). *Time Series Analysis and Its Applications*. Springer.

Stigler, S.M. (1986). *The History of Statistics: The Measurement of Uncertainty Before 1900*. Harvard University Press. ISBN: 9780674403413. URL: https://books.google.ch/books?id=M7yvkERHIIMC.

Stone, Henry (1961). "Approximation of curves by line segments". In: *Mathematics of Computation* 15, pp. 40–47. URL: https://api.semanticscholar.org/CorpusID:121847482.

Taylor, Sean and Benjamin Letham (Sept. 2017). "Forecasting at Scale". In: *The American Statistician* 72. DOI: 10.1080/00031305.2017.1380080.

Wang, Shiyu et al. (2024a). *TimeMixer: Decomposable Multiscale Mixing for Time Series Forecasting*. arXiv: 2405.14616 [cs.LG]. URL: https://arxiv.org/abs/2405.14616.

Wang, Yuxuan et al. (2024b). *Deep Time Series Models: A Comprehensive Survey and Benchmark*. arXiv: 2407.13278 [cs.LG]. URL: https://arxiv.org/abs/2407.13278.

Warwicker, J. A. and S. Rebennack (2021). "A Comparison of Two Mixed-Integer Linear Programs for Piecewise Linear Function Fitting". In: *INFORMS J. Comput.* 34, pp. 1042–1047. DOI: 10.1287/ijoc.2021.1114.

Witten, Ian H., Eibe Frank, and Mark A. Hall (2011). *Data Mining: Practical Machine Learning Tools and Techniques*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN: 0123748569.

Wolpert, David H. (1992). "Stacked generalization". In: *Neural Networks* 5.2, pp. 241–259. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(05)80023-1. URL: https://www.sciencedirect.com/science/article/pii/S0893608005800231.

Wu, Haixu et al. (2022). *Autoformer: Decomposition Transformers with Auto-Correlation for Long-Term Series Forecasting*. arXiv: 2106.13008 [cs.LG]. URL: https://arxiv.org/abs/2106.13008.

Wu, Haixu et al. (2023). *TimesNet: Temporal 2D-Variation Modeling for General Time Series Analysis*. arXiv: 2210.02186 [cs.LG]. URL: https://arxiv.org/abs/2210.02186.

Zeng, Ailing et al. (2022). *Are Transformers Effective for Time Series Forecasting?* arXiv: 2205.13504 [cs.AI]. URL: https://arxiv.org/abs/2205.13504.

Zhou, Haoyi et al. (2021). "Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting". In: 35.12, pp. 11106–11115.