Forecast-Driven Vehicle Routing for B2B Food Delivery using Neural Networks

MSc Thesis Applied Mathematics

Jasper Mulder

Delft University of Technology

TU Delft

Forecast-Driven Vehicle Routing for B2B Food Delivery using Neural Networks

by

Jasper Mulder

to obtain the degree of Master of Science in Applied Mathematics
at the Delft University of Technology,
to be defended publicly on Friday May 16, 2025 at 13:45.

Thesis committee

Daily supervisor TU Delft:	Dr. D. de Laat
Daily supervisor ORTEC:	R. Emmen (MSc)
Committee member:	Dr. ir. G.F. Nane
Project duration:	September, 2024 – May, 2025
Faculty:	Faculty of Electrical Engineering, Mathematics and Computer Science, Delft
Student number:	4830709

An electronic version of this thesis is available at http://repository.tudelft.nl/.

The cover features an image of a truck from the ORTEC website (cropped by author), https://ortec.com/en/solutions/b2b-delivery/vehicle-routing.

TU Delft

Abstract

This thesis explores the application of neural networks for forecasting order volumes. Additionally, this thesis presents a framework for integrating these forecasts into the route planning strategy of B2B wholesale distributors. The routing strategy involves a two-stage optimization process: provisional routes are generated by assuming the forecasts are correct, and final routes are generated using the provisional routes and actual orders after all orders are revealed.

During the week before delivery, customer orders are sorted at a cross-dock corresponding to the vehicle that will deliver to that customer. If the provisional route plan differs from the final route plan, customer orders that were already sorted need to be relocated. We want to minimize these relocations. Since actual relocations are not measurable through simulations, we define a swap as the event where a customer is scheduled on a different route than originally planned. The main goal of this study is to minimize the number of swaps between forecasted and actual routes for B2B food delivery while maintaining both feasibility and route efficiency. This is accomplished by improving forecasting and designing a framework to integrate forecasts into the route optimization process.

For route planning, we use the B2B Delivery software by ORTEC [37]. This optimizer allows you to input preplanned routes, but it is not yet able to include uncertainty. In this research, we investigate the effect of different optimizer configurations on the number of swaps and route efficiency. For implementation, we propose a two-stage optimization strategy, which, due to limitations of the optimizer, includes two rounds in the second stage. First, a full optimization is performed where customers from the provisional plan are fixed in their preplanned routes. This is followed by a light optimization round that encompasses all customers. In the light optimizer, raising the minimum estimated gain before attempting swaps had a more significant effect on reducing the number of swaps compared to increasing the number of swap attempts. Tests on forecasting qualities highlighted the importance of global metrics, such as the total number of customers and forecasted volume, on swap reduction and route efficiency.

Prior to this research, a proof of concept was conducted to forecast thousands of time series using traditional forecasting methods, which revealed two significant flaws. Firstly, the necessity to train separate forecasting models for individual time series resulted in the training of tens of thousands of models. Secondly, forecasting for new customers with limited historical data proved challenging. This motivated the adoption of neural network models as a unified solution, also capable of learning from similar patterns across multiple time series. This research tests three state-of-the-art neural network models: DeepAR from Amazon [44], Temporal Fusion Transformer (TFT) from Google [22], and N-HiTS from Nixtla [6].

All models struggled with forecasting the raw time series due to the zero-inflated nature of order data. Using a separate model to predict the occurrence of an order and another model to predict the ordered volume gave comparable or better results than the proof of concept. The neural network models also show improvements in terms of forecasting for new customers. However, the main advantage lies in the ability to maintain just two models that function across all customers, eliminating the need for separate models for each customer.

Forecasts from DeepAR gave superior results in route planning regarding consistency and efficiency, despite having the highest mean absolute error, likely due to better performance on global metrics. TFT, while achieving the highest accuracy, incurred more swaps due to volume underestimation. N-HiTS performed worst in forecasting and route planning but required significantly less training time.

Further research should focus on refining loss functions to potentially forecast zero-inflated data using a single model. Within the current framework, hierarchical forecasting should be further explored because of the importance of correctly forecasting the total number of customers and ordered volumes. The framework could be further improved by incorporating stochastic optimization elements, optimizing the initial routing decisions based on forecast uncertainties.

Preface

This thesis project has been conducted on behalf of ORTEC. ORTEC is a leading provider of optimization solutions, including route planning optimization, workforce scheduling, and supply chain design. The main objective of this case study was to integrate forecasting of customers' orders into the route-planning strategy of B2B delivery. This has been achieved. However, this work also serves a broader purpose within ORTEC. It demonstrates the feasibility of applying neural networks in a real-world supply chain context, laying the groundwork for adoption across other use cases. Additionally, this work shows that ORTEC's deterministic route optimizer can already accommodate the uncertainty inherent in forecasted orders. It also offers a clear roadmap for improving the optimizer to handle stochastic inputs.

This research helped me to develop on both a technical and personal level. On a technical level, this thesis provided the opportunity to work with various systems, including the B2B route optimizer from ORTEC. I also learned to train neural networks, navigating the many challenges and complications they can present, and gained experience in using cloud GPUs through Microsoft Azure. On a personal level, this project fueled my enthusiasm for applying operations research and data science to tackle real-world business problems. It also taught me how to scope broad and complex projects effectively.

I want to thank ORTEC for trusting me with the broad and challenging project. With countless potential research directions available, ORTEC granted me the freedom to pursue the most intriguing ones. I would also like to thank the people within ORTEC, particularly those from Center of Excellence and the B2B Delivery and Routing experts. They were always available for a quick coffee, engaged in brainstorming sessions, and offered essential support in setting up all the necessary software. Thanks to everyone who contributed to this project! Furthermore, there are a few people whose contribution I would like to highlight.

Firstly, Rogier Emmen played a major role in the success of this project and my professional growth during the nine months I spent at ORTEC. I truly appreciate your enthusiasm and the opportunity to work with you. I especially enjoyed our discussions about applying the results and methods to new use cases. Thank you!

Secondly, I would like to thank David de Laat for his scientific guidance and review of this thesis. His insights improved the readability and comprehensibility of this document and offered helpful directions to include to make the thesis scientifically engaging.

Lastly, I want to thank Juliette for her unwavering patience and support. She was always there to listen, assist, and provide feedback, even during late-night work sessions.

Jasper Mulder
Delft, May 2025

Contents

1

Introduction

1.1. Background on Case Study

1.1.1. Introduction to ORTEC

ORTEC, founded in the 1980s, is a leading provider of optimization solutions, including route planning optimization, workforce scheduling, and supply chain design. With over 1,000 employees across 13 countries, ORTEC assists more than 1,200 clients in making data-driven decisions in a constantly changing landscape [36].

1.1.2. Project Context

This thesis is conducted in collaboration with ORTEC and contributes to a project for one of their clients, a wholesale food distributor. The main objective of the project is to improve route planning for business-to-business (B2B) food deliveries from cross-docking transport centers (TC). The distributor serves a wide variety of customers, including restaurants, hotels, and healthcare institutions.

Cross-docking facilities typically have limited storage capacity. Customer orders are collected and prepared throughout the week, after which they are temporarily stored at the cross-docking terminal associated with the delivery truck assigned to that customer. A schematic overview of this process is shown in Figure 1.1.

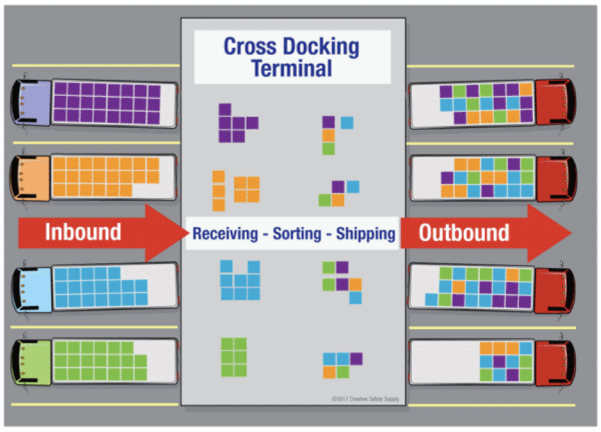


Figure 1.1: Illustrative example of a cross-docking terminal [23]. Incoming orders are sorted at the cross-dock corresponding to the vehicle that will deliver the products.

Customers of the distributor can place orders up to a certain cut-off time. This cut-off time can differ from customer to customer or even from product to product.

ORTEC already provides route planning solutions for this distributor. However, the distributor requested

to prepare delivery routes one week in advance. This requires forecasting customer orders and a framework for integrating them into the routes. To investigate the feasibility, ORTEC performed a proof of concept (PoC).

This thesis builds on that initiative by developing forecasting models for customer orders and proposing a framework for integrating these forecasts into the route planning process. It is important to note that this research focuses exclusively on outbound delivery optimization and does not address the planning of inbound shipments to the cross-docking centers.

This master's thesis has been written as part of the master Applied Mathematics at Delft University of Technology.

1.1.3. Current Route Planning Strategy

For quite a few years, the wholesale distributor has used the concept of "master routes". The planner of the TC pins down a set of routes for every day of the week that serve all customers. These master routes are determined based on the current customer base, historical order data from the last few weeks, and the experience of the planner. The routes are planned with a capacity of 130% as not every customer on the route places an order for each delivery. Depending on the size of the TC, the master routes are determined 1 to 4 times a year. Special weeks, such as Christmas, have different routes that are primarily based on the planner's experience.

> "*Currently, we look back 4 weeks in history to predict 40 weeks into the future.*"
>
> - Employee at wholesale distributor

Only the evening before delivery, all orders are definite. The planner of the TC will then make the actual routes for delivery. First, the planner uses ORTEC software to project the received orders on the predetermined master routes. However, several potential issues may arise:

1. Some routes could exceed the capacity of 100%, which means that not all orders will fit in the truck;
2. Companies that became customer after the master routes are determined can not be projected onto any of the master routes;
3. Customers that usually order on a different day are not included in the master routes for this specific day and, therefore, cannot be projected on the any of the routes as well.

Now, the planner can solve these issues by hand or use an optimizer designed by ORTEC. This optimizer uses simple local search methods to make sure all orders are on a route and that no route exceeds the capacity. With both methods it is possible that customers end up on a different route than was predetermined by the master route. When these orders have already been sorted and prepared on the cross-dock of the initial route, they now need to be relocated to the cross-dock of the new route.

The current route planning strategy is summarised in the Figure 1.2.

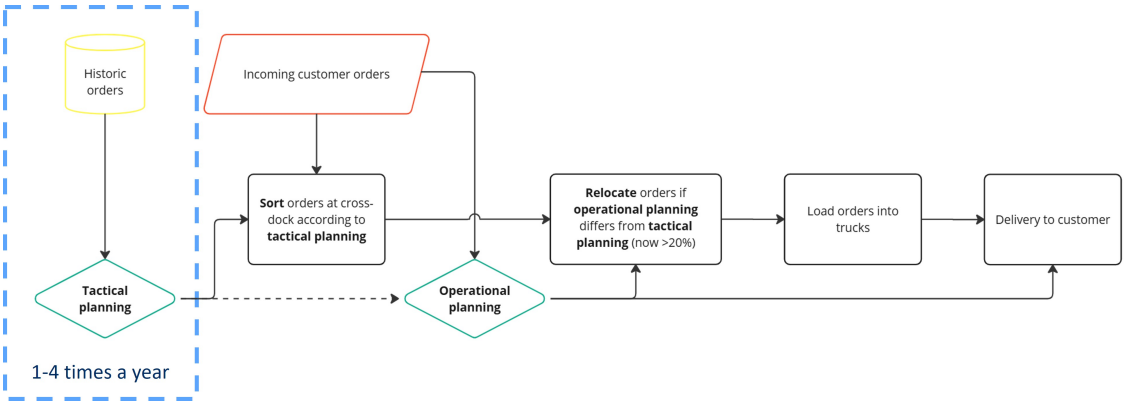


Figure 1.2: Current route planning strategy using master routes. In the current route planning strategy, more than 20% of the incoming orders that are sorted need to be relocated from one cross-dock to another due to a different final route plan.

Definition 1.1. A **relocation** refers to the event in which a customer's order needs to be moved from one cross-dock to another due to its scheduling on a different route than the predetermined one.

Such relocations are undesirable as they cause additional work just before the orders are packed into the trucks. Since there is also limited time for these relocations, it may not be possible to perform all relocations and thus drive less efficient routes. Therefore, we want to minimize the number of relocations. In the current route planning strategy, more than 20% of the incoming orders that are sorted need to be relocated from one cross-dock to another due to a different final route plan.

1.1.4. Dynamic Route Planning Strategy

With the current route planning strategy, the master routes are fixed for a relatively long time. One can imagine that these routes will work well in the first few weeks after the master routes are determined. However, as time passes, the customer base changes, and customer needs may have shifted due to a number of reasons. This results in the need for more and more swaps before loading the trucks, and it occurs that quite a few vehicles have to operate at only 60% capacity. Therefore, a more dynamic approach for the routing problem is desired.

The goal is to forecast the customer orders a week in advance and to compute optimal routes based on these forecasts. The routing software from ORTEC cannot yet handle stochastic data. It therefore just assumes the order forecasts to be true. Each customer predicted to place an order is assigned a route and, consequently, a specific cross-dock. If customers place their order for the predicted delivery day during the week before delivery, their orders can be prepared and sorted at the correct cross-dock. When the actual routes are computed on the day of delivery, it is still possible that some orders may need to be relocated. However, it is assumed that predicting routes a week in advance will result in fewer relocations than setting up master routes for a longer period of time.

The dynamic route planning strategy is summarised in Figure 1.3.

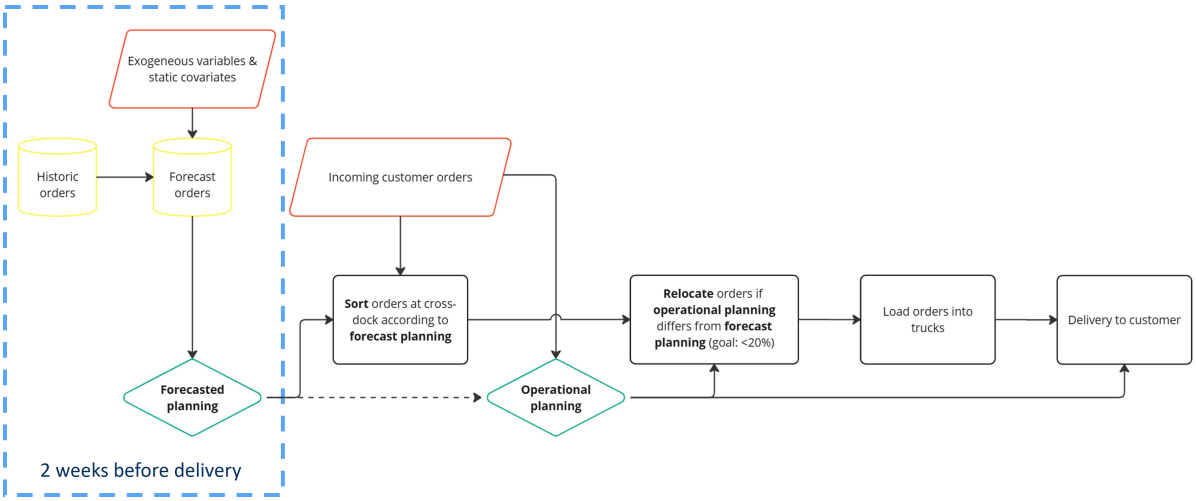


Figure 1.3: Desired dynamic route planning strategy that uses historic orders and exogenous variables to predict customer orders. These forecasts are used to generate a provisional route plan, which is handled in the same way as master routes. The goal is that, by employing a more adaptive provisional planning approach, the number of relocations can be reduced to below 20%.

There are, however, a few problems with this dynamic strategy, one of which is dealing with driver familiarity. Each customer has different instructions for delivery; for example, orders may need to be placed at back doors, in the kitchen, or in other specific locations. Not every driver can be expected to know the instructions for all customers. Additionally, customers also prefer to see a familiar face when receiving their orders. As a result, reduced driver familiarity increases delivery time and reduces customer satisfaction. With the concept of master routes, a driver responsible for a specific route is familiar with all the customers on that route. When one driver replaces another, they only need to learn

instructions for the customer on that particular master route. However, with dynamic route planning, where there are no fixed routes, it becomes challenging to ensure that drivers know the instructions for the customers they will be delivering to.

One idea would be to forecast customer orders several weeks in advance, such as six weeks prior to delivery, to allow for earlier anticipation and planning. However, it is not clear what the effect of different forecasting time frames might be on the overall efficiency of the delivery process. Given the challenging nature of this problem, particularly regarding driver familiarity, and the main goal of this project being to minimize the number of relocations of orders, the issue of driver familiarity will be excluded from the scope of this study. Additionally, the impact of different forecasting intervals will not be investigated. For the sake of simplicity, we will only consider forecasting a week before delivery.

1.1.5. Previous Work done by ORTEC

Prior to the start of this thesis, ORTEC conducted two initial proof of concept (PoC) studies to explore opportunities for improving the current forecasting and route planning processes. These PoCs were conducted separately: one focused on order forecasting, and the other on route optimization. The integration of both components into a single framework has not yet been realized.

Proof of Concept: Forecasting

In the forecasting PoC, individual product orders were aggregated into three main product streams: fresh goods, frozen products, and dry groceries. This resulted in three distinct time series per customer, each representing the volume of a specific product stream ordered on a given date. Forecasting was conducted independently for each of these time series.

The forecasting was approached using two separate models. A first model, called the delivery model, predicts whether a customer would place an order on a particular day, which is a binary classification task. A second model, called the volume model, forecasts the total order volume for the week, which is a regression task. These models are then combined: the volume forecast is only considered if the delivery model predicts that an order would occur on that day.

For predicting deliveries, logistic regression and random forest classifiers were used. For volume prediction, several regression models were evaluated, including linear regression, Prophet, Holt-Winters, and SARIMAX. The models are compared to a baseline forecast that simply copies the orders from the last week. The dataset was divided into training, validation, and test sets. All models were trained on the training set, and the optimal model combination was selected based on validation performance. Delivery models were evaluated using accuracy, defined as the percentage of correctly predicted delivery days. Volume models were assessed using Mean Absolute Error (MAE), which measures the average absolute deviation from the actual order volume.

The best-performing model combination was then evaluated on the test set and compared against the baseline approach. This naive baseline already provides a reasonable benchmark, given that many order patterns exhibit gradual changes driven by seasonality or business growth. In the PoC, the classification model obtained a 92% accuracy (baseline: 89%) and the combined model forecasted order volumes with an MAE of 43 liters (baseline: 51).

The most challenging customers to forecast were those with unpredictable ordering behavior or limited historical data. Additionally, having thousands of time series and multiple models to train per time series, the proof of concept involved training tens of thousands of time series models. This highlights the need for a unified forecasting model that can learn from the collective data of all customers. Using a single model potentially enables learning similar patterns across customers, while also providing a solution that is easier to maintain and manage.

Proof of Concept: Route Planning Optimizer

The route planning optimizer currently used by the client, as described in Section 1.1.3, is no longer actively maintained. In recent years, ORTEC has developed a new software solution for solving routing problems. This new optimizer, which will be discussed in more detail in Chapter 7, is capable of applying more advanced optimization algorithms. It is also more customizable in configuring optimization steps and incorporating the client's needs.

As part of the proof of concept (PoC), the performance of this new software was compared to the old system still in use by the wholesale distributor. Even without tailoring the new system to the client, it already demonstrated improved route efficiency and lower operational costs. However, it is important to note that this initial evaluation did not yet account for certain business rules, such as driver familiarity with specific routes, because these had not been implemented in the new system's configuration. In this thesis, no data or route plans from the old routing strategy, nor results of the PoC route planning, are available.

1.2. Problem Description

1.2.1. Research Objective

The main goal for ORTEC is to compute optimal[1] B2B delivery routes for a wholesale distributor. Because of the cross-docking system with limited storage, it is desired to know the routes in advance. The current strategy is to pin down a set of master routes only a few times a year. This thesis explores a more dynamic route planning strategy.

Demand forecasting will be used to generate routes a week in advance. After the cut-off time, when all orders are received, the actual optimal routes are computed. If customers are on different routes than the forecasted ones, it is still necessary to relocate the orders between docking stations. It is currently unknown if this dynamic strategy is better than the current situation, where more than 20% of the orders require a relocation.

For the case study of this thesis, no data is available on the actual number of relocations. Moreover, it is unclear how the wholesale distributor manages relocations. For example, if it is already known before the order cut-off time that a route is full, it is uncertain whether incoming orders are still directed to that route's corresponding cross-dock. Therefore, we need an alternative metric for the number of relocations that can be estimated through simulations.

Definition 1.2. A **swap** refers to the event in which a customer is scheduled on a different route than the originally predetermined route that customer was on.

It is important to note that swapping a customer from one route to another does not always result in a relocation of their order. For instance, if the order arrives at the transport center after the actual routes have already been generated, it can be sorted directly at the appropriate cross-dock without needing to be moved from the initial cross-dock. This is stated in Proposition 1.1.

Proposition 1.1. *The number of relocations is bounded from above by the number of swaps, i.e.,*

$$\#\textit{relocations} \leq \#\textit{swaps}.$$

However, the tightness of the bound is unknown due to insufficient knowledge on the daily operations.

The goal of this research is to evaluate the impact of the proposed framework by ORTEC on the swaps and to investigate new forecasting and route planning strategies to improve the impact, while maintaining the efficiency of routes. We will later see that there is a tradeoff to be made between route consistency and route efficiency.

One of the key challenges in forecasting is accurately predicting orders for customers with limited order history. Therefore, this research aims to improve forecasting for these customers by identifying trends and patterns that can be learned from similar customers. For route planning strategies, this research will explore different configurations of the route optimizer and their effects on both swaps and route efficiency.

[1] Of course, since the problem is NP-hard, finding the true optimal solution is not feasible. ORTEC aims to find a solution as good as possible.

1.2.2. Research Question

The central research question is formulated as follows:

> How can the **number of swaps** between forecasted and actual routes in the cross-docking warehouse system for B2B food delivery be **minimized** while **maintaining** both **feasibility** and **route efficiency**?

To answer this overarching question, the following subquestions are addressed:

- What is the impact of using ORTEC's existing order forecasts, in combination with the new B2B route planning software, on the number of swaps between forecasted and actual routes?
- How do changes in the input parameters of the B2B route optimizer affect the number of swaps and the overall efficiency of the resulting routes?
- What forecasting methods can be applied to multiple customers to learn patterns from similar customers and therefore improve the accuracy of the forecast?

Together, these questions guide the development and evaluation of a forecast-driven route planning approach.

1.2.3. Method

This section provides a high-level overview of the research method to answer the research question. A full description of the experimental setup can be found in Part II.

This thesis is structured into two main research parts, just as the two different PoC's by ORTEC. First, we will investigate the route optimizer with the integration of order forecasts. Second, we try to improve the initial order forecasts provided by ORTEC.

The main goal is to help the wholesale distributor optimize its routing. We will do this by evaluating two key metrics: the route consistency, measured in a number of swaps between a provisional route and a final route plan, and the route efficiency, measured in terms of plan costs. A schematic overview of investigating the route consistency is illustrated in Figure 1.4. First, we run the optimizer on forecasted orders to produce a provisional route plan. Then, using that provisional plan and the actual orders, we run the optimizer again to obtain a final route plan. Since the outcome of the optimizer is highly dependent on the configuration being used, we will test different configurations of this second optimizer round. The number of swaps will be determined by investigating the difference in customer allocation in the two route plans.

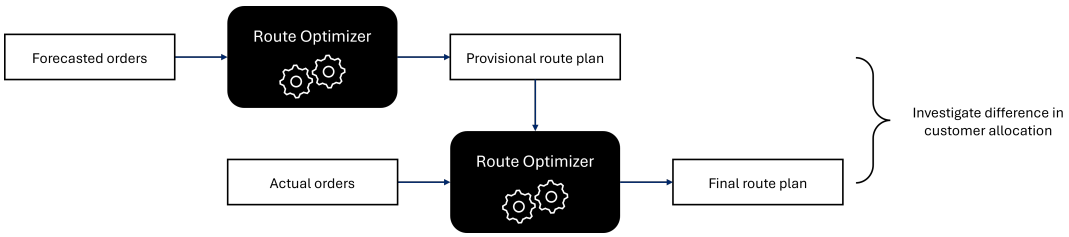


Figure 1.4: Forecasted orders are used to generate a provisional route plan using the route optimizer. Using the actual orders, along with the provisional route plan, the route optimizer generates a final route. We are interested in the difference between the provisional route plan and the final route plan in terms of customer allocations. Differences in these plans are referred to as swaps.

To investigate the route efficiency, we compare the cost of the forecast-based final plan with the cost of a plan generated solely from actual orders (no forecast input). Although this approach neither guarantees the true global optimum nor operates in a fully deterministic manner, it provides the most practical reference for assessing the additional cost from incorporating forecasts. A schematic overview of this reference for route efficiency is provided in Figure 1.5. We will repeat both these steps from Figures 1.4 and 1.5 for every optimizer configuration and forecasting method we want to test on the optimizer. More information on the workings of the route optimizer, and the configurations that are tested, can be found in Chapter 7.

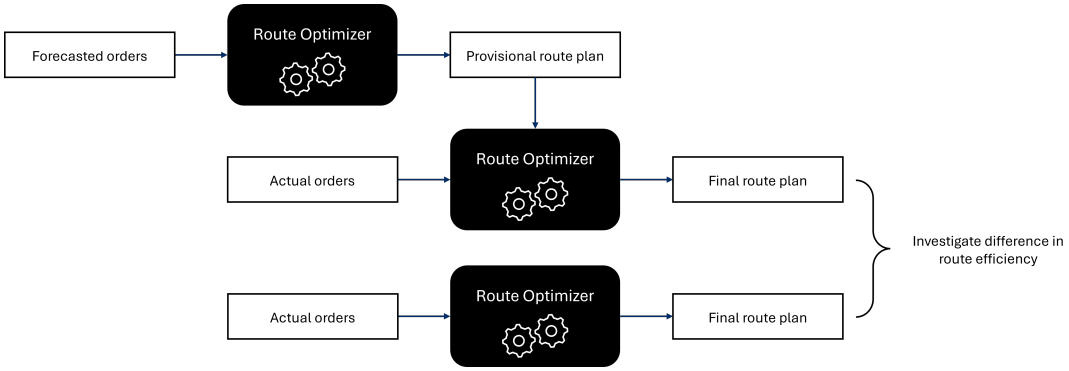


Figure 1.5: To assess the impact of using a provisional route, we also generate a final route plan based solely on actual orders. The difference in efficiency is then measured by comparing the plan costs of final route plans that either incorporate forecasted orders or rely exclusively on actual orders.

This research investigates thousands of time series[2] from the wholesale distributor over the period from 2022-03-21 to 2023-06-24. Raw orders per customer are provided at the product level and aggregated into three product groups, resulting in three distinct time series per customer. Every vehicle used for route planning has a capacity in terms of volume and weight. The focus of this research is on the volume of the orders, but can be repeated similarly on the weight. More information on the available data, filtering of the data and feature engineering can be found in Chapter 5.

Contrary to the PoC, which trained a different model per individual time series, this research employs three state-of-the-art neural network models capable of jointly forecasting all series. All models are implemented using the neuralforecast package from Nixtla, providing a uniform syntax for training the different models. To be able to equally compare results with PoC, we adopt the same train-validation-test split as done by ORTEC. More information on the implementation of the tested models, including tested hyperparameter spaces and used Python packages, can be found in Chapter 6.

Finally, we integrate the improved forecasts into the route optimizer and evaluate their impact on both swap count and route cost. Our aim of using improved forecasts is to minimize route reassignments while preserving or improving overall efficiency compared to the PoC results.

1.3. Thesis Outline

This thesis is structured as follows. In Part I, the relevant theory for the study is presented. This includes an introduction to the Vehicle Routing Problem in Chapter 2, an overview of time series analysis and traditional forecasting methods in Chapter 3, and an introduction to machine learning techniques for time series forecasting in Chapter 4.

Part II outlines the research methodology and experimental setup. It includes a description of the available data in Chapter 5, the implementation of forecasting models in Chapter 6, and a detailed explanation of the route optimization software and used configurations in Chapter 7.

The results are presented in Part III. We will start by evaluating the forecasts from ORTEC in Chapter 8, followed by assessing the impact of different optimizer configurations Chapter 9 using these forecasts. Chapter 10 presents the results of forecasting the customer orders using neural networks, and Chapter 11 examines the effect of integrating these neural network forecasts into the route planning process.

Finally, the findings are summarized and recommendations are provided in Part IV.

[2]For confidentiality reasons, the exact number of customers has been omitted. The number is available in the confidential version of this thesis within Appendix E.

Part I

Theoretical Background

2

The Vehicle Routing Problem and its Variants

The problem ORTEC solves for the wholesale distributor is known as the Vehicle Routing Problem (VRP). The VRP is one of the most studied problems in combinatorial optimization [13]. There are many different variants of the VRP. In this section, we will formalize the VRP. First, we will start with a general formulation and then discuss some variants that are relevant for this research.

The Vehicle Routing Problem is a generalization of the Traveling Salesman Problem (TSP). Instead of one salesman who has to serve every customer, the VRP considers a fleet of vehicles that can leave from one or more depots. The goal of the VRP is to determine a set of routes that meet all client requirements and operational constraints. The objective function typically aims to minimize operational costs or total distance traveled. A schematic overview of the VRP is provided in Figure 2.1.

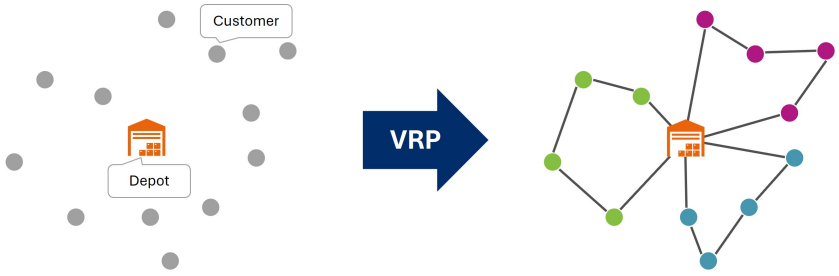


Figure 2.1: Illustrative example of a VRP with a single depot. In this scenario, three routes depart from the depot, which can be serviced by either a single vehicle or three separate vehicles.

The formulation of the VRP, as will be described in this section, is used a lot in literature. However, it is important to note that this formulation is a simplified version compared to a real-world scenario. The mathematical formulation outlined below will not be used explicitly in later chapters; it is included solely to provide background on the underlying mathematical problem being solved. More practical information on how ORTEC solves the VRP will be discussed in Chapter 7.

2.1. Two-index vehicle flow formulation

We will start with the vehicle flow formulation of the capacitated Vehicle Routing Problem (CVRP), as presented by Toth and Vigo [49]. In this formulation, all customers represent deliveries, and the demands are deterministic and known in advance. We consider one depot and assume the vehicles are identical, with only constraints imposed on the capacity. The objective is to determine a set of minimal-cost routes that satisfy all the requirements.

Let $\mathcal{C} = \{1, \ldots, n\}$ represent the set of customers. Each customer must be served, meaning they must be visited exactly once. Each customer i has a demand d_i with $d_i > 0$ for $i \in \mathcal{C}$. The objective is to design routes for a fleet of K vehicles available at a single depot. Each vehicle has a maximum capacity C, and every route must start and end at the depot. To ensure feasibility, we assume $d_i \leq C$ for all $i \in \mathcal{C}$. Given a set $S \subseteq \mathcal{C}$, let $d(S) = \sum_{i \in S} d_i$.

The problem can be modeled on a graph $G = (V, A)$. The set of vertices V is defined as $V = \mathcal{C} \cup 0$, where vertex 0 is the depot. Consequently, all routes start and end at vertex 0. The set of arcs, A, contains arcs (i, j) for each pair $i, j \in V$. The cost of traversing arc $(i, j) \in A$ is denoted by c_{ij}. If $c_{ij} = c_{ji}$, the problem is called the symmetric CVRP, and the set of arcs A may be replaced by a set of undirected edges E. Given a set $S \subseteq \mathcal{C}$, we denote by $r(S)$ the minimum number of vehicles required to serve all customers in S. Note that this is equal to the optimal solution of the Bin Packing Problem with item set S. A trivial lower bound for this problem would be $\lceil d(S)/C \rceil$.

We introduce the following decision variable:

$$x_{ij} = \begin{cases} 1, & \text{if a vehicle travels from vertex } i \text{ to vertex } j \text{ directly,} \\ 0, & \text{otherwise.} \end{cases} \tag{2.1}$$

Then, the optimization problem for the VRP, as described above, can be formulated as

$$\min \quad \sum_{i \in N} \sum_{j \in N} c_{ij} x_{ij} \tag{2.2}$$

$$s.t. \quad \sum_{i \in V} x_{ij} = 1 \qquad \forall j \in V \setminus \{0\} \tag{2.3}$$

$$\sum_{j \in V} x_{ij} = 1 \qquad \forall i \in V \setminus \{0\} \tag{2.4}$$

$$\sum_{i \in V} x_{i0} = K \tag{2.5}$$

$$\sum_{i \in V} x_{0j} = K \tag{2.6}$$

$$\sum_{i \notin S} \sum_{j \in S} x_{ij} \geq r(S) \qquad \forall S \subseteq V \setminus \{0\},\ S \neq \emptyset \tag{2.7}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in V \tag{2.8}$$

Here, constraints (2.3) and (2.4) ensure that exactly one route enters and leaves the vertex associated with a customer. Constrains (2.5) and (2.6) make sure that all vehicles enter and leave the depot.

Constraints (2.7), called the capacity-cut constraints, guarantees both the connectivity of the solution and the vehicle capacity requirements. An alternative formulation of the capacity-cut constraints can be obtained through subtour elimination, analogous to the subtour elimination constraints used in the Traveling Salesman Problem. The generalized subtour elimination constraints makes sure that at least $r(s)$ arcs leave each customer set S, and is given by

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - r(s) \qquad \forall S \subseteq V \setminus \{0\}, S \neq \emptyset. \tag{2.9}$$

Both sets of constraints (2.7) and (2.9) have a cardinality that grows exponentially with n. Consequently, directly solving the linear programming relaxation of this problem becomes practically infeasible. This highlights the need for relaxation techniques and heuristic approaches to obtain feasible solutions in a reasonable timeframe.

It is not hard to show that solving the VRP is NP-hard. Since the TSP is NP-hard [11], and the VRP strictly generalizes it, the VRP is NP-hard as well. Simply allow only one vehicle, and ignore capacities, to reduce the VRP to the TSP.

2.2. Extensions of the Vehicle Routing Problem

ORTEC must accommodate various extensions to the routing problem, which can vary per client. This section presents the extensions involving a heterogeneous fleet, time-constrained routes, and time windows, as these are common across nearly all clients. We will continue with the formulation from Toth and Vigo [49]. Additional extensions to the VRP include multiple depots, pickup-and-delivery requests, driver work-hour restrictions, driver familiarity, and more. While each of these variants introduces its own unique modeling and computational challenges, we will not explore them further in this thesis.

2.2.1. Heterogeneous and Excess Vehicles

First, consider the scenario when the number of available vehicles, K, is greater than the minimal amount of vehicles required, $K_{min} = r(\mathcal{C})$. Then, it may be possible to leave some vehicles unused. In this case, some sort of minimization of used vehicles/routes is often added to the objective function, like fixed costs for the use of a vehicle or an additional objective requiring the minimization of used vehicles. Also, the constraints (2.5) and (2.6) need to be replaced by

$$\sum_{i \in V} x_{i0} \leq K,$$

and

$$\sum_{j \in V} x_{0j} = \sum_{i \in V} x_{i0}.$$

Instead of changing the constraints it is also possible to first compute K_{min}, by solving the Bin Packing Problem, and then define $K = K_{min}$.

Another frequently considered extension of the VRP is the case where the available vehicles are different. For example, the vehicles could have different capacities C_k, for $k = 1, \ldots, K$.

With the model becoming more complex, the two-index may not be as suitable anymore. For example, a solution $(x_{ij})_{i,j \in V}$ to problem (2.2)-(2.8) only tells you if arc $(i, j) \in A$ is included in a route. It does not directly tell you which vehicle travels which route. A possible way to overcome this is by explicitly indicating which vehicles traverse which arcs. We introduce the following decision variables:

$$x_{ijk} = \begin{cases} 1, & \text{if vehicle } k \text{ traverses arc } (i,j) \in A, \\ 0, & \text{otherwise,} \end{cases} \tag{2.10}$$

$$y_{ik} = \begin{cases} 1, & \text{if customer } i \text{ is served by vehicle } k, \\ 0, & \text{otherwise.} \end{cases} \tag{2.11}$$

Now, the three-index vehicle flow formulation of the CVRP, with heterogeneous fleet, is given by

$$\min \quad \sum_{i\in V}\sum_{j\in V} c_{ij} \sum_{k=1}^{K} x_{ijk}, \tag{2.12}$$

$$s.t. \quad \sum_{k=1}^{K} y_{ik} = 1 \qquad \forall i \in V \setminus \{0\}, \tag{2.13}$$

$$\sum_{k=1}^{K} y_{0k} = K, \tag{2.14}$$

$$\sum_{j\in V} x_{ijk} = \sum_{j\in V} x_{jik} = y_{ik} \qquad \forall i \in V,\ k = 1,\dots,K, \tag{2.15}$$

$$\sum_{i\in V} d_i y_{ik} \le C_k \qquad \forall k = 1,\dots,K, \tag{2.16}$$

$$\sum_{i\in S}\sum_{j\in S} x_{ijk} \le |S| - 1 \qquad \forall S \subseteq V \setminus \{0\},\ |S| \ge 2,\ k = 1,\dots,K, \tag{2.17}$$

$$y_{ik} \in \{0,1\} \qquad \forall i \in V,\ k = 1,\dots,K, \tag{2.18}$$

$$x_{ijk} \in \{0,1\} \qquad \forall i,j \in V,\ k = 1,\dots,K. \tag{2.19}$$

$$\tag{2.20}$$

Here, constraints (2.13)-(2.15) ensure that each customer is visited exactly once, that K vehicles leave the depot, and that the same vehicle enters and leaves a given vertex, respectively. Constraints (2.16) are the capacity restrictions for each vehicle. The connectivity is guaranteed by the subtour elimination constraints (2.17), which impose that for each vehicle k at least 1 arc leaves each vertex set S visited by k. Note that this three-index formulation is a generalization of the two-index formulation as we can always take $x_{ij} = \sum_k x_{ijk}$ and $y_i = \sum_k y_{ik}$.

In this case study, the wholesale distributor delivers products that require transportation in a refrigerated compartment (freezer), while other products do not. This requires yet another generalization to the problem where we now need to manage a multi-compartment fleet. Ostermeier et al. [39] introduced a formulation to this problem, and will be presented here with some slight variations. First, we introduce a set of product types $p \in P$. Now, we can specify the capacity of each vehicle per compartment, so every vehicle k has capacity C_{kp} for every $p \in P$. Also the demand per customer can be specified per product type, d_{ip}. We can use the same decision variable for x_{ijk} as in (2.10), and change (2.11) slightly to

$$y_{ikp} = \begin{cases} 1, & \text{if product type } p \text{ is delivered to customer } i \text{ by vehicle } k, \\ 0, & \text{otherwise.} \end{cases} \tag{2.21}$$

Now, with the included index for product type formulation (2.12)-(2.19) becomes

$$\min \quad \sum_{i\in V}\sum_{j\in V} c_{ij} \sum_{k=1}^{K} x_{ijk}, \tag{2.22}$$

$$s.t. \quad \sum_{k=1}^{K} y_{ikp} = 1 \qquad \forall i \in V \setminus \{0\},\ p \in P, \tag{2.23}$$

$$\sum_{k=1}^{K} y_{0kp} = K \qquad \forall p \in P, \tag{2.24}$$

$$\sum_{j\in V} x_{ijk} = \sum_{j\in V} x_{jik} = y_{ikp} \qquad \forall i \in V,\ k = 1,\dots,K,\ p \in P, \tag{2.25}$$

$$\sum_{i} d_{ip} y_{ikp} \le C_{kp} \qquad \forall k = 1,\dots,K,\ p \in P, \tag{2.26}$$

$$\sum_{i\in S}\sum_{j\in S} x_{ijk} \le |S| - 1 \qquad \forall S \subseteq V \setminus \{0\},\ |S| \ge 2,\ k = 1,\dots,K, \tag{2.27}$$

$$y_{ik} \in \{0,1\} \qquad \forall i \in V,\ k = 1,\dots,K, \tag{2.28}$$

$$x_{ijk} \in \{0,1\} \qquad \forall i,j \in V,\ k = 1,\dots,K. \tag{2.29}$$

(2.30)

The above assumes that the capacities of vehicles per product type, C_{kp}, are known. If the compartments are adjustable and only the total capacity of a vehicle is given we may introduce the constraint

$$\sum_{p} C_{kp} \le C_k \qquad \forall k.$$

2.2.2. Time-Constrained Routes

Companies often require conditions on the total time of a route, such as maximum driving or working times for a driver. To include this into the model, we need to assign non-negative times t_{ij} to each arc $(i,j) \in A$. Then, the total time of a route cannot exceed the maximum time T. If the fleet is heterogeneous, then the maximum route lengths are T_k, $k = 1, ..., K$. This can be implemented in the three-index formulation using the constraint

$$\sum_{i\in V}\sum_{j\in V} t_{ij} x_{ijk} \le T_k \qquad \forall k = 1,\dots,K. \tag{2.31}$$

Additionally, a service time s_i may be associated with each customer, or depot, i, denoting a time period for which a vehicle must stop at a vertex. These service times may also be integrated into the travel times of the arcs

$$t'_{ij} = \frac{s_i}{2} + t_{ij} + \frac{s_j}{2},$$

where t_{ij} is the travel time of arc $(i,j) \in A$ without the service times and s_i, s_j are the service times of $i,j \in V$ respectively.

This extension is called the Time-Constrained VRP, sometimes also called the Distance-Constrained VRP with analogous conditions on the total length of a route. Note that often the cost of traversing an arc corresponds more or less to the time it takes to traverse that arc. Hence, the driving time may already be, though indirectly, minimized through the objective function.

2.2.3. VRP with Time Windows

Often, customers want to specify a time window in which they want their order to be delivered. A restaurant, for example, may want to have the orders delivered at the start of the day before its own customers arrive.

The VRP with Time Windows (VRPTW) is the extension of the CVRP in which each customer i is associated with a time window $[a_i, b_i]$ in which it needs to be serviced. To accommodate for this, we add time variables w_{ik} for $i \in V$ and $k \in K$, specifying the start of service at customer i when serviced by vehicle k. To ensure feasibility, the start of service time for subsequent customers i and j in a vehicle must include at least the service time s_i and travel time t_{ij} between them, i.e.,

$$x_{ijk}(w_{ik} + s_i + t_{ij} - w_{jk}) \leq 0 \qquad \forall i, j \in V, k = 1, \ldots, K. \tag{2.32}$$

Then, for customer i to be serviced within the time window $[a_i, b_i]$, we require that

$$a_i \sum_{j \in V} x_{ijk} \leq w_{ik} \leq b_i \sum_{j \in V} x_{ijk} \qquad \forall i \in V, k = 1, \ldots, K. \tag{2.33}$$

Since x_{ijk} is binary, Equation (2.32) can be linearized as

$$w_{ik} + s_i + t_{ij} - w_{jk} \leq (1 - x_{ijk})\, M_{ij} \qquad \forall i, j \in V, k = 1, \ldots, K,$$

where M_{ij} are large constants. Toth and Vigo take $M_{ij} = \max\{b_i + s_i + t_{ij}, 0\}$ [49].

If the vehicles have an earliest possible departure time E and latest possible arrival time L at the depot, then we can add the constraint:

$$E \leq w_{ik} \leq L \qquad \forall i \in V, k = 1, \ldots, K. \tag{2.34}$$

2.3. VRP with Uncertain Customers and Demand

In this research, we have to deal with uncertain customers and demand. The challenge lies in the fact that ORTEC employs a deterministic solver, which complicates managing uncertainty. Managing demand uncertainty is crucial, as it directly impacts the capacity constraints of vehicles and the efficiency of routing decisions. Vehicles can, for example, become overloaded or underutilized. In literature, this problem is known as the Vehicle Routing Problem with Stochastic Customers and Demands (VRP-SCD). Gendreau, Laporte, and Séguin (1995) described this problem as ”exceedingly difficult.” We assume that customers place an order with probability p_i and that their demand, d_i, is uncertain under an unknown distribution.

There are two main solution approaches from stochastic programming that address this problem [7]. Chance Constrained Programming tackles the problem by ensuring constraints are satisfied with a certain probability. For instance, a failure threshold α can be set, such that routes fail with a probability no more than α. For the capacity constraints, one might use:

$$\mathbb{P}\left(\sum_i d_i y_{ik} \leq C_k\right) > 1 - \alpha \qquad \forall k = 1, \ldots, K. \tag{2.35}$$

Alternatively, the problem can be addressed as a two-stage problem. In this framework, the process is not as straightforward as making decisions first, then observing the uncertainty and computing the costs: [41]

$$\underset{\text{decision}}{x} \rightarrow \underset{\text{uncertain parameter observed}}{z}.$$

Instead, once the uncertainty is observed, we can change some of the routing decisions. The simplest case is a two-stage sequence is given by:

$$\underset{\text{decision}}{x} \rightarrow \underset{\text{uncertain parameter observed}}{z} \rightarrow \underset{\text{decision}}{y}.$$

Ideally, x is optimized by considering all possible outcomes of z and the corresponding optimal decisions y that would respond to z. Again, optimizing this is difficult given the deterministic solver. Cordeau et al. [7] refers to this approach as Stochastic Programming with Recourse. The recourse policy, decision y, is a modeling choice that depends on the client's preferences and requirements.

Stochastic programming often requires some knowledge of the probability distributions, while robust optimization focuses on extreme scenarios using uncertainty sets. In practice, this information is often unavailable. Additionally, stochastic optimization can face practical challenges due to its complexity and computational demands. Despite being extensively researched in the literature, stochastic optimization is frequently not implemented in the industry [42].

Given these challenges, parametric optimization emerges as a viable alternative [42]. Parametric optimization involves adjusting parameters in the optimization model to account for uncertainties indirectly. A deterministic solver is sufficient for simulating and solving problems under this approach. For uncertain demand, a parameterized capacity constraint might be introduced, such as:

$$\sum_i d_i y_{ik} \leq C_k - \theta_k \qquad \forall k = 1, \ldots, K, \tag{2.36}$$

where θ_k is a parameter to plan empty space for each vehicle k. We could also impose an uncertainty factor θ_i per customer i:

$$\sum_i (d_i + \theta_i) y_{ik} \leq C_k \qquad \forall k = 1, \ldots, K. \tag{2.37}$$

There is extensive literature on stochastic optimization, offering numerous ideas to enhance route planning in this case study. However, the goal of this thesis is to utilize ORTEC's solver and explore its effectiveness in handling uncertainty. We will therefore mainly focus on simulating various scenarios using a two-stage solution approach, as we have seen in Figure 1.4. Different configurations of the route optimizer define the recourse strategies.

3

Time Series and Traditional Forecasting Methods

In this chapter, we will define the concept of time series and outline the general framework for forecasting them. Understanding this foundation is also essential when applying more advanced forecasting methods later. Therefore, it is particularly recommended that the reader pays close attention to Section 3.2. Additionally, we will introduce some traditional statistical methods for forecasting time series. As ORTEC has already implemented these methods to this customer case, this research will not focus on them. However, they will still be presented here for context.

3.1. Time Series and its Characteristics

Time series analysis is the area of statistics and data science that focuses on analyzing data points indexed by time. Examples of time series are daily stock market prices, hourly weather data, and annual sales figures. Observing and analyzing these series helps to understand the underlying processes that generated the data. This section is mostly based on [46].

3.1.1. Definition and Decomposition of Time Series

A time series is essentially a sequence of random variables indexed by time. Each data point has a distribution function associated with it. This probability distribution may depend on previous values of the series, seasonal effects, or external influences, but more on that later. The stochastic process can be described by the joint distribution function of all random variables in the stochastic process.

Definition 3.1. A **time series** is a double infinite sequence $\{\ldots, x_{-2}, x_{-1}, x_0, x_1, x_2, \ldots\}$ of random variables indexed by time. A finite collection of observed values from the stochastic process is called a **realization**.

Definition 3.2. A time series is called **causal** if it only depends on past values.

The behavior of a time series can often be characterized by a trend, seasonality, and an error term. The trend component represents the long-term progression of the series, which does not have to be linear. The seasonal component captures periodic changes that occur at regular intervals, such as daily, monthly, or yearly cycles. Finally, the error term accounts for the random noise in the data that cannot be explained by the trend or seasonal patterns.

Definition 3.3. A time series x_t can be decomposed **additively** as follows:

$$x_t = T_t + S_t + \varepsilon_t,$$

where T_t represents the trend component, S_t denotes the seasonal component, and ε_t is the error term. Likewise, the time series x_t can also be expressed **multiplicatively** as:

$$x_t = T_t \cdot S_t \cdot (1 + \varepsilon_t),$$

where T_t, S_t, and ε_t still represent the trend, seasonality, and error, but their interaction in the multiplicative model defines the components differently than in the additive model.

3.1.2. Measures of Dependence in Time Series

For a time series, we can analyze the relationships between observations at different time points. The following statistical measures are often used to quantify these relationships. Firstly, the mean process captures changes in the average value of the time series over time. If, for example, a time series has a trend component, it will be reflected in the mean process.

Definition 3.4. The **mean process** of time series x_t is defined by expected value over the time series

$$\mu_x(t) = \mathbb{E}[x_t], \tag{3.1}$$

provided it exists.

The auto-correlation measures the degree to which current values of the series are related to past values, indicating the presence of dependencies over time.

Definition 3.5. The **auto-correlation** of time series x_t between times t and s is defined by[1]

$$\gamma_x(t, s) = \mathbb{E}[x_t x_s], \tag{3.2}$$

which is the correlation between the values of the series at different times.

Finally, the auto-covariance measures how deviations from the mean at one time point relate to deviations at another time point.

Definition 3.6. The **auto-covariance** of time series x_t, with mean process $\mu_x(t)$, is defined by[1]

$$c_x(t, s) = \mathbb{E}[(x_t - \mu_x(t))(x_s - \mu_x(s))]. \tag{3.3}$$

3.1.3. Stationarity of Time Series

For some time series models, it is required for the time series to be independent of time. Here, independence of time means that if we observe a time series now, we would observe the same statistical properties if we started the observations one hour later. For stochastic processes, we refer to time invariance as stationarity.

Definition 3.7. A time series x_t is **strictly stationary** if the distribution vector $(x_{t_1}, x_{t_2}, \ldots, x_{t_k})$ is equal to $(x_{t_1+h}, x_{t_2+h}, \ldots, x_{t_k+h})$ for every $h, t_1, \ldots, t_k \in \mathbb{R}$ and every $k \in \mathbb{N}$.

Since strict stationarity is quite a hard requirement and often unnecessary for practical applications, it is typically sufficient to consider weak stationarity.

Definition 3.8. A time series x_t is **weakly stationary** (or wide sense stationary) if the following conditions hold:

1. The mean process $\mu_x(t)$ is finite and constant over time.
2. The auto-correlation function $\gamma(t, s)$ only depends on the time difference (lag) between the two points, i.e. $\gamma(t, s) = \gamma(t - s, 0)$.
3. The variance $c_x(t, t)$ is finite for all t.

For a weakly stationary process, we often write the autocorrelation function only as a function of the lag τ, so $\gamma_x(t, t + \tau) = \gamma_x(\tau)$. Then condition (3) is equivalent to $c_x(0) < \infty$.

If a time series has trend or seasonal components, it is obviously not weakly stationary. This issue of non-stationarity can be solved by differencing. If a time series, for example, has a linear trend, it can be removed by taking the difference between observations at consecutive time points.

[1] To be precise, the second term in the expectation should be the complex conjugate. However, this can be omitted for real-valued processes.

Definition 3.9. The **difference operator** on a time series x_t, denoted by Δ, is defined by

$$\Delta x_t = x_t - x_{t-1}. \tag{3.4}$$

For ease of use, we also introduce an operator that allows us to reference previous time points directly.

Definition 3.10. The **backshift operator** on a time series x_t, denoted by B, is defined by

$$Bx_t = x_{t-1}. \tag{3.5}$$

We can now also write the difference operator as $\Delta = (1 - B)$. The backshift operator also extends to higher powers, e.g. $B_2 x_t = B(Bx_t) = Bx_{t-1} = x_{t-2}$.

Sometimes, a first-order difference on the time series is insufficient to make it stationary. It may be necessary to take a second or even higher order difference.

Definition 3.11. The **difference of order** d on time series x_t is defined as

$$\Delta^d x_t = (1 - B)^d x_t. \tag{3.6}$$

If a time series contains seasonal patterns, it is also possible to take the difference between observations corresponding to the same time in a season. This helps to remove the seasonal component from the time series.

Definition 3.12. The **seasonal difference** with seasonality m on time series x_t is defined as

$$\Delta_m x_t = (1 - B^m)x_t = x_t - x_{t-m}. \tag{3.7}$$

Determining which differences to apply for achieving stationarity is not always straightforward and can involve a degree of subjectivity.

3.2. Forecasting of Time Series

3.2.1. Types of Input Data

When forecasting time series, we are interested in predicting how the sequence of observed values will continue in the future. The easiest way to predict the future is to simply extrapolate the trend and seasonal patterns based on only the historical values of the variable of interest. Examples of such methods are exponential smoothening and ARIMA models, which will be discussed in Sections 3.4 and 3.5.

It is also possible to enhance time series models by incorporating additional exogenous information, often referred to as covariates or features. The aim is not only to learn from the historical data of the target variable but also to include relevant external factors such as the day of the week, holidays, or temperature data, which may also influence the target variable. Examples of such approaches include regression models, introduced in Section 3.3.1, and the SARIMAX extension of ARIMA models.

In this thesis, we will consider three types of covariates: static, future, and past covariates.

Definition 3.13. Static covariates are variables that do not change over time. Examples are product information or geographical features.

Definition 3.14. Past covariates are variables that have been observed in the past and are yet unknown for future time periods. Examples are temperature readings or historical sales data.

Definition 3.15. Future covariates are variables that are known for the future time periods. Examples are days of the week, holidays, or upcoming promotional events.

Different forecasting models handle covariates differently, and not all models can handle every type of covariate.

3.2.2. Training and Validation of Time Series Model

Training a time series model is done by fitting the model to the time series of interest. This can be achieved by estimating the model parameters such that a predefined error function is minimized. Alternatively, one could maximize the likelihood function of a parameter.

Some frequently used error functions are the mean squared error, mean absolute error, and mean error. Given target variable y, with true values y_i and predicted values $\hat{y}_i$ for $i = 1, \ldots, n$, we define the error functions below.

Definition 3.16. The **mean squared error** (MSE) is defined as the average of squared errors, i.e.

$$\text{MSE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2. \tag{3.8}$$

Definition 3.17. The **mean absolute error** (MAE) is defined as the average of absolute errors, i.e.

$$\text{MAE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|. \tag{3.9}$$

Definition 3.18. The **mean error** (ME) is defined as the average of errors, i.e.

$$\text{ME}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i). \tag{3.10}$$

These error functions can be used in different situations. The MSE and MAE both minimize all errors, whether positive or negative. However, the MSE emphasizes larger deviations due to squaring the errors. The ME can be used in cases where negative errors may cancel out positive errors.

When predicting binary variables, we can use different loss functions designed for binary classification. If the predictions are also binary, we will use the accuracy to measure the model's performance.

Definition 3.19. The **accuracy** of a binary forecast $\hat{y}$ for binary variable y is given by

$$\text{acc}(y, \hat{y}) = \frac{\#\text{correct predictions}}{\#\text{total number of predictions}} = 1 - \text{MAE}(y, \hat{y}). \tag{3.11}$$

To evaluate the performance of a time series model, we want to know how it will perform on future, unseen data. Not on historic data. Testing on data that is unknown to the model is referred to as out-of-sample testing. This is typically achieved by splitting the available dataset into a training set and a testing set, as illustrated in Figure 3.1. The parameters of the model are then selected such that the chosen error function is minimized over the training data. Subsequently, the model's performance is assessed based on the prediction errors in the testing set.

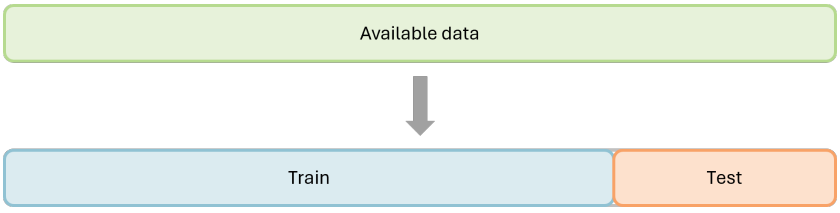


Figure 3.1: Train-test split of the available time series dataset. The train set is used to configure the model parameters such that the model best fits the training data. The test set is used to evaluate the model's performance on out-of-sample data.

3.2.3. Cross-validation on Time Series Models

Cross-validation is a well-known technique used in machine learning. The idea is to better capture the model's out-of-sample performance by applying the train-test split principle multiple times on the same dataset. The dataset is divided into K randomly generated folds. Then, for every fold, the model is trained on all data except the fold and tested on the fold. The final metric of the model is the average value of the metrics over all folds.

However, one can easily see that it is not possible to generate random folds for time series data because of its sequential nature. It makes no sense to use future observations to predict the past. What can be used for time series data is a rolling window of test folds. For the first iteration, the model is trained on a small part of the available data. Using this model, the next few time steps are predicted and compared to the test set, the first fold. For the next iteration, this first fold is included in the training set on which the model updates its parameters. Using the updated model, the next time steps are predicted and compared to the second fold. This continues as long as there is available data left. The process of cross-validation on time series is illustrated in Figure 3.2.

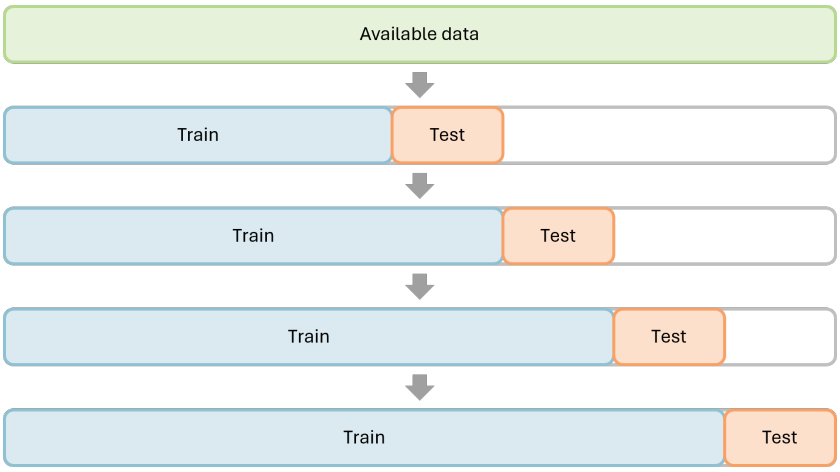


Figure 3.2: Cross-validation for time series using a rolling window of test folds. The model is trained for each fold on an increasing size of training dataset. The performance of the model is based on the average performance over all test folds.

3.2.4. Hyperparameter Tuning and Model Selection

During training, we aim to find the optimal parameters such that the model best fits the data. However, there are also parameters we can tune before the training of a model. These parameters are called hyperparameters. Examples of hyperparameters can be the dimensions of the model, the number of features to include, or the number of time steps to look back when making a prediction.

Definition 3.20. Hyperparameters are parameters of a model to be set prior to the training process, in contrast to parameters that the model learns from data.

The hyperparameter settings of a model can have a significant impact on its forecasting quality. Therefore, it is often necessary to tune the hyperparameters as well. This can be achieved by training the model with different hyperparameter settings, resulting in distinct forecasts for every configuration. The best setting can be chosen based on an out-of-sample set. We call this set the validation set. Finally, we want to test the best settings on another out-of-sample set, called the test set. This additional out-of-sample step is necessary to ensure the chosen configuration still works well on unseen data and not just the validation dataset. The train-validation-test split of available data is visualized in Figure 3.3.

The train-validation-test split can also be used for model selection. Then, all models are tested and compared using the validation set. The model that scores the best is chosen as the final model. This model can then be tested again on a new out-of-sample test set.

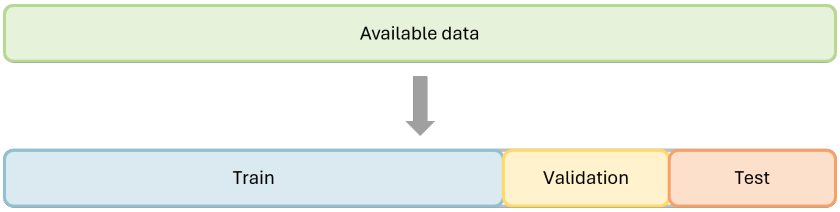


Figure 3.3: Train-validation-test split of available time series dataset. The train set is used to configure the model parameters for multiple models or hyperparameter settings. The validation set is used to find the best model or hyperparameter settings. The test set is used to evaluate the final model's performance on out-of-sample data.

In classical statistics, model complexity is often considered as well in choosing the best model. One would prefer a model with fewer parameters over a more complex model if they hold almost similar results. To quantify the model fit, we could use the likelihood function, which determines the probability of observing the given data under the specified model.

Definition 3.21. Given the density function $f(x \mid \theta)$ for x given a model with parameters θ, the **likelihood** of this model is defined as

$$L(\theta) = L(\theta \mid x_1, \ldots, x_n) = f(x_1, \ldots, x_n \mid \theta). \tag{3.12}$$

It is crucial to understand that the likelihood does not give the probability of the data itself, but rather the probability of observing data x under the assumption that the model with parameters θ is accurate.

Two often used criteria that balance model fit and complexity are the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC).

Definition 3.22. The Akaike's Information Criterion (**AIC**) for a model with k parameters is given by

$$\text{AIC} = 2k - 2\ln L, \tag{3.13}$$

where L is the maximum likelihood of the model.

Definition 3.23. The Bayesian Information Criterion (**BIC**) for a model with k parameters is given by

$$\text{BIC} = k\ln n - 2\ln L, \tag{3.14}$$

where L is the maximum likelihood of the model.

These criteria generally favor models with fewer parameters and less complexity, helping to prevent overfitting. In more advanced neural network models, which we will see in Chapter 4, these criterions are not used as models often have thousands or milions of parameters.

3.3. Regression Models

3.3.1. Linear Regression

In the simplest case, we may say that the time series of interest, y_t, is linearly dependent on another known time series x_t, called the predictor variable. The time series y_t is then given by

$$y_t = \beta_0 + \beta_1 x_t + \varepsilon_t, \tag{3.15}$$

where parameters β_0, β_1 denote the intercept and the slope of the linear line, and ε_t is the error or noise process.

Now, to forecast the series y_t, we may generate a linear fit to the past data.

An often used method for estimating these parameters is the least squares method. This method provides a way of finding the parameters in such a way that the sum of squared errors is minimized. Given a training set with data points indexed by times $t \in T$, the parameters are given by

$$\min_{\hat{\beta}_0,\hat{\beta}_1} \sum_{t=1}^{T} \varepsilon_t^2 = \min_{\hat{\beta}_0,\hat{\beta}_1} \sum_{t=1}^{T} (y_t - \hat{y}_t)^2 = \min_{\hat{\beta}_0,\hat{\beta}_1} \sum_{t=1}^{T} \left(y_t - \hat{\beta}_0 + \hat{\beta}_1 x_t\right)^2.$$

The parameters can be found using simple calculus, i.e., solving $\partial \sum_t \varepsilon_t^2 / \partial \beta_i = 0$ for $i = 0, 1$. We can use these parameters to make predictions about future values of y_t based on new observations of x_t. Specifically, for a new value x_{t+h} (where h represents the forecast horizon), the forecasted value of y can be computed as

$$\hat{y}_{t+h} = \hat{\beta}_0 + \hat{\beta}_1 x_{t+h}.$$

To use linear regression for forecasting a time series, we, of course, assume that the time series of interest is linearly dependent on the predictor variable. Additionally, we require errors to be independent and identically distributed normal random variables with zero mean and constant variance.

We can extend this idea of linear regression to multiple predictor variables. Now, assume the time series of interest, y_t, is being influenced by a collection of possible independent series, say $x_{1,t}, x_{2,t}, \ldots, x_{k,t}$.

Definition 3.24. The **linear regression** model with k regressors is defined by

$$y_t = \beta_0 + \beta_1 x_t + \beta_2 x_{2,t} + \cdots + \beta_k x_{k,t} + \varepsilon_t \tag{3.16}$$

where $\beta_0, \beta_1, \ldots, \beta_k$ are unknown fixed regression parameters.

The coefficients can be estimated in a similar manner as before, by minimizing the sum of squared errors:

$$\min_{\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_k} \sum_{t=1}^{T} \varepsilon_t^2 = \min_{\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_k} \sum_{t=1}^{T} \left(y_t - \hat{y}_t\right)^2.$$

3.3.2. Logistic Regression

Linear regression is very versatile for forecasting time series but may not be suitable when the time series has nonlinear relationships. The simplest way to then model these nonlinear relationships is to transform the variable of interest y, or the predictor variables x, before estimating the model parameters. Common transformations include logarithmic transformations. However, we will not discuss these types of nonlinear regression any further.

A commonly used method to forecast binary observations is logistic regression. This model estimates the log-odds of one of two possible events occurring. It employs the logit model, which is represented by the sigmoid function

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

Assume the variable of interest is a Bernoulli random variable with probability p of taking the value 1 and probability $q = 1 - p$ of taking the value 0. In the logit model, e^x can be seen as the odds for value 1. Therefore, the probabilities can be expressed as

$$p = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}, \quad \text{and} \quad q = \frac{1}{1 + e^x}.$$

In the logistic regression model, the log-odds, x, are determined using a linear combination of the predictor variables, similar to linear regression. We end up with the following model.

Definition 3.25. The **logistic regression** model with k regressors is defined by

$$y_t = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_t + \beta_2 x_{2,t} + \cdots + \beta_k x_{k,t})}}.$$

While linear regression predicts continuous outcomes, logistic regression predicts binary outcomes because the sigmoid function maps the predicted values to probabilities.

3.4. Exponential Smoothening Models

3.4.1. Simple Smoothening

The idea of smoothening methods is to use a weighted average of past observations to make predictions about the future. This can be seen as linear regression, where the predictor variables are solely past data points. Naively, we can say the next observation will just be the same as the previous data point, i.e.

$$y_{t+1|t} = y_t.$$

However, older data points also contain information that we may want to use. Another straightforward approach would be to say the next observation will be the average of all previous observations, i.e.

$$y_{t+1|t} = \frac{1}{t}\sum_{i=1}^{t} y_i.$$

Exponential smoothening serves as a middle ground between these naive methods. It incorporates all previous data points but places greater emphasis on more recent observations. This is implemented by a weighted average, where weights decay exponentially as observations get older. So, more recent observations are considered more important. The most simple exponential smoothening model is defined below.

Definition 3.26. The **simple exponential smoothening** model with parameter α is defined by

$$y_t = \alpha y_{t-1} + \alpha(1-\alpha)y_{t-2} + \alpha(1-\alpha)^2 y_{t-3} + \cdots,$$

where $0 \leq \alpha \leq 1$. Note that for $\alpha = 1$, we have the naive case.

Note, however, that this model only works if the time series has no clear trend or seasonal patterns. Predictions for the next observations can be made using a weighted average of past observations. Given observations up to time t, the one-step-ahead prediction of the time series is given by

$$\begin{aligned}\hat{y}_{t+1|t} &= \alpha y_t + \alpha(1-\alpha)y_{t-1} + \alpha(1-\alpha)^2 y_{t-2} + \cdots \\ &= \alpha y_t + (1-\alpha)\hat{y}_{t|t-1}.\end{aligned}$$

Note that if we make a multi-step prediction of horizon h, where the observations are known until time t, the forecast simply projects a constant line. This can be explained by the fact that instead of the observation y_t, we use the forecast $\hat{y}_t$ for the unobserved time points. Since both terms in the forecast are now the same, we can simply add them. This happens for every forecast where we do not have the actual observation to adjust the weighted average. So,

$$\hat{y}_{t+h|t} = \alpha\hat{y}_{t+h-1|t} + (1-\alpha)\hat{y}_{t+h-1|t} = \hat{y}_{t+h-1|t} = \cdots = \hat{y}_{t+1|t}.$$

Similar to linear regression, the optimal parameter α over a training set $t \in T$ can be determined using the least squares method:

$$\min_{\alpha} \sum_{t=1}^{T} (y_t - \hat{y}_t)^2.$$

For convenience, we will write the forecasting equation in the following form:

$$\hat{y}_{t+h|t} = \ell_t, \tag{3.17}$$

$$\ell_t = \alpha y_t + (1-\alpha)\ell_{t-1}, \tag{3.18}$$

where we call Equation (3.17) the forecasting equation and Equation (3.18) the smoothening equation.

For the exponential smoothening model, there are two options for the error term. The error of the model can be assumed to be additive or multiplicative. For an additive error term, the error and simple exponential smoothening model are defined to be

$$\varepsilon_t = y_t - \hat{y}_{t|t-1} = y_t - \ell_{t-1}, \qquad \begin{aligned} y_t &= \ell_{t-1} + \varepsilon_t, \\ \ell_t &= \ell_{t-1} + \alpha\varepsilon_t. \end{aligned}$$

For a multiplicative error, the error and simple exponential smoothening model are defined to be

$$\varepsilon_t = \frac{y_t - \hat{y}_{t|t-1}}{\hat{y}_{t|t-1}} = \frac{y_t - \ell_{t-1}}{\ell_{t-1}}, \qquad \begin{aligned} y_t &= \ell_{t-1}(1+\varepsilon_t), \\ \ell_t &= \ell_{t-1}(1+\alpha\varepsilon_t). \end{aligned}$$

3.4.2. Holt-Winters

Holt expanded the simple exponential smoothing model in 1957 to allow time series data with a trend. This method extends the framework of Equations (3.17) and (3.18) to a model with a forecast equation and two smoothening equations, one for the level and one for the trend. The trend component is based on the difference between the two adjacent level values, i.e., $\ell_t - \ell_{t-1}$. Similarly, as for the level, the smoothening principle can be applied to all past observations of the trend. Assuming a constant linear trend, we can again make a multi-step prediction of horizon h.

Definition 3.27. Holt's linear trend method [15], with smoothening parameters α and β, for forecasting time series with a trend, is given by the equations

$$\begin{aligned} \hat{y}_{t+h|t} &= \ell_t + hb_t, \\ \ell_t &= \alpha y_t + (1-\alpha)(\ell_{t-1} + b_{t-1}), \\ b_t &= \beta(\ell_t - \ell_{t-1}) + (1-\beta)b_{t-1}. \end{aligned}$$

Here, ℓ denotes the estimated level, and b_t denotes the estimated trend of the time series at time t. The parameter α is the smoothening coefficient for the level and β the smoothening coefficient for the trend. Both parameters are required to be in the range $\alpha, \beta \in [0, 1]$.

Holt's linear trend method assumes a constant linear trend. However, many real-world time series do not show a trend that keeps increasing or decreasing indefinitely. Assuming a constant increase or decrease, therefore, often overestimates the real observations for predictions over longer horizons. Gardner and Mckenzie (1985) extended the linear trend model with a dampening parameter.

Definition 3.28. The **damped trend method** [10], with smoothening parameters α, β, and damping parameter ϕ, for forecasting time series with a trend, is given by the equations

$$\begin{aligned} \hat{y}_{t+h|t} &= \ell_t + \sum_{i=1}^{h} \phi^i b_t, \\ \ell_t &= \alpha y_t + (1-\alpha)(\ell_{t-1} + \phi b_{t-1}), \\ b_t &= \beta(\ell_t - \ell_{t-1}) + (1-\beta)\phi b_{t-1}. \end{aligned}$$

Here, ℓ_t and b_t represent the level and trend components, with smoothening parameters α and β, respectively. The damping parameter ϕ satisfies $0 \leq \phi \leq 1$. When $\phi = 0$, there is no trend. A ϕ value between 0 and 1 indicates a damped trend. For $\phi = 1$, the model exhibits a linear trend, similar to Holt's method.

Holt and Winters further expanded Holt's trend method to also include seasonality. This method extends the framework of Equations (3.17) and (3.18) to a model with a forecast equation and three smoothening equations: one for the level, one for the trend component, and one for the seasonal component. The corresponding smoothening parameters are α, β and γ.

There are two types of Holt-Winters models, distinguished by the way they deal with the seasonal component. First, the additive model defines the seasonal component in absolute terms in the scale of the observed series. In the level equation, the seasonal component is subtracted to account for the seasonal patterns. The seasonal equation s_t calculates a weighted average between the current seasonal value and the seasonal value m time periods ago, representing a full cycle.

Definition 3.29. Holt-Winters' additive model [15, 52], with smoothening parameters α, β and γ, for forecasting time series with trend and seasonality with period m, is given by the equations

$$\begin{aligned}\hat{y}_{t+h|t} &= \ell_t + hb_t + s_{t-m+T},\\ \ell_t &= \alpha(y_t - s_{t-m}) + (1-\alpha)(\ell_{t-1} + b_{t-1}),\\ b_t &= \beta(\ell_t - \ell_{t-1}) + (1-\beta)b_{t-1},\\ s_t &= \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m},\end{aligned}$$

where $T = [(h-1) \mod m] + 1$ is the specific period in the season. Here, ℓ_t, b_t, and s_t represent the level, trend, and seasonal components, with smoothening parameters α, β, and γ, respectively.

In the multiplicative model, the seasonal component is represented as a relative measure. To account for the seasonal effect, the series is divided by the seasonal component.

Definition 3.30. Holt-Winters' multiplicative model [15, 52], with smoothening parameters α, β and γ, for forecasting time series with trend and seasonality with period m, is given by the equations

$$\begin{aligned}\hat{y}_{t+h|t} &= (\ell_t + hb_t)s_{t-m+T},\\ \ell_t &= \alpha\frac{y_t}{s_{t-m}} + (1-\alpha)(\ell_{t-1} + b_{t-1}),\\ b_t &= \beta(\ell_t - \ell_{t-1}) + (1-\beta)b_{t-1},\\ s_t &= \gamma\frac{y_t}{(\ell_t + b_{t-1})} + (1-\gamma)s_{t-m},\end{aligned}$$

where $T = [(h-1) \mod m] + 1$ is the specific period in the season. Here, ℓ_t, b_t, and s_t represent the level, trend, and seasonal components, with smoothening parameters α, β, and γ, respectively.

The models discussed above, along with combinations of them, form the basis of the ETS model, which stands for Error, Trend, and Seasonality. The error component can be either additive (A) or multiplicative (M). For the trend component, the options include none (N), additive (A), and additive damped (A_d). It is also possible for the trend to be multiplicative (M) or multiplicative damped (M_d). For seasonality, the choices are none (N), additive (A), and multiplicative (M). For example, the models introduced in Definitions 3.26 to 3.30 can be classified under the ETS framework as follows:

- Simple exponential smoothing: ETS(A,N,N)
- Holt's linear trend method: ETS(A,A,N)
- Damped trend method: ETS(A,Ad,N)
- Holt-Winters additive method: ETS(A,A,A)
- Holt-Winters multiplicative method: ETS(A,A,M)

We will not delve further into these models, as only the Holt-Winters model is considered in this research. The interested reader can read more on the classification of exponential smoothing methods in the book by Hyndman et al. [17].

3.5. ARIMA Models

3.5.1. ARMA models for Stationary Time Series

The general ARMA model, which stands for Autoregressive Moving Average, was popularized by Box and Jenkins [5]. This model combines both autoregressive and moving average components to analyse a time series. An important assumption of the ARMA model is that the time series being analyzed must be stationary. We will start by introducing the autoregressive model.

Autoregression of a time series means that regression is used on its own past values. So, an autoregressive model is essentially a linear combination of previous observations. The order of the model specifies how many time steps back are considered in the regression.

Definition 3.31. The **autoregressive model** [5], of order p, is defined by

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \varepsilon_t = \sum_{i=1}^{p} \phi_i B^i y_t + \varepsilon_t,$$

where ε_t is white noise and ϕ_i, $i = 1, \ldots, p$, are the model weights. For convenience, we define the autoregressive operator of order p

$$\boldsymbol{\phi}(B) = (1 - \sum_{i=1}^{p} \phi_i B^i)$$

Then, the autoregressive model can be written as $\boldsymbol{\phi}(B) y_t = \varepsilon_t$.

Instead of past observations, the moving average model uses past forecast errors in a regression model.

Definition 3.32. The **moving average model** [5], of order q, is defined by

$$y_t = \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots + \theta_q \varepsilon_{t-q} = \varepsilon_t + \sum_{i=1}^{q} \theta_i B^i \varepsilon_t,$$

where ε_t is white noise and θ_i, $i = 1, \ldots, q$, are the model weights. For convenience, we define the moving average operator of order q

$$\boldsymbol{\theta}(B) = (1 + \sum_{i=1}^{q} \theta_i B^i).$$

Then, the autoregressive model can be written as $y_t = \boldsymbol{\theta}(B) \varepsilon_t$.

We can combine the autoregressive and moving average model to obtain the ARMA model or order p and q.

Definition 3.33. The **ARMA model** [5], with autoregressive order p and moving average order q, is given by

$$y_t = \sum_{i=1}^{p} \phi_i B^i y_t + \sum_{i=1}^{q} \theta_i B^i \varepsilon_t + \varepsilon_t, \tag{3.19}$$

where ε_t is white noise and ϕ_i, $i = 1, \ldots, p$, and θ_i, $i = 1, \ldots, q$, are the weights of the model. In shorthand notation, we can write $\boldsymbol{\phi}(B) y_t = \boldsymbol{\theta}(B) \varepsilon_t$.

To fit an ARMA(p, q) model to a time series, we have to find the optimal parameters ϕ_i, $i = 1, \ldots, p$, and θ_i, $i = 1, \ldots, q$, such that the error between the observed values and the model's predicted values is minimized. Selecting the right parameters for an ARMA model is essential to ensure optimal performance. However, multiple parameter sets might provide satisfactory outcomes. While higher-order ARMA models, which incorporate more past observations, may improve model accuracy, they also have a higher risk of overfitting. To ensure a balance between model fit and model complexity, the AIC and BIC from Definitions 3.22 and 3.23 are commonly used to select the best parameters.

Various packages in Python and R offer implementations that automatically select the optimal model parameters. This automation simplifies the model selection process and reduces the manual effort involved. However, potential drawbacks are the reduced control over the modeling process and a lack of transparency. While estimating the parameters for an AR model can be simply done, for example, using least squares. This is, however, not possible for the moving average model as the errors cannot be observed. This makes estimating the parameters more complicated. The `auto.arima` from R uses maximum likelihood estimation [16].

After fitting the ARMA model, we can use the model to forecast future data points. Given data up to time t, and Equation (3.19), we can produce the point forecast $t + h$ by stating with $h = 1$ and repeating for $2, 3, \ldots$. iteratively. For any known time point, we directly use the observed value y_{t-i} or the observed error value ε_{t-i} for any $i \geq 0$. For future time points, we assume $\varepsilon_{t+h} = 0$ for $h > 0$. For intermediate future values y_{t+j} where $0 < j < h$, we can utilize the forecasts from previous iterations.

3.5.2. ARIMA models for Nonstationary Time Series

ARMA models require the assumption of stationarity of the data. As discussed in Section 3.1.3, we can try to make a time series stationary through differencing. This idea is incorporated in the ARIMA model, which stands for Autoregressive Integrated Moving Average. It extends the ARMA model by first applying differencing on the time series.

Definition 3.34. The **ARIMA models** [5], with autoregressive order p, moving average order q and differencing of order d, is given by

$$(1 - \sum_{i=1}^{p} \phi_i B^i)(1 - B)^d y_t = (1 + \sum_{i=1}^{q} \phi_i B^i)\varepsilon_t, \tag{3.20}$$

where ε_t is white noise and ϕ_i, $i = 1, \ldots, p$, and θ_i, $i = 1, \ldots, q$, are the weights of the model. In shorthand notation, we can write $\boldsymbol{\phi}(B)\Delta^d y_t = \boldsymbol{\theta}(B)\varepsilon_t$.

It is important to note that for an ARIMA model with parameters (p, d, q), the parameter count k, as used in Definitions 3.22 and 3.23, is given by $k = p + q + 1$. This count includes the orders of the autoregressive and moving average components. The additional one in the equation accounts for the variance of the error term. The differencing order is not directly included in k, although it is implicitly reflected in the likelihood calculation.

3.5.3. Seasonal ARIMA Models

A further extension of ARIMA model is the Seasonal ARIMA, or SARIMA, which includes seasonality. The model is denoted by parameters $(p, d, q)(P, D, Q)_m$, where the lowercase notation is for the non-seasonal part and the uppercase notation is for the seasonal part of the model. The length of the seasonal cycle is denoted by m. The seasonal part works similarly to the non-seasonal part already present in the ARIMA model, except for the seasonal part, we use the seasonal backshift parameter B^m instead of B.

Definition 3.35. The **Seasonal ARIMA** non seasonal part (p, d, q) and seasonal part $(P, D, Q)_m$ where m is the seasonal period

$$(1 - \sum_{i=1}^{p} \phi_i B^i)(1 - \sum_{i=1}^{P} \Phi_i B^{im})(1 - B)^d (1 - B^m)^D y_t = (1 + \sum_{i=1}^{q} \theta_i B^i)(1 + \sum_{i=1}^{Q} \Theta_i B^{im})\varepsilon_t, \tag{3.21}$$

where ε_t is white noise and ϕ_i, $i = 1, \ldots, p$, θ_i, $i = 1, \ldots, q$, Φ_i, $i = 1, \ldots, P$, and Θ_i, $i = 1, \ldots, Q$, are the weights of the model. In shorthand notation, we have

$$\boldsymbol{\phi}(B)\boldsymbol{\Phi}(B^m)\Delta^d \Delta_m^D y_t = \boldsymbol{\theta}(B)\boldsymbol{\Theta}(B^m)\varepsilon_t.$$

3.5.4. ARIMA Models with Exogenous Variables

Another extension is the ARMAX model, which stands for Autoregressive Moving Average with eXogenous variables. This model incorporates exogenous variables into the ARIMA framework. This model is particularly useful when the time series is thought to be influenced by external factors that cannot be explained solely by its past values. The covarites, denoted by $\boldsymbol{x}$, are added to the model similar to linear regression.

Definition 3.36. The **ARMAX model** [5] for forecasting time series with exogenous variables x_i, $i = 1, \ldots, r$, is given by

$$\boldsymbol{\phi}(B)y_t = \boldsymbol{\theta}(B)\varepsilon_t + \sum_{i=1}^{r} \beta_i x_t^i, \tag{3.22}$$

where $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ are the autoregressive and moving average operator of orders p and q respectively.

Similar to the extension to the ARMA model, we can also include exogenous variables in the ARIMA or SARIMA model. The SARIMAX model, for example, would be defined by

$$\boldsymbol{\phi}(B)\boldsymbol{\Phi}(B^m)\Delta^d \Delta_m^D y_t = \boldsymbol{\theta}(B)\boldsymbol{\Theta}(B^m)\varepsilon_t + \sum_{i=1}^{r} \beta_i x_t^i.$$

3.6. Prophet

Prophet [48] is a forecasting model developed by Taylor and Letham from Facebook in 2017. The model is engineered to handle various time series data characteristics with ease. It employs an additive model where time series data is decomposed into three main components: trend, seasonality, and holidays. The Prophet model is defined by

$$y(t) = g_t + s_t + h_t + \varepsilon_t. \tag{3.23}$$

Here, g_t represents the trend function, s_t represents the periodic changes such as daily, weekly, or yearly seasonality, and h_t accounts for the influence of holidays. The error term ε_t is assumed to be normally distributed.

Unlike traditional time series models that focus on the temporal dependencies in the data, Prophet frames forecasting as a curve-fitting problem. This results in faster training and allows for measurements that are not equally spaced.

To fit the model, Prophet makes use of Stan's L-BFGS optimization algorithm, which facilitates Bayesian forecasting by finding a maximum a posterior estimate. In Bayesian inference, we aim to maximize a posterior distribution $f(\theta \mid x)$, a combination of the likelihood $f(x \mid \theta)$ and the prior distribution $f(\theta)$.[2] Here, we need to assume a prior distribution for the parameters.

3.6.1. Trend Model

Prophet models the trend as linear, logistic, or flat growth. A linear growth model is given by

$$g_t = kt + m,$$

where k represents the growth rate and m is an offset parameter. Facebook also implemented a logistic growth model, since it often characterizes population growth, reflecting the most prominent data-generating processes found in Facebook data. A logistic growth model is given by

$$g_t = \frac{C}{1 + \exp(-k(t-m))}.$$

Here, C is the carrying capacity, k represents the growth rate, and m is an offset parameter. Facebook refined this model by introducing a time-varying capacity C_t.

For both linear and logistic growth, Prothet allows for changes in the growth rate over time. These changes are incorporated by defining changepoints at which the growth rate k can shift. Suppose there are S changepoints at times t_j, $j = 1, \ldots, S$. Then define a vector of rate adjustments $\boldsymbol{\delta} \in \mathbb{R}^S$, where δ_j is the change that occurs at time t_j. The rate at time t is a base rate k plus all the adjustments up to that point, i.e., $k + \sum_{j:\, t>t_j} \delta_j$. For the rate adjustments, Taylor and Letham use a prior $\delta_j \sim \text{Laplace}(0, \tau)$, where τ controls the flexibility in altering the rate. If $\tau \to 0$, the model reduces back to a standard linear or logistic growth model.

3.6.2. Seasonal Model

Seasonal effects in Prophet are captured using Fourier series, i.e.,

$$s_t = \sum_{n=1}^{N} \left(a_n \cos\left(\frac{2\pi n t}{P}\right) + b_n \sin\left(\frac{2\pi n t}{P}\right) \right).$$

This approach estimates the parameter vector $\boldsymbol{\beta} = [a_1, b_1, \ldots, a_N, b_N]^T$ with $2N$ parameters. By constructing a matrix of seasonality vectors $\boldsymbol{X}_t$ for each t, the seasonality can be expressed as $s_t = \boldsymbol{X}_t \boldsymbol{\beta}$. For example, $\boldsymbol{X}_t$ for yearly seasonality with $N = 10$ is given by

$$\boldsymbol{X}_t = \left[\cos\left(\frac{2\pi(1)t}{365.25}\right), \ldots, \sin\left(\frac{2\pi(10)t}{365.25}\right) \right].$$

For the seasonality parameters, the prior $\boldsymbol{\beta} \sim \mathcal{N}(0, \sigma)$ is used. The authors found that $N = 10$ works well for yearly seasonality and $N = 3$ works well for weekly seasonality.

[2] Using Bayes' rule, we find $f(\theta \mid x) = \frac{f(x \mid \theta) f(\theta)}{f(x)} \propto f(x \mid \theta) p(\theta)$, which is the likelihood times the prior.

3.6.3. Holidays and Events

Holidays and events introduce significant, yet somewhat predictable, disruptions to business time series that do not conform to regular periodic cycles covered in the seasonal part. Prophet allows for the input of a list of past and future events, identifiable via unique names, into the forecasting model.

For each holiday i, let D_i be the set of dates corresponding to that event. An indicator function is implemented to determine whether time t falls within holiday i, and each holiday is assigned a parameter κ_i reflecting its specific impact on the forecast. The holiday part of the forecast, h_t, is modeled as $h_t = \boldsymbol{Z}_t\boldsymbol{\kappa}$, with

$$\boldsymbol{Z}_t = [\mathbf{1}(t \in D_1), \ldots, \mathbf{1}(t \in D_1)]\,.$$

Similar to the seasonal part, $\boldsymbol{\kappa} \sim \mathcal{N}(0, \nu^2)$ is used as prior distribution.

The model also enables taking into account a window of days surrounding holidays by treating each day in the window as a holiday itself. Note that for the model to learn the effect of certain holidays, it needs to have multiple data points in the set D_i. For the specific case study addressed in this thesis, which involves only one year of data, this condition does not hold true.

3.7. Hierarchical Forecasting

3.7.1. Hierarchical Time Series

Hierarchical time series are structured in a tree-like manner, featuring various levels of aggregation. At the lowest level, there are numerous individual time series. Going up in the hierarchy, each series represents the summation of the series directly beneath it. For example, as illustrated in Figure 3.4, at the top level, we have y_0, which is the sum of series y_1 and y_2 below it. Going down to the bottom level, we have $y_1 = y_{1,1} + y_{1,2}$ and $y_2 = y_{2,1} + y_{2,2} + y_{2,3}$.

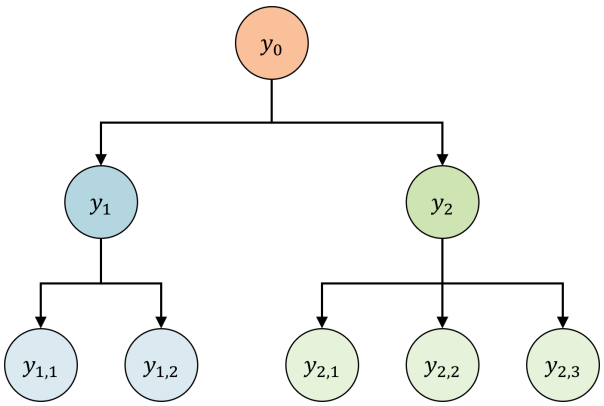


Figure 3.4: Example of a hierarchical structure in time series. Here, $y_0 = y_1 + y_2$, $y_1 = y_{1,1} + y_{1,2}$, and $y_2 = y_{2,1} + y_{2,2} + y_{2,3}$.

Forecasting hierarchical time series is essential for decision-makers who require consistency across different levels of data aggregation. Consider a scenario where a retailer needs to decide how much stock to purchase based on customer demand forecasts. If forecasts are generated for individual customer segments and then summed, the total may not align with a forecast generated directly from the total demand. This difference poses a dilemma: Should the purchasing decision be based on the sum of individual forecasts or the forecast of the total series? Hierarchical forecasting addresses this issue by ensuring consistent forecasts at different hierarchical levels.

3.7.2. Single-Level Approaches

Hierarchical forecasting can be approached in several ways: bottom-up, top-down, or middle-out.

The bottom-up approach is the most straightforward method. It starts by forecasting the series at the bottom level of the hierarchy. The individual forecasts are then summed to generate predictions for higher levels. This method ensures consistency but does not use the forecasts of higher levels.

In contrast, the top-down approach begins with forecasting the top level of the hierarchy. These forecasts are then proportionally allocated to lower levels based on one of the following criteria: average historical proportion, proportions of the historical averages, or forecast proportions. This method is useful when high-level data is more reliable or when lower-level data is noisy.

The middle-out approach combines elements of both previous methods. It starts by forecasting at a middle level of the hierarchy. Forecasts for higher levels are generated by aggregating these middle-level forecasts, while forecasts for lower levels are derived by disaggregating the middle-level forecasts. This approach can be effective when the middle level represents a significant segmentation of the data.

3.7.3. Forecast Reconciliation

In hierarchical forecasting, it is also possible to combine the forecasts of all series to ensure coherence. This process, proposed by Hyndman et al. (2011), is called forecast reconciliation. To understand this method, first define the summing matrix $\boldsymbol{S}$ such that

$$\boldsymbol{y}_t = \boldsymbol{S}\boldsymbol{b}_t$$

where $\boldsymbol{b}_t$ are the time series at the lowest level and $\boldsymbol{y}_t$ are all time series across all hierarchies. For example, the summing equation for the structure of Figure 3.4 is given by

$$\begin{bmatrix} (y_0)_t \\ (y_1)_t \\ (y_2)_t \\ (y_{1,1})_t \\ (y_{1,2})_t \\ (y_{2,1})_t \\ (y_{2,2})_t \\ (y_{2,3})_t \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} (y_{1,1})_t \\ (y_{1,2})_t \\ (y_{2,1})_t \\ (y_{2,2})_t \\ (y_{2,3})_t \end{bmatrix}.$$

Now, forecast reconciliation is defined by a transformation matrix $\boldsymbol{P}$ that maps the forecasts of all levels into the bottom layer. Then, using the summing matrix $\boldsymbol{S}$, we can obtain the coherent forecast of all levels. So, given the forecasts of all levels $\hat{\boldsymbol{y}}_t$, the reconciled forecast is given by

$$\tilde{\boldsymbol{y}}_t = \boldsymbol{S}\boldsymbol{P}\hat{\boldsymbol{y}}_t.$$

For example, the bottom-up approach on Figure 3.4, is given by

$$\boldsymbol{P} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Similarly, the top-down approach on Figure 3.4, is given by

$$\boldsymbol{P} = \begin{bmatrix} p_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ p_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ p_3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ p_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ p_5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where p_i are the estimated proportions for disaggregation.

A commonly used reconciliation method is the min-trace method by Wickramasuriya, Athanasopoulos, and Hyndman (2015). They derive the mapping matrix $\boldsymbol{P}$ by minimizing the sum of variances of the reconciled forecast errors.

$$\boldsymbol{P} = (\boldsymbol{S}^T \boldsymbol{W}_h^{-1} \boldsymbol{S})^{-1} \boldsymbol{S}^T \boldsymbol{W}_h^{-1}$$

where $\boldsymbol{W}_h$ is the covariance matrix matrix of the h-step-ahead forecast errors. Specifically,

$$\boldsymbol{W}_h = \mathbb{E}[\hat{e}_t(h)\hat{e}_t^T(h)|\mathcal{I}_t],$$

where

$$\hat{e}_t = y_{t+h} - \hat{y}_t(h).$$

Now, since the trace of the covariance matrix is the sum of variances, minimizing the sum of variances of the reconciled forecast errors is equal to

$$\min \mathsf{tr}[SPW_hP^TS^T].$$

4

Machine Learning Methods for Time Series Forecasting

The previously introduced methods for time series forecasting, like ARIMA and regression, rely on a predefined mathematical model that has to be able to represent the patterns in the data. In contrast, in machine learning we often want to learn these, sometimes complex, patterns without explicitly defining a mathematical formula for the structure of the series.

In this chapter, we will discuss various machine learning techniques, starting with the random forest algorithm. This chapter also presents three state-of-the-art deep learning models for time series forecasting, which are central to this research. To understand these advanced models, we will start with an introduction to the fundamental concepts of neural networks and we will explore some of the building blocks that constitute each model.

4.1. Decision Trees and Random Forest

A random forest is a popular ensemble learning technique that combines the output of multiple decision trees. One big advantage of the random forest is its ability to provide insights into the feature importance, adding a level of explainability to the predictions. We will first introduce the concept of tree-based models to later present the algorithm that generates the random forest. This section is mostly based on the book by Hastie, Tibshirani, and Friedman [14].

4.1.1. Decision Trees

Given a target variable y and p input features $\boldsymbol{x} = (x_1, x_2, \ldots, x_p)$, a decision tree aims to predict the target by recursively partitioning the input feature space into distinct regions. The partitioning is based on tests performed on the feature values. This creates a tree-like structure of decisions and their possible outcomes that result in a final decision or value.

Definition 4.1. The structure of a decision tree is defined by the following elements:

- An **internal node** represents a test on a feature. This node often makes a binary or multi-way decision, based on the value of the feature. This leads to a further split in the tree. It is also possible to have an internal node with a probabilistic outcome.
- **Branches** are the outcomes of the test of an internal node. Each branch corresponds to a specific value or range of values of the feature being tested.
- A **leaf node** represents the final decision taken after completing all the tests along a path from the root to that leaf.

An example of a simple decision tree is given in Figure 4.1. Using a very simplified model, we want to determine whether a customer will place an order for today. This example has two internal decision

nodes and three leaf nodes. The branches here are the answers to the yes/no questions in the nodes. Using this decision tree we can follow a path from the root node to a leaf node to answer our question.

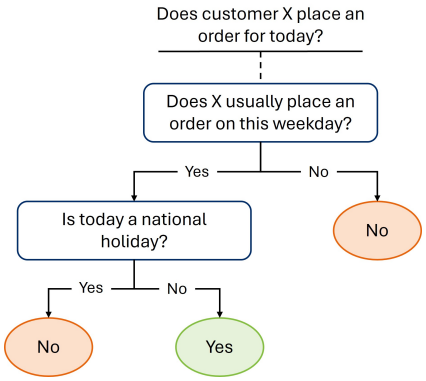


Figure 4.1: Example of a simple decision tree to answer the question: Does customer X place an order for today? Answers to the questions on the internal nodes lead to an answer to the top question in the leaf nodes.

There are two types of decision trees. A regression tree is a type of decision tree used for predicting continuous numerical outcomes. Each leaf node represents a predicted value for the target variable. A classification tree is a type of decision tree used for predicting categorical outcomes. Each leaf node of the tree corresponds to a predicted class label for the target variable.

Although decision trees are not designed for forecasting time series, they can still be adapted for this purpose. By including lagged values of the variable of interest, y_t, as features, the tree can make a decision based on previous observation.

There are multiple ways to grow a decision tree, which we will not discuss in detail here. In general, the aim of a tree building algorithm is to find the best partitions that divide the feature space. Often, a greedy algorithm[1] is applied to find the best split at the current node, as it is computationally very hard to find the optimal partitions. This is achieved by minimizing the node impurity, which is a measure of how well a node splits the data into homogeneous sets. Essentially, this process measures the informational gain provided by each split. For regression trees, we can use the mean square error as impurity measure. For classicification, the Gini index or Entropy can be used utilized as impurity measure [14].

To prevent overfitting, one could tune the following hyperparameters: the minimum number of samples required for splitting, the maximum tree size, the maximum tree depth, and the minimum number of samples for each leaf node. If a decision tree has the same number of leaf nodes as training data points, it will perform perfectly on the training data but will likely fail to generalize to new data. Therefore, it is crucial to find a good balance between model complexity and generalization. The optimal hyperparameters may vary for each dataset.

4.1.2. Ensemble Methods

Suppose we have observed some data set $\boldsymbol{x} = (x_1, x_2, \ldots, x_N)$ with N observations and assume it is independent and identically distributed according to an unknown distribution F. When we cannot repeat the experiment anymore, we are not able generate more samples as F is unknown. The idea of the bootstrap method is to generate more samples by replacing the distribution F with the empirical distribution function $\hat{F}$. Since we know this empirical distribution function, it is possible to sample from this distribution as often as desired.

Definition 4.2. A **bootstrap sample** is defined to be

$$\boldsymbol{x}^* = (x_1^*, x_2^*, \ldots, x_N^*),$$

where each x_i^* is drawn randomly from the original sample $(x_1, x_2, \ldots, x_N)$, with replacement.

[1] A greedy algorithm makes the locally optimal choice at each step of the algorithm, in the hope of finding a global optimial solution.

We can also use the concept of bootstrapping to obtain multiple predictions from a single dataset. Say we have a training data set $Z = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)\}$ where $\boldsymbol{x}_i$ are the p feature variables and y_i is the variable of interest. Using a forecasting method, like regression, we can find a prediction $\hat{f}(\boldsymbol{x})$ for y at input $\boldsymbol{x}$. Bootstrap aggregation, or bagging, averages the prediction over a collection of bootstrap samples.

Definition 4.3. The **bagging estimate** for training data set Z, using B bootstrap samples Z^{*b} for $b = 1, \ldots, B$, is given by[2]

$$\hat{f}_{\mathsf{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(\boldsymbol{x}), \tag{4.1}$$

where $\hat{f}^{*b}(\boldsymbol{x})$ is the prediction determined from a bootstrap sample Z^{*b}.

Say we are using a decision tree for forecasting, where $\hat{f}(\boldsymbol{x})$ denotes the prediction of the tree for input $\boldsymbol{x}$. Each bootstrap tree will typically have different features than the original, and might have a different number of leaf nodes. The bagged estimate is the average prediction at $\boldsymbol{x}$ from these B trees.

Since each tree generated in bagging is identically distributed, we have that the expected value of an average of B trees is the same as the expected value of any individual tree. Therefore, the bias of bagged trees is also the same as for one tree. The variance of the average of B trees can be obtained using the following theorem.

Theorem 4.1. *The variance of an average of n independent and identically distributed random variables X_i, $i = 1, \ldots, n$, is given by*

$$\mathsf{var}\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = \rho\sigma^2 + \frac{1-\rho}{n}\sigma^2,$$

where ρ is the pairwise correlation, $\rho_{XY} = \frac{\mathsf{cov}(X,Y)}{\sigma_X \sigma_Y}$, and σ^2 is the variance of random variable X_i.

Proof.

$$\begin{aligned}\mathsf{var}\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) &= \frac{1}{n^2}\mathsf{cov}\left(\sum_{i=1}^{n} X_i, \sum_{i=1}^{n} X_i\right) = \frac{1}{n^2}\left(\sum_{i=1}^{n}\mathsf{var}(X_i) + \sum_{i\neq j}\mathsf{cov}(X_i, X_j)\right) \\ &= \frac{1}{n^2}\left(n\sigma^2 + n(n-1)\rho\sigma^2\right) = \rho\sigma^2 + \frac{1-\rho}{n}\sigma^2.\end{aligned}$$

□

Note that as the number of trees increases, the second term in Theorem 4.1 disappears. The first term stays. So the variance of the average of B trees is determined by the correlation of pairs of bagged trees. This is where random forest comes into play.

4.1.3. The Random Forest Algorithm

The idea of a random forest is to improve the variance of a bagged collection of decision trees by reducing the correlation between trees. This is done by randomly selecting a subset of m feature variables at each step in the tree-growing process. This feature randomness ensures a low correlation between different decision trees. The inventors recommend a value of $m = \sqrt{p}$ for classification problems and $m = p/3$ for regression problems [14]. However, the best value for $m \leq p$ can differ case-by-case. The algorithm for generating a random forest is described in Algorithm 4.1.

[2] To be precise, Equation (4.1) represents a Monte Carlo estimate of the true bagging estimate, which converges to it as $B \to \infty$. For more details, see page 282 [14] .

Algorithm 4.1 Random Forest for Regression or Classification [14]

for $b = 1$ **to** B **do**

1. Draw a bootstrap sample Z^* of size N from the training data.
2. Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the stopping criterion (maximum tree size, minimum node size, ...):
   (a) Select m variables at random from the p feature variables.
   (b) Pick the best variable and split-point among the m variables.
   (c) Split the node into two daughter nodes.

Output the ensemble of trees $\{T_b\}_{b=1}^{B}$.

Making decisions using a random forest is done by taking the average of the outcomes of all decision trees in the forest. For regression trees with prediction $\hat{T}_b(\boldsymbol{x})$ of tree b, the final prediction of a random forest with B trees is given by

$$\hat{f}_{rf}^{B}(\boldsymbol{x}) = \frac{1}{B}\sum_{b=1}^{B}\hat{T}_b(\boldsymbol{x}).$$

For classification trees with class prediction $\hat{C}_b(\boldsymbol{x})$ of tree b, the final prediction of a random forest with B trees is given by the majority vote of all trees in the forest, i.e.

$$\hat{C}_{rf}^{B}(\boldsymbol{x}) = \text{majority vote}\left\{\hat{C}_b(\boldsymbol{x})\right\}_1^B.$$

4.1.4. Advantages of Random Forests

The random forest algorithm is a popular machine learning technique because of its ability to learn nonlinear relationships between features and the target variable. Because of the aggregation of many random trees, the outcome of the random forest is quite robust. This reduces the risk of overfitting but also helps against outliers. Moreover, because of its rule based structure, random forests are flexible at handling various data types. For the same reason, the algorithm does not require normalization or scaling of data, unlike many other algorithms.

A major advantage of using random forests is their capability to provide insights into the importance of each feature in making predictions. The importance of a feature is determined based on how frequently it is used to split nodes across all trees and how much information those splits gave (amount of decrease in impurity).

While there exist other decision tree algorithms, like gradient boosting, that powerful as well, this research only focuses on random forests.

4.2. Introduction to Neural Networks

Decision trees remain popular due to their ability to learn complex patterns from data. However, with recent advancements in computational power, more advanced models, particularly neural networks, have gained popularity. These deep learning methods are capable of learning patterns from even larger and more complex datasets. In this research, we will focus on three neural network models designed for time series forecasting.

Neural networks are inspired by the structure of the human brain. They consist of connected layers of artificial neurons that each learns from data through training. The idea is that neural networks automatically learn complex patterns, without giving too much structure to the model beforehand. In this section, we will explore the fundamental concepts of neural networks, based on elements from [1].

4.2.1. Neurons and Learning

The most basic building blocks of a neural network are artificial neurons. Each neuron receives input signals, processes them, and produces an output signal that is passed on to other neurons in the network. It does this by applying an activation function on a weighted sum of the input. The output of a neuron with weights $\boldsymbol{w}$, bias b and activation function g is given by

$$\hat{y} = g(\boldsymbol{w} \cdot \boldsymbol{x} + b). \tag{4.2}$$

A schematic overview of an artificial neuron is given in Figure 4.2.

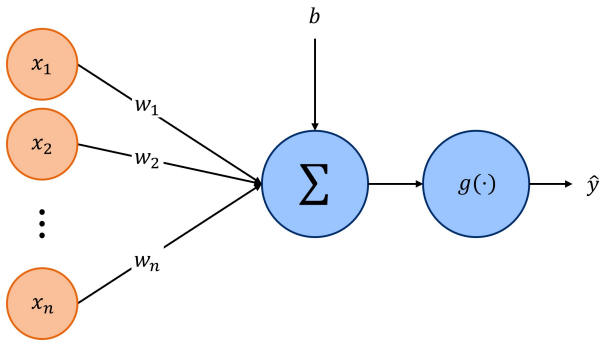


Figure 4.2: Schematic overview of a neuron in a neural network. A neuron takes a weighted sum of inputs $x_1, \ldots, x_n$ and adds a bias b. It then performs an activation function on this value. The output of a neuron is therefore given by $\hat{y} = g(\boldsymbol{w} \cdot \boldsymbol{x} + b)$.

In a neural network, neurons are organized into layers: an input layer, one or more hidden layers and an output layer. The input layer receives the raw data, the hidden layers perform transformations based on what is learned from previous data and finally, the output layer produces a final prediction.

The most simple type of a neural network architecture is a Feedforward Neural Network (FNN), shown in Figure 4.3. Information flows in one direction from the input layer, through the hidden layers, and finally to the output later without any cycles or loops. Each layer consists of a set of neurons that are connected to every other neuron in the next layer. FNNs can be particularly effective for tasks such as regression or classification problems. However, they may struggle with time series data as they lack the mechanisms to capture dependencies across time steps. Still, as we will see later, FNNs serve as building blocks for more advanced architectures.

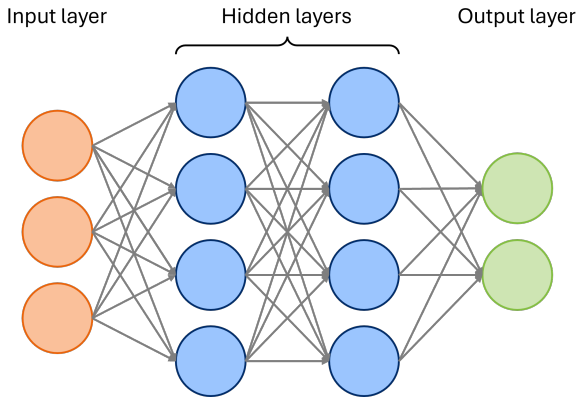


Figure 4.3: Example of a Feedforward Neural Network (FNN). A FNN is defined by an input layer, one or more hidden layers, and an output layer. Information always flows in a forward direction. In this example, the input size is 3, there are two hidden layers of size 4, and the output size is 2. This FNN is fully connected.

Learning in neural networks happens by training the network, where the network adjusts its weights and biases based on training data. A training data set contains input and target data. Through forward propagation, the network can generate some predicted output based on the input data.

Definition 4.4. Forward propagation is the process where input data is passed through the network's layers, according to its learned weights and biases, to generate an output.

Now, we want to choose the weights and biases that minimize the difference between the predicted outputs and the actual target values. This is typically done by using optimization algorithms like gradient descent. The process of computing the gradients of the loss function with respect to the weights and biases is called backpropagation.

Definition 4.5. Backpropagation is an algorithm to compute the gradient, used to determine the updates of the weights and biases. It works by applying the chain rule to propagate the error backwards from the output layer back through the hidden layers to the input layer.

The training of a neural network on a training set consists of multiple iterations of forward and backpropagation. An epoch is defined as one complete pass through the entire training dataset. If the data set is divided in multiple batches, one epoch involves multiple iterations of both forward and backpropagation. After each epoch, the model parameters are updated. A network is commonly trained using multiple epochs. After training, new predictions can be made by applying forward propagation to new input data.

4.2.2. Activation functions

Every neuron in a neural network has its own activation function. An activation function is a mathematical function that determines whether a neuron should contribute to the network's output or not. The activation function is a crucial component of the neural network component, allowing it to learn complex patterns. A few commonly used activation functions will be discussed here.

The most trivial activation function would be the linear, or identity, activation function.

Definition 4.6. The **linear** activation function is defined by

$$g(x) = x \tag{4.3}$$

However, only using linear functions would restrict the network's capacity to learn from data, limiting it to linear mappings. So, the network must have non-linear activation functions to actually learn complex patterns and make non-linear decisions. Additionally, many activation functions are piecewise continuously differentiable, as this is necessary for optimization algorithms like gradient descent.

In the early days of neural networks, it was common to use the sigmoid function to incorporate non-linearity [1]. It simulates biological neurons, i.e., firing when the input exceeds a certain threshold. The sign function also has this behaviour, but is non-differentiable.

Definition 4.7. Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.4}$$

The hyperbolic tangent (tanh) function also became popular as an activation function. Since the tanh outputs values in the range of $[-1, 1]$, it is prefered over the sigmoid when the desired outputs are both positive and negative.

Definition 4.8. The **tanh** activation function is defined by

$$g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{4.5}$$

One problem of the sigmoid and tanh function is the vanishing gradient problem. When the input becomes too high or too low, the gradients become very small, resulting in slow learning or stopping altogether.

One of the most popular activation functions in neural networks is the rectified linear unit (ReLU) activation function. It is popular as it is computationally efficient, allowing for faster convergence during training.

Definition 4.9. The rectified linear unit activation function (**ReLU**) is defined by

$$g(x) = \max(0, x) = \begin{cases} 0 & \text{if } x \leq 0, \\ x & \text{if } x > 0. \end{cases} \tag{4.6}$$

However, the ReLU can lead to the ”dying ReLU” problem, where neurons become inactive and stop learning if they consistently output zero.

To address the dying ReLU problem, the Leaky ReLU activation function introduces a small slope for negative input values.

Definition 4.10. The leaky rectified linear unit activation function (**Leaky ReLU**), with slope parameter $0 < \alpha \ll 1$, is defined by

$$g(x) = \begin{cases} \alpha x & \text{if } x \leq 0, \\ x & \text{if } x > 0. \end{cases} \tag{4.7}$$

Note that the slope parameter α is determined before training. By allowing this small, non-zero gradient, the Leaky ReLU helps to maintain the learning capability of neurons that might otherwise become inactive.

Another parametric alternative to the ReLU is the Exponential Linear Unit (ELU), which also aims to solve the dying ReLU problem.

Definition 4.11. The exponential linear unit activation function (**ELU**), with parameter $\alpha > 0$, is defined by

$$g(x) = \begin{cases} \alpha(\exp(x) - 1)x & \text{if } x \leq 0, \\ x & \text{if } x > 0. \end{cases} \tag{4.8}$$

The activation functions introduced above are applied on the weighted sum of inputs, as defined in Equation (4.2). It is also possible to apply an activation function on the inputs directly. One often used function to downsample the feature space, called pooling, is the Maxout function.

Definition 4.12. The **Maxout** activation function simply outputs the maximum of the input values without computing a weighted average,

$$g(x) = \max_i x_i. \tag{4.9}$$

The Softmax function converts the output values of the previous neurons into probabilities that sum to one. It is typically used in the output layer of a neural network for multi-class classification problems, where the goal is to assign an input to one of several possible classes. The class with the highest probability outputted by the Softmax function is usually considered the predicted class.

Definition 4.13. The **Softmax** activation function is defined by

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}, \tag{4.10}$$

where N is the number of classes and x_i is the value for class i.

There are many more possible activation functions, which will not be further discussed in this paper. The graphs of some of the activation functions discussed above are shown in Figure 4.4.

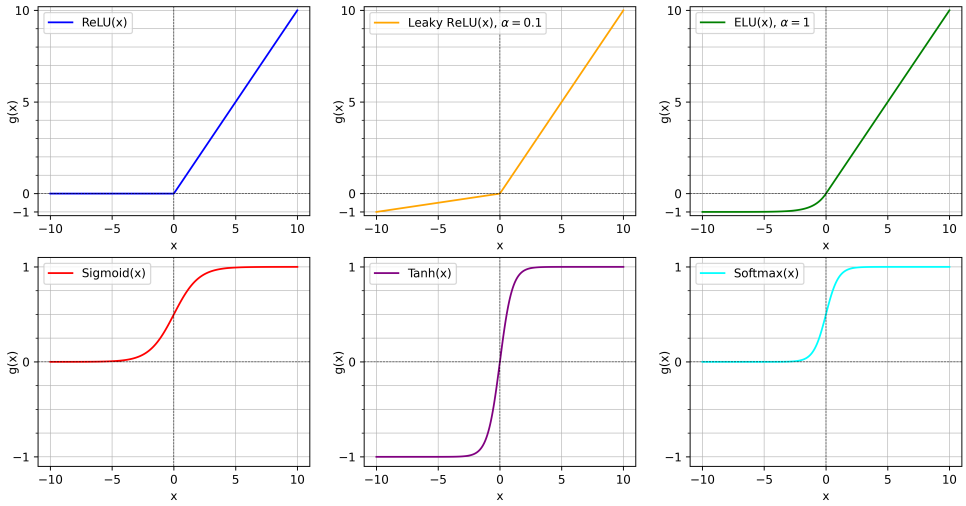


Figure 4.4: Plots of six commonly used activation functions in neural networks. Top row: ReLU, Leaky ReLU, and ELU. Bottom row: Sigmoid, Tanh, and Softmax.

4.3. Building Blocks of Neural Network

In this section, we will introduce several standard neural network architectures that may be used for time series forecasting but can also serve as building blocks for more complex architectures later.

4.3.1. Multilayer perceptron (MLP)

In deep learning, a multilayer perceptron (MLP) refers to a fully connected feedforward neural network. A neural network may only consist of an MLP, but the MLP can also be part of a larger network architecture. For example, the MLP can serve as the final classification layer or as a component that processes features extracted by other layers.

Definition 4.14. The output of a linear **multilayer perceptron** (MLP) layer with input vector $\boldsymbol{x}$ is given by

$$\texttt{MLP}(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

where $\boldsymbol{W}$ and $\boldsymbol{b}$ are the weights and biases of the nodes in the MLP.

Even though MLP's are often nonlinear, making it hard to express them in a simple form, we can still describe the network by its weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$. We will also refer to the output by $\texttt{MLP}(\boldsymbol{x})$.

4.3.2. Normalization

Normalization is a crucial preprocessing step in neural network modeling, especially for time series forecasting. It serves to stabilize learning and accelerate convergence by making sure that the input features to the network are on a similar scale.

A common technique in deep learning is batch normalization [19]. In the context of neural networks, a batch refers to a group of data samples that are processed together during a single training iteration. So, when forecasting time series, this form of normalization operates over all time series that are grouped in a batch. Here, each batch has its own statistics for normalization.

Another form of normalization is layer normalization [2], which normalizes the input features within each individual layer of the neural network. It calculates normalization statistics based on the summed inputs to the neurons within a hidden layer. Under layer normalization, all the hidden units in a layer share the same normalization terms, but different training batches can have different normalization terms.

Unlike the other two methods, temporal normalization [33] specifically targets the temporal dimensions of the data. This means that each time series will be scaled individually with its own normalization statistics. This allows for focusing on time-dependent fluctuations unique to each series.

The dimensions targeted by the normalization techniques are visually demonstrated in Figure 4.5.

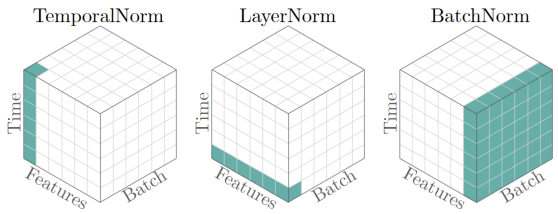


Figure 4.5: From left to right: temporal normalization, layer normalization, and batch normalization. The highlighted entries indicate the normalization statistics [33].

Some models, like TFT [22] and DeepAR [44], already implement scale-robust learning, meaning these models have strategies within their architectures that automatically manage variations in the scale of the data. Other models might need additional normalization.

Below, we will define some common normalization functions. Consider the following notation. Say we have data $\boldsymbol{x}$, then $\boldsymbol{x}_{[i][:t][c]}$ represents that data for feature/variable c with batch index i, up till time t.

Standard normalization standardizes the data by subtracting the mean and scaling by the variance.

Definition 4.15. Standard normalization [33] adjusts the data to have a mean of zero and a standard deviation of one, transforming $\boldsymbol{x}_{[i][:t][c]}$ using:

$$\boldsymbol{z} = \frac{(\boldsymbol{x}_{[i][:t][c]} - \overline{\boldsymbol{x}}_{[i][c]})}{\hat{\boldsymbol{\sigma}}_{[i][c]}}, \tag{4.11}$$

where $\overline{\boldsymbol{x}}_{[i][c]}$ and $\hat{\boldsymbol{\sigma}}_{[i][c]}$ are the mean value and standard deviation of $\boldsymbol{x}$ for batch i and feature c.

Minmax normalization scales the data to a specified range, typically between 0 and 1.

Definition 4.16. Minmax Normalization [33] scales the input $\boldsymbol{x}_{[i][:t][c]}$ to the range [0, 1] using:

$$\boldsymbol{z} = \frac{(\boldsymbol{x}_{[i][:t][c]} - \min(\boldsymbol{x}_{[i][:t][c]})_{[i][c]})}{(\max(\boldsymbol{x}_{[i][:t][c]})_{[i][c]} - \min(\boldsymbol{x}_{[i][:t][c]})_{[i][c]})}. \tag{4.12}$$

Similarly, we can scale the input to the range $[-1, 1]$ using:

$$\boldsymbol{z} = 2\frac{(\boldsymbol{x}_{[i][:t][c]} - \min(\boldsymbol{x}_{[i][:t][c]})_{[i][c]})}{(\max(\boldsymbol{x}_{[i][:t][c]})_{[i][c]} - \min(\boldsymbol{x}_{[i][:t][c]})_{[i][c]})} - 1 \tag{4.13}$$

Robust normalization scales the data similarly to the standard normalization, but using the median and mean absolute deviation instead of the mean and standard deviation. This method is particularly useful when outliers negatively influence the mean and/or variance. In this case study, however, due to the zero-inflated nature of the data, the median will be zero for most customers, which complicates the scaling process.

Definition 4.17. Robust normalization [33] transforms $\boldsymbol{x}_{[i][:t][c]}$ using:

$$\boldsymbol{z} = \frac{(\boldsymbol{x}_{[i][:t][c]} - \text{median}(\boldsymbol{x}_{[i][:t][c]}))}{\text{mad}(\boldsymbol{x})_{[i][:t][c]}} \tag{4.14}$$

where mad is the median absolute deviation $\text{mad}(\boldsymbol{x}) = \text{median}(|\boldsymbol{x} - \text{median}(x)|)$.

4.3.3. Pooling layers

Pooling layers are used in neural networks to reduce the dimension of feature maps. It does this by sliding a filter over the input data and summarizing the features lying within the region covered by the filter. It has two purposes. First, it reduces the number of parameters or weights, thereby lowering the required computational cost. Second, it helps to control overfitting in the network. An effective pooling method should extract only the relevant information while discarding unnecessary details.

First, we introduce the parameters stride and padding that, together with the filter size, control the size of the output map of the pooling layer.

Definition 4.18. Stride refers to the number of pixels/data points by which the filter moves across the input feature map during the pooling operation. A stride of $s = 1$ means that the filter shifts one pixel at a time, while a stride of 2 means it moves two pixels at a time.

Definition 4.19. Padding is the process of adding extra pixels/data around the border of the input feature map before applying the pooling operation. A padding of $p = 1$ add a layer of 1 pixel/data point around the border.

Given an input feature map of size I and a filter of size f with stride s and padding p. The size of the output map, O, is then given by

$$O = (I - f + 2p)/s + 1.$$

There are multiple ways to implement a pooling layer in a neural network. Here, we will only discuss two examples to sketch the general idea.

Definition 4.20. A **max pooling** layer takes the maximum value from each patch of the feature map that the filter overlaps.

Assume a 2D input matrix $\boldsymbol{X}$, a filter size of $f \times f$ and stride s, The output of the `MaxPool` layer is given by

$$\texttt{MaxPool}(X \mid f, s)_{i,j} = \max(X_{is:is+f-1,js:js+f-1}).$$

Definition 4.21. An average pooling layers takes the average value from each patch of the feature map that the filter overlaps.

Assume a 2D input matrix $\boldsymbol{X}$, a filter size of $f \times f$ and stride s, The output of the `AvgPool` layer is given by

$$\texttt{AvgPool}(X \mid f, s)_{i,j} = \frac{1}{f^2} \sum_{k=is}^{is+f-1} \sum_{k=js}^{js+f-1} X_{k,l}.$$

4.3.4. Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) are a class of neural networks designed to process structured grid-like data, such as images. They leverage the spatial hierarchies in the data to learn patterns at various levels of abstraction.

A convolutional neural network typically consists of a combination of convolutional layers, pooling layers, and fully connected layers. A convolutional layer operates similarly to a pooling layer but does not necessarily reduce the dimensions of a feature map. Instead, a convolutional layer applies the convolution operation on the input feature map using a filter (or kernel) to extract relevant features. The filter scans the input and produces a corresponding feature map, highlighting specific patterns.

CNNs are primarily designed for processing spatial data, but may be applied to time series data. It is important to note two limitation. First, CNNs require inputs of the same size due to the use of fixed filter sizes, limiting the ability to analyse time series of different lengths. Also, they have limitations when it comes to capturing long-term dependencies. The filters focus on a small, localized portion of the input data. While this is effective for capturing local patterns and features, it limits the network's ability to consider long-term dependencies across the entire sequence. CNNs also do not have a built-in mechanism for maintaining a memory of previous inputs, so they cannot learn from any sequential patterns.

4.3.5. Recurrent Neural Network (RNN)

Recurrent Neural Networks (RNNs) are a class of neural networks specifically designed to handle sequential data by maintaining a form of memory across time steps. Unlike feedforward networks, RNNs have connections that loop back on themselves, allowing them to retain information from previous inputs. This makes RNNs particularly suitable for applications where the order of the input data is crucial, like time series or natural language processing.

The architecture of an RNN consists of a series of recurrent layers, where each layer processes an input sequence one time step at a time. At each time step, the RNN takes the current input and combines it with the hidden state from the previous time step, allowing the network to incorporate information from earlier inputs sequentially. A schematic overview of an RNN is given in Figure 4.6.

Definition 4.22. The output $\boldsymbol{h}_t$ of a **Recurrent Neural Network** (RNN) at time step t is given by

$$\boldsymbol{h}_t = \mathtt{A}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}),$$

where $\boldsymbol{h}_{t-1}$ is the hidden state at time $t-1$ and $\boldsymbol{x}_t$ is the input vector at time t. In the RNN, $\mathtt{A}(.)$ can be a single node with activation function or a more complex network.

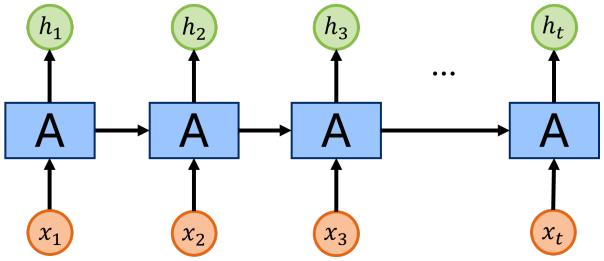


Figure 4.6: Schematic overview of a Recurrent Neural Network (RNN). An RNN is composed of multiple recurrent hidden layers, where the output of each hidden layer serves as the input to the next. Here, $\mathtt{A}(.)$ can be a single neuron or a more complex neural network itself.

When the RNN has too many layers, training of the network may become impossible. During back-propagation, the gradients of the earlier layers are calculated by multiplying the gradients of the later layers. If the gradients of these later layers are less than one, their multiplication can lead to a gradient approaching zero. Conversely, if the gradients are greater than one, they can grow excessively large, resulting in exploding gradients. This phenomenon is known as the vanishing or exploding gradient problem.

4.3.6. Long Short-Term Memory (LSTM)

The LSTM is a type of recurrent neural network model that is designed to solve the vanishing or exploding gradient problem. In a LSTM, the recurring cell $\mathtt{A}(.)$ has two hidden states instead of one. As the name of the model suggests, one is for keeping track of a short term memory and the other is for keeping track of a long term memory. The architecture is summarized in Figure 4.7.

As mentioned, the LSTM network has a long term memory, called the cell state c_t, and short term memory, called the hidden state h_t. An LSTM cell $\mathtt{A}(.)$ starts by deciding whether to keep or throw away the cell state. This is decided by a ”forget gate layer”, consisting of an $\mathtt{MLP}(.)$ together with a sigmoid function on the hidden state h_{t-1} and new information x_t. The forget parameter $f_t \in [0, 1]$ defining how much information to keep is given by

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

The next step is to decide what new information to store in the cell state. An $\mathtt{MLP}(.)$ layer with tanh function first creates a candidate vector $\tilde{c}_t$ from the hidden state h_{t-1} and x_t. Then, similar to the forget parameter, we can define the input parameter $i_t \in [0, 1]$, that defines how much information of $\tilde{c}_t$ we want to keep. The input parameter and new candidate vector are given by

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i),$$
$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_C).$$

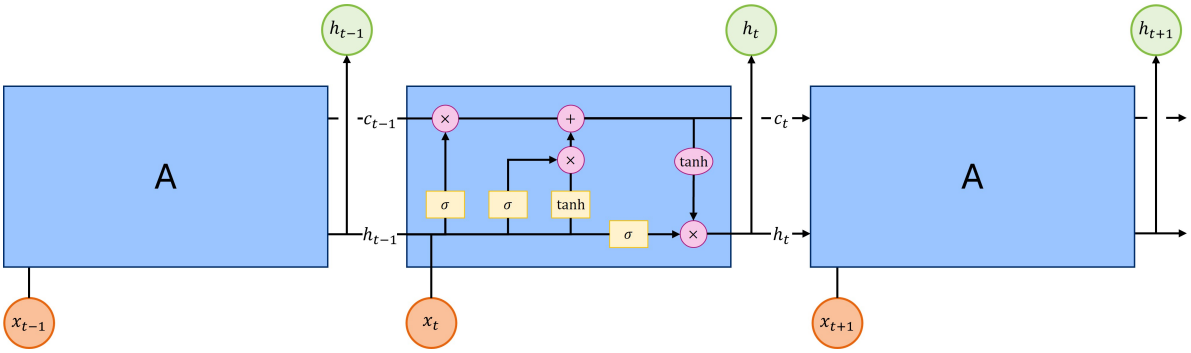


Figure 4.7: Architecture of an LSTM neural network, based on [31]. An LSTM extends a standard RNN by introducing two hidden states: c_t (long-term memory) and h_t (short-term memory). Here, pink circles represent pointwise operations, and yellow squares represent MLP layers.

Combining the forget layer and the input layer, we can update the long term memory in the following way

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t.$$

where $\odot$ is the Hadamard product. The LSTM cell outputs the hidden state h_t. In the last step, the output is updated using a combination of the previous hidden state h_{t-1}, the new information x_t and the updated cell state c_t in the following way

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o),$$
$$h_t = o_t \odot \tanh(C_t).$$

The LSTM is already quite capable of forecasting time series data. However, training an LSTM network still requires to to train the network on individual series. We will see how the LSTM can be used as a building block for more complex architectures later on.

4.3.7. Sequence to Sequence (Seq2Seq)

Sequence to sequence (Seq2Seq) models are a neural network architecture used for tasks that involve transforming one sequence into another. This approach allows for handling input and output sequences of variable-length, The original Seq2Seq model was introduced by Google in 2014 [47] and was applied to translate English text to French.

The Seq2Seq architecture uses a series of RNN layers to encode the input sequence into a context vector. Another series of RNN layers is then used to decode this vector into the target sequence. Often, the LSTM is used as both encoder and decoder. A schematic overview of an Seq2Seq is illustrated in Figure 4.8.

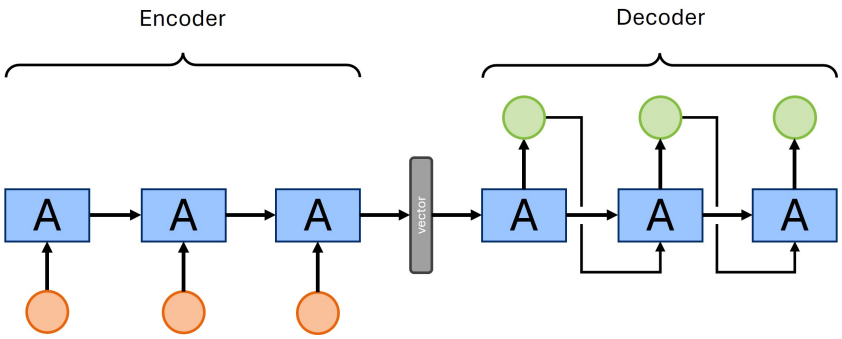


Figure 4.8: Seq2Seq architecture, with the encoder shown on the left and the decoder on the right. Here, the encoder processes the input sequence and compresses it into a fixed-length context vector, which is then used by the decoder to generate the output sequence.

The Seq2Seq architecture is popular for several reasons. First, it enables the model to handle sequences of varying lengths. Second, by separating the encoding and decoding processes, the model

can learn to represent the input sequence effectively while generating the output sequence in a flexible manner. This architecture has become foundational in natural language processing, and we will see it later in neural networks for time series.

In the context of time series forecasting, the number of encoder layers, encode length, defines the lookback window and the number of decoder layers, decoder length, defines the forecasting horizon. So, this strucutre can be used to predict time series for a given horizon H. However, since the forecast of the previous time point is fed back as input, it may be possible that errors propagate though the forecast.

4.3.8. Attention Mechanisms

Google improved on the Seq2Seq model for natural language processing even further. In 2017, they introduced attention mechanisms in the paper ”Attention is all you need” [50]. The problem was that in a Seq2Seq model, the encoder only outputs the final hidden state to the decoder. Therefore, the model has trouble decoding longer sequences. They extended the Seq2Seq architecture to output the hidden state of every RNN layer in the encoder to the decoder. Then, at every decoding step, the attention mechanism tells the decoder which part of the encoded sequence to focus on. They called this architecture a ”Transformer”.

The idea behind the attention mechanism is that it scales values $\boldsymbol{V}$ based on the similarities between keys $\boldsymbol{K}$ and queries $\boldsymbol{Q}$. In the Transformer model, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence.

Definition 4.23. The output of a **single-head attention mechanism** is given by

$$\texttt{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = f(\boldsymbol{Q}, \boldsymbol{K}) \cdot \boldsymbol{V}$$

where $f(.)$ is a normalization function, e.g. the Softmax function.

To further enhance the learning capacity of this mechanism, a multi-head attention mechanism was also proposed in [50]. This combines the output of multiple single-head attentions mechanisms.

Definition 4.24. The output of a **multi-head attention mechanism** is given by

$$\texttt{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \begin{bmatrix} \boldsymbol{H}_1 & \dots & \boldsymbol{H}_{m_H} \end{bmatrix} \boldsymbol{W}_H,$$

with

$$\boldsymbol{H}_h = \texttt{Attention}(\boldsymbol{Q}\boldsymbol{W}_Q^{(h)}, \boldsymbol{K}\boldsymbol{W}_K^{(h)}, \boldsymbol{V}\boldsymbol{W}_V^{(h)})$$

where $\boldsymbol{W}_Q^{(h)}, \boldsymbol{W}_K^{(h)}, \boldsymbol{W}_V^{(h)}$ are head-specific weights for keys, queries and values and $\boldsymbol{W}_H^{(h)}$ linearly combines the outputs from all heads $\boldsymbol{H_h}$.

Both the single head and multi-head attention mechanisms are illustrated in Figure 4.9.

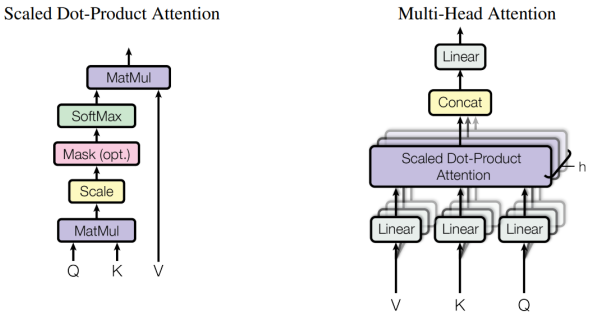


Figure 4.9: Visualization of attention mechanisms as introduced in [50]. (left) The standard attention mechanism, $\texttt{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$, computes weighted combinations of value vectors based on similarity between queries and keys. (right) The multi-head attention mechanism, $\texttt{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$, applies several attention layers in parallel.

4.4. State-of-the-Art Neural Network Models for Time Series Forecasting

Building upon the foundational architectures discussed in the previous section, we will now focus on more advanced neural network models. These state-of-the-art approaches have complex mechanisms designed to improve the predictive performance by training on multiple time series simultaneously. Notably, we will explore Amazon's DeepAR (2019), a global forecasting model using autoregressive recurrent networks which provides probabilistic forecasts. We will also examine Google's Temporal Fusion Transformer (2020), a model that provides interpretable forecasts using attention mechanisms. Lastly, we will discuss N-HiTS (2022), a model designed by Nixtla that uses hierarchical interpolation and promises improvements in accuracy and computation time over the transformer models.

4.4.1. DeepAR

DeepAR is an autoregressive recurrent neural network architecture introduced by Amazon in 2019 [44]. The network is able to produce probabilistic forecasts by training an autoregressive recurrent network on a large number of related time series. An important feature of DeepAR is that it is a global model. So, the network learns seasonal patterns and dependencies based on given covariates across multiple time series. As a result, by learning from similar items, DeepAR is able to provide forecasts for time series with little history, which would not be possible for single-time series models. Also, DeepAR is able to make probabilistic forecasts by using Monte Carlo samples [44].

Say we have N time series $y_{i,t}$, $i = 1, \dots, N$. Denote vector $\boldsymbol{y}_t = \begin{pmatrix} y_{1,t} & \cdots & y_{n,t} \end{pmatrix}^T$. Let $\boldsymbol{x}^{(s)}$ be the static exogenous variable and $\boldsymbol{x}_t^{(f)}$ be the future exogenous variables available at the time of prediction. Given the "past" time series $\boldsymbol{y}_{[1:t]} = [\boldsymbol{y}_1, \dots, \boldsymbol{y}_t]$, we are interested in the "future" time series $\boldsymbol{y}_{[t+1:t+H]} = [\boldsymbol{y}_t, \dots, \boldsymbol{y}_{t+H}]$ with forecasting horizon H. DeepAR aims to model the conditional distribution

$$\mathbb{P}(\boldsymbol{y}_{[t+1:t+H]} \mid \boldsymbol{y}_{[1:t]}, \boldsymbol{x}^{(f)}_{[1:t+H]}, \boldsymbol{x}^{(s)}).$$

The Architecture

The DeepAR model uses the encoder-decoder structure described in Section 4.3.7, making it a recurrent neural network. The model is autoregressive since it takes $\boldsymbol{y}_{t-1}$ as input for the next time step. At every time step, the RNN has output

$$\boldsymbol{h}_t = \mathtt{A}(\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x}_t),$$

where $\boldsymbol{x}_t = (\boldsymbol{x}_t^{(f)}, \boldsymbol{x}^{(s)})$ and $\mathtt{A}$ is an LSTM layer. The output of the LSTM layer at every time step is used as input for a fully connected layer. This MLP estimates the distribution parameters θ_t of a distribution function that is chosen before training. Then, the likelihood function $\ell(y \mid \theta)$ can be used to generate Monte Carlo samples. This likelihood function is chosen before training and can be for example a Gaussian likelihood for real-valued data or a negative-binomial likelihood for positive count data.

For example, for a Gaussian distribution with parameters $\theta = (\mu, \sigma)$, the likelihood is given by

$$\ell_G(y \mid \mu, \sigma) = (2\pi\sigma^2)^{-\frac{1}{2}} \exp(-(y-\mu)^2/(2\sigma^2)),$$

where

$$\mu(\boldsymbol{h}_t) = \boldsymbol{w}_\mu^T \boldsymbol{h}_t + b_\mu \quad \text{and} \quad \sigma(\boldsymbol{h}_t) = \log(1 + \exp(\boldsymbol{w}_\sigma^T \boldsymbol{h}_t + b_\sigma)).$$

The architecture model is illustrated in Figure 4.10.

Training

During training, DeepAR generates multiple instances from each time series by selecting windows with different starting points. The encoder and decoder length are kept fixed across training.

All parameters of the model, both for the RNN $\mathtt{A}(.)$ and the MLP, are learned by maximizing the log-likelihood

$$\mathcal{L} = \sum_{i=1}^{N} \sum_{t=t_0}^{t_0+H} \log \ell(z_{i,t} \mid \theta(\boldsymbol{h}_{i,t})), \tag{4.15}$$

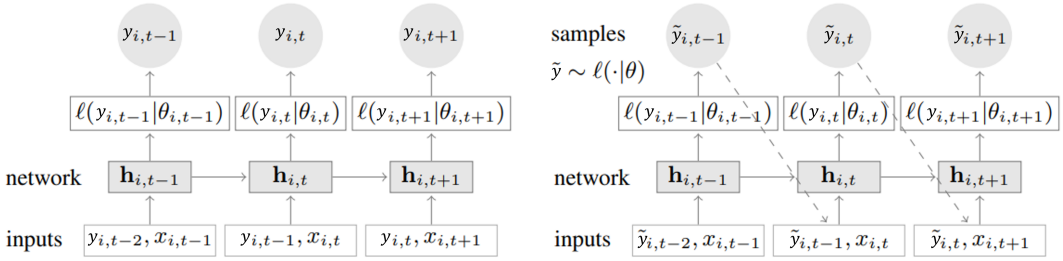


Figure 4.10: DeepAR architechture, adapted from [44]. Here, the variable z from the original paper has been replaced with y for consistency. The left side illustrates the encoder structure, while the right side shows the decoder structure. The output of the RNN is fitted to a distribution function to enable probabilistic forecasting.

via stochastic gradient descent. Here, t_0 is the first time of the decoded sequence.

The DeepAR model is proven to work well on a wide range of data sets, starting from a few hundred series, with little to no hyper parameter tuning [44].

4.4.2. Temporal Fusion Transformer

The Temporal Fusion Transformer (TFT) is an advanced neural network architecture designed for multi-horizon forecasting, introduced by Google in 2020 [22]. This section is based on the original paper by Google. The TFT is able to learn patterns from multiple time series, including static covariates, known future inputs, and exogenous time series. Unlike traditional deep learning models that function as "black boxes", the TFT provides interpretable insights into how these various inputs interact with the target variable. The output prediction is, just like DeepAR, a probabilistic forecast with quantile predictions. The estimated conditional distribution is given by

$$\mathbb{P}(\boldsymbol{y}_{[t+1:t+H]} \mid \boldsymbol{y}_{[1:t]}, \boldsymbol{x}^{(h)}_{[1:t]}, \boldsymbol{x}^{(f)}_{[1:t+H]}, \boldsymbol{x}^{(s)}),$$

where $\boldsymbol{y}_{[t+1:t+H]}$ is the predicted variable of interest for horizon H, $\boldsymbol{y}_{[1:t]}$ is the known past of the variable of interest, $\boldsymbol{x}^{(h)}_{[1:t]}$ are past covariates known until time t, $\boldsymbol{x}^{(f)}_{[1:t+H]}$ are future covariates that are also known for the forecasting horizon and $\boldsymbol{x}^{(s)}$ are the static covariates.

Because of its complex architecture, the TFT can, just like a random forest, identify which inputs are most significant for the predictions, giving insights into the model's decision-making process. Additionally, it can recognize and visualize temporal patterns and significant periods on a global level.

The Architecture

The architecture includes specialized components for feature selection and gating layers to filter out irrelevant inputs. We will briefly explain them here.

- **Gating Mechanisms**: The TFT uses so called Gated Residual Networks (GRNs), as shown in Figure 4.11, to determine the relationship between exogenous inputs $\boldsymbol{x}$ and targets $\boldsymbol{y}$. These mechanisms enable the model to skip over unnecessary features or parts of the network that are not required for a given dataset. The `GRN` network is defined by

$$\begin{aligned}
\texttt{GRN}_\omega(\boldsymbol{a}, \boldsymbol{c}) &= \texttt{LayerNorm}(\boldsymbol{a} + \texttt{GLU}_\omega(\boldsymbol{\eta}_1)) \\
\boldsymbol{\eta}_1 &= \boldsymbol{W}_{1,\omega}\boldsymbol{\eta}_2 + \boldsymbol{b}_{1,\omega} \\
\boldsymbol{\eta}_2 &= \texttt{ELU}(\boldsymbol{W}_{2,\omega}\boldsymbol{a} + \boldsymbol{W}_{3,\omega}\boldsymbol{c} + \boldsymbol{b}_{2,\omega}),
\end{aligned}$$

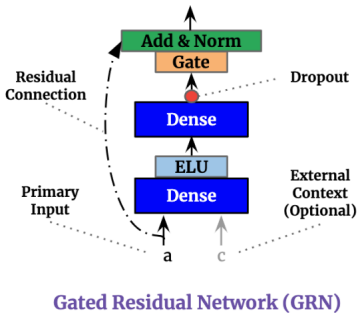


Figure 4.11: Gating mechanism used in the TFT architecture [22].

where $\boldsymbol{a}$ is the primary input and $\boldsymbol{c}$ is an optional context vector. $\boldsymbol{W}_\omega$ and $\boldsymbol{b}_\omega$ are the weights and biases of the dense MLP layers. $\boldsymbol{\eta}_{1,2}$ are results of intermediate layers. $\texttt{ELU}(.)$ is the Exponential Linear Unit activation function and $\texttt{LayerNorm}(.)$ is a standard layer normalization.

$\texttt{GLU}(.)$ is a Gating Linear Unit [9], defined by

$$\texttt{GLU}_\omega(\boldsymbol{\gamma}) = \sigma(\boldsymbol{W}_{4,\omega}\boldsymbol{\gamma} + \boldsymbol{b}_{4,\omega}) \odot (\boldsymbol{W}_{5,\omega}\boldsymbol{\gamma} + \boldsymbol{b}_{5,\omega})$$

where $\sigma(.)$ is the sigmoid activation function and $\odot$ is the element-wise Hadamard product.

During training, dropout is applied before the gating layer. Dropout means randomly setting a fraction of the neurons to zero. This means that during each training iteration, a subset of neurons is ”dropped out” or ignored, which prevents the network from becoming overly reliant on any particular features.

- **Static Covariate Encoders**: The TFT integrates static metadata by using seperate $\texttt{GRN}$ encoders that create four different context vectors, $\boldsymbol{c}_s$, $\boldsymbol{c}_e$, $\boldsymbol{c}_c$ and $\boldsymbol{c}_h$. These context vectors are used as input at different locations of the temporal fusion decoder. Specifically, for temporal variable selection ($\boldsymbol{c}_s$), local processing of temporal features ($\boldsymbol{c}_c, \boldsymbol{c}_h$) and enriching of temporal features with static information ($\boldsymbol{c}_e$).
- **Variable Selection Networks**: To determine the relevance of the many input variables on the target $\boldsymbol{y}$, variable selection is performed on both static and time-dependent covariates. These input covariates are transformed by the TFT itself. For the categorical variables, entity embeddings are used to map them on dense vectors and continuous variables undergo a linear transformation.

  W.l.o.g. consider the variable selection network for past inputs. Let $\boldsymbol{\xi}_t^{(j)}$ denote the transformed input of the j-th variable at time t, with $\boldsymbol{\Xi}_t = \left[\boldsymbol{\xi}_t^{(1)^T} \quad \dots \quad \boldsymbol{\xi}_t^{(2)^T}\right]^T$ being the flattened vector of all past inputs at time t. The weights of variable selection are generated by using a $\texttt{GRN}$ with $\boldsymbol{\Xi}_t$ and context vector $\boldsymbol{c}_s$ as input. So,

$$\boldsymbol{v}_{\chi_t} = \texttt{Softmax}(\texttt{GRN}_{\nu_\chi}(\boldsymbol{\Xi}_t, \boldsymbol{c}_s))$$

  where $\boldsymbol{v}_{\chi_t}$ is the vector variable selection and $\boldsymbol{c}_s$ is obtained from a static variable encoder. At each time step, an additional $\texttt{GRN}$ is applied on every $\boldsymbol{\xi}_t^{(j)}$

$$\tilde{\boldsymbol{\xi}}^{(j)} = \texttt{GRN}_{\tilde{\xi}(j)}\left(\boldsymbol{\xi}^{(j)}\right)$$

  where $\tilde{\boldsymbol{\xi}}_t^{(j)}$ is the processed feature vector for variable j. Processed features are then weighted by their variable selection weights,

$$\tilde{\boldsymbol{\xi}} = \sum_{j=1}^{m_\chi} v_{\chi_t}^{(j)} \tilde{\boldsymbol{\xi}}^{(j)}$$

  where $v_{\chi_t}^{(j)}$ is the j-th element of vector $\boldsymbol{v}_{\chi_t}$.

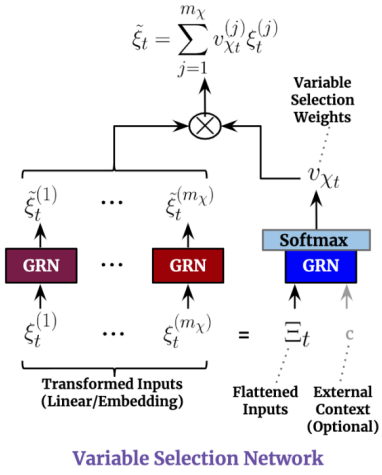


Figure 4.12: Variable selection method used in TFT architecture [22].

- **Seq2Seq layer**: Similar to DeepAR, the TFT uses an encoder-decoder structure. Here, $\tilde{\boldsymbol{\xi}}_{t-k:t}$ are feeded into the encoder and $\tilde{\boldsymbol{\xi}}_{t+1:t+H}$ is used in the decoder, where k is the encoder length and H is the forecasting horizon. The context vectors $\boldsymbol{c}_c$ and $\boldsymbol{c}_h$ are used in the first LSTM to initialize the cell state and the hidden state resp.

  The output of the decoder layer, denoted by $\phi(t, n)$ with $n \in [-k, H]$ being a position index. This output is used as input into the temporal fusion decoder.

Additionally, the output $\phi(t,n)$ is combined with features $\tilde{\boldsymbol{\xi}}_{t+n}$ using a gated skip layer

$$\tilde{\phi}(t,n) = \texttt{LayerNorm}\left(\tilde{\boldsymbol{\xi}}_{t+n} + \texttt{GLU}_{\tilde{\phi}}(\phi(t,n))\right).$$

- **Temporal Fusion Decoder**: This part can be seen as the brain of the network, and is mostly what makes the TFT special over the DeepAR model.

  First, the output from the Seq2Seq layer is further enriched using static covariates,

  $$\boldsymbol{\theta}(t,n) = \texttt{GRN}_{\theta}\left(\tilde{\boldsymbol{\phi}}(t,n), \boldsymbol{c}_e\right)$$

  where weights of $\texttt{GRN}_{\theta}$ are shared across entire layer, and $\boldsymbol{c}_e$ is a context vector from the static covariate encoder. Then all static-enriched temporal features are grouped into a single matrix $\boldsymbol{\Theta}(t) = \begin{bmatrix}\boldsymbol{\theta}(t,-k) & \dots & \boldsymbol{\theta}(t,H)\end{bmatrix}^T$.

  Next, the TFT applies a self-attention mechanism to learn long-term relationships across different time steps. This is a modified version from the multi-head attention introduced in Section 4.3.8 to improve explainability. The idea is to make sure each head has the same values such that each can learn different temporal patterns, but still contribute to the same set of input features.

  Definition 4.25. The interpretable multi-head attention is given by

  $$\texttt{InterpretableMultiHead}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = \tilde{\boldsymbol{H}}\boldsymbol{W}_H$$

  $$\tilde{\boldsymbol{H}} = \frac{1}{H}\sum_{h=1}^{m_H}\texttt{Attention}(\boldsymbol{Q}\boldsymbol{W}_Q^{(h)}, \boldsymbol{K}\boldsymbol{W}_K^{(h)}, \boldsymbol{V}\boldsymbol{W}_V)$$

  where $\boldsymbol{W}_V$ are value weights shared across all heads.

  The interpretable multi-head attention is applied at each forecast time

  $$\boldsymbol{B}(t) = \texttt{InterpretableMultiHead}(\boldsymbol{\Theta}(t), \boldsymbol{\Theta}(t), \boldsymbol{\Theta}(t)),$$

  where $\boldsymbol{B}(t) = [\boldsymbol{\beta}(t,-k), \dots, \boldsymbol{\beta}(t,H)]$. Following the self-attention layer, an additional skip gating layer is applied,

  $$\boldsymbol{\delta}(t,n) = \texttt{LayerNorm}(\boldsymbol{\theta}(t,n) + \texttt{GLU}_{\delta}(\boldsymbol{\beta}(t,n)))$$

  An additional non-linear processing layer is applied to the outputs of the self-attention layer.

  $$\psi(t,n) = \texttt{GRN}_{\psi}(\boldsymbol{\delta}(t,n))$$

  where the weights of $\texttt{GRN}_{\psi}$ are shared across the whole layer. Finally, a gated residual connection which skips the entire temporal fusion decoder block is applied. This gives the model the possibility to skip the complexity if it is not necessary. The output is given by

  $$\tilde{\psi}(t,n) = \texttt{LayerNorm}\left(\tilde{\phi}(t,n) + \texttt{GLU}_{\tilde{\psi}}(\psi(t,n))\right).$$

- **Quantile predictions**: The quantile predictions of the TFT are generated by using a linear transformation of the output from the temporal fusion decoder

  $$\hat{y}(q,t,\tau) = \boldsymbol{W}_q\tilde{\boldsymbol{\psi}}(t,\tau) + b_q,$$

  where $\boldsymbol{W}_q$, b_q are the weights and bias for the specified quantile q. The forecasts are only generated for the decoder sequence.

The components described above combine into the full TFT neural network, as presented in Figure 4.13.

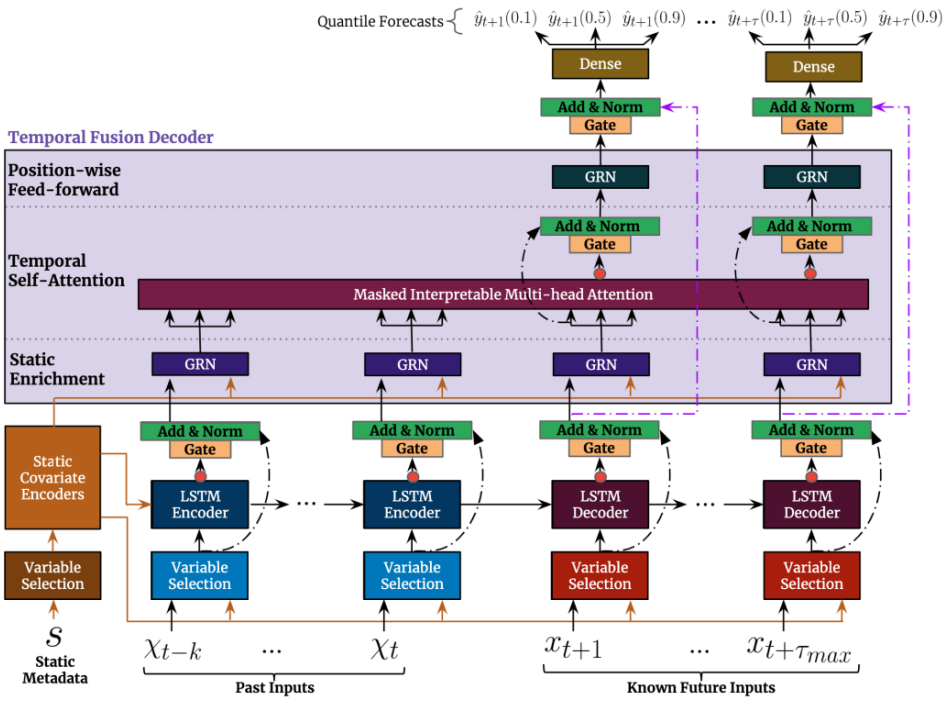


Figure 4.13: TFT architecture from [22]. TFT inputs static, time past, and future covariates. It applies variable selection to identify relevant features at each time step. Gating mechanisms and skip connections allow efficient information flow through the network. Time-dependent relationships are captured using LSTM layers for local processing, while a multi-head attention mechanism integrates information across all time steps.

Training

The TFT is trained by minimizing the quantile loss, summed across all quantile outputs

$$\mathcal{L} = \sum_{y_t \in \Omega} \sum_{q \in \mathcal{Q}} \sum_{\tau=1}^{H} \frac{QL(y_t, \hat{y}(q, t-\tau, \tau), q)}{M \cdot H}, \tag{4.16}$$

where Ω is the domain of the training data containing M samples and $\mathcal{Q}$ is set of output quantiles. The quantile loss QL for a quantile q, for example 80% or 90%, is given by

$$QL(y, \hat{y}, q) = q(y - \hat{y})_+ + (1 - q)(\hat{y} - y)_+,$$

where $(.)_+$ denotes $\max(0, .)$.

Note that because of the complexity of the network, it is computationally more expensive to train than the other networks presented in this research.

4.4.3. N-HiTS

N-HiTS is a more recent neural network model for long-horizon forecasting designed by Nixtla in 2022 [6]. The architecture is based on the Neural Basis Expansion Analysis (N-BEATS) model [35], which utilizes a stack of fully connected feedforward networks organized into blocks. N-HiTS applies this same principle in a hierarchical way to focus on different parts of the time series. The developers of the model promise a more efficient approximation of long horizon forecasts. Experiments show that N-HiTS can outperform state-of-the-art Transformer architectures, while significantly reducing computational time by up to 50 times [6].

Architecture

The workings of N-HiTS can, in some way, be compared to a Fourier decomposition where each frequency (basis function) is predicted locally by its own neural network, called a stack. Each stack s is build from a grouped number of B blocks and each block consists of an MLP, which learns to produce coefficients for the backcast and forecast outputs of its basis. The full architecture is presented in Figure 4.14.

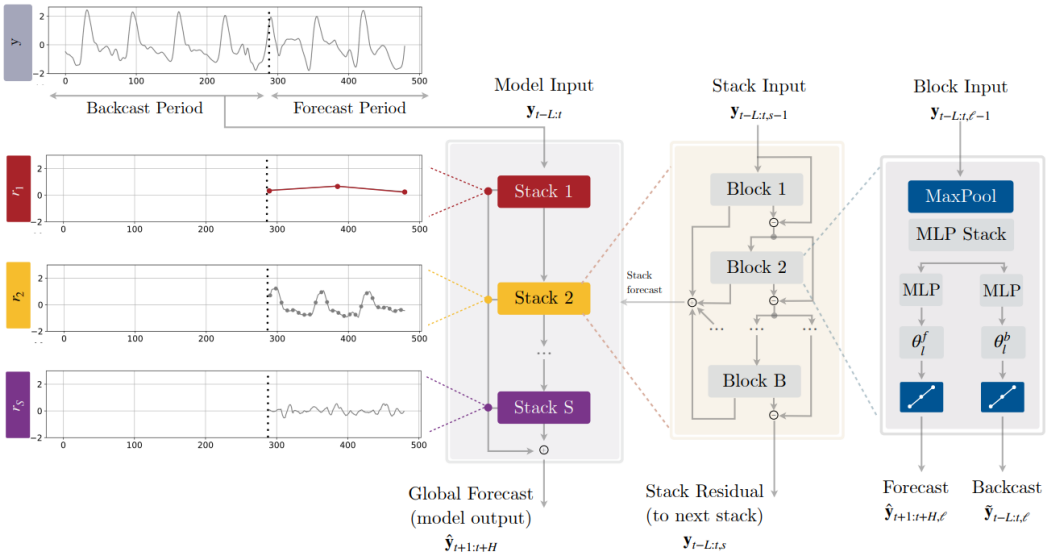


Figure 4.14: N-HiTS architecture from [6]. The model is composed of several stacks, each containing multiple MLP blocks. Each block generates both a backcast and a forecast, where the backcast is subtracted from the input for the next block. Hierarchical interpolation, with varying expressiveness ratios across blocks, allows the model to specialize in different frequencies of the time series.

The model starts with an input of the historic data of the target time series, $\boldsymbol{y}_{t-L:t}$, in the first block of the first stack. We denote this a block by $[s, l]$ for its stack $s \in \{1, \dots S\}$ and block in the stack $l \in \{1, \dots B\}$. For simplicity, we will for now focus on a single stack and only denote the block number l.

Given block l, it starts with a `MaxPool` layer with kernel k_l. Here, a larger kernel cuts more high-frequency input, so forces the block to focus on lower frequencies. One can already see here that if each block has a different kernel size, every block focuses on a different frequency of the time series. Challu et al. calls this multi-rate signal sampling. The operation is given by

$$\boldsymbol{y}^{(p)}_{t-L:t,l} = \texttt{MaxPool}(\boldsymbol{y}_{t-L:t,l}, k_l).$$

After the pooling layer, an MLP layer performs non-linear regression to produce interpolation coefficients. Both the forward interpolation coefficients $\boldsymbol{\theta}^f_l$ and backward interpolation coefficients $\boldsymbol{\theta}^b_l$ are derived from a hidden state $\boldsymbol{h}_l$, which is linearly projected. We have

$$\begin{aligned}
\boldsymbol{h}_l &= \texttt{MLP}_l\left(\boldsymbol{y}^{(p)}_{t-L:t,l}\right), \\
\boldsymbol{\theta}^f_l &= \texttt{Linear}_f(\boldsymbol{h}_l), \\
\boldsymbol{\theta}^b_l &= \texttt{Linear}_b(\boldsymbol{h}_l).
\end{aligned}$$

These coefficients are used to create a backcast $\tilde{\boldsymbol{y}}_{t-L:t,l}$ and forecast a $\hat{\boldsymbol{y}}_{t+1:t+H,l}$. The size dimensions of the coefficients are determined by the expressiveness ratio r_l of that block. It controls the number of parameters per unit of output time, i.e., $|\boldsymbol{\theta}^f_l| = \lceil r_l H \rceil$. Now, the fore- and backcast are determined using temporal interpolation with a predefined interpolation function $g(.)$,

$$\begin{aligned}
\hat{y}_{\tau,l} &= g(\tau, \boldsymbol{\theta}^f_l), \quad \forall \tau \in \{t+1, \dots, t+H\}, \\
\tilde{y}_{\tau,l} &= g(\tau, \boldsymbol{\theta}^b_l), \quad \forall \tau \in \{t-L, \dots, t\}.
\end{aligned}$$

The time partition for the interpolation is given by $\mathcal{T} = \{t+1, t+1+1/r_l, \ldots, t+H-1/r_l, t_H\}$.

The default setting of N-HiTS uses linear interpolation, which fits a straight line between the two nearest time points $t_1, t_2 \in \mathcal{T}$ surrounding τ. The interpolation function is then given by

$$g(\tau, \theta) = \left(\theta[t_1] + \frac{\theta[t_2] - \theta[t_1]}{t_2 - t_1}\right)(\tau - t_1) \quad \text{with} \quad t_1 = \arg\min_{t \in \mathcal{T}: t \leq \tau} \{\tau - t\},\ t_2 = t_1 + 1/r_l.$$

The resulting forecast and backcast of each block are output to the next parts. The backcast is subtracted from the input of the block, such that the next block can forecast another part of the time series

$$\boldsymbol{y}_{t-L:t,l+1} = \boldsymbol{y}_{t-L:t,l} - \tilde{\boldsymbol{y}}_{t-L:t,l}.$$

The subtracted part of the last block, the stack residual, is used as input for the next stack.

The forecasts of all blocks in the stack summed together to create the stack forecast.

$$\hat{\boldsymbol{y}}_{t+1:t+H} = \sum_{l=1}^{L} \hat{\boldsymbol{y}}_{t+1:t+H,l}.$$

The summed total of all stack forecasts form the global hierarchical forecast.

Each block specializes on its own scale of input and output signal, because of the different r_l and k_l. This clearly illustrates the hierarchical nature of the architecture. Blocks closer to the input have smaller r_l and larger k_l to first focus on the larger changes in the time series. Each stack can specialize in modeling a different known cycle of the time-series (weekly, daily etc.) using a matching r_l.

Training

The N-HiTS model does not have a specific loss function. Challu et al. (2022) tested the model using a simple MAE loss, however the model is also able to produce probabilistic forecasts when using a distribution loss (Equation (4.15)) or multi-quantile loss (Equation (4.16)).

Since different blocks focus on different frequencies of the time series, it is important that the hyperparameters r_l and k_l align with the frequencies or seasonality in the data.

Part II

Research Setup

5

Data Availability and Exploration

In this chapter, we explore the data provided by the wholesale distributor. Specifically, order data, vehicle data, and customer data. The available data also later determines the features we can use in training the time series models. Furthermore, we will investigate the customer behavior surrounding holidays. Due to the distributor not delivering on most holidays, customers often choose a substitute delivery a day before or after the holiday. Finally, we address the conversion from ordered volumes to roll containers. While the time series data is expressed in volumetric terms, orders are transported to the customer per roll container. So, the ordered volumes per customer need to be discretized to the volume of a roll container.

5.1. Available Data and Preprocessing

5.1.1. Order Data

In this case study, we are originally provided with order data from customers served by two transport centers situated in different cities across the Netherlands. However, for the purposes of this research, we focus on one transport center, the larger of the two.

Daily orders are specified at the product level. As requested by the client, these orders are aggregated by product stream, resulting in three distinct streams: frozen goods (FRZ), fresh products (FRSH), and dry groceries (DRY). Consequently, we end up with three time series for each customer.

Each order comes with a time window requested by the customer, as introduced in Section 2.2.3. For some, this window is restricted to 2 hours, while others may have a flexible time window that can span the entire day. Furthermore, each customer is linked to a specific address, which is essential for route planning.

5.1.2. Vehicle Data

At the transport centre of interest, various vehicles from different categories, such as B, C, and CE, are available. Within these categories, there are still some slight variations. Every vehicle is defined by capacity in terms of weight and volume. Each vehicle has a freezer and a non-freezer section, both with its own capacities. The available vehicles with their capacities are detailed in Appendix E.

Additionally, every truck has associated costs for operation. These costs include a constant base cost, variable costs per kilometer and per hour, and additional charges for overtime hours. The exact costs are detailed in Table B.1. The maximum working time of a truck is 8 hours per day, with normal operating hours scheduled between 04:30 and 18:00. Additional work-time regulations, as outlined by the client and Dutch law, including maximum working duration before a break and break duration, are specified in Table B.2.

5.1.3. Available Features

As explained in Section 3.2, it is possible to enhance the training of time series models by incorporating additional features. As discussed, features can be categorized into past, future, and static covariates. This section covers the covariates that are available in this research.

Past covariates are time-dependent features that are available only for previous time steps. In this research, we utilize daily temperature data from the KNMI [21] as a past covariate. Specifically, the measuring station closest to the transport center is selected.

Future covariates include information that can be predicted or is planned. For example, we can use seasonal variables, such as the day of the week, month, or year. Information on holidays is not provided by the client, so we consider Dutch national holidays obtained from the Python library `workalendar`.

Static covariates are attributes that remain constant over time for each time series. For this research, we have the customer ID, product type, market segment, and address. Customers are categorized into 18 distinct market segments, including restaurants, hotels, hospitals, events, and others. For geographic analysis, we can leverage the first four digits of the postal code to assess regional patterns within the city or the full six-digit postal code for patterns within an even smaller region.

5.1.4. Data Filtering

The available customer order data ranges from 2018-06-25 to 2023-06-24. It is important to acknowledge that a large portion of this dataset was significantly impacted by the COVID-19 pandemic, as can be seen in Figure 5.1. For instance, numerous restaurants were forced to close during this time, eliminating the necessity for them to place orders. Because the orders during the pandemic period cannot be compared to the post-COVID period in terms of ordered volume but also in terms of order consistency, it was decided to only use data from after 2022-03-21 for this research. At this date, all COVID measures had expired, and customer orders are considered to be back to normal [43].

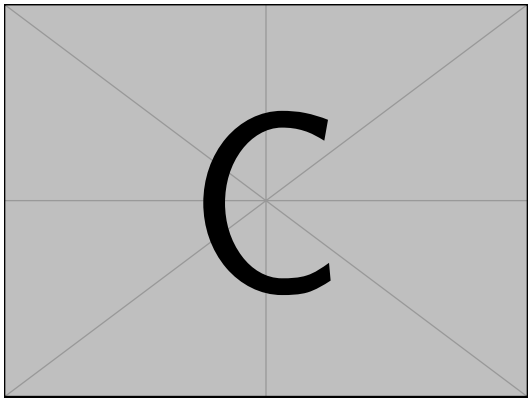


Figure 5.1: [*This figure contains confidential information and is therefore only available in the confidential appendix.*] Aggregated ordered volume per day, normalized by the maximum volume. The effect on the ordered volumes during the COVID period is clearly visible. The red line presents the cutoff date, 2022-03-21, used for training data.

It should be noted that the new dataset encompasses only about a year of data. This makes it harder to learn seasonal patterns over different years. Although the pre-COVID data might offer some insights, it was decided not to include it because it could be too outdated.

Outliers due to typos, such as 100 mL in the data being listed as 100 L, are removed by ORTEC. This research only considered the data that was filtered by ORTEC in the proof of concept.

5.2. Customer Behaviour around Holidays

Each customer has specific designated order days, outlined in a contract with the wholesale distributor. These order days are determined during the sales process. However, unfortunately, the planning department does not have access to this information. As a result, we must infer these patterns from customers' historical orders. It's important to note that customers do not place orders every designated day. For example, if a customer can order on Mondays, Wednesdays, and Fridays, they might order on Monday and Wednesday one week, and on Monday and Friday another week.

The exact business rules for each holiday are unknown in this research, but typically, the wholesale distributor does not deliver on national holidays. If a customer's designated order day coincides with a holiday, they are provided with a substitute day. During the week before or after the holiday, they can place orders on a day that is not their usual order day. This substitute day varies by customer and is not standardized. Customers with enough order days are less likely to utilize the substitute day, while those with fewer order days are more likely to use it.

A small exploratory study is conducted to examine customer behavior surrounding holidays. Here, it is assumed that a customer is a regular customer on a specific weekday if it has an order consistency higher than 90%. The goal is to identify which days customers prefer as substitutes for their designated order days when those coincide with holidays. An alternative day was counted only if it did not already qualify as a designated order day, specifically, if the customer's order consistency on that weekday was under 10%. Figure 5.2 shows the number of orders by customers that normally order on the weekday of the holiday and usually do not order on other weekdays. In this figure, all non-Sunday Dutch national holidays between 2022-03-21 and 2023-06-24 are considered, as the wholesale distributor does not deliver on Sundays.

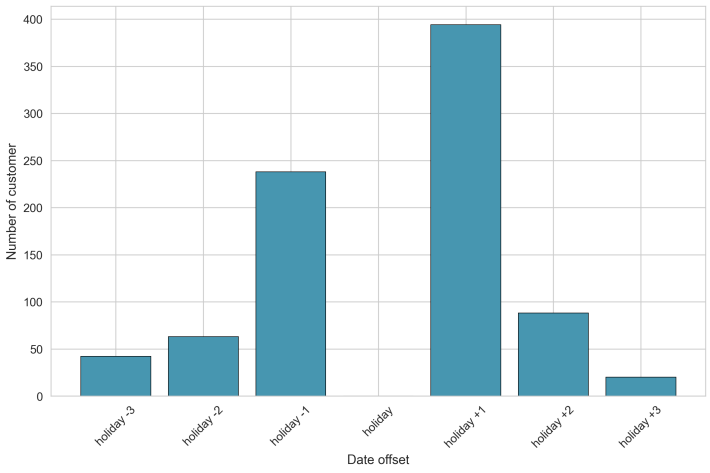


Figure 5.2: Number of orders 3 days before and after the holidays between 2022-03-21 and 2023-06-24 (13 in total) by customers that usually order on the weekday of the holiday (threshold of $> 90\%$) and not on the other weekdays (threshold of $< 10\%$). Most popular substitute day is the day before or the day after the holiday.

The frequency of substitutions on the day before the holiday is lower than the day after due to the fact that four out of thirteen holidays occur on Mondays, meaning the previous day (Sunday) is unavailable for deliveries. For holidays that are not on Monday, the day before the holiday is the preferred substitute. An example of the substitute days for Easter Monday and King's Day is presented in the Appendix, Figures A.1a and A.1b. It can be concluded that the most popular substitute day is either the day before or the day after the holiday.

In addition, it can be found from the data that the volume of orders increases around holidays. This can be explained because often holidays are busier days for customers, depending on the market segment. This will be reflected in the ordered volume before and after the holiday. It is found that the volume increases 1 or 2 days before and after the holiday.

5.3. Volume Conversion

The volume of orders is measured in liters. However, orders are transported to customers using roll containers. We are therefore interested in the number of roll containers required for transportation rather than the transported volume itself. It is assumed that a roll container can only be used for one customer, meaning that all customer orders need to be discretized to the volume of one or multiple roll containers.

When packing products into a roll container, it is often impossible to utilize the container's entire volume efficiently. For instance, packing circular objects typically results in unused space. To address this inefficiency, the client has provided volume conversion ratios, ρ, for each product group to convert product volume to effective volume. These ratios convert the product volume into the effective volume it occupies within a roll container. Due to different ways of packing or a different product mix, it might be possible for these conversion rates to differ across different depots. The exact conversion rates for each product group for the transport centre of interest are detailed in Appendix E. It is reasonable to expect that the conversion ratio of FRZ is the lowest as freezer products are often packed in boxes. In contrast, products from the FRSH group have the highest conversion ratio. This is primarily because items like fresh vegetables often can't be easily stacked and require additional space due to their irregular shapes. The volume conversion from product volumes to the effective volume is then given by the following equation:

$$v_{\text{effective}} = \rho_{\text{product}} \cdot v_{\text{product}}. \tag{5.1}$$

The wholesale distributor has two types of roll containers: one designed for transportation in the freezer compartment and another for the non-freezer section. Consequently, products from the DRY and FRSH groups can be transported on the same roll container. The volume of a roll container is denoted by V. The exact sizes of both types of roll containers are given in Appendix E.

Now, the number of roll containers required for a customer's freezer products, denoted as $\#RC_{\text{FRZ}}$, is computed by

$$\#RC_{\text{FRZ}} = \left\lceil \frac{\rho_{\text{FRZ}} \cdot v_{\text{FRZ}}}{V_{\text{FRZ}}} \right\rceil,$$

where ρ_{FRZ} represents the volume conversion rate for the FRZ group, v_{FRZ} is the volume of the ordered products, and V_{FRZ} is the size of a roll container for freezer items. Since products from the FRSH and DRY groups can be transported on the same roll container, the number of roll containers needed to transport a customer's non-frozen goods is calculated as

$$\#RC_{\text{DRY+FRSH}} = \left\lceil \frac{\rho_{\text{DRY}} \cdot v_{\text{DRY}} + \rho_{\text{FRSH}} \cdot v_{\text{FRSH}}}{V_{\text{DRY+FRSH}}} \right\rceil,$$

where ρ_{DRY} and ρ_{FRSH} are the conversion rates of the DRY and FRSH products respectively, v_{DRY} and v_{DRY} denote the respective volumes of the ordered products, and $V_{\text{DRY+FRSH}}$ represents the capacity of a roll container for transporting DRY and FRSH products.

6

Forecasting Approach

6.1. Forecasting Approach using Traditional Methods

This section outlines the decisions ORTEC made during the initial proof of concept for forecasting customer orders using traditional forecasting methods. It is essential to note that these decisions are not part of this research; they are provided here solely for reference. We begin by discussing the general concept of forecasting customer order data using a two-step approach. Next, we will specify the models utilized and the Python packages from which they originate. We will also address the features considered and the approach for hyperparameter tuning. Finally, we outline the criteria for selecting the optimal models. The end of this section includes a part on the implementation of hierarchical forecasting, which was not part of the initial proof of concept.

6.1.1. Two-Step Approach for Intermittent Time Series Forecasting

Forecasting intermittent order data can be challenging due to its highly non-linear and sporadic nature, which often makes traditional linear methods less effective. Croston's method [8] provides a solution to intermittent demand data by estimating the time intervals between demand and the average demand over the interval. While this may work for inventory scheduling, it does not give any information on the estimated day of delivery. Our objective is to forecast precise volumes for days when orders are placed and zero for days with no orders.

ORTEC addresses this by splitting the forecasting process into two distinct steps. First, we determine whether a customer will place an order on a specific day using a binary classification model. Then, if an order is to be placed, we predict the volume of that order using a regression model. For the first question, 2 models are tested: logistic regression and a random forest classifier model. For the second question, 4 models are tested: linear regression, Holt-Winters, SARIMAX, and Prophet. The final forecast is given by combining the result of the best classifier model with the result of the best regression model. This combined forecast is labeled ”BestCombined” by ORTEC.

It is noteworthy that the 6 models are trained for each time series individually. Given thousands of time series, the proof of concept involves training tens of thousands of different models separately.

6.1.2. Implementation of the Models

For forecasting whether a delivery is placed, logistic regression and a random forest classifier are employed. The models are implemented using the `LogisticRegression` and `RandomForestClassifier` classes from the scikit-learn library [40].

To predict the volume of orders, the process begins with forward filling zeros, ensuring every day is assigned a volume. This step creates an overestimated time series on which the following models are trained: linear regression, Holt-Winters, SARIMAX, and Prophet. For implementation, the `LinearRegression` class from scikit-learn [40] is used, while `ExponentialSmoothing` for Holt-Winters and `sm` for SARIMAX are used from the statsmodels library [45]. The `prophet` class is utilized from the

Prophet library [48]. Note that this method of filling zeros makes some assumptions. The idea is that if a customer does not order today, it will probably order a similar amount tomorrow. While this approach captures seasonal patterns over the year, it may be less accurate in reflecting daily fluctuations.

It can be easily understood that this two-step approach heavily relies on the classifier forecast being accurate. Otherwise, the combination of both methods can easily over- or underestimate the final forecast.

6.1.3. Features

Features enrich time series data by providing additional information that can improve model predictions. In the proof of concept, a few straightforward features are explored.

Since holidays have a big impact on the order patterns and the volumes ordered, a holiday flag is implemented. This feature uses one-hot encoding to indicate whether a given day is a holiday, the day before a holiday, or the day after a holiday. In addition, weather conditions can influence the volume of orders as well. For instance, inclement weather over consecutive days might lead to reduced traffic in restaurants, consequently decreasing the amount of products they need to order. To explore this effect, ORTEC introduced a temperature feature to assess if the model can learn from these variations. It should be noted that the temperature primarily reflects a yearly seasonal effect, which might already be learned by the models. It remains uncertain how daily temperature fluctuations specifically affect order patterns.

Most time series models inherently integrate some level of time indexing or seasonality. However, models such as linear regression, logistic regression, and random forest do not have a built-in mechanism for keeping track of time. To address this, weekday features are used, represented through one-hot encoded columns for each day of the week. Additionally, to improve the classification models (logistic regression and random forest), features incorporating lagged information about deliveries are included. Specifically, lags of 7, 14, 21, and 28 days. The lagged values enable the models to capture patterns such as weekly, biweekly, and monthly ordering behavior by asking if the customer placed an order one week prior, two weeks prior, and so forth.

6.1.4. Model Selection and Validation

As previously mentioned, the final model per customer is a combination of two models: a classification model and a regression model. The outputs from these models are combined to produce a final forecast. Specifically, if the classification model predicts that an order will occur, the regression model is used to determine the order's volume. The best classification model is selected based on accuracy, as defined in Definition 3.19. The best regression model is selected based on the Mean Absolute Error (MAE), as defined in Definition 3.17.

The best combination of models is selected based on the performance during a 6-week validation period. Following this, the chosen combination is tested over an additional 6 weeks. Thus, the entire process involves cross-validation using a train-test split over a span of 12 weeks. For each fold, the model predicts the next two weeks, but only the results for the second week are kept. Then, the model is updated by training on the following week's data, and the cycle continues accordingly. One can imagine that this process, when repeated individually for each customer, is computationally quite demanding.

The final model is compared to a baseline model called "CopyLastWeek", which operates by simply copying the results from the previous week to forecast the next horizon. It should be noted that this baseline model is also part of the model selection. So, if the baseline method works better than the other options for a specific time series, the BestCombined method chooses the baseline. This, of course, makes it easy for the BestCombined method to improve over the baseline method.

6.1.5. Hyperparameter Tuning

Each of the six models has specific hyperparameters that can be optimized. For instance, the random forest model incorporates hyperparameters such as maximum tree depth and minimum samples per node split, while the SARIMAX model is characterized by parameters p, d, q and P, D, Q. Moreover, determining the optimal set of features is also considered part of hyperparameter tuning. Ideally, only the most informative features would be included. However, identifying these beforehand can be

challenging.

Due to the computational expense associated with tuning hyperparameters for every model across all customers, hyperparameter tuning is only done over a subset of 3 customers. The best parameter set for every model is then applied to all customers. While this approach may not guarantee optimal parameters for each individual dataset, it aims to achieve a good enough solution that balances performance and resources. This method showed satisfactory results for the proof of concept.

Grid search is used to systematically explore all possible combinations of the specified hyperparameter values and identify the best combination. The specified grids for each regression and classification model can be found in Tables C.1 and C.2, respectively.

6.1.6. Hierarchical Forecasting

Hierarchical forecasting, as introduced in Section 3.7, is implemented using the hierarchicalforecast library from Nixtla [32]. We use `HierarchicalReconciliation` class alongside reconciliation methods such as `TopDown`, `MiddleOut`, and `MinTrace`. These methods are tested on a subset of all customers.

The standard implementation does not allow zeros in the dataset. Consequently, we test hierarchical reconciliation on the extended volume series, which contains no zeros.

For the hierarchical structure, we employ four levels: the top level, which reflects the sum of all series; the market segment level; the customer level; and the product level. Each level above is the aggregate of all series beneath it. In the `MiddleOut` approach, we set the market segment as the middle level.

6.2. Forecasting Approach using Neural Networks

Since neural networks are, by design, non-linear and have the ability to learn complex patterns, the potential exists for using a single model in place of the two-step approach discussed in Section 6.1.1. The models described in Section 4.4 are also designed to handle multiple customers in a single model. In total, this would reduce the amount of models to train from roughly 20,000 to a single model. The aim is for this approach to achieve comparable or superior results to that of two separate models applied to individual customers.

This section outlines the implementation of the tested neural network models. Additionally, the explored features and their embeddings are presented. Finally, it explains how the hyperparameters of the models are tuned.

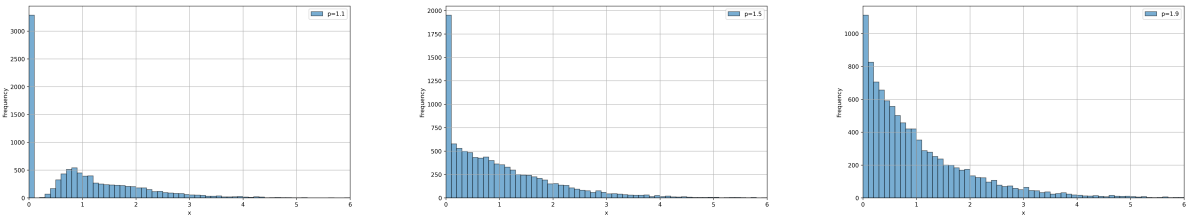
6.2.1. Implementation of the Models

The models evaluated in this study include DeepAR, TFT, and N-HiTS, all of which are implemented using Nixtla's Neuralforecast package [34]. Nixtla is a company focused on time series research and deployment. They provide a platform that integrates various forecasting packages, which allows us to use different models within a unified environment using consistent syntax. The models are trained and used for prediction using functions like `fit()`, `predict()`, and `cross_validation()`. A comprehensive explanation of the code implementation for training, predicting, and cross-validation is provided in Appendix D. To evaluate the models' ability to learn the non-linear behavior, we train them on the original time series data without replacing zeros.

A non-trivial step of the implementation is the two-week forecasting approach, where we are only interested in the second week predicted. This is accomplished by setting a forecasting horizon of 14 days and adjusting the window step size to 7 days (by default equal to the horizon length). To resolve the overlapping forecasts, we only keep the second week from each prediction. To make sure the loss function is only minimized over the predictions of the second week, a horizon weight of 0 is assigned to the first week and 1 to the second week.

Nixtla uses a collection of PyTorch loss functions. We specifically apply the `DistributionLoss` and `MQLoss` from the module `neuralforecast.losses.pytorch`. For TFT and N-HiTS we use the Multi-Quantile loss, as introduced in Section 4.4.2. For DeepAR, we can only use a distribution loss due to its design. The distribution loss can take different distributions. This research explores the normal and

Tweedie distributions, where the latter is tested because of the zero-inflated nature of the data (days without customer orders have zero volume). The Tweedie loss function was notably employed in the M5 forecasting competition [24], highlighting its effectiveness in handling similar intermittent demand data. PyTorch implements the Tweedie distribution with a variance power $1 < p < 2$, yielding a compound Poisson Gamma distribution. A variance power close to 2 reflects a Gamma distribution, and a variance power near 1 reflects a Poisson distribution, thus giving greater weight to zeros [20]. Figure 6.1 shows examples of the Tweedie distribution with varying variance powers.



(a) $p = 1.1$ **(b)** $p = 1.5$ **(c)** $p = 1.9$

Figure 6.1: Histograms of Tweedie distributions with $\mu = 1$ and $\phi = 1$ for different variance powers p. Each histogram is generated from 10,000 samples.

6.2.2. Features and Embeddings

With the neural network models, we conduct two tests: one using the same features as the proof of concept and another incorporating additional features. The goal is to compare the performance of traditional methods with neural network models while also enhancing the model's capabilities by leveraging additional data.

In the proof of concept, weekdays were represented using one-hot encoding for each day of the week. For neural networks, we evaluate both one-hot encoding and a categorical encoding, which utilizes a single feature column with integers ranging from 0 to 6 for weekdays rather than seven separate columns. In addition, a feature is added for the yearday to capture yearly seasonality, enhancing the model's ability to recognize annual patterns. Holidays are one-hot encoded to represent the holiday itself, along with 1 or 2 days immediately before or after them, creating a total of five feature columns. Finally, a trend feature is added to keep track of time.

The models aim to handle all time series at the same time, so we need a method to differentiate each series. To achieve this, we include static information such as customer ID and product type. We can also use the static information to enrich the data even further. The market segment is incorporated to capture segment-specific patterns. The postal code, using both 4 and 6 digits of precision, is included to capture local patterns related to neighborhoods and street-level variations within a city.

The models require static information to be formatted as either one-hot or integer embeddings. To limit the number of feature columns, the latter is used in this research. Thus, each static feature is represented using integers corresponding to the total number of options available for that feature. For instance, the customer ID is embedded as an integer within the range $0, \ldots, \#\text{customers}$.

For numerical data, we apply temporal normalization as discussed in Section 4.3.2. The methods tested include Identity, Standard, Robust, and Minmax scaling. Identity leaves data unchanged and preserves original values. Standard scaling removes the mean and scales data to unit variance. Robust scaling uses the median and mean absolute deviation for standardization, offering better handling of noisy data with outliers. Minmax Scaling resizes features to a specified range, often [0, 1] or [-1, 1], enhancing convergence during model training. For instance, temperature data is normalized using a minmax scaler with the range [-1, 1].

The used features in the neural network models, with their embeddings, are summarized in Table 6.1.

[1] if this day is a holiday as well, this feature is considered 0

Feature	Description	Embedding
	Temporal, past	
Volume	target variable, ordered volume	float, scaled
Delivery	1 if there is a delivery on that day, 0 else	one hot
Weather	temperature (min, max, mean) of that day	minmax scaled [-1,1]
	Temporal, future	
Date	the date of delivery	Pandas datetime
Weekday	the weekday of the delivery	one hot & categorical
Yearday	yearday, for seasonality	cyclic & categorical
Trend	timeseries to keep track of time	int $\in \{0, \ldots, \#\text{ time steps}\}$
Holiday	1 if day is holiday, 0 else	one hot
Holiday $\pm 1, 2$	1 if 1/2 days before/after holiday[1], 0 else	one hot per $\{-2, -1, +1, +2\}$
	Static	
Customer ID	unique ID for every customer	int $\in \{0, \ldots, \#\text{ customers}\}$
Product type	DRY, FRSH, FRZ	int $\in \{0, 1, 2\}$
Market segment	market segment of customer	int $\in \{0, \ldots, \#\text{ segments}\}$
Postal code number	first 4 digits of postal code	int $\in \{0, \ldots, \#\text{ 4 digit codes}\}$
Postal code full	full 6 digit postal code	int $\in \{0, \ldots, \#\text{ 6 digit codes}\}$

Table 6.1: Overview of static, past, and future exogenous variables used in this research for training the neural network models, along with a description and the embedding applied for each feature.

6.2.3. Hyperparameter Tuning

Hyperparameter tuning is crucial in deep learning, as models are highly sensitive to the choice of hyperparameters. Nixtla provides a built-in implementation for hyperparameter tuning [25], utilizing the `hyperopt` library [3]. Each model features an Auto variant, e.g., the Temporal Fusion Transformer (TFT) becomes AutoTFT. These Auto models operate in the same way as the standard models. The `fit()` and `cross_validation()` functions perform hyperparameter optimization, and `predict()` forecasts the time series using only the optimal hyperparameter settings.

Defining an Auto model requires specifying a loss function, a configuration with the search space, a search algorithm, the backend (Ray Tune or Optuna), and the number of configurations to test. A detailed explanation of the code implementation for hyperparameter tuning is provided in Appendix D.

For the backend, we use Ray's Tune library. Each Auto model includes a default search space, tested across several datasets. Search spaces are defined using dictionaries where keys correspond to the model's hyperparameters and values use Tune functions to determine how these hyperparameters are sampled. Examples of tune functions are `tune.choice()`, `tune.loguniform()`, and `tune.randint()`. We customize the search grid by defining a hyperparameter search space dictionary. The search spaces for each model employed in this research can be found in Table C.3.

For the search algorithm, we employ the HyperOptSearch function, which uses the Tree-structured Parzen Estimators (TPE) algorithm. TPE is, like Bayesian Optimization, a sequential model-based optimization method, meaning it approximates the performance of hyperparameters based on previous results and selects new hyperparameters for testing based on this approximation. The key difference lies in TPE estimating $p(x \mid y)$ and $p(y)$, whereas Bayesian optimization estimates $p(y \mid x)$ directly, the probability of a loss y given a hyperparameter setting x. Given observations $x^{(1)}, \ldots, x^{(k)}$, $p(x \mid y)$ is defined using two densities:

$$p(x \mid y) = \begin{cases} l(x) & y < y^*, \\ g(x) & y \geq y^*. \end{cases}$$

Here, $l(x)$ represents the density of the 'good observations', i.e., the observations with a loss function lower than a threshold y^*. Conversely, $g(x)$ denotes the density of 'bad observations', i.e., the observations with a loss function higher than y^*. Note that y^* has to be larger than the best observed loss to include some observations within $l(x)$. The TPE algorithm selects the threshold y^* to be some quantile of the observed y values. For the next hyperparameter setting, the algorithm chooses a candidate x with a high probability under $l(x)$ and a low probability under $g(x)$. The tree-structured form of l and g makes it easy to draw many samples from $l(x)$ and return the x^* that maximizes $l(x)/g(x)$ [4].

7

Route Planning Software

This chapter delves into the route optimization software, the B2B Delivery Product Suite [37], developed by ORTEC, which is used in this research. A schematic representation of the route optimizer is given in Figure 7.1. The focus of this chapter will be on the general solution approach of the optimizer and the specific configurations applied throughout this thesis. We will start by discussing the format of an optimization request. Then, we will explain the heuristic and meta-heuristic algorithms employed by the software to generate routes. We will also briefly state the specific configurations that are used in this research. Lastly, we will explain how the optimizer's responses are analyzed.

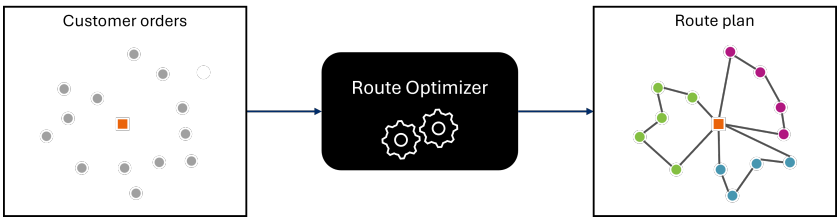


Figure 7.1: Schematic representation of optimizer. The route optimizer takes customer orders as input and outputs a route plan.

The content of this chapter is based on internal documentation by ORTEC [38] and meetings with routing experts at ORTEC.

7.1. Optimization Requests

An optimization request begins with a properly structured input, which is formatted as a JSON file. This input consists of lists with the following elements:

- **Depots**: Locations from which the vehicles can depart, in latitude and longitude coordinates.
- **Routes**: Available vehicles along with their relevant details, such as start and end depot, time constraints, preparation and completion times, capacities and break regulations.
- **Customers**: A list of customers along with their respective locations and handling durations.
- **Tasks**: The actual tasks that must be executed. Information on the tasks include the customer, depot to deliver from, required quantities (f.e. kg or volume), handling duration, time windows for service, and any restrictions on vehicle usage.

Additionally, the software allows the input of preplanned routes. Although this feature has not been tested yet for the client of this research, we will use this to give a forecasted route planning to the optimizer. A schematic overview of the used method is given in Figure 7.2. Specifically, a provisional route plan is generated using forecasted orders with the route optimizer. Then, given the actual customer

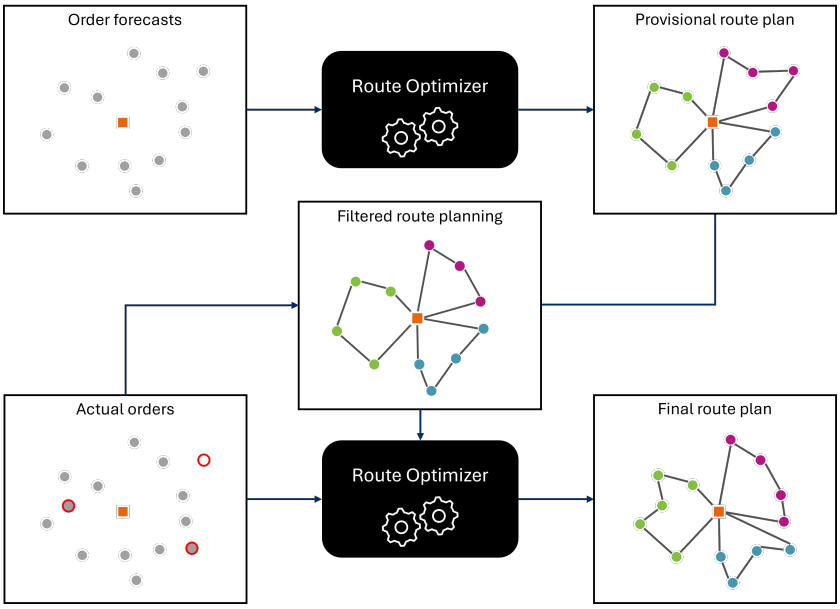


Figure 7.2: Schematic representation of optimizer, with input planning. The input planning is filtered based on the customers who actually placed an order.

orders, the provisional route plan is filtered such that it only contains the correctly forecasted customers. Note here that the customers in the filtered route plan may have ordered different volumes than were predicted in the provisional route plan. Finally, with this input plan and the additional customers that were not forecasted, the route optimizer can generate a final route plan.

For the input route plan, it is possible to fix certain tasks in a route or not. This gives us the possibility to, for example, optimize the tasks that were not included in the forecasted orders while maintaining the planning of the correctly forecasted ones.

Optimization requests are sent to the optimizer together with a specific configuration of the optimizer. This configuration determines the objective function and specifies which (meta-) heuristics to employ in order to find a solution to the routing problem. In other words, the configuration acts as a recipe book, instructing the optimizer on what algorithms to apply. One can easily understand that the output route of the optimizer is highly dependent on the configuration that is sent together with the request.

To better understand the possibilities of the optimizer and what configurations might be interesting to test, we will first explore its individual building blocks.

7.2. Solution Approach of the Optimizer

The optimizer solves the vehicle routing problem in three steps, summarized in Figure 7.3:

1. Construct a feasible solution by one-by-one insertion of tasks. The objective is to plan as many orders as possible (or the ones with the most profit).
2. Improve the current solution using heuristics that locally search for improvements of the objective function.
3. Try to find a better solution using metaheuristics that locally and globally search for improvements.

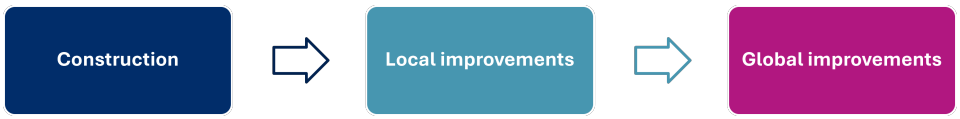


Figure 7.3: Solution approach followed by the optimizer, consisting of three main stages: initial construction, local improvements, and global improvements.

To determine whether one solution is better than another solution, we compare them based on a hierarchical objective function.

7.2.1. Hierarchical Objective Function

The optimizer evaluates solutions according to a hierarchical set of objectives. This hierarchical objective function first compares solutions based on the first objective. If two solutions are found to be equal in this regard, the optimizer then assesses them according to the second objective, and so on. Objectives that are lower in the hierarchy will be used less frequently as they serve to refine the evaluation process only when higher-priority objectives yield equivalent results.

The total plan cost is, besides the number of planned tasks, the main objective to be optimized. This metric consists of a weighted average of multiple operational costs like cost per vehicle, cost per kilometer, cost per hour, and cost per stop. The components of the objective function and the weights in the plan costs are determined during the PoC route planning in consultation with the client. These weights can be found in Table B.1. The weights for the constraints are presented in Table B.3.

The objectives, in hierarchical order, used in the configurations of this research are as follows:

1. Number of planned tasks: $\max$
2. Plan cost: $\min$
3. Route duration: $\min$
4. Distance: $\min$
5. Number of used routes: $\min$
6. Driving time: $\min$
7. Wait time: $\min$

Note that objectives 3-7 are indirectly included in the plan costs as well.

Constraints like a time window constraint or a capacity constraint were not initially included in the objective function. The reason for this is that the construction phase will not propose infeasible solutions. However, in this research, we will make use of an input planning that is based on forecasted orders. If the actual ordered volumes are higher than the forecasted volumes, this may result in a violation of the capacity constraint. The improvement phase will only propose new solutions if they improve the objective function. Therefore, to resolve possible capacity violations, we need to include the capacity constraint somewhere in the objective function. More on this in Section 7.4.

7.2.2. Construction

The initial feasible solution to the VRP is constructed using a greedy heuristic. It starts with the pre-planned tasks and fill the unplanned tasks one-by-one based on sequential or parallel insertion. The difference between two methods lies in how the routes are filled: one route at a time or multiple routes simultaneously. Given that the pre-planned route is feasible, the use of both the sequential and parallel insertion will result in a feasible solution.

Definition 7.1. Sequential insertion methods construct a feasible solution to the Vehicle Routing Problem (VRP), one route at a time. This is illustrated in Figure 7.4.

Definition 7.2. Parallel insertion methods construct a feasible solution to the Vehicle Routing Problem (VRP) for a number of routes simultaneously. This is illustrated in Figure 7.5.

When planning a new task in a route, the task that is cheapest to insert is selected, based on a predefined metric like distance. It is also possible for the optimizer to focus on the difficult-to-plan orders first.

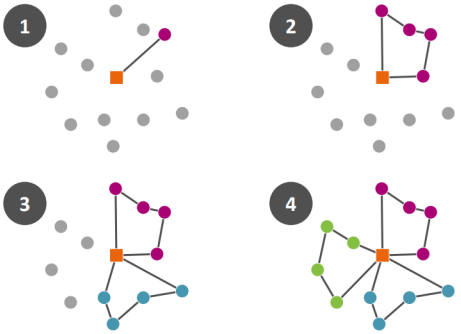


Figure 7.4: Sequential Insertion [38]. Tasks are assigned sequentially to each route. When a route reaches its capacity, the next route is started.

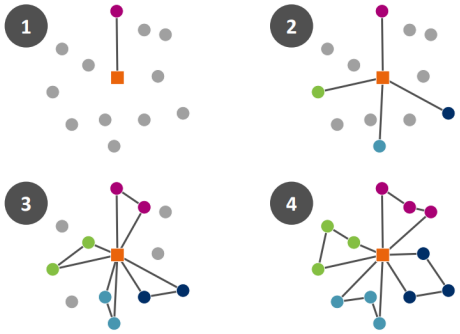


Figure 7.5: Parallel Insertion [38]. Tasks are assigned to multiple routes simultaneously, with routes being filled in parallel.

7.2.3. Local Search Methods

Given a feasible solution from the construction phase, we want to use local search methods to improve the solution. Local search methods are algorithms that move from one solution to a better one, defined by a space of possible candidates or a set of operations to perform on a solution. Since we are searching for improvements of the objective function, we will keep the outcome of an operation if it actually improves the objective function. We will disregard it if it doesn't improve the objective function.

The most simple example of an operation we can perform on a solution is to simply move a task from one route to another, as illustrated by Figure 7.6. This principle is exploited in Algorithm 7.1.

Figure 7.6: Move operation [38]. A task is moved to another route.

One can imagine that the number of moves to try will explode with the problem size. For this reason, ORTEC implemented a mechanism to only try operations with a minimum estimated gain for a specified metric (e.g. distance of travel time). Therefore, all algorithms have the attributes `estimateWith` and `minimumEstimatedGain` to control this mechanism.

To control whether local search methods are applied within the same route or over the whole solution, most algorithms also have the attribute `onlyWithinRoute` that can be true or false. This attribute determines whether the operations of that algorithm can only be applied within a route or can also be applied between multiple routes routes.

Algorithm 7.1 Move Algorithm

Attributes: `estimateWith` (e.g., distance, driving time), `minimumEstimatedGain` (value), `onlyAllowChangesWithinSameRoute` (true/false)

1: Create all combinations of groups of tasks in the current solution.
2: **for** every group **do**
3: **if** the estimated gain of the operation, in terms of `estimateWith`, is larger than `minimumEstimatedGain` **then**
4: Move the group to a different location in the solution (only on same route or on a different route, depending on `onlyAllowChangesWithinSameRoute`).
5: Compute the new objective value of the resulting solution.
6: Keep a new solution if it is better than the current one.

Instead of moving a task (or group of tasks), we can also exchange it with another task (or group of tasks), as illustrated in Figure 7.7. Note that this type of swap differs from the one defined and investigated in this research. However, since this terminology is used in ORTEC's internal documentation, we retain the name here for consistency. This operation can be implemented as an algorithm in a similar fashion as Algorithm 7.1.

Figure 7.7: Swap operation [38]. Here, a group of tasks is exchanged with another group of tasks between routes.

Algorithm 7.2 Swap Algorithm

Attributes: `estimateWith` (e.g., distance, driving time), `minimumEstimatedGain` (value), `onlyAllowChangesWithinSameRoute` (true/false)

1: Create all combinations of groups of tasks in the current solution.
2: **for** every pair of groups (within the same route or also between different routes, depending on `onlyAllowChangesWithinSameRoute`) **do**
3: **if** the estimated gain of the operation, in terms of `estimateWith`, is larger than `minimumEstimatedGain` **then**
4: Swap the location of the groups in the solution.
5: Compute the new objective value of the resulting solution.
6: Keep a new solution if it is better than the current one.

Now, let us define the 2-opt operation on a pair of edges. In the form of an algorithm, this operation can also be applied on all pairs of edges within the same route or on pairs of routes depending on whether the attribute `onlyAllowChangesWithinSameRoute` is true or false.

Definition 7.3. Given a pair of non-contiguous edges, a **2-opt** operation rearranges the connections between the four nodes such that it doesn't create subtours, as illustrated in Figure 7.8.

Figure 7.8: 2-opt operation [38]. Two connections between tasks are swapped to remove route crossings.

Algorithm 7.3 TwoOpt Algorithm

Attributes: `estimateWith` (e.g., distance, driving time), `minimumEstimatedGain` (value), `onlyAllowChangesWithinSameRoute` (true/false)

1: **for** every pair of edges (within the same route or also between different routes, depending on `onlyAllowChangesWithinSameRoute`) **do**
2: **if** the estimated gain of the operation, in terms of `estimateWith`, is larger than `minimumEstimatedGain` **then**
3: Apply the 2-opt operation on the pair of edges.
4: Compute the new objective value of the resulting solution.
5: Keep a new solution if it is better than the current one.

Similar to a 2-opt swap, we can do the same operation on three edges. Where in a 2-opt there is only one way to reconnect the nodes, there are multiple ways to perform a 3-opt move. This makes

the implementation computationally more expensive. Due to this increase in complexity, Algorithm 7.4 only performs this operation on edges within the same route.

Definition 7.4. Given three non-contiguous edges, a **3-opt** operation rearranges the connections between the six nodes. A 3-opt move can always be written as 1,2 or 3 subsequent 2-opt moves.

Algorithm 7.4 ThreeOpt Algorithm

Attributes: `estimateWith` (e.g., distance, driving time), `minimumEstimatedGain` (value)

```
1: for every triplet of edges within the same route do
2:     if the estimated gain of the operation is larger than minimumEstimatedGain then
3:         Apply the 3-opt operation on the triplet of edges.
4:         Compute the new objective value of the resulting solution.
5:         Keep a new solution if it is better than the current one.
```

The operations defined above can also be utilized in algorithms that exploit them in different ways to find improved solutions. Algorithm 7.5, for example, tries to resolve all overlapping route parts between routes, as illustrated in Figure 7.9. This can be seen as a swaps operation with specific groups. However, this algorithm is still capable of new improvements, because of a different way of looking at the solution.

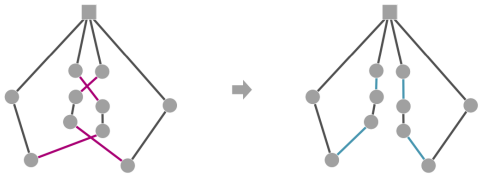


Figure 7.9: Cross Exchange operation [38]. Segments of tasks are exchanged between two different routes

Algorithm 7.5 CROSSExchange Algorithm

Attributes: `estimateWith` (e.g., distance, driving time), `minimumEstimatedGain` (value)

```
1: for every pair of overlapping route segments between routes do
2:     if the estimated gain of the operation is larger than minimumEstimatedGain then
3:         Apply a swap operation on the overlapping segments.
4:         Compute the new objective value of the resulting solution.
5:         Keep the new solution if it is better than the current one.
```

Algorithm 7.6 aims to insert tasks in routes that are already near full capacity. To facilitate this insertion, it unplans certain tasks to create the necessary space. The unplanned tasks are reallocated to other routes. This algorithm is computationally expensive.

Algorithm 7.6 EjectionChain Algorithm

Attributes: `estimateWith` (e.g., distance, driving time), `minimumEstimatedGain` (value)

```
1: for each task in a route that is near capacity do
2:     Unplan the task to create space for insertion.
3:     for each candidate task to insert do
4:         if the estimated gain from the insertion is larger than minimumEstimatedGain then
5:             Insert the candidate task into the route.
6:             Plan the unplanned task in another route.
7:             Compute the new objective value of the resulting solution.
8:             Keep the new solution if it is better than the current one.
```

Finally, the Algorithm 7.7 tries to move all tasks in a route to a different route to reduce costs.

Algorithm 7.7 MoveTasksToBetterRoute Algorithm

Attributes: `estimateWith` (e.g., costs), `minimumEstimatedGain` (value)

1: **for** every route with planned tasks, and a destination route **do**
2: **if** the estimated gain of the operation is larger than `minimumEstimatedGain` **then**
3: Move all tasks in the route to the destination route.
4: Compute the new objective value of the resulting solution.
5: Keep the new solution if it is better than the current one.

Table 7.1 provides an overview of the available algorithms, indicating whether each algorithm can be applied within a single route or across multiple routes.

Algorithm	Only within route	Over different routes
Move	v	v
swap	v	v
MoveAndswap	v	v
TwoOpt	v	v
ThreeOpt	v	x
CROSSExchange	x	v
EjectionChain	x	v
MoveTaskToBetterRoute	x	v

Table 7.1: Overview of the available local search algorithms, indicating their applicability within a single route or across multiple routes.

7.2.4. Meta-Heuristics

The algorithms discussed in Section 7.2.3 only accept new solutions if they yield an improved objective function value over the current one. Consequently, these algorithms can become trapped in local minima, which means the algorithms will never find a better solution, even though better ones may exist. Meta-heuristics aim to solve this problem by exploring a larger solution space. This idea is illustrated in Figure 7.10. A common approach within meta-heuristics is the temporary acceptance of worse solutions, with the hope of eventually finding a better solution.

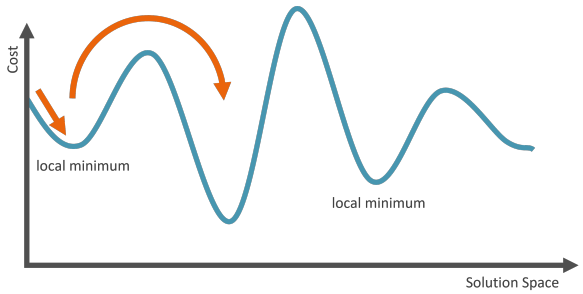


Figure 7.10: Illustration of the concept behind meta-heuristics. By exploring a larger solution space, we hope to escape local minima and find improved global solutions [38].

ORTEC has implemented meta-heuristics using the principle of ruin and recreate, which can be explained through Figure 7.11. Initially, the solution is intentionally ruined by removing customers from routes. Subsequently, the solution is reconstructed by reinserting customers back into routes. After performing several local search iterations on this new solution, a decision is made on whether to accept it. This process can be repeated as many times as desired.

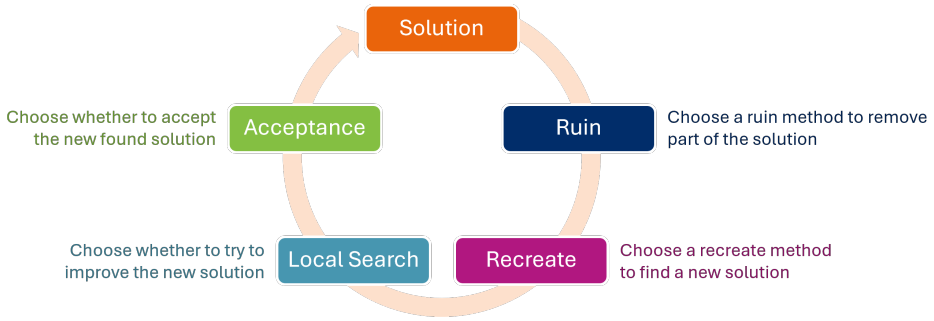


Figure 7.11: Overview ruin and recreate process [38]. Starting from an initial solution, it is 'ruined' by removing part of it, after which a solution is 'recreated' using a reconstruction method. Optionally, local search is applied before deciding whether to accept the new solution.

An example of ruin and recreate is given in Figure 7.12. First, a number of random tasks are removed from the route. Then, these tasks are inserted into a route where they fit best.

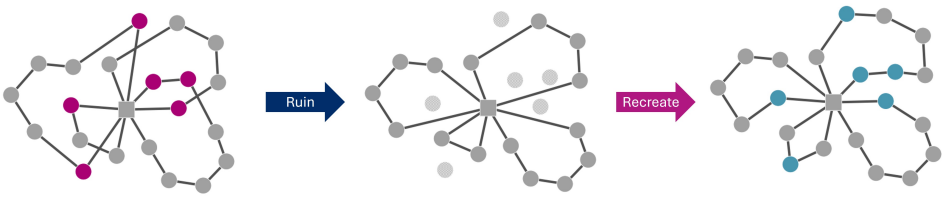


Figure 7.12: Example of ruin and recreate with random removal [38].

Possible removal methods to ruin a solution include the removal of random customers, the removal of the worst tasks in a route, the removal of random clusters, or the removal of a whole trip. These methods are summarized in Figure 7.13.

(a) Random removal.

(b) Worst removal.

(c) Random cluster removal.

(d) Trip removal.

Figure 7.13: Examples of removal methods used in the ruin phase of the ruin and recreate process [38].

To recreate a solution, tasks need to be reinserted. This can be based on cheapest insertion or largest 'regret'. In the regret-based approach, for all removed tasks, the additional costs incurred when inserting a task into its second-best position compared to its best position are calculated. The task with the highest 'regret' is prioritized for insertion. Cheapest insertion can be performed in parallel or based on a predefined order of tasks.

The ruin and recreate algorithm includes attributes like `PercentageOfTasksToRemove` and `NumberOfTasksToRemove`, which specify the amount of tasks to remove. Additionally, it can be specified whether local search should only be performed on the new solution if it is worse than the current one. For this, an allowable threshold can be specified to govern whether local search is applied. Similar to

local search heuristics, the algorithm allows selection of an objective `EstimateWith` (such as distance, driving time, plan cost, etc.) for comparing solutions.

The selection of methods is implemented by using a so-called roulette wheel. After a method is applied, the probabilities for each method are adjusted based on if the new objective value is better than the current one (a better value increases the probability of the selected method, a worse value decreases the probability of the selected method). This process of selecting a method using the roulette wheel, finding a new solution, and adjusting the probabilities is repeated for a predefined number of iterations.

7.3. Tested Configurations

As said before, the configuration is an ordered lists of algorithms to apply in finding a solution to the VRP. It is beyond the scope of this research to build a configuration from scratch or to design new algorithms that solve the objective of this research. In this thesis, we will work with a configuration from ORTEC that is used in the PoC for route planning and is optimized to return efficient routes (not considering the amount of swaps). This configuration includes a set of heuristics that are applied within routes, a set of heuristics that are applied across routes and a set of ruin and recreate methods. Each set has multiple recursions. We will tweak some parameters in this configuration to investigate the effect of certain changes.

In this research, we want to investigate the tradeoff between keeping the pre-planned routes as much as possible and planning the most efficient routes. We try to find the right balance by tweaking the following parameters. First, we can decide to fix the input planning or not. This means the input planning will not change for sure, therefore also making it impossible for the optimizer to swap a task if it is planned really poorly in the forecasted planning. Furthermore, we can choose which (meta-) heuristics to use. For simplicity, they are grouped into Ruin & Recreate, Local Search outside route, and Local Search within route. Finally, we can tweak the minimum estimated gain and the number of recursions to control the amount of operations that are performed on a solution.

The tested configurations are summarized in Table 7.2. The configurations are named full optimizer, light optimizer, and minimum optimizer, indicating the size of the search space. The full optimizer is the configuration constructed by ORTEC. Since the set of ruin & recreate methods has too big of an impact on the number of swaps, it is turned off for the light optimizers. As a results, swaps will only occur from local search outside the route. The aim of the different ”light optimizers” is to compare the effect of the higher estimated gain with a lower number of recursions. The minimum optimizer only applies local search within the same route. A configuration is added with only the construction phase as a baseline to compare.

Configuration	Input orders	Input planning	Ruin & Recreate	LS outside route `minEstGain`	LS outside route `maxNofRec`	LS within route
Full opt, forecast orders	forecast	x	v	-1000	3	v
Full opt, actual orders	actual	x	v	-1000	3	v
Full opt, fixed input	actual	v, fixed	v	-1000	3	v
Full opt, with input	actual	v	v	-1000	3	v
Light optimizer 0	actual	v	x	1000	3	v
Light optimizer 1	actual	v	x	5000	3	v
Light optimizer 2	actual	v	x	1000	1	v
Minimum optimizer	actual	v	x	x		v
Construction only	actual	v	x	x		x

Table 7.2: Overview of used optimizer configurations, detailing input types, use of ruin and recreate, and local search parameters. Minimum estimated gain is determined in terms of distance.

Note that according to the table above, we can either keep the input planning fixed and fully optimize the planning of all other tasks or optimize all tasks while making no difference between pre-planned tasks and non-forecasted tasks. Ideally, we would like to have the flexibility to only allow adjustments to the input planning if it gives a certain gain in efficiency and fully optimizes the other tasks. Unfortunately,

this is not possible with the current software. To work around this limitation, we employ a double optimization approach:

1. In the first optimization round, we keep the input route plan fixed and use the full optimizer to optimize the tasks that were not forecasted.
2. In the second optimization round, we allow the optimizer to make adjustments to all tasks.

This approach is illustrated in Figure 7.14.

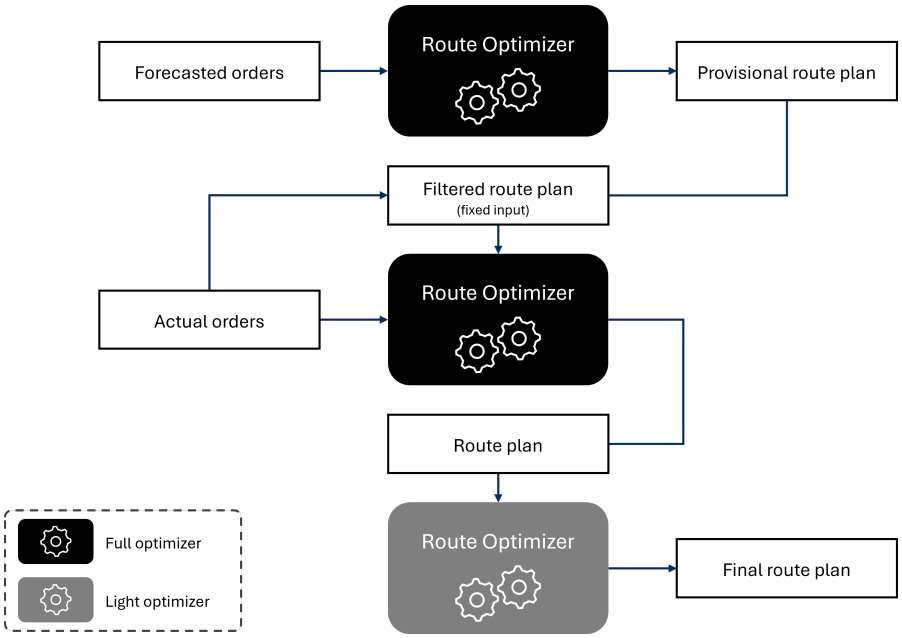


Figure 7.14: Schematic overview of the double optimization method applied in the second stage of route optimization. In the first round, optimization is performed using a fixed input route plan, so optimization is only limited to customers who were not predicted to place an order. In the second round, changes can be made to all tasks for further optimization.

7.4. Addressing the Capacity Constraint

The original ORTEC configuration was not specifically designed to integrate an input planning. While technically feasible to include an input planning, the configuration lacked methods dedicated to resolving capacity violations. During testing with forecasts from ORTEC's BestCombined method, which slightly underestimated volumes, we found that most route plans had high capacity overloads. To resolve this, it is important to include the capacity constraint in the objective function, so that the optimizer recognizes a reduction in the capacity constraint as an improvement. Then, we need to actively try to resolve the capacity constraint using local search methods.

In this research, it is chosen to implement the capacity constraint as a weighted sum with the plan cost. The weight of the constraint is chosen such that, roughly, planning one roll container too many equals the cost of using an additional truck to deliver that roll container. Planning more roll containers above the capacity yields a quadratic increase in the penalty value. We also tried to implement the capacity constraint as a separate objective above the plan cost in the hierarchy. In preliminary tests, the weighted-sum approach delivered the best results, most consistently satisfying the capacity constraint and providing efficient routes.

When the constraint is added to the objective function, the optimizer would accept a new solution if it reduced capacity violations. However, the optimizer did not actively seek to decrease these violations, as all algorithms used distance as the metric for `estimateWith`. To improve the handling of capacity constraints, we added two heuristics to the local search outside the route phase: MoveAndswap and MoveTasksToBetterRoute. These heuristics were implemented using a metric for `estimateWith` that

includes penalties (PenaltiesAndCost), with a minimum estimated gain set to match the capacity violation of one roll container. The aim is to relocate a customer to a different vehicle if the current vehicle exceeds its capacity. This improvement successfully addressed capacity issues for most days. Further optimization is necessary to focus more thoroughly on capacity constraints.

7.5. Evaluation of the Optimization Responses

Upon completing the optimization process, the software outputs another JSON file with information on the generated solution. The main result is, of course, a list of tasks assigned to each vehicle, outlining the execution sequence. In addition to the output routes, various Key Performance Indicators (KPIs) are given to evaluate the performance and efficiency of a solution. Key metrics include total optimization costs, number of vehicles utilized, total distance traveled by each vehicle, and driving duration.

The number of swaps is determined by comparing the output routes of the provisional plan to the final route plan. Since tasks are planned per vehicle, each customer's assignment can be checked to see if they remain on the same vehicle in both the provisional and final route plans. The pseudo-code provided in Algorithm 7.8 outlines this process. The algorithm calculates the swap percentage, allowing for comparisons between different days.

If we compare route plans from different forecasts, it may be an unfair comparison to simply count the number of swaps. If one forecasting method predicts that none of the actual customers will place an order, it will have no swaps in the current definition. We therefore extend the definition of a swap to include non-forecasted orders. Algorithm 7.8 includes the calculation of the percentage of swaps+ relative to the total actual customers for a specific day.

Definition 7.5. The number of **swaps+** is defined as the number of swaps between the provisional and final route, plus the number of customers that were not forecasted.

Algorithm 7.8 Compute swaps

```
Input: provisional_route_plan, final_route_plan
customers_forecast = set of customers in provisional_route_plan
customers_actual = set of customers in final_route_plan

customers_correctly_forecasted = customers_actual.intersection(customers_forecast)
customers_not_forecasted = customers_actual.difference(customers_forecast)

same_route_count = 0
different_route_count = 0
for customers in customers_correctly_forecasted do
    route_prov = route customer is planned on in provisional_route_plan
    route_act = route customer is planned on in final_route_plan
    if route_prov == route_act then
        same_route_count += 1
    else
        different_route_count += 1
swap_percentage = different_route_count / customers_correctly_forecasted * 100
swap+_percentage = (different_route_count + customers_not_forecasted) / customers_actual * 100
```

Part III

Results

8

Forecasting with Traditional Methods

This chapter presents the results of the proof of concept on forecasting from ORTEC, specifically the predictions derived from the BestCombined method compared to a baseline approach. These predictions will be used in Chapter 9 as input for the route planning software. Later in Chapter 10, we will explore the use of neural networks to potentially achieve even better forecasting results. The primary focus of this chapter is to establish a baseline for evaluating the future improvements of this study. Subsequently, we highlight some improvements attempted on the traditional forecasting method through modifying holiday features and exploring hierarchical forecasting methods.

8.1. BestCombined method from ORTEC

We begin by examining the baseline method used by ORTEC in the proof of concept, which serves as a benchmark for evaluating the new forecasting approach. Specifically, this baseline is a copy of the data from the most recent week available. Figure 8.1 shows the total predicted volume by the baseline method, where we can clearly see a two-week shift. This can be explained by the fact that we forecast two weeks ahead. Clearly, this approach fails to accurately predict volumes during holidays, as these days do not follow the typical weekly pattern. Moreover, copying holiday order volumes to regular days two weeks later also results in incorrect forecasts. It can be concluded that, while this method may be adequate when order patterns remain constant from week to week, it becomes ineffective when there are large fluctuations in order volumes.

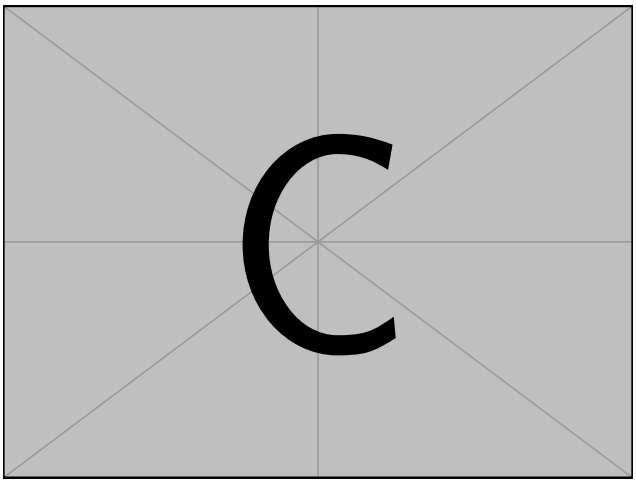


Figure 8.1: [*This figure contains confidential information and is therefore only available in the confidential appendix.*] Aggregated volume forecast using the baseline method. Both actual and predicted volumes are normalized by dividing by the maximum observed actual volume.

Figure 8.2 presents the forecast results from ORTEC's BestCombined method, which is the best combination, per customer, of a classification model and a volume model. While this approach seems to align well with the actual volume, we can see that it slightly underpredicts the overall volume.

Examining the holidays, the BestCombined method produces forecasts closer to the actual volume compared to the baseline method, though some discrepancies remain. The challenge of accurately predicting holiday volumes in this scenario is caused by the absence of explicit information concerning the distributor's business rules during holidays, which could provide valuable insights into these variations. On certain holidays, all customer orders are restricted, while on others, select customers seem to be able to still place orders. It can also be that, on certain holidays, the distributor may operate as usual, but customer closures prevent orders from being placed.

Additionally, the actual volume data shows notable peaks one to two days before and after holidays such as Easter, King's Day, and Pentecost. These peaks are not captured by the BestCombined method, probably because the days before and after a holiday are not explicitly included in the features.

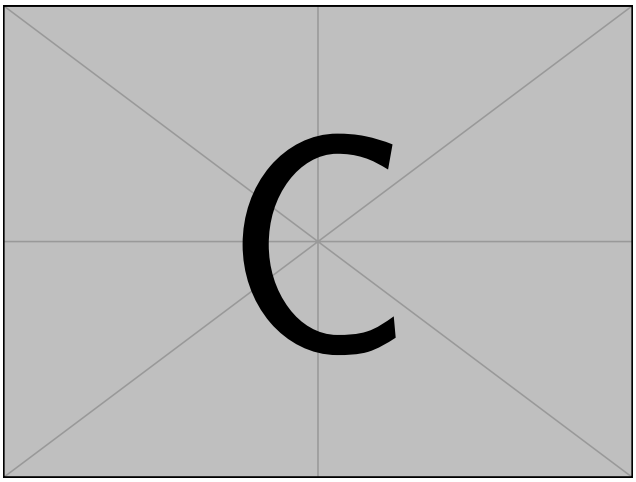


Figure 8.2: [*This figure contains confidential information and is therefore only available in the confidential appendix.*] Aggregated volume forecast using ORTEC's BestCombined method. Both actual and predicted volumes are normalized by dividing by the maximum observed actual volume.

The accuracy, mean absolute error (MAE), and mean error (ME) of the CopyLastWeek baseline and ORTEC's BestCombined method are detailed in Table 8.1. These metrics are averaged over all predictions within the test window. In this table, we observe a 16% improvement in MAE and a 1% increase in accuracy with the BestCombined method compared to the baseline. However, it is noteworthy that the mean error is higher for the BestCombined approach, indicating a tendency to underpredict the total volume on average.

The most significant difference in accuracy between the two methods can be attributed to the forecasts on holidays. Both methods exhibit a peak of false positives[1] on holidays, although this peak is higher with the CopyLastWeek method. Furthermore, the CopyLastWeek approach also shows a notable peak of false negatives two weeks after holidays, because it copied the lower number of orders that was on the holiday.

Method	Accuracy	MAE	ME
CopyLastWeek (BaseLine)	0.914201	50.952548	0.601498
BestCombined (ORTEC)	0.924149	42.648651	4.402192

Table 8.1: MAE and accuracy of baseline method and ORTEC's BestCombined method over test window.

[1] A false positive occurs when the model predicts that a customer will place an order, when they do not. Conversely, a false negative occurs when the model fails to predict an order that a customer actually places.

8.2. Improvements on Holidays

To enhance forecasting accuracy during holiday periods, several tests were conducted to evaluate the holiday feature and its influence on prediction results. The proof of concept initially incorporated a single feature column indicating whether a day was a holiday or adjacent to one. However, as observed, order patterns significantly differ between actual holidays and the surrounding days. To address this, we employed a random forest model with different sets of feature columns to investigate their effect.

In one test scenario, we isolated true holidays from adjacent days by using a column specific to the actual holiday, which resulted in no significant change in accuracy for non-holidays but about a 1% increase in accuracy for both true holidays and the surrounding days. Further refinement involved creating separate feature columns for each specific holiday, improving accuracy for true holidays by approximately 2% and for adjacent days by about 1%.

Despite these minor improvements in holiday-specific forecasting accuracy, this research will continue to use the original BestCombined forecast from ORTEC as reference.

8.3. Limitations of Hierarchical forecasting

Hierarchical forecasting was initially expected to improve the predictions by utilizing aggregated data across different hierarchical levels. However, its application revealed two limitations.

Firstly, hierarchical forecasting was applicable solely to the volume series without the zeros, providing no insights into whether individual customers would place orders. The Nixtla implementation only supported this limited scope. A potential improvement could be to develop smarter methods to decompose higher-level data, perhaps on a per-weekday basis. For instance, one approach might be to forecast the number of customers per day and subsequently employ a top-down method to predict which customers will place orders based on probabilities derived from predictions at the lowest hierarchical level.

Secondly, the reconciliation process, intended to ensure consistency across hierarchy levels, inadvertently introduced higher forecasting errors at the lowest levels. This approach failed to offer actionable insights regarding specific market segment orders. While it was hoped that this technique would enhance predictions at lower levels, its effectiveness was limited. An alternative application of hierarchical forecasting could involve predicting customer orders on a vehicle basis. This approach is beneficial because, ultimately, it may not matter which individual customer places larger orders as long as collectively they fit within the vehicle's capacity. However, implementing this strategy is challenging without established master routes.

Despite its potential, it has been chosen not to explore hierarchical forecasting further in this study.

9

Evaluating Different Configurations of Route Optimizer

In this chapter, we evaluate various configurations of the route optimizer developed by ORTEC, with a focus on understanding how these configurations affect route planning when incorporating forecasted orders. We will first asses the configurations based on the number of swaps and optimization costs over a three-week test period, using ORTEC's BestCombined method for forecasting. Additionally, we simulate different forecasting qualities using three dimensions: the percentage of correct customers, the number of forecasted customers, and the accuracy of the volume forecast. The goal is to explore how varying these forecasting qualities affects the optimizer's output. Ultimately, we aim to find a configuration or set of configurations that minimize both swaps and optimization costs.

9.1. Performance of Different Configurations

We begin this analysis by testing various configurations of the route optimizer, as introduced in Section 7.3, using the forecasts from ORTEC's BestCombined method. These configurations aim to assess how different settings influence the route planning outcomes. Our primary focus is on two metrics: the number of swaps and the optimization costs. Swaps are expressed as the proportion of customers assigned to a different route of all customers who were correctly forecasted to place an order, represented as a percentage, as explained in Algorithm 7.8. Optimization costs are derived by comparing the costs of the route plan based on forecasted orders against those using actual orders without forecasts, illustrating the additional costs incurred from relying on forecasts. The route optimizer is executed with various configurations for the first three weeks of the test window.

The percentage of swaps over the three weeks of forecasted orders is illustrated as a boxplot, per configuration, in Figure 9.1. First, consider the configurations with zero swaps. The forecasted orders naturally have zero swaps since it is compared to its own output. The fixed input configuration yields zero swaps by preserving the input planning, preventing the tasks from being reassigned to other routes. Similarly, the minimum optimization results in zero swaps, as it focuses solely on optimizing tasks within routes. The construction-only approach skips optimization entirely, resulting in no swaps as well.

Then, for the full optimizer, it is expected that the full optimization using actual orders, not relying on forecasts, displays nearly 100% swaps compared to forecasted planning, as it does not take the forecasted planning as input. Interestingly, the full optimizer, even with input planning, swaps almost every customer due to the included ruin and recreate method, which is absent in light optimizer configurations. Of the three light optimizer configurations, light optimizers 0 and 2 show similar outcomes, indicating that the number of recursions in the local search phase has minimal impact. Light optimizer 1, however, shows significantly reduced swap numbers.

Further comparison of the light optimizer configurations reveals a substantial improvement with the two-step approach from Figure 7.14, compared to the single-step optimization approach from Figure 7.2. The two-step optimization noticeably decreases the number of swaps. Here, we observe that light optimizer 1 consistently maintains a swap percentage below 20% over the three weeks, excluding the two outliers, which satisfies the distributor's preference.

The outliers observed in some of the boxplots correspond to the two holidays occurring within the test window.

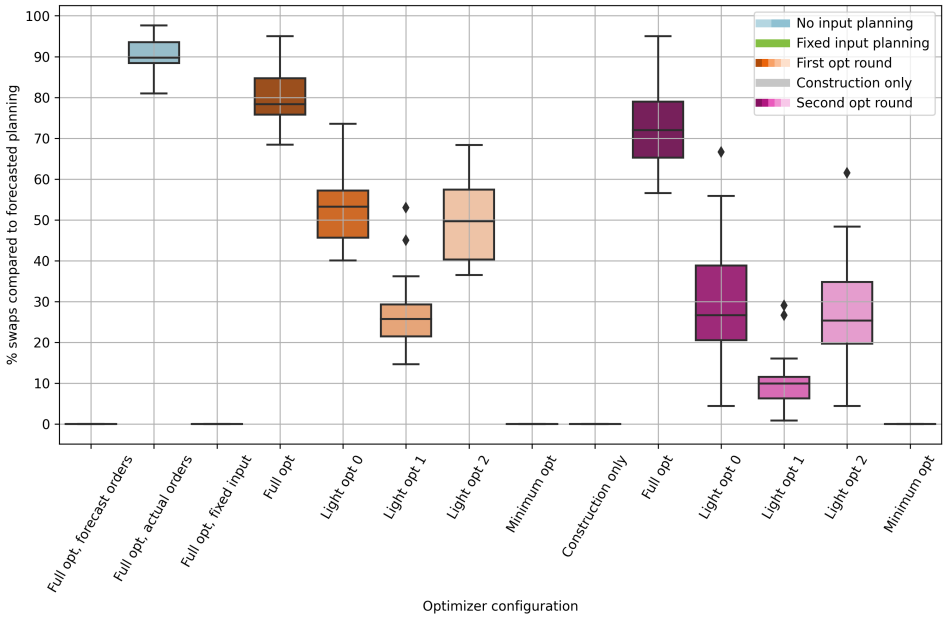


Figure 9.1: Boxplots showing the percentage of swaps from the provisional route plan to the final route plan per optimizer configuration, evaluated over the first three weeks of the test window.

We now focus on the optimization costs, presented relative to actual costs, as a boxplot per configuration in Figure 9.2. Actual costs here are defined as those incurred when forecasts are excluded, and the optimizer derives a heuristic solution to the VRP using only actual orders. Because these actual costs are used as a reference, the boxplot for the full optimizer using actual orders consistently displays a value of zero.

In the leftmost boxplot, we observe that the costs of the provisional route plans are mostly below zero. This indicates an underestimation of actual orders, as also previously noted in Table 8.1. This makes sense because transporting less volume typically requires fewer vehicles, resulting in lower costs.[1] Notably, two outliers, both holidays, have high plan costs. This is because the forecast method predicts too much volume on those holidays, as we have seen in Figure 8.2, resulting in higher plan cost. The bottom outlier is the day after Pentecost, where the forecast heavily underpredicted the volume and the number of customers that placed an order.

It is interesting to see that the full optimizer with fixed input, displayed in green, seems to work quite well. For most days, it has a lower plan cost than the light optimizers for the first optimization round, shown in orange. This indicates that the full optimizer, applied to a subset of customers (those not forecasted), achieves greater efficiency than a light optimizer working across all customers. It also maintains route assignments for correctly forecasted customers without swapping. However, the fixed input approach encounters two notable issues. Firstly, during holidays, the forecast over-predicts the number of customers, resulting in excess vehicle usage. With correctly forecasted customers fixed in their respective vehicles, the optimizer is unable to reduce the number of vehicles involved in the route plan. Secondly, the fixed input approach often violates the capacity constraint when fixed customers

[1] Note that this relationship does not always hold true. In instances where the model predicts too many customers with insufficient volume, route planning costs might actually increase, even if the total volume remains an underestimation.

order more volume than anticipated, as the optimizer cannot resolve the issue by swapping customers. This capacity constraint violation is also a problem for the minimum optimizer and construction-only configurations for the same reason.

For the light optimizers used in the first optimization round, depicted in orange, we observe that light optimizers 0 and 2 produce similar results, consistent with findings in Figure 9.1. Light optimizer 1, however, has slightly higher optimization costs. This is likely due to this configuration making fewer swaps, which reduces opportunities to improve route efficiency.

All configurations in the two-step optimization approach, illustrated in purple, result in lower costs compared to the fixed method. This is because they use the route plan from the fixed approach as their starting point, allowing them to enhance the fixed solution. The minimum optimizer, however, makes no changes, suggesting that the route plan is already optimal within individual routes. Notably, the other configurations minimize or eliminate holiday-related outliers. Additionally, most capacity violations found in the full optimizer with fixed input are solved during this second optimization round.

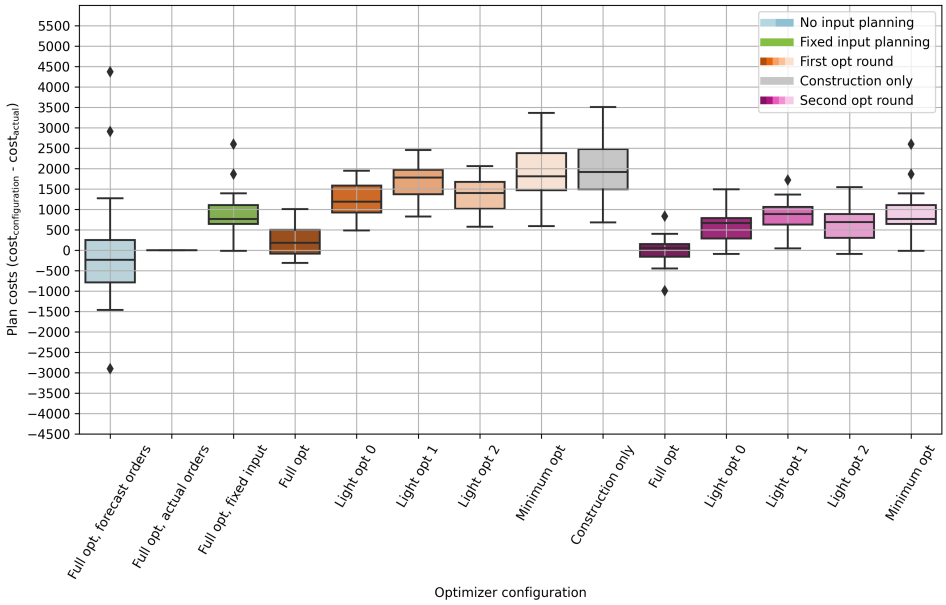


Figure 9.2: Boxplots showing the difference in plan costs for the final route plan when using a provisional input planning versus no input planning, presented for each optimizer configuration and evaluated over the first three weeks of the test window.

One can already see that there is a trade-off to be made between route efficiency and route consistency. If maintaining forecasted routes is not a priority, a single round of optimization with the full optimizer without an input plan would suffice. However, this research specifically focuses on integrating forecasted routes into the route planning process. Therefore, we aim to accept slightly higher plan costs to achieve greater route consistency. To provide a better insight into this trade-off, we visualize the results in a single graph, displaying swaps on the y-axis and plan costs on the x-axis.

Figure 9.3 illustrates this trade-off using mean values for each configuration, rather than the boxplots in Figures 9.1 and 9.2. From this graph, we identify that the light optimizers, when used as a second optimization round, are most interesting. Accordingly, our subsequent research will focus on these three configurations.

This type of graph can also serve as a tool for decision-makers looking to assess configuration performance on specific dates, rather than just the average over the test period. By plotting individual results for particular days, the decision-makers can select the preferred configuration by weighing optimization costs against the number of swaps based on real-time operational needs.

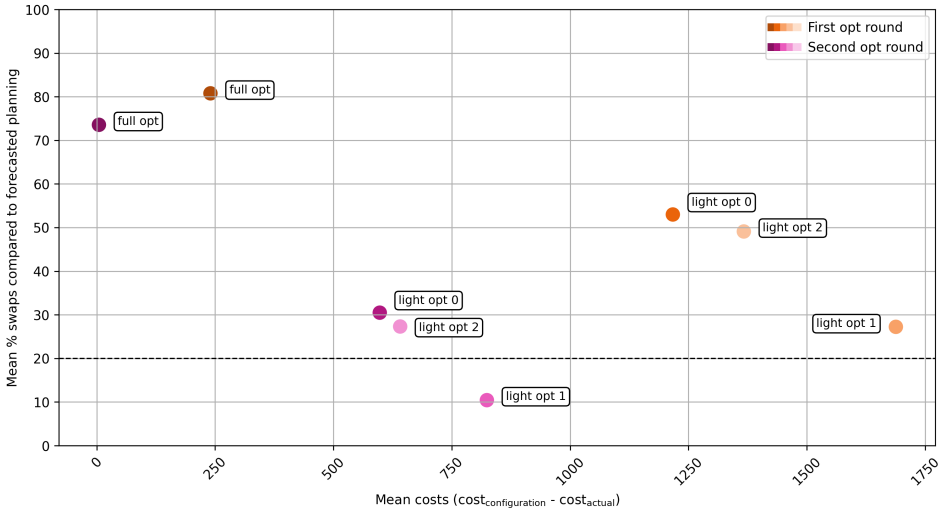


Figure 9.3: Mean percentage of swaps versus mean difference in plan cost for configurations using either a single or double optimization round in the second stage of route planning. This figure illustrates the trade-off between these two key performance indicators.

9.2. Comparison with Baseline Forecast

For reference, we compare the results from the route optimizer using forecasts from the BestCombined method against its baseline, CopyLastWeek, across the three best configurations from the previous section. As shown in Figure 9.4, the BestCombined method yields modest improvements in both the number of swaps+ and plan costs. Considering the improvements in MAE and the accuracy provided by BestCombined, one might have expected a greater impact on the number of swaps and plan costs. In the following section, we further explore how different forecasting qualities influence the routing metrics.

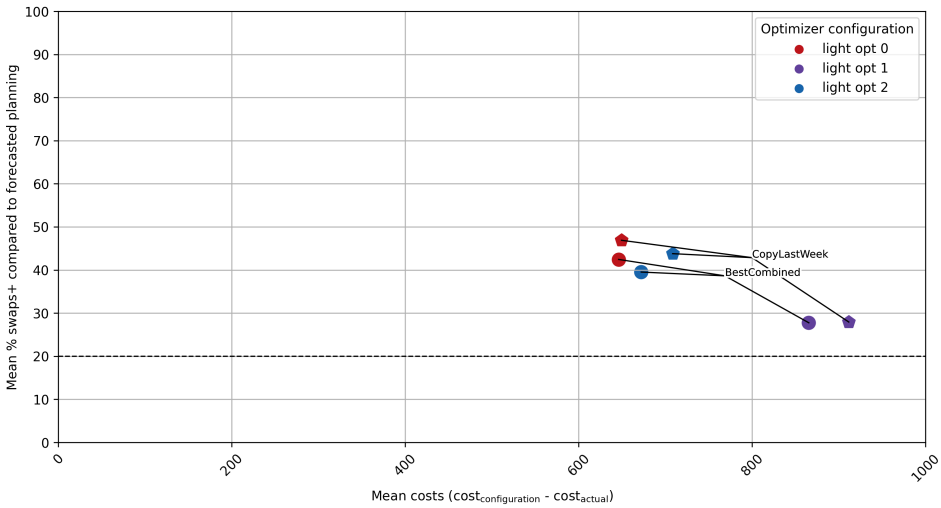


Figure 9.4: Comparison of route optimizer results using BestCombined forecasts and the CopyLastWeek baseline for the three light optimizer configurations within the double optimization framework.

9.3. Effect of Different Forecast Qualities

In this section, we investigate how varying forecasting qualities impact the output of the route optimizer. Here, we want to determine the extent to which optimization results can be enhanced through improvements in forecasting quality. We consider three dimensions of forecasting quality: the percentage of correctly forecasted customers, the ratio of forecasted customers to actual customers, and the quality of volume predictions. These dimensions are tested for the three configurations found in the previous section, specifically the light optimizer settings implemented as a second optimization round following a full optimization with fixed input. We use the same test weeks as the previous section.

Since this section compares different forecasting qualities, we will now use swap+ from Definition 7.5 as a metric. This definition accounts for non-forecasted customers, ensuring an equal comparison across varying forecasting qualities.

We begin by analyzing the first dimension of forecasting quality. In this simulation, we keep the number of forecasted customers equal to the actual number of customers. The simulated forecasts for a specific day are generated by randomly selecting a percentage of customers from the actual orders, while the remaining customers and their ordered volumes are sampled from orders on other days. It is important to ensure that these new, intentionally incorrect, customers are not present in the set of actual orders. Given that the number of forecasted customers matches the actual customer count, each simulated forecast has an equal number of false positives and false negatives. Such balance is not typically found in real-world forecasts.

Figure 9.5 shows the mean swaps+ over mean plan costs across different percentages of correctly forecasted customers. The endpoints of the graph, with forecasting qualities of 0% and 100%, are intuitive. At 0% quality, no customers are correctly forecasted, resulting in 100% swaps+ by definition. The plan cost should be almost zero because the optimizer runs the full optimization with fixed input without any correctly forecasted data. Conversely, at 100% quality, the provisional route plan in the first optimization step already includes all correct customers, thereby establishing an optimal route plan that requires no swaps and no additional costs. The slight deviation to the left side indicates that subsequent optimization steps after the provisional plan still achieved minor improvements. It is also intuitive for the graph to exhibit a parabola-like shape between the endpoints, as we know that a forecasting quality between 0 and 100 encounters additional plan costs, as observed with the BestCombined method.

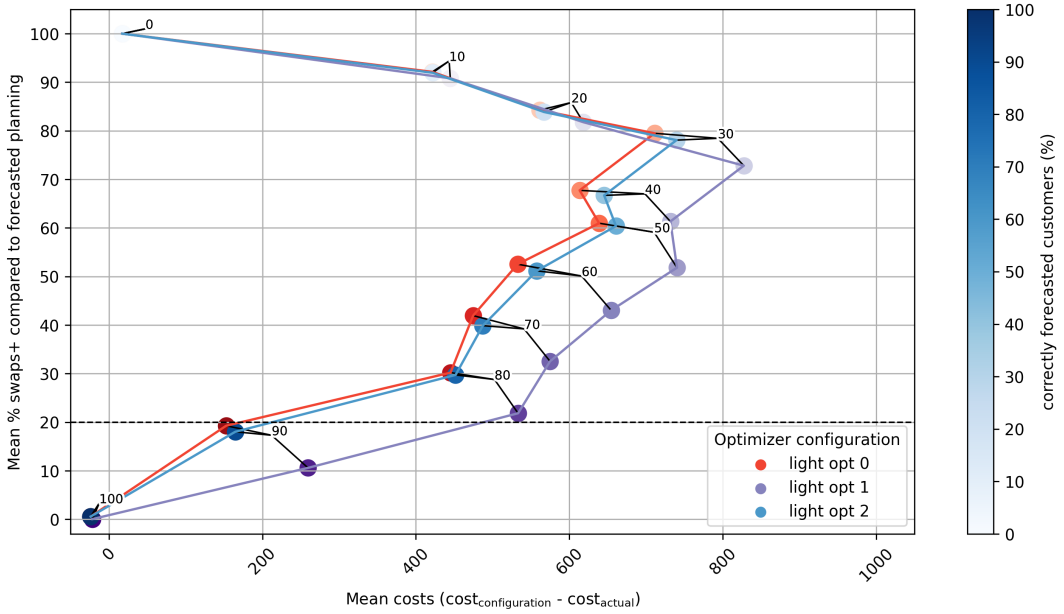


Figure 9.5: Mean percentage of swaps+ and difference in plan cost for different forecasting qualities $x \in [0, 10, 20, \ldots, 90, 100]$, where x denotes the percentage of correctly forecasted customers (with the total number of customers fixed to the actual value). Evaluated for the three light optimizer configurations within the double optimization framework.

It is interesting to see that the graph exhibits a skewed, non-symmetric shape, likely attributable to the implementation design of the forecasted orders (Figure 7.2) and the used configurations. First, a full optimizer is applied to the forecasted orders, followed by a two-step optimization process using actual orders. These steps effectively function as one optimization step with some restrictions: first, a full optimizer with fixed input, then a light optimizer. This restricted optimization on the actual orders results in a non-symmetric optimization process. As a result, the peak of the parabola occurs at 30% rather than the midpoint of 50%. Furthermore, the graph reveals that the most significant improvement in optimization costs occurs between forecasting qualities of 80% and 100%, emphasizing the importance of high forecasting accuracy.

Additionally, it is noteworthy that for light optimizer 1, nearly all swaps+ originate from non-forecasted customers rather than customers that were correctly forecasted to place an order. This observation is evident as the swaps+ values closely correspond to the percentage of customers that were not forecasted.

Continuing to the second dimension of forecast quality, we analyze the impact of the number of customers forecasted. In this setup, forecasting is set to correctly predict 80% of the customers that actually placed an order. These customers are sampled randomly from the actual customers. The forecasts are filled up to a specified percentage of the actual customer count. For example, if there are 100 actual customers, a forecast quality of 120% would include 80 correctly forecasted customers alongside 40 incorrectly forecasted ones. Note that the 100% point in this simulation corresponds to the 80% point on Figure 9.5.

Figure 9.6 shows the mean swaps+ over mean plan costs across different amounts of forecasted customers. The swaps+ stay relatively constant but seem to decrease slightly as the number of forecasted customers decreases. Notably, the light optimizer 1 configuration again shows that swaps+ are mainly originating from non-forecasted customers (20%) rather than from the correctly forecasted ones. In terms of optimization costs, we observe a zigzag pattern around 100%, indicating no clear trend. However, it is apparent that once the forecast exceeds 110%, optimization costs begin to rise consistently. While this zigzag behavior makes it hard to draw definitive conclusions, the overall result suggests that a more careful and conservative approach to forecasting customer numbers is beneficial.

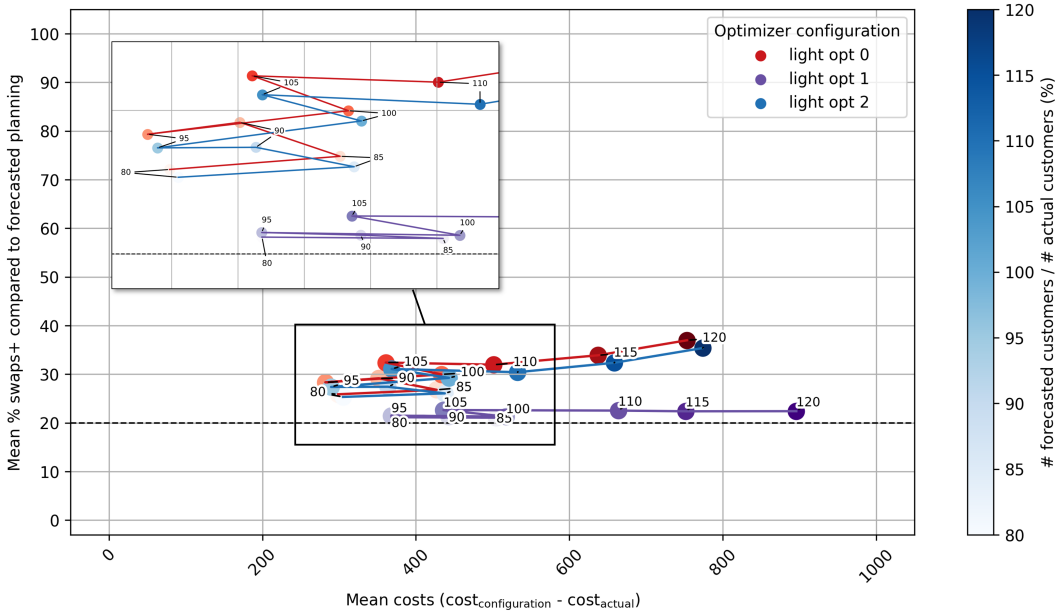


Figure 9.6: Mean percentage of swaps+ and difference in plan cost for different forecasting qualities $x \in [80, 85, \ldots, 115, 120]$, where x denotes the number of forecasted customers (with the number of correct customers fixed to 80% of actual customers). Evaluated for the three light optimizer configurations within the double optimization framework.

Lastly, we explore the third dimension of forecasting quality, concentrating on the accuracy of volume predictions. Simulated forecasts are obtained by taking all customers from the actual orders and adjusting their ordered volumes. Specifically, the volume for each customer is modified by multiplying it by a percentage to decrease or increase it.

Figure 9.7 presents the results for varying forecasting qualities in terms of volume accuracy. When the forecasted volume underestimates the actual demand, it directly leads to swaps and higher optimization costs. The swaps occur because customers no longer fit within the provisional route's vehicles, requiring swaps to avoid breaching the capacity constraint. The increase in optimization costs is likely due to inefficient routing adjustments. On the other hand, forecasting excessive volume leads to increased optimization costs without significantly affecting swaps. This occurs because the provisional route plan already allocates enough vehicles to accommodate actual orders. Only when the excess space from overestimated forecasts becomes too high does the optimizer seem to address this by swapping customers to enhance route efficiency. For light optimizers 0 and 2 (represented in red and blue), it is interesting to see that the results start to diverge when forecasting quality deteriorates. So, a reduced number of recursions leads to slightly fewer swaps but higher plan costs.

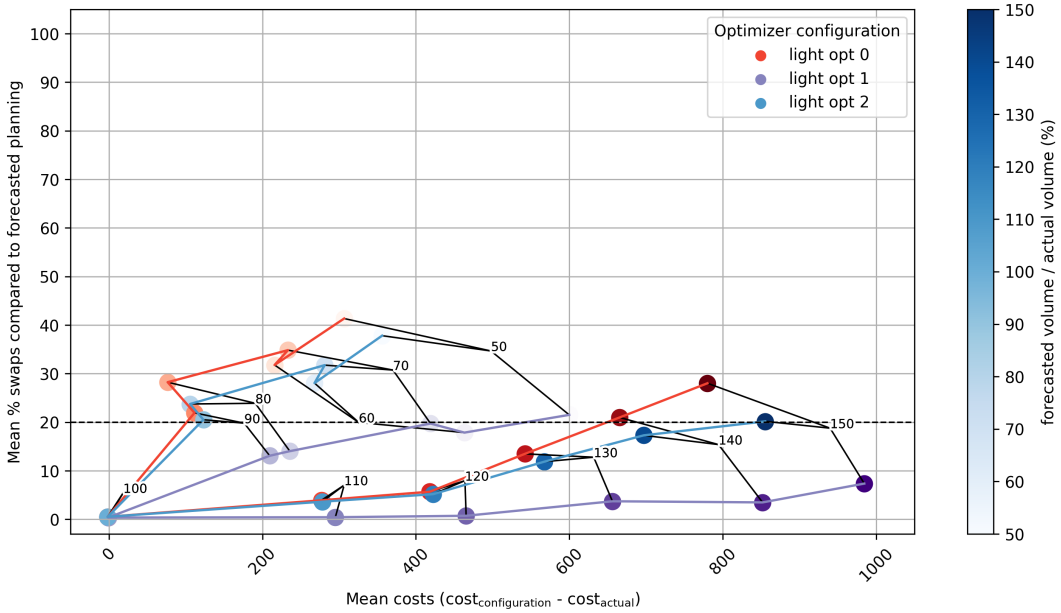


Figure 9.7: Mean percentage of swaps and difference in plan cost for different forecasting qualities $x \in [50, 60, \ldots, 140, 150]$, where x denotes the percentage of actual volume ordered per customer (with accuracy fixed at 100%). Evaluated for the three light optimizer configurations within the double optimization framework.

This analysis of volume accuracy highlights the importance of precise volume predictions to minimize swaps and unnecessary plan costs. Both underestimating and overestimating forecast volumes can lead to suboptimal planning outcomes, with underestimation primarily affecting swaps and overestimation primarily affecting plan costs.

9.4. Discussion

This study used a standard configuration from ORTEC, designed to find the most efficient route plan. Although the optimizer allows for inputting a route, ORTEC had not tested it with forecasted routes. Given the scope limitations, designing a configuration from scratch was not pursued. Instead, we modified the standard setup to explore its effectiveness in managing forecasted orders. While these adjustments provided valuable insights, there is potential for further optimization with more time and resources, specifically regarding the optimizer's approach to integrating input planning and determining when to swap customers to another route.

Within the configuration, tuning hyperparameters can improve the optimizer's output. Currently, only the minimum estimated gain and recursion count parameters for the local search method have been explored to a limited extent. These parameters, particularly the minimum estimated gain, could be refined further. It is also possible to determine the minimum estimated gain based on other metrics, such as optimization costs or driving time. Furthermore, the parameters can be individually tailored for each heuristic algorithm to enhance performance.

To evaluate swaps, we calculated the number of customers planned in different vehicles when comparing the provisional route plan to the final route plan. Thus, our analysis focused primarily on swaps at a per-vehicle level. However, it did not account for scenarios where groups of customers are swapped collectively to a new vehicle. If, for example, all customers in a vehicle are swapped to another vehicle, the wholesale distributor can simply move the new vehicle to the cross-dock instead of reallocating all individual orders to a different cross-dock. It is, however, hard to define a good metric for swaps that effectively tracks customer groups rather than individual vehicle assignments. Nonetheless, such an approach could potentially reduce the number of observed swaps.

A major challenge for route planning is managing vehicle capacity. In this case study, violations of the capacity constraint arose when customers fixed to a particular route ordered more volume than was originally forecasted. Resolving violations of the capacity constraint proved to be difficult using the standard configuration provided by ORTEC. This research introduced several improvements to the standard configuration to better handle potential violations, as previously described in Section 7.4. These measures appeared to be effective. However, a more detailed assessment of capacity constraint violations should be conducted before proceeding with implementation.

10

Forecasting using Neural Networks

In this chapter, we explore the application of the neural network models introduced in Section 4.4 for forecasting customer orders. We will follow the methodologies as described in Section 6.2. The aim is to assess whether neural networks can achieve similar or superior forecasting results compared to traditional methods.

10.1. Initial Forecasting Results

In this section, we present the results from training the neural network models across all time series and features outlined in Table 6.1. Despite extensive testing, the DeepAR model did not yield reasonable results, and hence it has been excluded from consideration here. A more detailed explanation can be found in Section 10.3.The raw forecasts from TFT and N-HiTS already demonstrated some improvements in terms of the Mean Absolute Error (MAE). Something interesting to note is that the TFT shows significantly higher MAE in the validation period, but improved results in the test window.

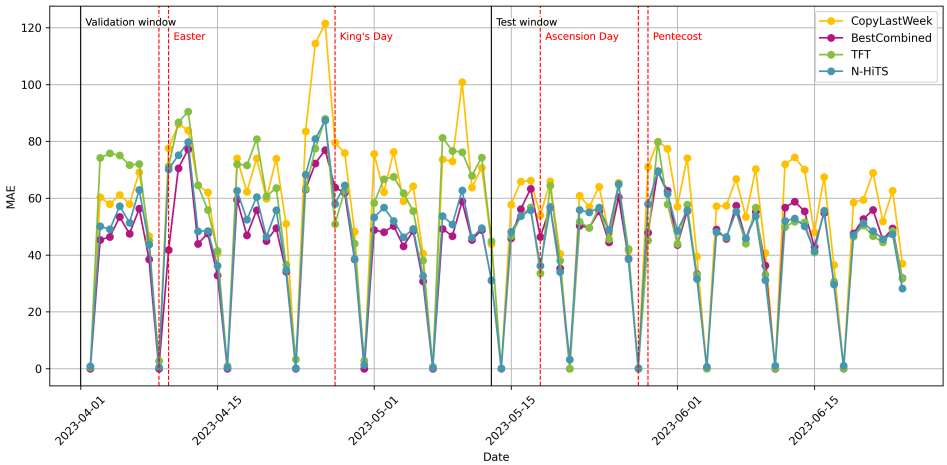


Figure 10.1: MAE of filtered forecasts from TFT and N-HiTS models compared to the MAE of the BestCombined method from ORTEC and the CopyLastWeek baseline. Neural network forecasts are filtered by setting forecasts below the series-specific minimum volume to zero.

Upon further inspection, it became evident that the outputs from the neural network models contained noise, including small positive and negative volumes. This can be observed, for example, in the forecasts for Sundays, which are incorrectly predicted as non-zero. To address this issue, we implemented a straightforward postprocessing step to eliminate the noise. For each individual forecast, we determined the minimum ordered volume based on historical data before the validation window. If a forecast

fell below this minimum volume, it was set to zero.

The post-processed results, including MAE and Accuracy, are presented in Table 10.1. Additionally, two dimensions of forecasting quality from Section 9.3 are given, as they significantly impact route planning. Note that the values here are averaged, whereas in section Section 9.3, the same quality was applied to each day and every customer. Although the MAE shows improvements over the traditional methods, we can see that on the other metrics, the neural network models perform poorly. The reason the accuracy is lower can already be seen in the forecasting quality regarding the number of customers. Despite removing small positive orders during postprocessing, we continue to predict an excessive number of customers. Specifically, the neural network models do predict fewer false positives but many more false negatives, which gives this imbalance. To address this, we will try to enhance the postprocessing step in Section 10.2, aiming to reduce the number of false negatives and thereby improve the accuracy.

			forecast quality (%)	
Method	MAE	Accuracy	# customers	Volume
CopyLastWeek (Baseline)	50.952548	0.914201	101.917882	**102.677288**
BestCombined (ORTEC)	42.648651	**0.924149**	**98.812771**	96.244327
TFT	**41.724360**	0.873693	117.795832	74.494962
N-HiTS	42.024162	0.828477	131.948974	76.587383

Table 10.1: MAE and accuracy over the test period for all methods. Neural network forecasts are filtered by setting forecasts below the series-specific minimum volume to zero.

In Table 10.1, it is evident that the neural network models significantly underpredict the total ordered volume, indicating their poor performance in forecasting these volumes. This issue is likely attributable to the loss function, which only looks at differences in the volume and thus struggles with predictions involving numerous zeros. For the loss function, a minor deviation above zero is considered equivalent to a small deviation above a peak, which might work well for the MAE but undermines accuracy. Additionally, these models tend to predict the volumes with caution when uncertainty is high. An example of a forecasted time series is given in Figure 10.2, which illustrates that both models underpredict the actual volumes. We will tackle this underestimation of the volume in Section 10.4.

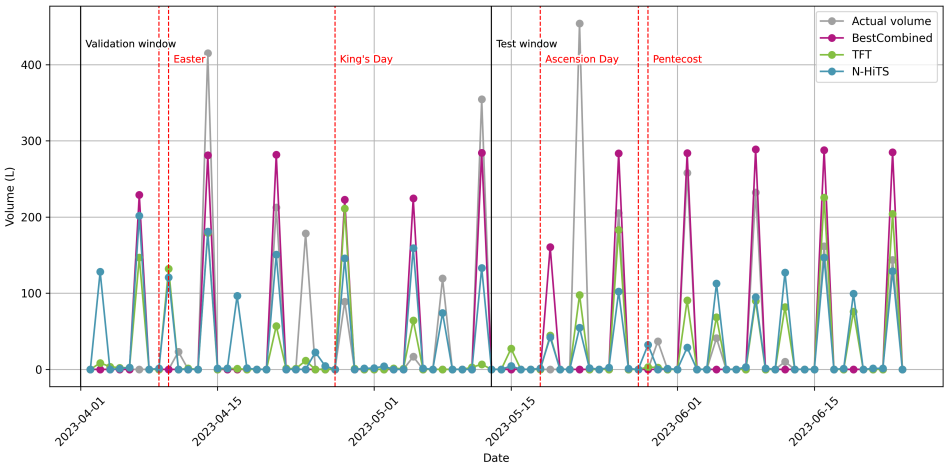


Figure 10.2: Example time series with forecasts from TFT, N-HiTS, and the BestCombined method.

Table 10.2 presents the running times required for validation and testing.

Method	Validation time (s)	Test time (s)
TFT	23466	448
N-HiTS	3548	90

Table 10.2: Running times for predicting the validation window (including hyperparameter tuning) and test window for TFT and N-HiTS. N-HiTS shows significantly lower training times compared to TFT.

10.2. Postprocessing

The previous section implemented a straightforward postprocessing step. This section aims to enhance the accuracy of the neural network forecasts by implementing more sophisticated postprocessing techniques. The Mean Absolute Error (MAE) and accuracy of the different postprocessing steps are summarized in Table 10.3.

To begin, let's revisit the concept of setting a forecast to zero if it was predicted below a certain threshold. In the previous section, we used the minimum volume from historical data as the threshold. Here, we consider using higher threshold values based on quantiles of historical orders. This threshold can be applied to individual time series, but since each customer has three time series (one for each product group), we determined the quantiles based on the customer's total daily volume. Therefore, if the cumulative volume of the three product types is below the customer-specific threshold, all three values are set to zero. A common quantile value could be applied across all customers, but we decided to determine a distinct quantile value for each customer. The optimal quantile value is selected to maximize accuracy during the validation period, with quantiles ranging from 90% to 95%.

This approach of using quantile thresholds showed improvements in terms of accuracy compared to using the minimum volume (equivalent to the 100% quantile). However, it still underperformed when compared to ORTEC's BestCombined method. While higher thresholds effectively reduced the number of false positives, they eventually led to an increase in false negatives. Therefore, it appears that further refinement of quantile thresholds is unlikely to yield significant gains in accuracy.

Finally, we attempted to improve the accuracy by separately forecasting whether a delivery would occur, using binary 0/1 values instead of volume series. The neural network models were trained using the same hyperparameter space as used for the volume forecast. For the loss function, we used a distribution loss with a Bernoulli distribution to account for the binary values. Additionally, lagged values for 7, 14, 21, and 28 days were introduced as features, similar to the approach taken in the proof of concept. The results of this binary forecast are then multiplied by the volumes obtained in Section 10.1. Note, however, that this postprocessing strategy closely aligns with the two-step approach detailed in Section 6.1.1. Notably, the training durations are significantly lower than those of the volume models. Detailed information on running times and the accuracy of these models can be found in the Appendix, Tables A.1 and A.2.

Method	Postprocessing	MAE	Accuracy	# customers (%)
BestCombined	min_volume	42.516238	0.924310	98.812771
	quantile	42.505094	0.923666	95.917864
	nn_delivery	x	x	x
TFT	min_volume	41.724360	0.873693	117.795832
	quantile	41.786533	0.880345	**100.747083**
	nn_delivery	**40.784442**	**0.934939**	94.108934
N-HiTS	min_volume	42.024162	0.828477	131.948974
	quantile	41.849958	0.872752	105.984739
	nn_delivery	42.067071	0.917918	87.302258

Table 10.3: MAE, accuracy, and number of forecasted customers over the test period for each method after different post-processing steps. The min_volume sets forecasts below the series-specific minimum volume to zero; quantile determines an optimal customer-specific threshold below which forecasts are set to zero; and nn_delivery refers to an additional neural network-based delivery filtering step.

Table 10.3 shows that the third postprocessing step achieves significantly higher accuracy than other postprocessing steps and even surpasses the BestCombined approach. This already suggests that training a separate model for the delivery series could be crucial for enhancing forecasting outcomes. It is important to note that these postprocessed results lead to further underestimation of total volumes compared to Table 10.1 because the postprocessing only focuses on excluding additional customers. This motivates training the neural network models solely on volume data without zeros as well, which will be discussed in Section 10.4.

10.3. Limitations of DeepAR

DeepAR promises to leverage hundreds to thousands of similar time series, by effectively fitting the outcomes of LSTMs to a distribution. However, despite testing various hyperparameters, the results obtained were disappointing. The generated forecasts failed to align with the actual orders meaningfully.

When using the normal distribution, the results seemed to mimic Croston's method, producing averaged outcomes rather than capturing the true nature of zero and large orders. The Tweedie distribution, which is zero-inflated, initially seemed promising but ultimately fell short. This limitation is likely attributable to the varied distribution patterns among different customers, as can be seen in Figure 10.3. For instance, some customers have almost no orders (Figure 10.3a), some have biweekly orders, and some order every weekday (Figure 10.3c). Although the neural network model is capable of fitting the mean and standard deviation per time series, the power variance, which determines the weight of zeros, must be set consistently for the entire model. Consequently, fitting a singular distribution to accommodate all customers proved unfeasible.

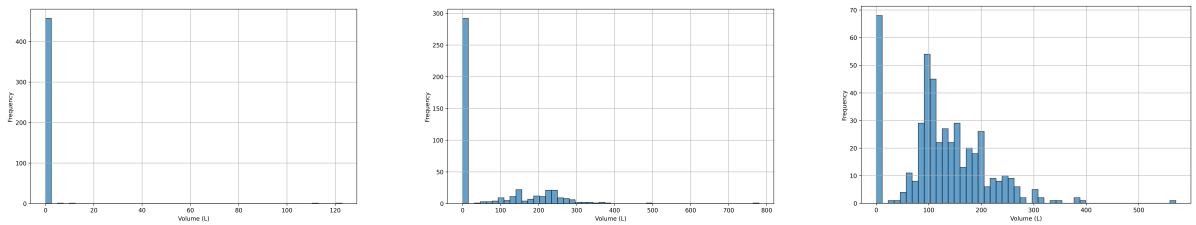


Figure 10.3: Example volume distributions for different customers, illustrating the variation in zero-inflation across customer profiles. The x-axis represents volume, and the y-axis represents frequency.

The issue seems to lie in the dissimilarity of the time series data, which makes it difficult to fit a suitable distribution to these diverse series. However, it's important to note that we cannot definitively conclude that this is the sole reason for the poor results. There is a possibility that untested hyperparameters may yield satisfactory outcomes.

10.4. Two-step Approach with Neural Network Models

The neural network models presented a challenge in accurately predicting the total volume, particularly due to their tendency to heavily underpredict it. An ideal resolution would be a more sophisticated loss function, that combines a binary loss with one for the volume series, such as MAE. However, such a function was not available, and the scope of this research did not accommodate the development and testing of a new loss function. Consequently, we tested a strategy similar to ORTEC's BestCombined method, separating volume and delivery series.

Using linear interpolation from Python's pandas library, we constructed the volume series. The neural networks are trained using this interpolated series as the target variable. The resulting forecast is then multiplied by the delivery forecast from the last postprossessing step presented above. Since this separation in volume and delivery series resolved the problem with different zero-inflated series, we revisited the DeepAR model. This model is tested using a Bernoulli distribution for the delivery series and a Normal distribution for the volume series.

The results of the two-step approach for the three neural network models are presented in Table 10.4. The training times for the separate models are listed in the Appendix, Table A.2. The accuracy and predicted number of customers for TFT and N-HiTS are consistent with the results from the final post-processing step, showing only slight improvements, as expected. The MAE seems to be slightly higher than was observed when training a single model. However, more importantly, the volume continues to be underpredicted, although to a lesser extent than previously observed. This indicates that even without zeros in the volume data, the models still underpredict the total volume. It is difficult to say

whether this issue is attributed to the model design, the chosen hyperparameters, or the loss function, although the latter is the most likely factor.

			forecast quality (%)	
Method	MAE	Accuracy	# customers	Volume
CopyLastWeek (Baseline)	50.952548	0.914201	**101.917882**	**102.677288**
BestCombined (ORTEC)	**42.648651**	0.924149	98.812771	96.244327
TFT	42.699000	**0.935979**	94.108934	87.832563
N-HiTS	42.693970	0.921225	87.302258	89.701742
DeepAR	43.943460	0.933898	95.779833	102.738534

Table 10.4: MAE, accuracy, number of customers, and quality of total volume prediction over the test period for each method. Here, the neural network forecasts use a two-step approach, similar to BestCombined, separating the prediction of delivery occurrence and the delivery volume series.

Surprisingly, the DeepAR method performs remarkably well in terms of accuracy. While the MAE is slightly higher compared to other models, DeepAR exceeds the other neural network models in predicting both the number of customers and the total volume. This is also visible in Figure 10.4, where the total forecasted volume is presented. Additionally, DeepAR provides the most accurate predictions among all methods for holidays that do not fall on a Monday (King's Day and Ascension Day). However, it tends to over-predict more than some other methods on Easter Monday and Whit Monday. N-HiTS, on the other hand, displays the most unusual behavior in total volume, appearing to mimic the CopyLastWeek method by repeating the effect of holidays one week later.

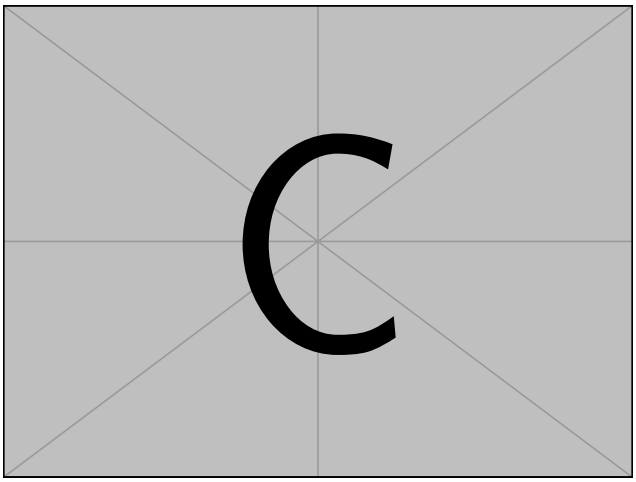


Figure 10.4: [*This figure contains confidential information and is therefore only available in the confidential appendix.*] Aggregated volume forecast using two-stage forecasting models: BestCombined, TFT, N-HiTS, and DeepAR. Both actual and predicted volumes are normalized by dividing by the maximum observed actual volume.

In Chapter 11, we will examine how these metrics ultimately impact route planning.

10.5. Additional Insights

Finally, we share some additional insights gained during the testing of neural networks, including various configurations and methodologies.

10.5.1. Forecasting New Customers

The hope of using a global model, which uses data from all time series in one model, is that it can learn patterns from similar customers. This allows the model to leverage these learned patterns for new customers who have limited historical data.

We define a new customer as one who has placed their first order after the cut-off date from the validation window. Applying this criteria yields 92 individual time series, corresponding to approximately 30 customers.

Table 10.5 presents the accuracy and mean absolute error (MAE) for the models applied to these new customers. We observe that all neural network models show improved performance in both accuracy and MAE.

Method	MAE	Accuracy
CopyLastWeek (Baseline)	17.727565	0.937371
BestCombined (ORTEC)	15.621669	0.940994
TFT	**11.736780**	**0.960404**
N-HiTS	13.710071	0.941770
DeepAR	14.511910	0.956522

Table 10.5: MAE and accuracy of new customers with their first order after the validation cut-off date. Here, the neural network forecasts use a two-step approach, similar to BestCombined, separating the prediction of delivery occurrence and the volume.

Upon examining the forecasts for new customers, we find that the BestCombined approach struggles with these predictions, often defaulting to forecasting zeros or merely replicating the previous week's orders. This pattern suggests that the models integrated within the BestCombined method struggle to forecast new customer behaviors accurately.

An example illustrating a new customer is shown in Figure 10.2, where its first non-zero value is after the cut-off date. We can see that the BestCombined approach chooses the CopyLastWeek forecast, meaning its first prediction is two weeks after the first order. By simply copying orders, the method visually makes some mistakes in prediction. Meanwhile, the neural network models take longer in their predictions, indicating a need for higher confidence that the customer will continue ordering. Furthermore, the neural network models fail to capture peaks in ordered volume on days near holidays, despite having these days included as features.

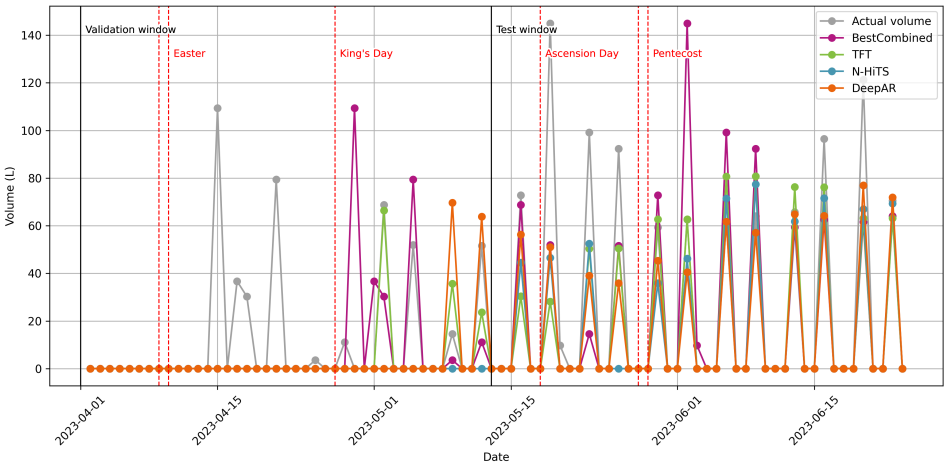
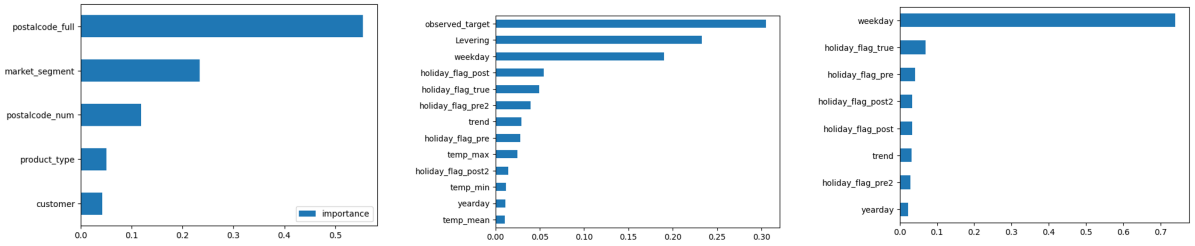


Figure 10.5: Example time series of a new customer with forecasts from BestCombined, TFT, N-HiTS, and DeepAR.

10.5.2. Feature Importance TFT

Understanding feature importance in machine learning models can provide valuable insights into the factors influencing predictions. For the traditional models, random forests are able to provide feature importances. The TFT model offers a sophisticated mechanism for assessing feature importance in a similar manner. The features are split into three categories: static, historical, and future covariates. The feature importance distributions for the features are presented in Figure 10.6.

Among the static features, full postal code and market segment emerge as the most significant contributors to the model's predictions. Conversely, the customer feature shows the least importance, likely due to its applicability being restricted to only a limited number of series (maximum of three) per value, which constrains its impact across broader datasets. For the past covariates, the observed target, which is the volume, along with delivery information ('levering'), is considered most important, as would be expected. Following these, the weekday and holiday features also hold notable importance. Surprisingly, all features with some information on the yearly seasonality, like yearday or temperature, have a low feature importance. For future data, the weekday stands out as the most important feature.



(a) Static covariates. **(b)** Past covariates. **(c)** Future covariates.

Figure 10.6: Feature importances of the TFT trained on raw time series data without splitting into delivery and volume components.

Figure 10.7 shows the attention distribution over time. The time axis shows the lookback window used by the model, which is 28 days. In this figure, we can see clear peaks of attention at 7-day intervals, underscoring a weekly periodic pattern. Additionally, the model also places high attention on the day before the forecast window starts.

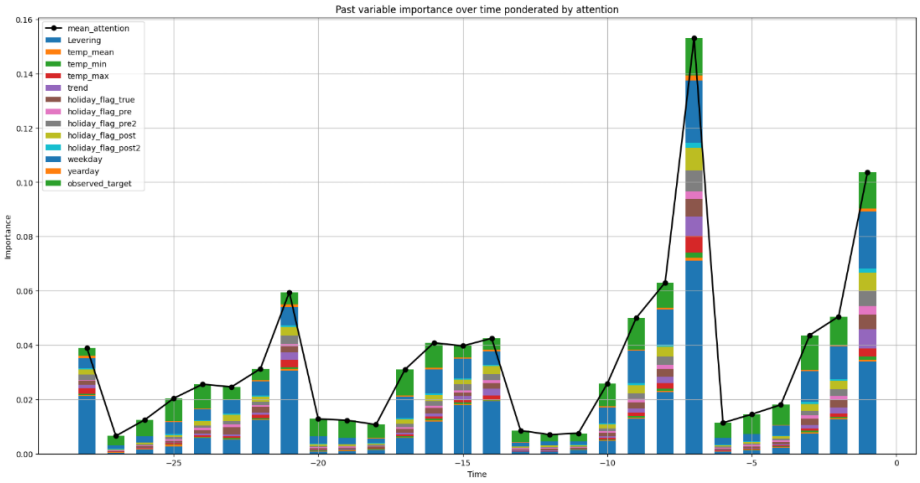


Figure 10.7: Attention over a look-back window of 28 for the TFT trained on raw time series data without splitting into delivery and volume components.

The feature importance allows for a more transparent understanding of model behavior, providing clients with insights beyond the standard output of a blackbox model. This transparency can help validate the model's reliability and build trust. However, it is important to weigh the benefits against the increased computational costs, as training time is considerably longer for the TFT compared to the other models.

10.5.3. Different Embeddings

This section explores different embedding strategies for temporal features such as weekdays and yeardays in neural network models. This investigation was conducted on a subset of 20% of the customers using a single product type per customer.

Two embedding approaches were tested. First, a one-hot encoding for weekdays and a cyclic encoding for yeardays, resulting in 7+2 feature columns, similar the temporal features in the PoC. Second, a categorical encoding for both weekdays and yeardays with a single feature column that is scaled between 0 and 1, resulting in 1+1 feature columns.[1]

Results indicated that the first embedding slightly underperformed compared to the second. Moreover, the first embedding required twice the amount of training and prediction time for the Temporal Fusion Transformer. This is probably due to the complex mechanisms used for feature selection. In conclusion, while traditional forecasting techniques often benefit from one-hot encoding, this approach may not be as effective for neural network models.

10.6. Discussion

At the beginning of this research, the primary focus was on improving the MAE and accuracy. However, as research progressed in route planning, it became clear that other metrics, particularly the number of customers and volume deviations, can be more important. If these factors had been identified earlier, more attention would have been paid to evaluating models and postprocessing steps based on these metrics. For instance, hierarchical forecasting was initially set aside due to its negative impact on predictions at the lowest hierarchy level. Nonetheless, later it showed that sacrificing precision at lower levels could be advantageous if it leads to improved forecasting accuracy at higher hierarchical levels. Additionally, postprocessing on quantiles is now performed by finding the best quantile based on accuracy. However, integrating additional dimensions could provide better results.

As explained, one potential improvement lies in the application of hierarchical forecasting. Since there are no hierarchical forecasting implementations for the purpose of this study, we would require a manual implementation. One idea would be to predict the number of customers per day and then refine the customer selection by sampling using a weight based on forecasted probabilities.

Furthermore, there remains potential for further research into the impact of varied loss functions on model performance. While N-HiTS and TFT employed a multiquantile loss for volume predictions, DeepAR utilized a distribution loss. The superior performance of DeepAR may be attributed not only to its distinct model architecture but also to this choice of loss function. Similarly, while successful outcomes have been achieved by separating delivery and volume forecasts, there remains the potential for unified forecasting using a single model with a combined loss function to produce equally promising results.

Regarding holidays, true holiday adjustments have been implemented as an improvement, but the variability per holiday still gives some complexities. Given the substantial increase in training time associated with additional features, it is not recommended to create separate feature columns for each distinct holiday. Grouping holidays with similar characteristics could be an effective strategy for addressing these disparities. Having business rules specified for each holiday by the distributor would have made the process of integrating holiday features easier.

[1] The weekday column has values from 1 to 7 and the yearday column has values from 1 to 365, both scaled to the range [0,1].

11

Combining Forecasts from Neural Networks with Route Optimizer

In this final results chapter, we evaluate the performance of various forecasting models within the context of a route optimization process. Specifically, we compare the BestCombined method from ORTEC with the forecasts from the neural network models. Using the same three-week test period and configurations outlined in Chapter 9, we examine how each forecast impacts the routing consistency and efficiency. The results of this evaluation are illustrated in Figure 11.1.

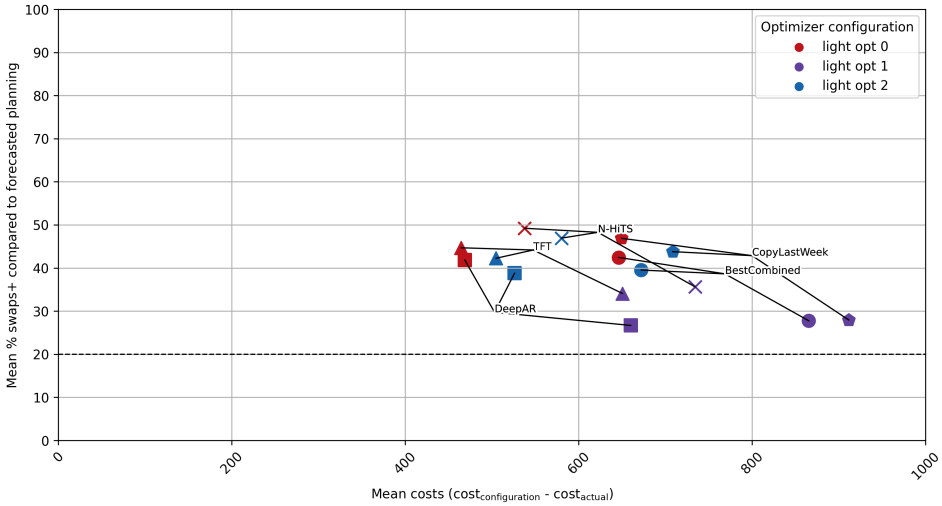


Figure 11.1: Comparison of route optimizer results using forecasts from neural network models, ORTEC's BestCombined method, and the CopyLastWeek baseline for the three light optimizer configurations within the double optimization framework.

We can see that the DeepAR model provides the best results in terms of both swaps+ and route plan costs. Despite having a higher MAE in forecast prediction compared to other models, DeepAR strictly outperforms the BestCombined method from ORTEC. Conversely, the TFT achieves similar route plan costs but exhibits higher swap+ rates, even though it has the highest accuracy. This increase in swap+ rates may be attributed to systematic underestimations in volume predictions, as we have seen in Section 9.3, where it was noted that such volume underestimations contribute to a higher number of swaps. This highlights the importance of accurate forecasting on a global scale rather than for individual forecasts.

Comparing the improvement from CopyLastWeek to BestCombined with the improvement from BestCombined to DeepAR, we observe that the results in this thesis show a significantly larger improvement than what was achieved in the proof of concept.

A noteworthy finding is that all forecasts and configurations exceeded the 20% swap+ threshold. While this exceeds the client's specified limit, it is crucial to consider that the client's exact objectives may not be fully understood, particularly in terms of handling unforecasted customers. When evaluating the swap percentages while excluding non-forecasted orders, as shown in the Appendix, Figure A.2, the forecasts indicate a potential for achieving swap percentages of around 10%. So, if there is an efficient way to handle incoming orders that were not forecasted, it is feasible to accomplish a swap percentage below the threshold. From Figure 11.1, but more clearly from Figure A.2, we can also conclude that the number of swaps primarily depends on the optimizer configuration used. Improvements in forecasting lead to more efficient routings, probably because accurate demand predictions enable better route planning in the first stage.

Part IV

Conclusion and Recommendations

12

Conclusion

The goal of this research was to evaluate the impact of the proposed framework by ORTEC on the swaps and to investigate new forecasting and route planning strategies to improve the impact while maintaining the efficiency of routes.

The examination of the optimizer revealed several insights into the effectiveness of different configurations. The ruin and recreate method in the full optimizer proved to play a major role in finding optimal routes, albeit without maintaining the customer assignments of the input routes. On the other hand, employing fixed input routes yielded promising results, though it led to capacity violations. This can be attributed to the forecasts from ORTEC underpredicting the volume of customers. These violations were be addressed by applying a secondary optimization phase after executing the full optimizer with fixed input routes. Within the light optimizers, we found that increasing the minimum estimated gain, in local search methods that look outside the routes, has a bigger impact on the swaps than reducing the number of recursions. However, fewer swaps do come with a higher plan cost. A trade-off has to be made by the decision maker, possibly on a daily basis, between the number of swaps and plan costs.

Additionally, simulating different forecast qualities highlighted the importance of evaluating forecasting performance on a global level beyond the MAE and accuracy. Specifically, overpredicting the number of customers by more than 10% notably increases planning costs, leading to inefficient resource allocation. Furthermore, the tendency to overpredict the total volume similarly increases plan costs, while underpredicting the total volume significantly increases the number of swaps required.

Turning to the forecasting results, we assessed three models: DeepAR, TFT, and N-HiTS. All models demonstrated their capability to operate effectively on a global dataset consisting of thousands of time series. The models provide comparable or even superior outcomes to ORTEC's BestCombined method, which is trained on individual customers. This demonstrates the feasibility of using neural network models for supply chain forecasting at ORTEC, suggesting the potential for a more streamlined and concise forecasting workflow.

Predicting zero-inflated data using a single model proved challenging for all the neural network models. DeepAR seemed unable to fit a singular zero-inflated distribution across diverse series due to their inherent differences in weight on zeros. Although TFT and N-HiTS showed improvements in MAE, they did not achieve good accuracy. They both overpredicted the number of customers, where every customer was forecasted to have low order volumes. Postprocessing failed to rectify this issue. Moreover, total volume predictions were significantly underpredicted. A potential solution could involve a loss function that balances both binary loss and volume loss. Such an implementation was not explored in this study. Notably, the neural network models showed improvements in predicting new customers.

Finally, despite its higher training costs, the TFT offers valuable insights by providing feature importances within its forecasts, enhancing model interpretability.

When combining the forecasting and optimization processes, DeepAR provided the best results in terms of swaps and plan costs, despite having the highest MAE among the forecasting models. TFT achieved similar route efficiency, yet resulted in more swaps, possibly attributable to the underestimation of volume. The number of swaps primarily depends on the optimizer configuration, where route efficiency is improved with better forecasts. Depending on how swaps are defined, a reduction to less than 20% swaps is feasible with all forecasting methods. The definition of swaps ultimately depends on the client's preferences and how they manage orders coming from customers that were not initially forecasted, which remains unknown. Nonetheless, developing a strategy to accommodate such incoming orders is necessary.

In conclusion, this research demonstrated the feasibility of ORTEC's proposed framework and showed improvements in both forecasting and route planning. A key achievement in forecasting is the ability to predict a large number of time series effectively using a single model. However, further refinement of the loss function offers a possibility for future research. In terms of route planning, it is crucial to prioritize not only metrics that assess individual predictions but also those that evaluate the aggregate forecasted outcomes, ensuring more efficient and cost-effective strategies.

13

Recommendations

The wholesale distributor can begin by implementing the selected configurations for the route optimizer, specifically the light optimizers employed after a full optimization cycle using fixed input. These configurations can be tailored and selected on a daily basis to meet operational needs best. Additionally, several improvements could be made beyond this thesis that could be adopted to achieve better outcomes.

This research utilized order data spanning approximately one year. Expanding this dataset for training purposes should be considered, as a broader data range has the potential to increase model reliability and effectiveness. The limited timeframe prevented the model from learning yearly seasonality, and with most holidays appearing only once in the dataset, it was impossible for the model to accurately learn their impact. Testing the TFT model during the COVID period is also recommended, as Lim et al. [22] suggests it can effectively understand crisis situations.

Forecasts could further be improved by integrating client-specific business rules for holidays and specific customers. The current method flags all holidays similarly, as using a different feature for every holiday leads to unnecessarily high training times. It may therefore be beneficial to categorize holidays into distinct groups based on their varying effects on operations. Specifically, understanding which days the distributor continues to deliver, which holidays result in fewer customer orders, and which holidays have no impact at all. In this research, all customers are treated equally regarding how they are fixed into routes, when the optimizer considers swaps, and in counting the number of swaps. However, the wholesale distributor may wish to prioritize reducing swaps for regular or larger customers over others. To achieve this, additional business rules concerning customer prioritization should be provided.

In the early stages of this research, we discontinued our exploration of hierarchical forecasting due to its negative impact on forecasting quality at the lowest hierarchy level, which was initially considered critical. However, later findings in route planning revealed that accurate global forecasting holds greater significance. Thus, hierarchical forecasting should be reconsidered. Although Nixtla's implementation is unsuitable for zero-inflated data, an alternative approach could involve manually forecasting the daily number of customers and sampling customers based on probabilities predicted by the neural network.

Moreover, the wholesale distributor might explore a multi-stage approach to route planning. We now use a two-stage strategy, where decisions are made and then adjusted based on final order confirmations. It is also possible to revise routing decisions dynamically each time a customer places a new order. Although the effects of this multi-step approach are currently unknown, and the optimal planning strategies at each stage remain unclear, investigating this adaptive method could potentially lead to a more flexible and responsive solution.

In the short term, ORTEC should integrate the double optimization approach from Figure 7.14 efficiently into its current system. A refined strategy for handling customers can be developed by implementing a flexible system for flagging customers, rather than simply fixing a customer in a route. Introducing a variable minimum estimated gain per customer can provide nuanced control over route adjustments. For instance, customer A could be swapped if it improves routing, whereas customer B would require a minimum estimated gain before being considered for swapping. Additionally, there should be the ability to selectively fix certain customers during specific optimization phases, such as during the ruin and recreate process. Moreover, including a penalty for swaps, possibly depending on the customer, into the objective function could help the optimizer to balance between minimizing swaps and route efficiency.

For neural networks, it is recommended to conduct further investigations into alternative loss functions. Observations indicate that the TFT and N-HiTS models continue to underpredict total volume, potentially due to the chosen loss function. Additionally, it may still be possible to forecast zero-inflated data in a single model if the loss function accommodates it. One approach could be using a combined loss function that balances binary and volume losses. Alternatively, the model could be designed to output two data points: a binary value indicating delivery and a separate volume prediction. It is then important to ensure that both points have different loss functions.

Another unexplored area in this research is parametric optimization, which offers potential benefits, especially when forecasts tend to underpredict total volume. A whole different route planning strategy could involve focusing solely on forecasting predictable customers. For unpredictable customers, one could either allocate empty space within provisional routes or deploy additional vehicles to handle them. The former strategy might lead to more efficient routing. By using forecasts of the total ordered volume and subtracting the predicted volume for predictable customers, an estimate can be made of the amount of empty space (or θ_k in Equation (2.36)) required to accommodate unpredictable customers.

In the long term, ORTEC should explore the integration of stochastic optimization elements into its route optimizer, motivated by an increasing demand from the business sector. In the current framework, the provisional routes are generated using a full optimizer, assuming the forecasts are true. This research investigated some recourse strategies when the uncertainty is disclosed. ORTEC should explore ways to incorporate uncertainty into the first provisional route plan. For instance, by using confidence intervals generated by neural network models or the probability of a customer placing an order, ORTEC can develop robust routes that adeptly manage uncertainty. It is essential to determine which aspects of these uncertainties should be leveraged and how the optimizer can best interpret and respond to them. Furthermore, it could be interesting to explore reinforcement learning. In this framework, an agent can learn to adjust the inputs and forecasts using a reward structure tied to the optimizer's outcomes. Reinforcement learning can potentially lead to improved adaptability and more effective decision-making processes in dynamic environments.

References

[1] Aggarwal, C. C. *A Textbook Second Edition Neural Networks and Deep Learning*. 2nd ed. Springer, 2023. ISBN: 978-3-031-29642-0.

[2] Ba, J. L., Kiros, J. R., and Hinton, G. E. *Layer Normalization*. Tech. rep. July 2016. URL: http://arxiv.org/abs/1607.06450.

[3] Bergstra, J., Yamins, D., and Cox, D. D. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *TProc. of the 30th International Conference on Machine Learning (ICML 2013)* 28 (2013).

[4] Bergstra, J. et al. "Algorithms for Hyper-Parameter Optimization". In: *Advances in Neural Information Processing Systems* 24 (2011).

[5] Box, G. E. P. et al. *TIME SERIES ANALYSIS: Forecasting and Control*. John Wiley and Sons Inc., 2015. ISBN: 978-1-118-67502-1. DOI: 10.1111/jtsa.12194.

[6] Challu, C. et al. *N-HiTS: Neural Hierarchical Interpolation for Time Series Forecasting*. Tech. rep. Nov. 2022. URL: http://arxiv.org/abs/2201.12886.

[7] Cordeau, J. F. et al. "Chapter 6 Vehicle Routing". In: *Handbooks in Operations Research and Management Science*. Vol. 14. C. 2007, pp. 367–428. DOI: 10.1016/S0927-0507(06)14006-2.

[8] Croston, J. D. "Forecasting and Stock Control for Intermittent Demands". In: *Operational Research Quarterly (1970-1977)* 23.3 (Sept. 1972), p. 289. ISSN: 00303623. DOI: 10.2307/3007885.

[9] Dauphin, Y. N. et al. *Language Modeling with Gated Convolutional Networks*. Tech. rep. Dec. 2016. URL: http://arxiv.org/abs/1612.08083.

[10] Gardner Everette S, J. and Mckenzie, E. "Forecasting Trends in Time Series". In: *Management Science* 31.10 (Oct. 1985), pp. 1237–1246. ISSN: 00251909,15265501. URL: http://www.jstor.org/stable/2631713.

[11] Garey, M. R. and Johnson, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[12] Gendreau, M., Laporte, G., and Séguin, R. "EUROPEAN JOURNAL OF OPERATIONAL RESEARCH Stochastic vehicle routing". In: *European Journal of Operational Research* 88 (Jan. 1995), pp. 3–12.

[13] Golden, B., Raghavan, S., and Wasil, E. *THE VEHICLE ROUTING PROBLEM: LATEST ADVANCES AND NEW CHALLENGES*. Springer, 2008. ISBN: 978-0-387-77777-1. DOI: 10.1007/978-0-387-77778-8.

[14] Hastie, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning Data Mining, Inference, and Prediction*. Springer, Jan. 2017.

[15] Holt, C. C. "Forecasting seasonals and trends by exponentially weighted moving averages". In: *International Journal of Forecasting* 20.1 (Jan. 2004), pp. 5–10. ISSN: 01692070. DOI: 10.1016/j.ijforecast.2003.09.015.

[16] Hyndman, R. J. and Khandakar, Y. "Automatic Time Series Forecasting: The forecast Package for R". In: *Journal of Statistical Software* 27.3 (July 2008). DOI: 10.18637/jss.v027.i03. URL: http://www.jstatsoft.org/.

[17] Hyndman, R. J. et al. *Forecasting with Exponential Smoothing The State Space Approach*. Springer, 2008, pp. 11–28. DOI: 10.1007/978-3-540-71918-2.

[18] Hyndman, R. J. et al. "Optimal combination forecasts for hierarchical time series". In: *Computational Statistics and Data Analysis* 55.9 (Sept. 2011), pp. 2579–2589. ISSN: 01679473. DOI: 10.1016/j.csda.2011.03.006.

[19] Ioffe, S. and Szegedy, C. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: (Feb. 2015). URL: http://arxiv.org/abs/1502.03167.

[20] Jorgensent, B. “Exponential Dispersion Models”. In: *Journal of the Royal Statistical Society* 49.2 (1987), pp. 127–162. URL: https://www.jstor.org/stable/2345415.

[21] KNMI. *Daggegevens van het weer in Nederland*. URL: https://www.knmi.nl/nederland-nu/klimatologie/daggegevens.

[22] Lim, B. et al. *Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting*. Tech. rep. Sept. 2020. URL: http://arxiv.org/abs/1912.09363.

[23] Lopez, L. *When Is cross-docking Used in 3PL companies is used*. URL: https://go-freight.io/when-is-cross-docking-used-in-3pl-companies-is-used/.

[24] Makridakis, S., Spiliotis, E., and Assimakopoulos, V. “M5 accuracy competition: Results, findings, and conclusions”. In: *International Journal of Forecasting* 38.4 (Oct. 2022), pp. 1346–1364. ISSN: 01692070. DOI: 10.1016/j.ijforecast.2021.11.013.

[25] Nixtla. *Hyperparameter Optimization*. URL: https://nixtlaverse.nixtla.io/neuralforecast/docs/capabilities/hyperparameter_tuning.html.

[26] Nixtla. *Models: DeepAR*. URL: https://nixtlaverse.nixtla.io/neuralforecast/models.deepar.html.

[27] Nixtla. *Models: NHITS*. URL: https://nixtlaverse.nixtla.io/neuralforecast/models.nhits.html.

[28] Nixtla. *Models: TFT*. URL: https://nixtlaverse.nixtla.io/neuralforecast/models.tft.html.

[29] Nixtla. *NeuralForecast: PyTorch Losses*. URL: https://nixtlaverse.nixtla.io/neuralforecast/losses.pytorch.html.

[30] Nixtla. *NeurelForecast: End to End Walkthrough*. URL: https://nixtlaverse.nixtla.io/neuralforecast/docs/tutorials/getting_started_complete.html.

[31] Olah, C. *Understanding LSTM Networks*. Aug. 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[32] Olivares, K. G. et al. “HierarchicalForecast: A Reference Framework for Hierarchical Forecasting in Python”. July 2022. URL: http://arxiv.org/abs/2207.03517.

[33] Olivares, K. G. et al. “Hierarchically Coherent Multivariate Mixture Networks”. In: (May 2023). URL: http://arxiv.org/abs/2305.07089.

[34] Olivares, K. G. et al. *NeuralForecast: User friendly state-of-the-art neural forecasting models.* 2022. URL: https://github.com/Nixtla/neuralforecast.

[35] Oreshkin, B. N. et al. “N-BEATS: Neural basis expansion analysis for interpretable time series forecasting”. In: (May 2019). URL: https://arxiv.org/abs/1905.10437.

[36] ORTEC. *About us*. URL: https://ortec.com/en/about-us.

[37] ORTEC. *B2B Delivery*. URL: https://ortec.com/en-us/solutions/b2b-delivery.

[38] ORTEC Algorithm Knowledge Team. *Optimization Configuration Training (Internal document)*. 2024.

[39] Ostermeier, M. et al. “Multi-compartment vehicle routing problems: State-of-the-art, modeling framework and future directions”. In: *European Journal of Operational Research* 292.3 (Aug. 2021), pp. 799–817. ISSN: 03772217. DOI: 10.1016/j.ejor.2020.11.009.

[40] Pedregosa, F. et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[41] Postek, K. et al. *Hands-on Mathematical Optimization with Python*. Cambridge University Press, 2025. ISBN: 9781009493505.

[42] Powell, W. B. *REINFORCEMENT LEARNING AND STOCHASTIC OPTIMIZATION A unified framework for sequential decisions*. Wiley, Sept. 2019. ISBN: 3175723993.

[43] RIVM. *Tijdlijn van coronamaatregelen 2022*. URL: https://www.rivm.nl/gedragsonderzoek/tijdlijn-maatregelen-covid-2022.

[44] Salinas, D., Flunkert, V., and Gasthaus, J. *DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks*. Tech. rep. Feb. 2019. URL: http://arxiv.org/abs/1704.04110.

[45] Seabold, S. and Perktold, J. "statsmodels: Econometric and statistical modeling with python". In: *9th Python in Science Conference*. 2010.

[46] Shumway, R. H. and Stoffer, D. S. *Time Series Analysis and Its Applications*. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-52451-1. DOI: 10.1007/978-3-319-52452-8.

[47] Sutskever, I., Vinyals, O., and Le, Q. V. *Sequence to Sequence Learning with Neural Networks*. Tech. rep. Sept. 2014. URL: http://arxiv.org/abs/1409.3215.

[48] Taylor, S. J. and Letham, B. "Forecasting at scale". In: *PeerJ Preprints* 5 (Sept. 2017). DOI: 10.7287/peerj.preprints.3190v2. URL: https://peerj.com/preprints/3190v2.

[49] Toth, P. and Vigo, D. *THE VEHICLE ROUTING PROBLEM*. SIAM, 2002.

[50] Vaswani, A. et al. *Attention Is All You Need*. Tech. rep. Google, June 2017. URL: https://arxiv.org/abs/1706.03762.

[51] Wickramasuriya, S. L., Athanasopoulos, G., and Hyndman, R. J. "Optimal forecast reconciliation for hierarchical and grouped time series through trace minimization". In: *Journal of the American Statistical Association* 114.526 (Dec. 2015), pp. 804–819. DOI: 10.1080/01621459.2018.1448825. URL: https://robjhyndman.com/papers/MinT.pdf.

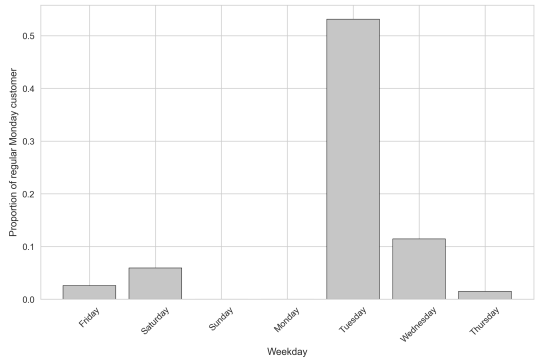
[52] Winters, P. R. "Forecasting Sales by Exponentially Weighted Moving Averages". In: *Management Science* 6.3 (Apr. 1960), pp. 324–342. ISSN: 00251909,15265501. DOI: 10.1287/mnsc.6.3.324. URL: https://about.jstor.org/terms.

A

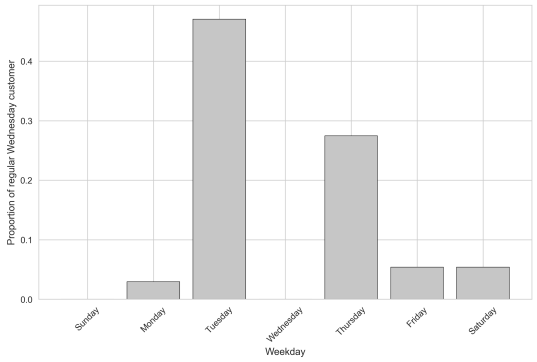
Additional Results

A.1. Substitute Order Day

In Section 5.2, we examined customer behavior surrounding holidays, focusing on their choice of substitute days for placing orders when they are unable to order on their regular days. We defined regular customers for a weekday as those with an order consistency of over 90%. We then counted how many of these regular customers placed orders on alternative days, provided they were not already regular customers for those specific weekdays. While this analysis does not capture the full spectrum of customer behavior during holidays, it offers insights into preferred alternative order days. Here, we present the effect on two specific holidays, instead of aggregated data for all holidays. Figure A.1 illustrates the preferred substitute days for Easter Monday and King's Day. For Easter Monday, the most popular substitute is the day after the holiday, as the wholesale distributor does not deliver on Sundays. For King's Day, which fell on a Wednesday, both the preceding and following days were popular substitutes, with the day before being slightly more favored.



(a) Holiday: Easter Monday (2022-04-18). The most popular substitute day is the day after Easter Monday. Sum of bars is 74.5%.



(b) Holiday: King's Day (Wednesday 2022-04-27). The most popular substitute day is the day after the King's Day. Sum of bars is 88.2%

Figure A.1: Number of orders 3 days before and after a holiday by customers that usually order on the weekday of the holiday (threshold of $> 90\%$) and not on the other weekdays (threshold of $< 10\%$). Presented as a proportion to the number of regular customers for that weekday.

A.2. Neural Network results on Two-Stage Forecast

In this section, we present the results of training neural networks using delivery data and volume data separately. Rather than using the complete time series of ordered volume, TFT and N-HiTS are trained explicitly on delivery data as part of the postprocessing step in Section 10.2. The delivery time series is characterized as a binary series: 0 when the ordered volume is zero and 1 when the volume is greater

than zero. The accuracy of these models is given in Table A.1, which is slightly higher than the result from using these forecasts as a postprocessing step.

Method	Accuracy
CopyLastWeek (baseline)	0.914201
BestCombined (ORTEC)	0.924149
TFT	0.935979
N-HiTS	0.921225
DeepAR	0.933898

Table A.1: Accuracy of neural network models on delivery data (binary classification) compared to baseline and ORTEC's BestCombined method, evaluated on the test period.

Additionally, in Section 10.4, we presented the results of training DeepAR, TFT, and N-HiTS using a two-stage forecast. The volume series is generated by interpolating the zeros linearly and filling the zeros at the beginning and end using forward and backward fill. The final forecasting results can be found in the results section. Table A.2 presents the training times of all models.

	Delivery		Volume	
Method	Validation time (s)	Test time (s)	Validation time (s)	Test time (s)
DeepAR	17434	350	18089	414
TFT	18509	298	24934	207
N-HiTS	2026	90	3050	64

Table A.2: Running times for predicting the validation window (including hyperparameter tuning) and the test window on delivery and volume series.

Figure A.2 shows the percentage of swaps per forecasting method, excluding the non-forecasted orders. It is important to note that this may not be an entirely fair comparison, since different methods might have varying amounts of non-forecasted orders. Nevertheless, the figure demonstrates that, if there is an efficient way to handle incoming orders that were not forecasted, it is feasible to achieve a swap percentage of approximately 10% among customers whose orders were correctly predicted. We can also observe that the number of swaps is primarily determined by the configuration applied, while the plan cost varies more depending on the forecasting method used, with improved forecasting leading to more efficient routes.

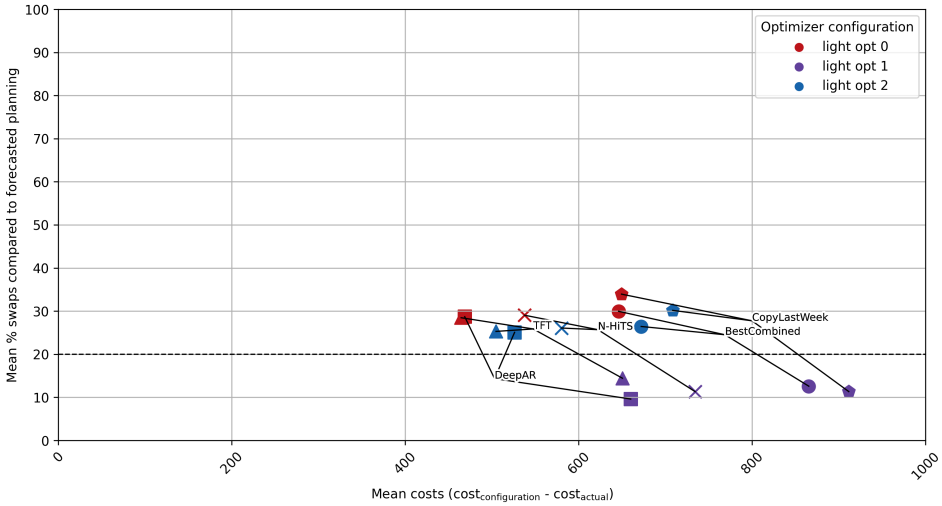
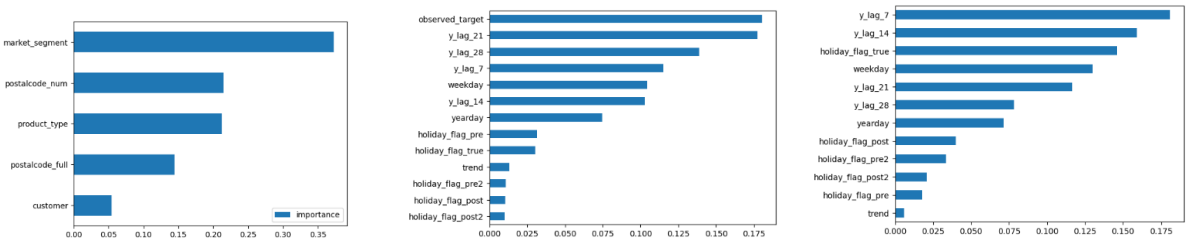


Figure A.2: Mean percentage of swaps and difference in plan cost using forecasts from neural network models, ORTEC's BestCombined method, and the CopyLastWeek baseline.

Feature Importances TFT

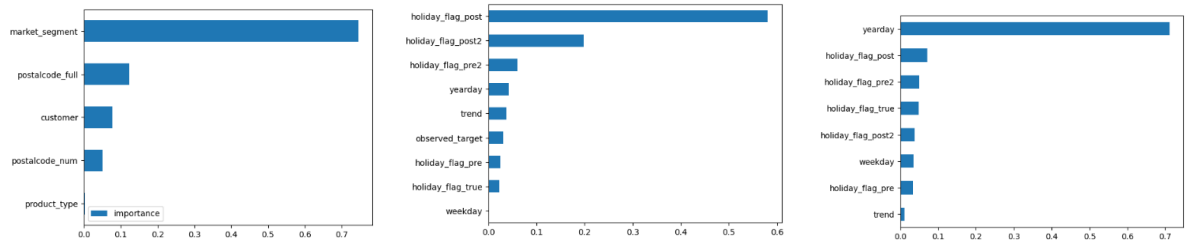
Figure A.3 shows the feature importances for static, past, and future covariates of the TFT model trained on the delivery data (binary classification).



(a) Static covariates. **(b)** Past covariates. **(c)** Future covariates.

Figure A.3: Feature importances of the TFT trained on delivery data series (binary classification).

Figure A.4 shows the feature importances for static, past, and future covariates of the TFT model trained on the volume data linearly interpolated to remove zeros.



(a) Static covariates. **(b)** Past covariates. **(c)** Future covariates.

Figure A.4: Feature importances of TFT trained on volume data series (interpolated using linear interpolation).

B

Optimizer Settings

As discussed in Section 7.2.1, the plan cost encompasses a weighted average of various factors. This includes expenses associated with vehicle usage, the distance to be traveled, working hours, and overtime. The specific weights assigned to each of these components are outlined in Table B.1.

Type	Cost
per vehicle	200
per kilometer	2.14
per hour	60
per hour overtime	6.14

Table B.1: Components of plan costs with their respective weights.

In addition to minimizing plan costs, the route optimizer must consider specific restrictions such as maximum working time per day, allowable overtime, minimum break duration, and maximum working time before a break is required. These values are detailed in Table B.2.

Type	Value
maximumDuration	8 hr
maximumOverTime	1.5 hr
maximumNumberOfTrips	5
breakDuration	0.5 hr
maximumWorkingTimeBeforeBreak	5.25 hr
maximumDrivingTimeBeforeBreak	4.5 hr

Table B.2: Constraints for feasible routes.

Constraints within the route optimizer are treated as soft constraints, meaning violations incur specific penalties. Each constraint is composed of a constant term, a linear term, and a quadratic term. The constant term is applied for every violation, while the linear and quadratic terms depend on the magnitude of the violation. The coefficients for the constant, linear, and quadratic terms associated with violations are provided in Table B.3.

Type	Constant (c)	Linear (b)	Quadratic (a)
Capacity	200	0	100
MaxWorkTime	50	0	0.001
RequiredCapabilities	100	0	0
RouteFinishTime	50	0	0.001
TaskTimeWindow	50	0	0.001

Table B.3: Coefficients of the penalty per constraint. A violation of x incurs penalty $= ax^2 + bx + c$.

C

Hyperparameters Forecasting

C.1. PoC

Table C.1 shows the tested hyperparameter grid for the traditional forecasting methods used on the volume series.

Parameter	Values
Linear Regression	
fit_intercept	True, False
features	holiday_flag, holiday_flag + weekday_cols, holiday_flag + weekday_cols + temp_mean
Holt-Winters	
trend	additive, multiplicative
seasonal	additive, multiplicative
damped_trend	True
use_boxcox	True, False
SARIMAX	
order_pdq	[0,0,0], [1,1,1], [1,0,0], [0,0,1], [1,0,1], [0,1,1]
seasonal_pdq	[0,0,0], [1,1,1], [1,0,0], [0,0,1], [1,0,1], [0,1,1]
features	holiday_flag, holiday_flag + temp_mean
Prophet	
growth	linear, flat
yearly_seasonality	False
weekly_seasonality	True
daily_seasonality	False
seasonality_mode	additive, multiplicative
seasonality_prior_scale	10
holidays_prior_scale	10
changepoint_prior_scale	0.05
features	holiday_flag, holiday_flag + temp_mean

Table C.1: Hyperparameter grid per regression model used in the PoC by ORTEC.

Table C.2 shows the tested hyperparameter grid for the classification methods used on the delivery series.

Parameter	Values
Logistic Regression	
fit_intercept	True, False
class_weight	None, balanced
features	holiday_flag, weekday_cols holiday_flag + weekday_cols + temp_mean, holiday_flag + weekday_cols + delivery_lags
Random Forest	
max_depth	None, 5
min_samples_split	2, 10
features	holiday_flag, weekday_cols holiday_flag + weekday_cols + temp_mean, holiday_flag + weekday_cols + delivery_lags

Table C.2: Hyperparameter grid per classification model used in the PoC by ORTEC.

C.2. Neural Networks

Table C.2 shows the tested hyperparameter grid for the classification methods used on the delivery series. All models used a batch size, the number of time series trained on simulataneously, of 32 and a window batch size, the number of forecasting windows trained simultaneously, of 1024 (both standard configurations). Increasing these numbers caused the used GPU to crash. A more powerful GPU could probably handle larger batch sizes, decreasing the training time.

Parameter	Values
DeepAR	
input_size	14, 28
learning_rate	tune.loguniform(1e-4, 1e-1)
max_steps	1000, 1500, 2000
scalar_type	None, standard, minmax, robust
lstm_n_layers	2, 3
lstm_hidden_size	32, 64, 128
lstm_dropout	0.1
decoder_hidden_layers	1, 2, 3
decoder_hidden_size	32, 64, 128
random_seed	tune.randint(1, 10)
stat_exog_list	customer_id + product_type + market_segment + postalcode_num + postalcode_full
futr_exog_list	weekday + yearday + trend + holiday_flag_true + holiday_flag_pre + holiday_flag_pre2 + holiday_flag_post + holiday_flag_post2
TFT	
input_size	14, 28
learning_rate	tune.loguniform(1e-4, 1e-1)
max_steps	1000, 1500, 2000
scalar_type	None, standard, minmax, robust
n_head	4, 8
hidden_size	32, 64, 128
lstm_dropout	tune.uniform(0,0.5)
random_seed	tune.randint(1, 10)
stat_exog_list	customer_id + product_type + market_segment + postalcode_num + postalcode_full
futr_exog_list	weekday + yearday + trend + holiday_flag_true + holiday_flag_pre + holiday_flag_pre2 + holiday_flag_post + holiday_flag_post2
past_exog_list	Levering + temp_mean + temp_max + temp_min
N-HiTS	
stack_types	3*['identity']
n_blocks	[1, 1, 1]
mlp_units	[[512, 512], [512, 512],[512, 512]]
input_size	14, 28
learning_rate	tune.loguniform(1e-4, 1e-1)
max_steps	1000, 1500, 2000
scalar_type	None, standard, minmax, robust
n_pool_kernel_size	[7, 3, 1], [7, 2, 1], [1,1,1]
n_freq_downsample	[28, 7, 1], [14, 7, 1], [7, 3, 1], [1,1,1]
random_seed	tune.randint(1, 10)
stat_exog_list	customer_id + product_type + market_segment + postalcode_num + postalcode_full
futr_exog_list	weekday + yearday + trend + holiday_flag_true + holiday_flag_pre + holiday_flag_pre2 + holiday_flag_post + holiday_flag_post2

Table C.3: Hyperparameter grid per neural network model used in this research.

D

Implementation of Neural Networks

The models for this research were implemented using the `NeuralForecast` class from Nixtla. This section outlines the general process for training and predicting with neural networks, as well as some best practices. Note that this is a general description of the implementation; the actual code used is more detailed.

The guidelines presented here are drawn from Nixtla's tutorial [30], alongside documentation for specific classes [26, 27, 29, 28]. Additionally, this section presents some techniques relevant to this case study that are not easily found in the documentation. These include employing cross-validation with an overlapping window, focusing the loss function exclusively on the second week of predictions, and after hyperparameter tuning with an AutoModel, using solely the best configuration for training on new data.

D.1. Training a Model and making Predictions

To train a model and make predictions, we need a dataframe, `Y_df`, containing time series data. This dataframe should be formatted as a pandas dataframe and must include at least three columns: `unique_id`, `ds`, and `y`. The `unique_id` represents the unique identifier for each time series, `ds` denotes the date and must be formatted as pandas datetime, and `y` indicates the value of interest, such as volume or a binary 0/1 value. An example is provided in Figure D.1.

unique_id	ds	y	weekday	yearday
custumer_1_product_A	2022-03-21 00:00:00	300,55	1	80
custumer_1_product_A	2022-03-22 00:00:00	0	2	81
custumer_1_product_A	2022-03-23 00:00:00	0	3	82
⋮	⋮	⋮	⋮	⋮
custumer_9999_product_C	2023-06-22 00:00:00	0	4	173
custumer_9999_product_C	2023-06-23 00:00:00	180,30	5	174
custumer_9999_product_C	2023-06-24 00:00:00	23,72	6	175

Figure D.1: Example of dataframe `Y_df` with time series data of all customers and 3 product types. `y` represents the ordered volume for the date given in `ds`. Here, weekday and yearday are future covariates.

Additionally, we require a dataframe containing static features. This `static_df` should include `unique_id` and all static features applicable to the task, such as customer ID, market segment, and product type. An example is provided in Figure D.2.

unique_id	customer	product_type	market_segment
custumer_1_product_A	1	1	1
custumer_1_product_B	1	2	1
⋮	⋮	⋮	⋮
custumer_9999_product_C	9999	3	5

Figure D.2: Example of dataframe `static_df` with static features representing the customer, product type, and market segment corresponding to the time series.

To proceed, we can import the desired models and loss functions. The `DistributionLoss` is used for DeepAR and delivery forecasts utilizing Bernoulli loss, whereas `MQLoss` is used for the volume forecasts with TFT and N-HiTS models.

We can also define a train-test split and set a forecasting horizon of two weeks:

```
from neuralforecast import NeuralForecast
from neuralforecast.models import DeepAR, NHITS, TFT
from neuralforecast.losses.pytorch import DistributionLoss, MQLoss

horizon = 14
split_date = Y_df['ds'].max() - pd.DateOffset(days=horizon)
Y_train_df = Y_df[Y_df.ds <= split_date]
Y_test_df = Y_df[Y_df.ds > split_date]
```

Next, we specify the model configuration for training. In this example, the N-HiTS model is initialized with some hyperparameters and a daily data frequency:

```
models = [NHITS(h = horizon,
                input_size = 2 * horizon,
                max_steps = 1000,
                loss = MQLoss(level=[80, 90]),
                stat_exog_list = ['customer', 'product_type', 'market_segment'],
                futr_exog_list = ['weekday', 'yearday'],
                hist_exog_list = [])]
nn_model = NeuralForecast(models=models, freq='D')
```

After setting up the model, we use the following functions to fit the model to the training data and predict the forecast horizon:

```
nn_model.fit(df=Y_train_df, static_df=static_info, val_size=horizon)
Y_hat_df = nn_model.predict(futr_df=Y_test_df)
```

Note that for training the model, we use the training dataframe and the static dataframe. The test dataframe can include future exogenous variables, but will not use it for predictions.

In this case study, we make predictions for two weeks, but we are specifically interested in the second week. To ensure the loss function only considers the second week, we introduce a horizon weight and adjust the loss function accordingly during model initialization:

```
horizon_weight = torch.tensor([0.0] * 7 + [1.0] * 7)

loss = MQLoss(level=[80, 90], horizon_weight=horizon_weight))
```

D.2. Hyperparameter Tuning

Hyperparameter tuning in `NeuralForecast` is streamlined through the use of Auto models. Each model within the library has an Auto version, such as AutoNHITS and AutoTFT, which enables automatic hyperparameter selection across either a default or a user-defined search space.

To initiate hyperparameter tuning, we import the necessary Auto models, along with a backend (Ray Tune) and search algorithm:

```
from neuralforecast.auto import AutoDeepAR, AutoTFT, AutoNHITS
from ray import tune
from ray.tune.search.hyperopt import HyperOptSearch
```

Next, we define the configuration settings. A standard configuration, tested across numerous datasets, is available. In this research, we chose to vary certain parameters using `tune` or set specific values to remain constant. Below is an example configuration for N-HiTS:

```
config_nhits = {
       "input_size":  tune.choice([7,14,21,28]),
       "learning_rate": tune.loguniform(1e-5, 1e-1),
       "max_steps": 1500,
       "scaler_type": tune.choice([None, "standard", "robust"]),
       "n_pool_kernel_size": tune.choice([[7, 3, 1], [7, 2, 1], [1,1,1]]),
       "n_freq_downsample": tune.choice([[28, 7, 1], [14, 7, 1], [7, 3, 1], [1,1,1]]),
       "early_stop_patience_steps": 2,
       "stat_exog_list": stat_exog,
       "futr_exog_list": futr_exog,
       "hist_exog_list": hist_exog,
    }
```

The setup of a model now includes the hyperparameter space and a search algorithm. The number of samples represents the number of hyperparameter configurations to be tested.

```
nn_auto = NeuralForecast(
        models=[AutoNHITS(h = horizon,
              loss = MQLoss(level=[80, 90]),
              config = config_nhits,
              search_alg = HyperOptSearch(),
              backend = 'ray',
              num_samples = 20)],
        freq='D')
```

The tuning of the Auto model involves using the same functions for fitting and predicting as standard models. Fitting tests a number of configurations specified in the model. Then, the prediction is based on the configuration that performed best on the training set, according to the loss function.

```
nn_auto.fit(df=Y_train_df, static_df=static_info, val_size=forecast_length)
nn_auto.predict(futr_df=Y_test_df)
```

It is important to note that the Auto models are saved as their base counterparts, using the best hyperparameters identified during tuning. So, an AutoNHITS will be stored as an NHITS model:

```
nn_auto.save(path='neural_nets/nhits_model/',
        model_index=[0],
        overwrite=True,
        save_dataset=True)
```

To access the results or the best configuration, we can use the following:

```
results = nn_auto.models[0].results.get_dataframe()
best_result = results.loc[results['loss'].idxmin()]
```

D.3. Cross-validation

For cross-validation, we want to repeatedly train and predict on larger datasets through a series of validation folds. This process can be implemented manually by sequentially using `.fit()` on `Y_train`, then `.predict()` on `Y_test`, updating `Y_train` with `Y_test`, and repeating the sequence for the required number of folds.

In this research, we used 6 validation folds and 6 test folds. The folds are defined in the following way:

```
window_step_size = 7
forecast_length = 14

num_val_folds = 6
num_test_folds = 6
num_folds = num_val_folds + num_test_folds
unique_dates = pd.to_datetime(Y_df['ds'].unique())

```

```
folds =  [(d, d+pd.DateOffset(days=window_step_size+1), d + pd.DateOffset(days=
    forecast_length)) for d in unique_dates[unique_dates.dayofweek == 5][-(num_folds+2):-2]]

list_validation_limits = folds[:num_val_folds]
list_test_limits = folds[num_val_folds:]

split_date = list_validation_limits[-1][-1]
Y_train_df = Y_df[Y_df['ds'] <= split_date].reset_index(drop=True)

Y_test_df = []
for i in range(num_folds):
    Y_test_fold = Y_df[(Y_df['ds'] > list_validation_limits[i][0]) & (Y_df['ds'] <=
    list_validation_limits[i][1])].reset_index(drop=True)
    Y_test_df.append(Y_test_fold)
```

Instead of manually fitting and predicting in a loop, Nixtla offers an implementation that simplifies this sequence. For this, we can use the `.cross_validation()` function. Remember that, since we want to predict the next two weeks for every week, we have overlapping forecasting windows. The `.cross_validation()` method automatically uses a window step size equal to the forecast length, starting the new prediction right after the other one. We can achieve the overlapping window by setting the step size to 7 and the forecast horizon to 14 days. Note that we no longer need to define the test set over multiple folds.

```
cv_valildation_auto = nn_auto.cross_validation(Y_train_df, static_df=static_info, val_size=
    forecast_length, n_windows=num_val_folds, step_size=window_step_size, refit=True)
nn_auto.save(path='neural_nets/nhits_val/',
        model_index=[0],
        overwrite=True,
        save_dataset=True)
```

A challenge with overlapping windows is having multiple predictions for a single date. The cross-validation function addresses this by adding a ”cutoff” column, indicating the used training set. We implemented the following function to use this information and extract the desired week:

```
def remove_overlap(df, list_limits):
    """
    df should be from NeuralForecast and have columns: unique_id, ds, cutoff
    list_limits should have tuple (cutoff, d1, d2)
        cutoff: cutoff date for trainingset
        d1: first date of forecast period of interest
        d2: last date of forecast period of interest
    """
    for cutoff, d1, d2 in list_limits:
        df = df[~((df['cutoff'] == cutoff) & (df['ds'] < d1))]
        df = df[~((df['cutoff'] == cutoff) & (df['ds'] > d2))]
    return df.drop(columns=['cutoff'])

Y_hat_val = remove_overlap(cv_valildation_auto, list_validation_limits)
```

Using the cross-validation function on an Auto model enables hyperparameter tuning through cross-validation, which identifies the best configuration across all folds. Saving the model at the validation period, and subsequently loading it, ensures that only the optimal configuration from the validation window is applied to the test window, preventing further hyperparameter tuning of the Auto model.

```
nn_model = NeuralForecast.load(path='neural_nets/nhits_val/')
cv_test = nn_model.cross_validation(Y_df, static_df=static_info, val_size=forecast_length,
    n_windows=num_test_folds, step_size=window_step_size, refit=True)
Y_hat_test = remove_overlap(cv_test, list_test_limits)
```

D.4. TFT Feature importance

As outlined in Section 4.4.2, the TFT provides insights into feature importance, enhancing the interpretability of the model. The feature importance can be obtained using:

```
feature_importances = nn_model.models[0].feature_importances()
feature_importances.keys()
```

This information can be visualized using the following approach. We obtain plots for static, past, and future covariates using:

```
feature_importances['Static covariates'].sort_values(by='importance').plot(kind='barh')
feature_importances['Past variable importance over time'].mean().sort_values().plot(kind='barh')
feature_importances['Future variable importance over time'].mean().sort_values().plot(kind='barh')
```

Feature importance over time for the lookback window, like in Figure 10.7, can be obtained through the following code:

```
        df= feature_importances['Past variable importance over time']
mean_attention = nn_model.models[0].attention_weights()[nn_model.models[0].input_size:,:].mean(axis=0)[:nn_model.models[0].input_size]
df = df.multiply(mean_attention, axis=0)

fig, ax = plt.subplots(figsize=(20, 10))
bottom = np.zeros(len(df.index))

for col in df.columns:
    p = ax.bar(np.arange(-len(df),0), df[col].values, 0.6, label=col, bottom=bottom)
    bottom += df[col]
ax.set_title('Past variable importance over time ponderated by attention')
ax.set_ylabel("Importance")
ax.set_xlabel("Time")
ax.legend()
ax.grid(True)
plt.plot(np.arange(-len(df),0), mean_attention, color='black', marker='o', linestyle='-', linewidth=2, label='mean_attention')
plt.legend()
plt.show()
```

E

Client Specific Information

The contents of this appendix have been removed to protect company information and maintain confidentiality.