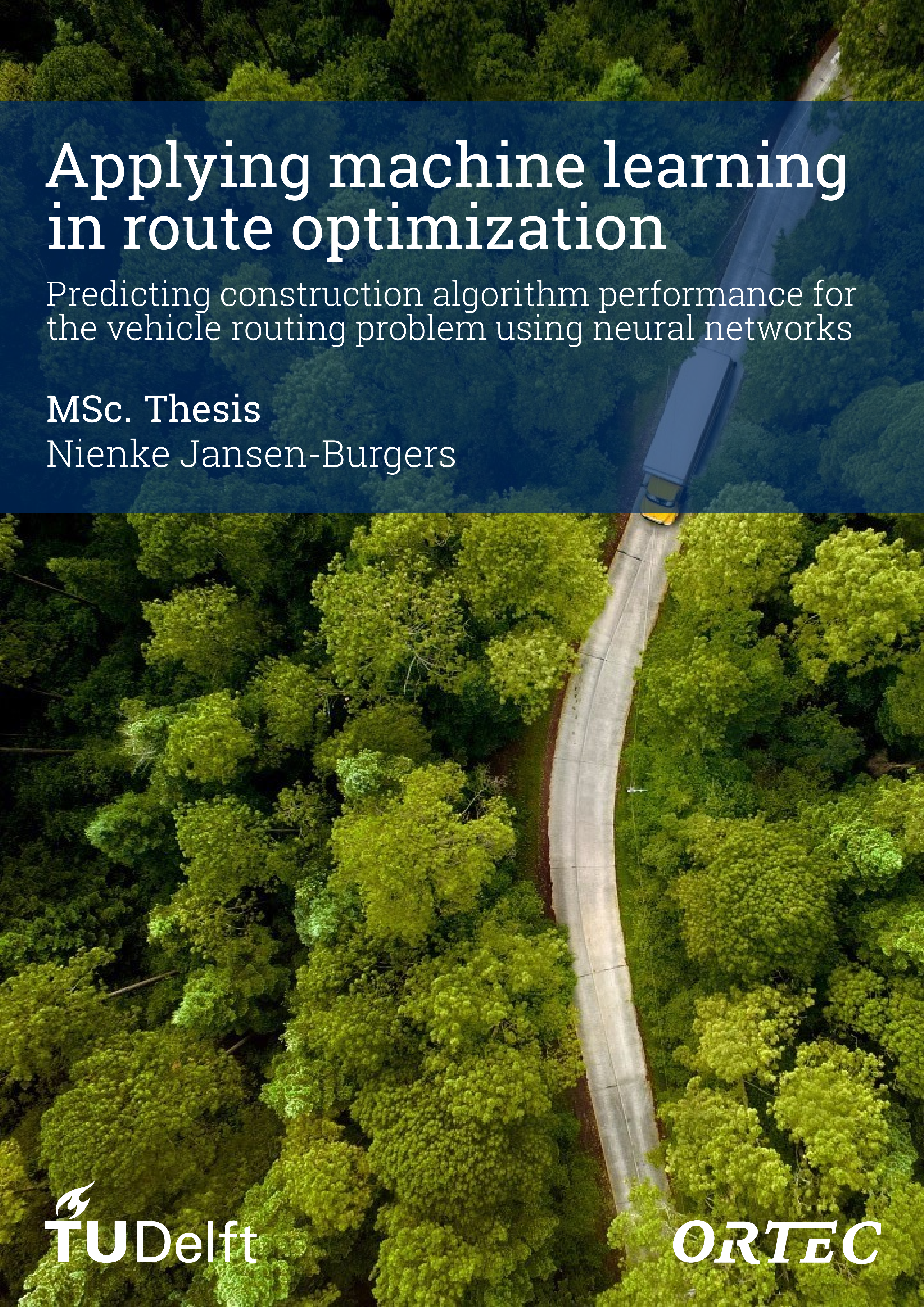


Applying machine learning in route optimization

Predicting construction algorithm performance for the vehicle routing problem using neural networks

MSc. Thesis

Nienke Jansen-Burgers

Applying machine learning in route optimization

Predicting construction algorithm performance for the vehicle routing problem using neural networks

by

Nienke Jansen-Burgers

A thesis submitted to the Delft Institute of Applied Mathematics for the completion of the degree
Master of Science in Applied Mathematics

at the Delft University of Technology,

to be defended publicly on June 12th 2023, 14:00.

Student number:	5426596	
Project duration:	October, 2022 – June, 2023	
Thesis committee:	Dr. ir. T. (Theresia) van Essen,	TU Delft, supervisor
	Dr. A. (Alexander) Heinlein,	TU Delft
	Q. (Quinten) Cederhout MSc,	ORTEC, supervisor
Institution:	Delft University of Technology	
	Faculty of Electrical Engineering,	
	Mathematics & Computer Science	
Master programme:	Applied Mathematics	
Specialisation:	Optimization	

Abstract

The real-life Vehicle Routing Problem (VRP) is the problem in which a set of vehicles needs to perform a set of tasks such that we have a shortest total driving distance. Such problems can be solved using construction algorithms. Finding the best-performing construction algorithm is time-consuming because these algorithms consist of many different elements, which differ between algorithms. In this thesis, a Neural Network (NN) model is built that predicts the best-performing algorithm from a pre-determined set of algorithms for a specific input case. These predictions are based on problem features extracted from real-life data. For our first NN model, we evaluate the best settings with a grid search, which leads to a model with a mean-squared error of 0.147 and an accuracy of 58.6%. We try to improve this original model by data balancing and varying the input features. First, we balance the data using downsampling and oversampling performed by an Integer Linear Program (ILP), which does not lead to a better-performing NN. Secondly, we add more input features to the model, which leads to a slight improvement because the model has more information about the problem at hand. After, we perform an elaborate feature analysis using permutation feature importance, SHAP, and Greenwell numbers. Based on this analysis, we reduce the number of input features to only 12. This reduction leads to the best-performing model with a mean-squared error loss of 0.125 and an accuracy of 61.7%. To investigate whether our prediction model indeed improves the routes using the predicted algorithm, we look at the distribution of predictions. For each company, we replace the algorithm in use with the algorithm most often predicted by our model. This replacement indeed improves the results for most of the considered companies.

Contents

List of abbreviations

BCVRP	Balanced Cargo Vehicle Routing Problem
CNN	Convolutional Neural Network
FI	Feature Importance
FFNN	Feed Forward Neural Networks
HVRP	Heterogeneous Vehicle Routing Problem
ILP	Integer Linear Program
MPE	Mean Percentage Error
MDVRP	Multiple Depot Vehicle Routing Problem
MOVRP	Multiple Objective Vehicle Routing Problem
MLP	Multi Layer Perceptron
NN	Neural Network
PVRP	Period Vehicle Routing Problem
RNN	Recurrent Neural Network
SHAP	SHapley Additive exPlanations
TSP	Traveling Salesman Problem
VRP	Vehicle Routing Problem
VRPTW	Vehicle Routing Problem with Time Windows

Preface

In my six years of studying Mathematics, I have always favored optimization subjects. During my studies at the TU Delft, I also learned more about machine learning and the mathematical optimization behind the models. This spiked my interest at least as much as classical optimization. To no surprise, this thesis touches on both the classical optimization problem of vehicle routing and the machine learning method neural networks.

This research could not have been completed without the help of Theresia, my supervisor at TU Delft. Her feedback on my writing and critical questions helped me to perfect my work. I also want to thank Alexander for sharing his insights into neural networks and for being part of my thesis committee.

This research has been conducted in collaboration with ORTEC, which provided not only data but also valuable knowledge and insights. I want to thank everyone at ORTEC for their support. From the first day, I felt part of the team, which made my time at ORTEC all the more enjoyable. In particular, I want to thank my supervisors Quinten, Anna, and Tom for their continuous support and help.

Lastly, I want to thank my family for supporting me in every decision that led to this. Especially, I want to thank my husband Sam for always being there for me, from the beginning to the end. Without you, I would not have been able to accomplish this.

Nienke Jansen-Burgers
Delft, May 2023

Chapter 1

Introduction

Everyone desires their home to be heated, whether using a boiler, a heat pump, or some other kind of appliance. All such appliances need to be installed, maintained, and repaired when broken. This work is performed by install centers that offer home service. When receiving several calls, the repairmen visit the various homes and perform their tasks. When assigning tasks to a specific repairman, the repairman needs to have the right skills to perform the task. Furthermore, an install center wants to find the most efficient routes to drive for all repairmen to perform their tasks in time. Finding the most efficient route is also known as the Vehicle Routing Problem (VRP). The challenge of assigning the tasks to the repairmen based on skills, is an additional constraint to the general VRP.

An elaborate description of the VRP can be found in Section 2.1. In short, the general VRP consists of a set of vehicles whose drivers need to perform a set of tasks at several addresses. The vehicles always start from a shared location: the depot. This is also shown in Figure 1.1. The question that arises is: what is the optimal route in terms of travel distance for the vehicles to drive, in order for all tasks to be performed?

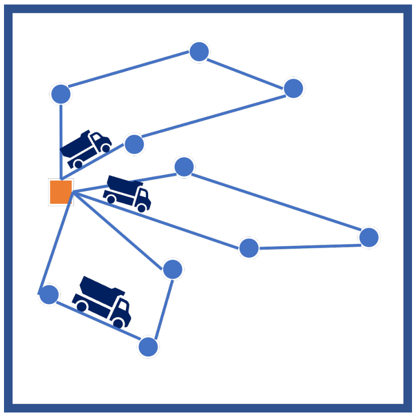


Figure 1.1: Solution to a simple vehicle routing problem

The VRP is an NP-hard problem (Lenstra and Kan [1981]). Therefore, there is no known algorithm that can find an optimal solution in polynomial time. To find a good enough solution in a limited time, heuristics and rules of thumb are used. A few known heuristics include Tabu Search and Simulated Annealing (Ho and Haugland [2004], Laporte et al. [2014]). New heuristics and rules of thumb are still devised, tested, and used, to improve upon already existing solution methods. Especially since home delivery has become more popular in recent decades, the interest in finding a good feasible solution to VRPs has increased. By improving a solution, a company can save costs and take on more customers. However, these real-life problems also bring extra constraints. This leads to many variations of the VRP. One of those variations is the repairman problem, as described above, in which extra constraints with regard to the skills of the repairmen are included. More on additional constraints is discussed in Section 2.2.

1.1 Problem description

At the software company ORTEC, various routing optimizers are implemented for customers to solve their VRPs for real-life cases. This is done by designing optimization algorithms tailored to the type of problem, the customer's data, and their preferences. The customers with similar problems as the repairman problem, are grouped. For them, the software of ORTEC for Field Service (OFS) is used. OFS is designed specifically to deal with characteristics of the repairman problem. These characteristics are based on the information of the vehicles and tasks, for example, the number of vehicles, the average distance from a task to a vehicle or the skills of a driver. These characteristics are also known as input features.

For each customer, a tailored algorithm is designed, but all algorithms follow the same layout: construction, local search, and ruin & recreate. During construction, an initial feasible solution is generated. This solution is constructed by selecting the first task for a route, whereafter the rest of the tasks are planned. Both of these steps can be performed in several ways, and therefore, there are many possible construction algorithms. ORTEC tries to find and use the construction algorithm that gives a best possible initial solution because this will benefit the outcome and runtime in the next two phases. After this initial solution is constructed, an extensive local search is done to find an improvement. After this, ruin & recreate is performed: a small part of the solution is deleted and rebuilt again, in the hopes of finding an improved solution. Especially this last step is very time-consuming. As one can imagine, the second and third steps are depended on the initial solution of the construction. Furthermore, the better the initial solution, the less runtime the local search and ruin & recreate take. It is even seen in practice that the better the initial solution, the better the final solution is. Therefore, a good initial solution has to be generated.

The question then arises of how to find the construction algorithm that generates the best initial solution. Right now, various construction algorithms are tried, based on expert knowledge and customer preferences about the solution. This, however, is a time-consuming activity. Therefore, the following question was asked by ORTEC: can a model be made which predicts the construction algorithm that generates the best initial solution? This is also known as the algorithm selection problem.

1.2 Research approach

The question of how to predict the right algorithm has first been asked by Rice [1976]. He proposed an abstract model that describes the steps for solving the algorithm selection problem. The model includes an overview of the necessary decisions to be made, such as the method used for algorithm selection, performance measure, and feature selection. Unfortunately, one of Rice's main conclusions was that the necessary machinery to solve this problem was still lacking. More on this and the algorithm selection problem can be found in Section 2.4.

Since Rice's proposal, we have come a long way. With machine learning techniques developing in the last decades, we now have better methods to tackle this problem. This has already been shown by the various research papers in the past decade that use machine learning techniques to solve the algorithm selection problem and optimize parameter settings (Guerri and Milano [2004], Guo [2019], Hutter et al. [2011]). A lot of this research, especially on parameter settings, has been done using the machine learning techniques tree regression and clustering. Unfortunately, the problems of nowadays are becoming more complex. Tree regression and clustering are sometimes not able to show any significant results because of their lack of representing complex systems. Therefore, more complex machine learning techniques need to be investigated (Guo [2019]). We are proposing to use the machine learning method Neural Networks (NNs) to solve the algorithm selection problem for finding the best construction algorithm.

An elaborate description of NNs is given in Section 2.5. In short, NNs is a machine learning technique that computes an array of numbers, given a certain input. This is done in the following way: a NN consists of a number of layers, as shown in Figure 1.2. Each layer consists of several nodes, the number of which might vary or be equal in different layers. There are three types of layers. The first layer is always the input layer and the last layer is always the output layer. In the input layer, numbers related to features of the problem are inserted. Based on this data, the model tries to predict the correct output. In between the input and the output layer are the hidden layers. In Figure 1.2, two hidden layers are shown, but a network can have many more. These hidden layers are the foundation for the connections between the input and the output layer. In Figure 1.2, these connections are shown as black arrows. When training a NN, these black arrows are assigned weights, either positive or negative. After these weights are determined in the training phase, a new input can be inserted into the input layer. By multiplying the value of each node with the corresponding weight, the NN computes the numbers of all the nodes in the hidden layers and the output layer. In this way, the NN outputs an array of numbers with the length of the output layer. From these output numbers, conclusions can be drawn or further steps can be taken depending on what the output nodes represent.

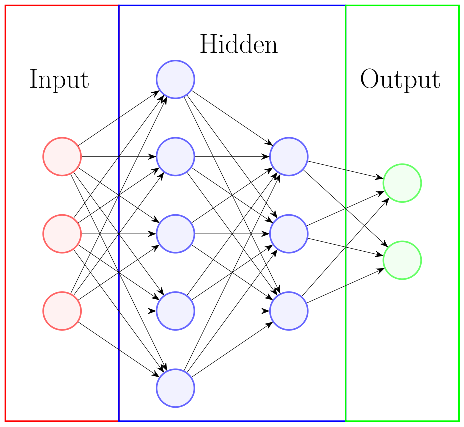


Figure 1.2: A neural network

So how does this connect to our problem? Our goal is to predict which construction algorithm is best to use, given certain problem features. If we translate this to a NN, the input layer consists of numerical or binary features of the data. The output layer consists of probabilities for each construction algorithm. Each probability represents the chance that that particular construction algorithm is best performing. Further explanation of this research setup is given in Section 3.

In this research, we use data from the field service route planning given by ORTEC. Field service is the name given by ORTEC to the VRP of the repairmen, as described in the example at the beginning of the introduction. Not only do we want to build a model which predicts the best construction algorithm accurately, but we also want to understand the influence of the input features on the model performance. This leads to the following research question:

How to find the most accurate Neural Network model, which predicts the best-suited construction algorithm for a given data instance, and what is the influence of the choice of input features on the performance of the model?

In this thesis, the research question is answered by selecting input features and construction algorithms, and building the corresponding NN with training data. After this, an extensive analysis of the built model is done, using test data. We try to improve the model based on this analysis, a data analysis and a feature analysis.

1.3 Thesis outline

In Chapter 2, we discuss the background knowledge needed for our research on VRPs, the algorithm selection problem, and NNs. This is followed by a discussion on our contribution to literature in Section 2.8. In Chapter 3, we discuss the research setup, including some details regarding the data and the performance measures of the model. The results are given in Chapter 4. Finally, we conclude our research and give recommendations for further research in Chapter 5.

Chapter 2

Literature study

In this thesis, we build models predicting which construction algorithm for the VRP is best to use, given certain input features. This problem is also known as the algorithm selection problem (Section 2.4). We predict the best algorithm using the machine learning method NNs. In this chapter, some elaborate information on the VRP, its origin (Section 2.1), different variations (Section 2.2), and existing heuristics (Section 2.3) is given. Also, an overview of machine learning and Neural Networks is given (Section 2.5). Additionally, we provide some background information on data imbalance (Section 2.6) and feature analysis (Section 2.7). Both provide information used to analyze the models in Chapter 4. We end this chapter by explaining our contribution to literature (Section 2.8).

2.1 Vehicle routing problem

Before introducing the VRP, we describe its precursor and one of the oldest optimization problems: the Traveling Salesman Problem (TSP). The classic version of the problem is as follows. We have our main protagonist, the Salesman. He wants to sell his goods in different cities, but as everyone knows, time is money. Therefore, he wants to take the shortest route. The problem of finding this shortest route to visit all cities is called the TSP. There are many different applications of this problem, like a repairman visiting different homes. A downside to the TSP is that it is NP-hard, so there is no known algorithm for solving each instance of the problem to optimality in polynomial time (Papadimitriou [1977]).

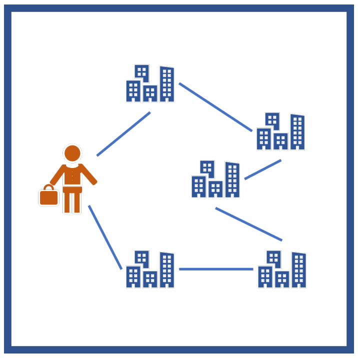


Figure 2.1: Solution to a simple traveling salesman problem

The VRP is an extension of the TSP and was first introduced by Dantzig and Ramser [1959]. In the VRP, we switch from a salesman to several vehicles. In the classic VRP, all vehicles start at the same place: the depot. The vehicles must drive the shortest possible distance while together visiting each address. These addresses represent stores, homes, etc. Again, our goal is to find routes with the shortest total distance. An example of a solution to the VRP can be found in Figure 1.1. Of course, there are many variations to this problem. In the next section, we discuss a few of these variations.

2.2 Variations of the VRP

In this section, a few relevant variations of the VRP are explained. All these extensions have characteristics that are also present in our data. Therefore, it is essential to understand the different problems and where to find further information on them.

2.2.1 Heterogeneous VRP

In the basic VRP, all vehicles are considered identical. However, in real-life cases, this is often not the case. For instance, a supermarket chain can have different sizes of trucks. Another example: one repairman might be able to fix electrical and mechanical problems, while another can only fix mechanical problems. In these cases, the vehicles have different characteristics and are not identical. Thus the routes might need to be adjusted to fit the heterogeneous properties of the vehicles. This is called the Heterogeneous VRP (HVRP). In literature, this problem is also referred to as the Heterogeneous Fleet VRP, and it was presented by Golden et al. [1984]. Since then, many variations to the HVRP have been considered, mostly in combination with time windows, which we discuss in Section 2.2.3. A comparison of some of the heuristics for the HVRP has been studied by Baldacci et al. [2008], while Kusuma et al. [2014] give a more recent overview of heuristics.

2.2.2 Multiple-depot VRP

In the basic VRP, all vehicles start from the same depot, while in real-life cases, this is not always true. For instance, supermarket chains often have several distribution centers from which trucks deliver to the stores. For the repairmen problem, most repairmen start driving from their homes, which can be considered depots. An example solution to such a problem is given in Figure 2.2. This problem is called the Multiple-Depot VRP (MDVRP). An extensive literature review on the MDVRP has been written by Montoya-Torres et al. [2015].

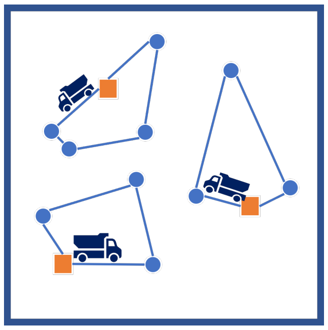


Figure 2.2: Solution to a multiple-depot vehicle routing problem

2.2.3 Time windows

In the basic VRP, all addresses must be visited without any restrictions on timing. Unlike in real-life cases, where there are often restrictions on the times when one can perform a task. For instance, supermarkets might need their products before the stores open, while repairmen can only visit addresses when the owners are home. Therefore, time windows are added as a restriction to the addresses. These time windows show when the address is allowed to be visited. This influences the solution to the problem. Time restrictions lead to the problem called VRP with Time Windows (VRPTW), which has been researched in depth as it applies to many real-life VRPs. There exist two kinds of time window restrictions: hard constraints and soft constraints. With soft constraints, the vehicles are allowed to visit the addresses outside the time windows, but this is penalized. With hard time window constraints, they cannot visit outside the time windows. In this research, we focus on hard time window constraints. For both types of constraints, the VRPTW is an NP-hard problem, so there is no known algorithm that can solve this problem to optimality in polynomial time (Savelsbergh [1985]). More information and an overview of the different exact methods and heuristics for the VRPTW can be found in the work of El-Sherbeny [2010].

2.2.4 Period VRP

In the basic VRP, we have a set of tasks that all need to be planned on one day. Some companies, however, have a set of tasks that can be planned during an entire month or year. This is especially true for repairmen that need to perform maintenance tasks. Those tasks are not restricted to a specific day but should be planned in a particular week or month. This causes the planning to have some freedom but it makes it more complex also, as not only the optimal routes need to be determined, but also the optimal day. In literature, this problem is called the Period VRP (PVRP). In the PVRP, a set of vehicles has to perform a set of tasks over a given period. The goal is to find the shortest total route over all the days. There are also many variations of the PVRP, such as the extension with time windows and multiple depots. Further information on these extensions and other extensions can be found in the works of Mourgaya and Vanderbeck [2006], Francis et al. [2008], and Rodríguez-Martín et al. [2019].

2.2.5 Workload balance VRP

Another variation of the VRP is the one that considers workload balance: the Balanced Cargo VRP (BCVRP). In this problem, the secondary goal, next to minimizing the distance, is to balance the load between the vehicles. This is mainly done for employee satisfaction. Unfortunately, not a lot of research has been done in this area of VRPs. Kritikos and Ioannou [2010] describe the problem in combination with time windows. Shahnejat-Bushehri et al. [2022] did a case study on balancing workload for COVID-19 testers in which workload balancing is essential to reduce the risk of getting infected. This literature only discusses workload balance between vehicles. However, for the repairmen problem, workload balance between days is important for companies and drivers. Balancing the already known tasks is needed because repairmen also need to be available for on-call jobs. By balancing the existing tasks over the days, there is room for new jobs called through during the day. We define this problem as the workload balance over days. To the best of our knowledge, there is no existing literature on this. However, Mancini et al. [2021] show how to optimize the VRP with workload balance between companies. This research uses balancing constraints, which apply to workload balance over days as well. Furthermore, Gulczynski et al. [2011] generated a heuristic that balances the workload of one day equally over the vehicles, also using balancing constraints. Although both of these problems are not exactly what is encountered in the repairmen problem, balancing constraints can be used to balance the workload of one vehicle equally over multiple days as well.

2.2.6 Multiple objectives

In the basic VRP, the optimal routes are determined by finding the routes with the shortest total distance, while all tasks are performed. However, in real-life cases, performing all tasks is often not possible. Therefore, planning as many tasks as possible is more important than the distance of the routes. By planning as many tasks as possible, the company can help more customers and make more profit. Therefore, planning as many tasks as possible is the main objective. However, costs and driving time are also crucial to make the work conditions of the driver and the profit of the company as good as possible (Jozefowiez et al. [2008], Bansal and Goel [2018]). Therefore, not one objective is considered, but multiple. Combining multiple objectives to find the best route, leads to the Multi-Objective VRP (MOVRP). All solution methods for the MOVRP can be divided into three groups: scalar methods, Pareto methods, and additional methods (Bansal and Goel [2018], Nahum and Hadas [2018]). Scalar methods are methods in which all the objectives give a weighted contribution to the total objective function. The main goal is to minimize this total objective function. Pareto methods use Pareto dominance directly. Pareto dominance is the concept that a solution is better if one objective is better, while the other objectives do not get worse. This concept is used in many evolutionary algorithms and local searches (Nahum and Hadas [2018]). The group of additional methods consists of methods that do not belong to either of the other two classes, like lexicographic strategies (Nahum and Hadas [2018]). In lexicographic optimization, the objectives are ordered from most important to least important. First, the most important objective is optimized, whereafter the second objective is considered. A small positive change in the first objective is more important than a big positive change in the second one, and so on. Further information on different types of objectives, the three groups, and links to further research on MOVRP can be found in the works of Bansal and Goel [2018], Jozefowiez et al. [2008], and Nahum and Hadas [2018].

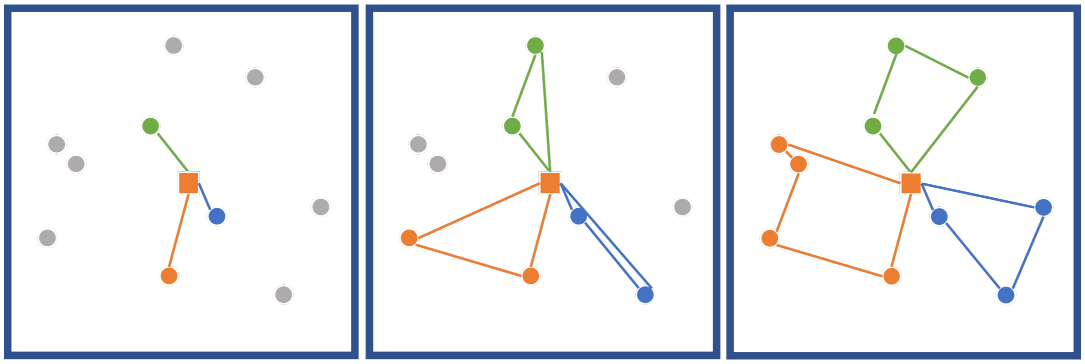
2.3 Heuristics for the VRP

The VRP is an NP-hard problem, just like the TSP. Therefore, there is no known algorithm that can find an optimal solution in polynomial time (Lenstra and Kan [1981]). To find good feasible solutions, many heuristics have been developed for the VRP (Ho and Haugland [2004], Laporte [1992], Laporte et al. [2014]). In this section, we discuss three groups that are important for our research: construction algorithms, local search, and ruin & recreate. Construction algorithms are self-explanatory: they construct an initial solution to a problem. Constructing an initial solution is done in many different ways. We give detailed descriptions of some heuristics in Section 2.3.1. Local search heuristics try to improve an already existing solution by making small changes. A short overview of some general ideas is given in Section 2.3.2. Finally, ruin & recreate is a metaheuristic that is used to escape local minima. A description of this method is given in Section 2.3.3.

2.3.1 Construction algorithms

Construction algorithms build a solution to a problem from scratch. In this section, we explain how a construction algorithm is designed. The general decisions when developing a construction algorithm are: choosing a parallel or sequential method, choosing a seed task, and choosing how to plan the rest of the tasks. After this explanation, a short paragraph is dedicated to the advantages and disadvantages of greedy heuristics. This section concludes with a few examples of standard construction heuristics. We give this detailed description of construction algorithms as our goal is to predict the best construction algorithm. Therefore, a good understanding of the various construction algorithms is in order.

Most construction heuristics use the method of insertion. This is also the easiest way to explain the difference between a parallel and a sequential method. Insertion is the method of inserting a new task somewhere in an existing route. Which task is inserted, and where, differs per method. There are two methods to construct routes with insertion: parallel insertion and sequential insertion. Parallel insertion builds all routes simultaneously, whereas sequential insertion builds per route. An example of parallel and sequential insertion can be seen in Figure 2.3. In more detail, the difference between parallel and sequential insertion is the following. Parallel insertion takes a predetermined number of vehicles and finds a seed task for all vehicles. After this, all of the routes are built simultaneously by inserting tasks, as shown in Figure 2.3a. Sequential insertion starts with one vehicle, selects a seed task, and builds the route until inserting a task is not feasible anymore. After this, the algorithm moves on to the next vehicle, as seen in Figure 2.3b. So, an important decision is whether we built all routes simultaneously (parallel) or per route (sequential). Parallel insertion is faster than sequential insertion (Crainic [2008]). However, the number of vehicles must be determined beforehand: this might be too many or too few. For sequential insertion, this is never the case.


(a) Parallel insertion

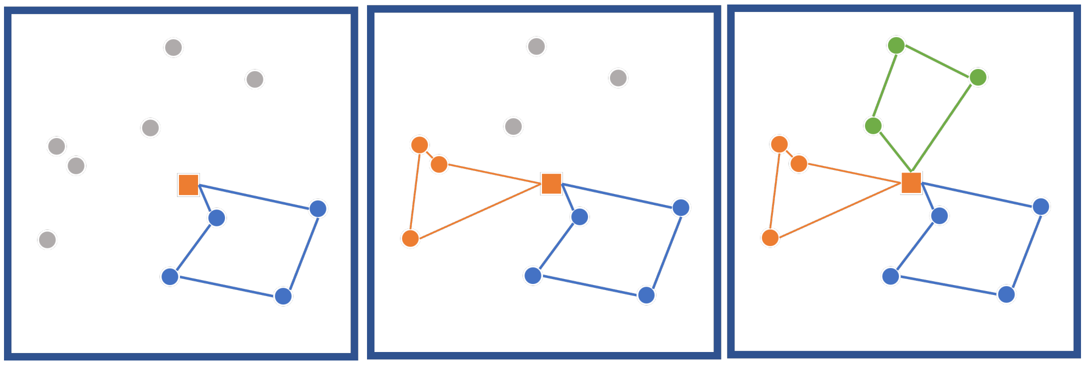
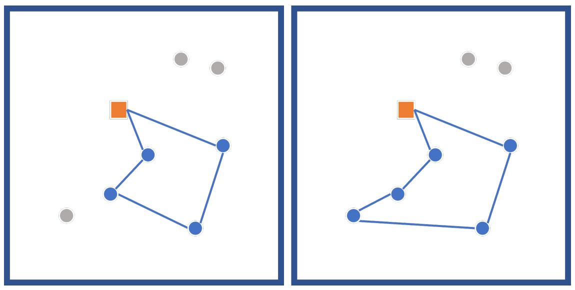

(b) Sequential insertion

Figure 2.3: Examples of parallel and sequential insertion

After selecting a parallel or sequential method, the method of selecting a seed task is chosen. The most classical ways to do this are by randomly selecting a task or by selecting the task closest in distance to the depot. But, there are many more methods. Often in real-life cases, a seed task that is very hard to plan is selected. This difficult task is selected, because there is a big chance it cannot be planned later in the process due to some difficult properties. To maximize the number of planned tasks, it should be planned first. Often considered tasks are the task furthest away from the depot, the biggest task, or the task with the shortest time window.

Planning the remaining tasks is often done by different criteria than selecting the seed task. For planning the remaining tasks, the task with the shortest distance to the current routes is often selected. In real-life cases, distance is often replaced by driving time. Some other criteria considered when planning the remaining tasks are the biggest task, shortest time window, or random. However, this is often done within a certain distance of the seed task, as it is highly illogical to plan two tasks that are very far away from each other on one route.

Construction algorithms are nearly always greedy. This means that decisions are made only considering the present. This results in choices, that are optimal at the moment but might not be optimal in the future. An example is shown in Figure 2.4. In Figure 2.4a, a task closest in distance to the current route is added, following the greedy construction algorithm. After adding this task, the route is full, and the two remaining tasks are performed by another vehicle. However, in Figure 2.4b, it is shown that adding the second closest task to the current route would have been favorable, as this leaves room for another task that is close by. This leads to only one task being performed by another vehicle. Therefore, we conclude that the greedy construction algorithm does not always lead to the optimal solution.


(a) Greedy construction algorithm solution

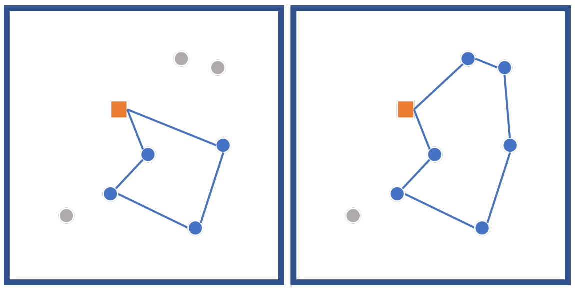

(b) Optimal solution

Figure 2.4: Examples of how a greedy method does not lead to the optimal solution

Cheapest insertion

The construction method cheapest insertion might be the most known construction algorithm for VRPs. It was introduced by Solomon [1987] and there are two versions: a parallel version, which is called parallel cheapest insertion, and a sequential version, which is called sequential cheapest insertion. We only explain sequential cheapest insertion. In sequential cheapest insertion, we build the routes one by one. We start with selecting the seed task for the first route. In cheapest insertion, this is the task that is considered cheapest: shortest in distance to the depot. All tasks are checked, and the closest, feasible task to the depot is planned. After this, all tasks are checked again, and the task that is feasible and has the shortest distance to the current route is selected and inserted. This is repeated until the route is full. After this, a new vehicle is selected, and the process of building the next route starts. This is repeated until all tasks have been planned or all vehicles are full. In real-life VRPs, this cheapest insertion algorithm is tweaked a bit, by not selecting the task closest in distance, but the task that leads to the shortest driving time.

Regret insertion

Regret insertion is a parallel insertion method that is part of the savings heuristics group. The savings heuristics all have in common that the tasks are ordered and inserted based on the largest savings (Schneider and Kirkpatrick [2006a]). In regret insertion, the seed task is selected by one of the criteria for seed tasks described above. After this, we start planning the remaining tasks. In contrast to cheapest insertion, not the cheapest task is planned, but the task with the largest regret. For each task, the shortest feasible distance to the routes is determined. If the task would be inserted there, it would be planned in between two other tasks. Then, the second shortest feasible distance to the routes is determined: if we do not plan the task on its best place, what place would then be preferable? The difference between those two distances is known as the regret value. The higher the regret value, the higher the priority is to plan the task in its first choice. Therefore, the task with the highest regret value is planned first. And the process is started over again. Diana and Dessouky [2004] generated a regret insertion algorithm for a large-scale problem with good results. A likewise algorithm for the TSP is described by Hassin and Keinan [2008]. Of course, regret insertion has a longer runtime than cheapest insertion as not only the cheapest place for a task needs to be determined, but also the second cheapest.

Nearest neighbor

The nearest neighbor algorithm was first designed for the TSP. However, with minor adjustments, it can also be used for the VRP. First of all, it should be noted that this greedy heuristic is not an insertion method, the routes that are built are not yet fully formed, as the way back to the depot is not taken into consideration. The nearest neighbor algorithm is very intuitive and the name already explains the algorithm. This algorithm can be done both sequentially and in parallel. We only discuss the sequential version. First, a seed task is selected at random. After this, the closest neighbor of this seed task is added at the end of the route. Each time, the task that is closest to the end of the route is added, until the route is full. Only then, the way back to the depot is added to the route and the route is completed. After the route is completed, a new seed task is chosen at random and the process repeats itself. This algorithm has similarities to cheapest insertion, but the heuristics do differ. In cheapest insertion, for each task, the distance to the entire route is considered, while in nearest neighbor only the distance to the last task of the route is considered. Furthermore, in cheapest insertion, the driving back of the vehicle to the depot is incorporated into the route from the beginning, in the nearest neighbor heuristic, this part of the route is added at the end. Again, an important note for the nearest neighbor heuristic is that the 'closest' in the general VRP is in regards to distance. However, in real-life VRP 'closest' is most of the time regarding driving time.

2.3.2 Local search

After the construction, a solution can often still be improved. This is done by performing a local search. In local search, a new solution is proposed within a neighborhood of the initial solution. After this, a check is performed on whether or not this new solution is better than the initial solution. In local search, the new solution is only accepted when it is better than the old solution (Gilli and Schumann [2010]). However, methods like threshold accepting have been developed, which act like local search, but also accept slightly worse solutions (Gilli and Schumann [2010]). Regardless of the outcome of this check, the process is started over again and a new neighboring solution is suggested. This makes local search an iterative method. Often, local search stops when there is no improvement within all the neighboring solutions or in several iterations.

The neighborhood of a solution can be defined in several ways, but often it is defined as the solutions that are generated by making a local change (Gil-Rios et al. [2021], Dechter [2003]). In this way, local search methods can be divided into two groups, based on the kind of local changes they make: intra-route optimization and inter-route optimization. Intra-route optimization focuses on optimizing the order of tasks within a route. The most used intra-route method is 2-OPT. In this method, we remove two connections of the same route and replace them with two new connections such that the route is different, as shown in Figure 2.5. A generalization of this method is k-OPT, in which k connections are removed and replaced such that the route is different. k can be an arbitrarily large number, but for practical reasons, it is often limited to 2-OPT or 3-OPT. Another intra-route method is complete enumeration per route, in which all orderings of the tasks are evaluated and the best one is selected. This is an exact and very expensive method. For problems with many tasks per route, this method is often not used.

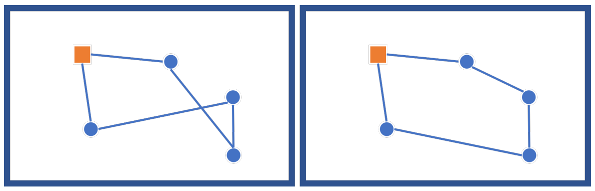
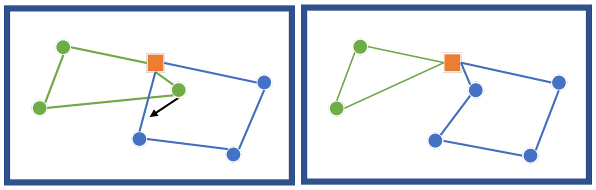
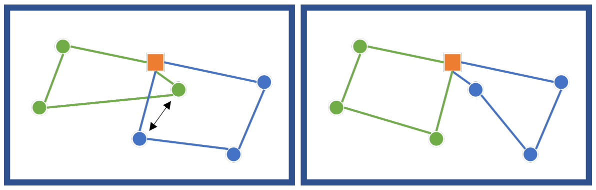


Figure 2.5: 2-OPT

Inter-route optimization focuses on moving tasks between routes. A well-known inter-route local search method is the move method, which removes one task from a route and inserts it in another route, see Figure 2.6a. A local search method that is similar to move, is swap. In the swap method, two tasks from different routes are swapped, as seen in Figure 2.6b. There are also two other versions of these local search methods: large neighborhood swap and large neighborhood move. These are identical to their single-task counterparts, but instead of only moving one task or swapping two, a set of tasks in the same neighborhood is moved or swapped. Due to this method, larger portions of the routes are moved. A last variation of these methods is large neighborhood swap & move, in which both methods are performed simultaneously.



(a) Move



(b) Swap

Figure 2.6: Examples of inter-route local search methods

All these local search methods can be used on their own, or they can all be applied to an initial solution. Any order of application can be used. The downside of local search is that it can easily get stuck in a poor-quality local minimum (Schneider and Kirkpatrick [2006b]).

2.3.3 Ruin & recreate

With local search methods, changes are made within a small neighborhood, and eventually will get stuck in a local minimum (Schneider and Kirkpatrick [2006b]). However, often the solution can still be improved by making changes in a larger neighborhood. This is called a large neighborhood search. We already saw three large neighborhood methods in the last section: large neighborhood swap, large neighborhood move, and large neighborhood swap & move. Another large neighborhood search method is called ruin & recreate. Schrimpf et al. [2000] were the first to suggest this method and found “record-breaking optimization results”. In ruin & recreate, a part of the solution is completely removed and rebuilt again, with the rest of the solution staying intact. In this way, routes can be rebuilt in a very different manner. After a part of the solution is removed and rebuilt, this new solution and the initial solution are compared. If the new solution performs better, or worse within a certain threshold, a local search is performed. After this, the new solution is only selected if the solution has indeed improved. Whether or not the new solution is accepted, the process is started again and a new part of the solution is deleted and rebuilt. This is done until a pre-set number of iterations has taken place.

The removal and rebuilding of the routes can both be done in several ways. A few removal methods are the following: random removal, removal per route, and a fixed percentage of removal. The rebuilding of the routes is also called construction. The construction algorithms discussed in Section 2.3.1 can be used to perform this rebuilding. In practice, we see that ruin & recreate can drastically improve the solution (Schrimpf et al. [2000]). Unfortunately, it is a very expensive method, and so, the runtime of ruin & recreate is very long.

2.4 Algorithm selection

The algorithm selection problem is the problem of finding the best model that maps an instance of a problem to the best-performing algorithm. This problem was first described by Rice [1976].

He proposed a framework on how to tackle this problem. Many studies use this framework (Guerri and Milano [2004], Guo and Hsu [2003], Guo [2019], Salisu et al. [2017], Smith-Miles et al. [2010]). In this section, we discuss this framework, the feature selection problem, and research on the algorithm selection problem in combination with machine learning.

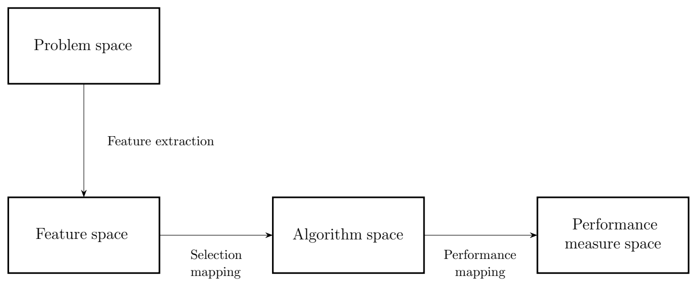


Figure 2.7: Framework of the algorithm selection problem Rice [1976]

The framework of Rice [1976] divides the problem into several spaces, between which mappings need to be determined, as seen in Figure 2.7. The basics of the framework are the first and third boxes in the chain: the problem space and the algorithm space. The problem space includes all the instances of a problem. The algorithm space includes all the algorithms that are considered. The mapping from the problem space to the algorithm space, in which we are interested, is not shown in Figure 2.7. This mapping would be identical to the algorithm selection problem: given an instance of the problem space, which algorithm is best to select? But, this question is hard to answer, and therefore, Rice expanded this question by adding two boxes in the chain.

First of all, it is important to determine what is meant by the best algorithm. For our research, we consider a pool of algorithms that compute a solution to the given VRP. To find out which algorithm performs best, we might look at the objective value computed by the algorithm. After computing this objective value for all algorithms in the algorithm space, we could select the algorithm that gives the best objective function. However, this might not always be considered the best algorithm. For instance, runtime and simplicity are also factors that are considered important in solving real-life VRP. Therefore, Rice suggested a performance mapping, which computes the different performance results: the mapping from box 3 to box 4.

Secondly, a feature space is included between the problem space and the algorithm space. Many problems, especially real-life problems, cannot be put into a model that selects the best algorithm, because the instances contain too much information and are not in the right format. Luckily, the problems can be described using features, which give an understanding of the problem, without using all the information. Using features gives way to many different mapping techniques, including machine learning techniques like classification trees, clustering, and NNs. However, adding a feature space results in a new question: how to select the features that help predict the right algorithm? This is known as the feature selection problem. Unfortunately, there is not a lot of understanding of the relation between input features and the algorithm performance, and not a lot of studies discuss their feature selection (Guo [2019]). For instance, Mocking [2017] uses a computable set of features for the VRP problem, but does not corroborate as to how these features were selected and how they influence the results. Smith-Miles et al. [2014] suggested a method in which a large pool of instance features is computed and different subsets of features are selected to build the models. There are several methods to select this subset of features. The most common are wrapper methods, filter methods, and embedded methods (Tanamala [2021]).

Wrapper methods are greedy search heuristics in which different subsets of features are selected (Tan [2020]). This method starts with an initial subset. For a few iterations, new features are added, or some features are eliminated. Based on the evaluation of the new models, the subset is updated, or not. In this greedy way, several subsets of input features are modeled and analyzed. Filter methods select features based on univariate statistics. Embedded methods embed the feature selection into the model, for instance, Lasso regularization can let unimportant features disappear by training a parameter to be zero (Tanamala [2021]). These methods help to show what features are important, but still the question remains how the features influence the outcome.

Rice investigates the algorithm selection problem for several examples, but concludes, as one can imagine, that the answer to finding the best selection method, is highly dependent on the problem at hand. An overview of some different methods for solving the problem, the research done on this, and future research suggestions are given by Salisu et al. [2017], Smith-Miles et al. [2010], and Smith-Miles et al. [2014]. Guo [2019] applied the methodology of Smith-Miles et al. [2014] to the real-life problem of crew pairing for an airplane schedule to gain knowledge on the relevant features for the algorithm selection. They applied machine learning (logistic regression and classification trees) to the feature extraction mapping, as shown in Figure 2.7. Interestingly, they suggest further research using NNs.

2.5 Neural networks

In this research, we solve the algorithm selection problem for data of ORTEC using machine learning. As a machine learning method, we use Neural Networks (NNs) to build a model to determine the best algorithm. In this section, we give a short overview of the different categories of machine learning and why we use NNs (Section 2.5.1), a technical explanation of a NN (Section 2.5.2), and information on a few important characteristics of NNs (Section 2.5.3). Lastly, a short overview of different types of NNs is given (Section 2.5.4).

2.5.1 Machine learning

Machine learning has been a fast-developing field of mathematics with many breakthroughs and new techniques. The downside of machine learning is that it is hard to translate the real world into a format in which a model can make decisions and predictions. However, once this challenge is overcome, the software makes faster decisions than a human being. A great example of this, was when AI beat the word champion in the complex board game Go in 2016 (Wong and Sonnad [2016]). With our fast-growing big data problem, smart models which can make accurate decisions fast are highly necessary.

Machine learning can be divided into three categories: supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, the model is given a labeled input. The model is built by both using the input and their corresponding correct label. In this way, the model learns what input corresponds to what output. An example of supervised learning is the problem of a computer recognizing hand-written digits. By showing the model a lot of hand-written digits and the corresponding answer, it learns what characteristics belong to what number. When a new number is given, the model will check all the important characteristics and decide which number it is. Supervised learning can be categorized into two subclasses: classification and regression. Classification is the problem of determining which category a certain input belongs to. The previous example belongs to classification supervised learning. In regression supervised learning, the data instance is not sorted into a category, but the model predicts the value of an output feature. For instance, when trying to forecast the weather, a model can predict the chance of rain. This is an example of regression supervised learning. Unsupervised learning trains the model using only the input data. It tries to cluster the input into a predetermined number of groups, based on characteristics of the data. For instance, if a model is fed with a pool of different geometric shapes, it might group the data based on shapes having sharp edges, or round edges. In this way, the data is divided into two groups. However, the model itself decides on what characteristics it groups the data. In reinforcement learning, the model repeatedly selects an action from a predetermined set of actions, the environment reacts to this action and gives back a reward or penalty. In this way, the model learns what actions have a positive effect and what actions have a negative effect. An example of reinforcement learning is when a model learns how to play chess, by playing a lot of games against opponents. This is how a computer learned how to play Go (Wong and Sonnad [2016]) and defeated the world champion a few years ago. This paragraph included only a short, general description of the three machine learning categories. More information on supervised learning, unsupervised learning, and reinforcement learning is given by Padmanabha Reddy and Mohan Krishna Varma [2020], Wu et al. [2022], and Nian et al. [2020], respectively.

The algorithm selection problem is a classification problem. We want to assign the data instances to the correct class, or in our case, algorithm. However, we are not only interested in the correct algorithm but also in acquiring more information about the difference in performance. Because a combination of different algorithms can also help find a better solution, it is valuable knowledge to know whether one algorithm performs a lot better than all the others, or whether a few algorithms perform equally well. Assigning a probability based on the relative performance of the algorithm is a natural way to compare the algorithms between data points. A high probability indicates a well-performing algorithm, while a low probability indicates a bad working algorithm. Algorithms with similar probability have similar performance. Building a model that assigns the right probabilities to the algorithms, does not lead to a classification problem, but rather a regression problem. From these probabilities, we can still conclude which algorithm performs best. A few regression methods are linear regression, random forest, and neural networks. In linear regression, we try to find the relation between the input and the output of the model by fitting a linear line. However, it is highly unlikely our data has a linear relation, and therefore, this method does not fit our problem. Furthermore, Kadam et al. [2019] investigated the prediction of water quality with the help of both linear regression and neural networks. They conclude that neural networks yield a more precise model. In a random forest, a set of decision trees are built. A data point is evaluated in each of them, and the average output of those decision trees equals the output of the random forest. Decision trees are a widely used machine learning method, as they are easily interpretable. Therefore, many studies prefer to use this method, as the model is relatively easy to interpret, and it can easily handle outliers and interactions between features (Singh et al. [2016]). This is also shown in the number of studies that use decision trees and random forests to solve the algorithm selection problem or that use them to study the impact of the input features, as a random forest can both be used for classification and regression problems (Guerri and Milano [2004], Guo [2019], Hutter et al. [2011]). In their study, Guo [2019] studied the relation between input features and algorithm performance, one of their methods being decision trees. Unfortunately, they did not find the results they had hoped. They, however, suggested that a more complex machine learning method like NNs, should be considered for further research. Although we do not have similar problem instances, we take their advice to heart. Their research shows that complex problems cannot be represented by a simple machine learning method. This is also confirmed by Singh et al. [2016]. Therefore, we want to investigate if a more complex method, like NNs, can find the complex connection between the input features and the algorithm selection. That NNs work well for non-linear and dynamic data, is confirmed by Singh et al. [2016]. A downside of NNs is that they train relatively slowly, which causes problems for large networks.

2.5.2 A neural network

In this section, we give a technical explanation of NNs (Heinlein and Postek [2022], Sanderson and Pullen [2017]). We explain the generic model, which is also known as a Multi-Layer Perceptron (MLP), a subclass of Feed Forward Neural Networks (FFNNs). These names are sometimes used interchangeably in literature. We discuss the difference between the two in Section 2.5.4. MLPs have one of the most basic structures: at least one hidden layer and all connections are forward connections. NNs with other structures are discussed in Section 2.5.4. The details on certain characteristics of the generic model are given in Section 2.5.3. A short introduction to NNs is already included in Section 1.2. To refresh the reader's mind: a NN is a model consisting of layers, which predict the outcome by learning the connection between the nodes of these layers, as seen in Figure 2.8.

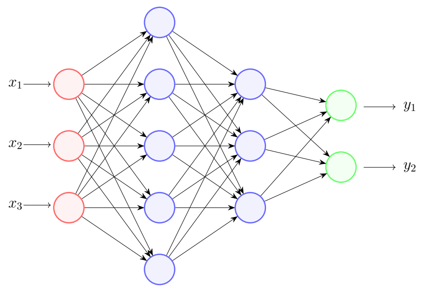


Figure 2.8: A neural network with input $\boldsymbol{x} \in \mathbb{R}^3$, output $\boldsymbol{y} \in \mathbb{R}^2$, number of hidden layers $L = 2$

A NN is a supervised learning technique. This means the model is trained with both the data instances $\boldsymbol{x} \in \mathbb{R}^n$ and their corresponding label $\boldsymbol{y} \in \mathbb{R}^m$, which represents the correct outcome. The model is built in three steps: determining the number of nodes, parameter initialization, and training. In machine learning, the nodes of a NN are called neurons. A neuron j of layer l is denoted as $a_j^{(l)}$, which is also used to refer to the value of this neuron. The dimensions of the data and labels determine the number of neurons in the input and output layer. If the dimensions of our data and label are n and m, then our input layer has n neurons and the output layer has m neurons, as shown in Figure 2.8. Furthermore, the number of hidden layers L and their corresponding width d_l is set. The width corresponds to the number of neurons in each layer and may vary between layers. The example in Figure 2.8 shows two hidden layers, with corresponding widths of five and three. In the notation of NNs, the number of hidden layers L is taken as a starting point. This means the input layer is denoted as layer 0, and the output layer is denoted as layer $L + 1$, with corresponding width d_0 and d_{l+1}.

Training the model will optimize the parameters, which leads to an accurate model. The parameters consist of several weight matrices and bias vectors. The weight matrices $W^{(l)}$, for $l = 1, \cdots, L+1$, group the weights between layers $l - 1$ and l in a logical way. The bias vector $\boldsymbol{b}^{(l)}$ for $l = 1, \cdots, L$ holds the bias values for each of the neurons of the hidden layers. The weights and biases need an initial value. Assigning these values is called parameter initialization, see Section 2.5.3. Furthermore, the model uses an activation function α, which needs to be selected. The function takes in a computed number, based on the weights and biases, and gives back an outcome, which is the value of the neuron. The reason for this activation function is to account for the non-linearity of the model. Therefore, the activation function is non-linear. The most common activation functions are the ReLU and sigmoid functions. More on this can be found in Section 2.5.3.

Before we move on to explaining the training of the model, it is good to know how the output of a NN is computed. We first show how the weight matrices, bias vectors, and activation function are used for computing the value of a single neuron, and then describe this process on the level of a layer. In a NN, the neurons of the input layer take on the values of the input data. The values of all other neurons are computed in the following way. For a node $a_j^{(l)}$, node j of layer l, all values of the previous layer $\boldsymbol{A}^{(l-1)} \in \mathbb{R}^{d_{(l-1)}}$ are taken into account, as shown in Figure 2.9. These values are multiplied by the weights of the corresponding edges. After this, the bias corresponding to node j is added, followed by evaluating this outcome in the activation function α. This results in the value of the node being $a_j^{(l)}$. This value is used in the next layer or is the outcome of the network. Doing this for all neurons in all layers is called forward propagation.

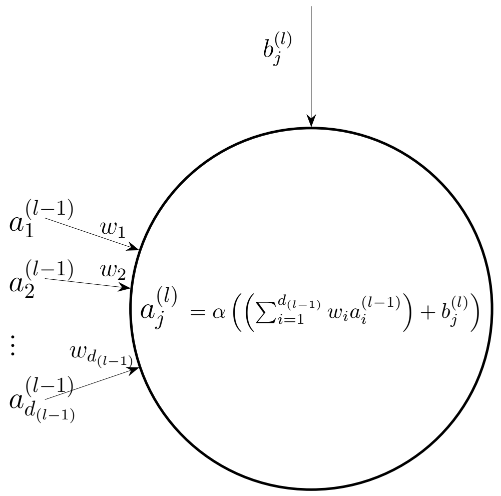

$b_j^{(l)}$
$a_1^{(l-1)}$
w_1
$a_2^{(l-1)}$
w_2
$\vdots$
$w_{d_{(l-1)}}$
$a_{d_{(l-1)}}^{(l-1)}$
$a_j^{(l)} = \alpha\left(\left(\sum_{i=1}^{d_{(l-1)}} w_i a_i^{(l-1)}\right) + b_j^{(l)}\right)$

Figure 2.9: Processing input for one node in a NN, based on Vieira et al. [2017]

This process is repeated for each neuron in the network. If we want to look at this process from a higher level, we can also investigate how the network computes the value of the neurons layer by layer. The state of the neurons in layer l is notated in vector $\boldsymbol{A}^{(l)}$. We then organize the weights $w_1 \cdots w_{d_{(l-1)}}$ for each of the neurons in layer l. Those weights are the connections between layer $l-1$ with width $d_{(l-1)}$, to layer l width with d_l. Structuring these weights gives matrix $W^{(l)} \in \mathbb{R}^{d_{(l-1)} \times d_l}$. This results in eventually $L+1$ weight matrices. The bias of each of the nodes of layer l with width d_l is stored in bias vector $\boldsymbol{b}^{(l)} \in \mathbb{R}^{d_l}$. The bias vectors are only used for computing the values of the hidden layers and, therefore, the network trains L bias vectors. The activation function which is set, could theoretically be different for every node and every layer. In practice, the activation function is often identical for the entire network. Computing the layers results in the following functions.

$$\boldsymbol{A}^{(1)} = \alpha\left(W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right) \tag{2.1}$$
$$\boldsymbol{A}^{(i+1)} = \alpha\left(W^{(i+1)}\boldsymbol{A}^{(i)} + \boldsymbol{b}^{(i+1)}\right), \text{ for } i = 1, \cdots, L-1$$
$$\hat{\boldsymbol{y}} = W^{(L+1)}\boldsymbol{A}^{(L)}$$

In these formulas, $\boldsymbol{A}^{(i)}$ is the state of the neurons in layer i, $\boldsymbol{x} \in \mathbb{R}^{d_0}$ is a data point and $\hat{\boldsymbol{y}} \in \mathbb{R}^{d_{l+1}}$ is the state of the neurons in the output layer, and therefore, the predicted outcome.

Now we have enough knowledge to move on to the training of the model. Training the model is done with backpropagation. First of all, all the training data points $\boldsymbol{x}_1 \cdots, \boldsymbol{x}_n$ are fed to the model, which results in prediction $\hat{\boldsymbol{y}}^i$ for each data point $\boldsymbol{x}_i$. After this, the performance of the model is evaluated using a cost function C. Several cost functions can be used, described in Section 2.5.3, but normally the model is evaluated using L2-loss. This means that the cost of a data point $\boldsymbol{x}_i$ with label $\boldsymbol{y}^i$ is:

$$C^i = \sum_{j=1}^{d_{l+1}} (\hat{\boldsymbol{y}}^i_j - \boldsymbol{y}^i_j)^2$$

in which m is equal to the number of neurons in the output layer. $\hat{\boldsymbol{y}}^i$ is the predicted outcome for a data point $\boldsymbol{x}_i$, and $\boldsymbol{y}^i$ is the label of this data point. In other words, for each of the neurons in the output layer, the prediction is compared to the label. Minimizing this cost, and so minimizing this function, leads to an good working model. This is disregarding overfitting, which we discuss in Section 2.5.3.

Normally, minimizing a function $f(\boldsymbol{x})$ is done by solving $f'(\boldsymbol{x}) = 0$, but unfortunately, this is not doable for the cost function, as $\hat{\boldsymbol{y}}_j$ is computed by the many functions shown in Equation (2.1). Therefore, this minimum is found by trying to steer the parameters in the right direction step by step. This is done by computing the local gradients of each of the parameters with respect to the cost function for one specific data point. We discuss how to compute these local gradients in the next paragraph. However, if we only use the local gradients of one data point, the model will only work well for that particular data point. The model needs to take into account all other data points as well. Therefore, the adjustments to the parameters are only applied after all data points are considered. An average of the gradients of all the data points is computed and used as the steering direction. The downside of this is that taking all data points into account is computationally heavy. If we have K data points, this results in the total cost function, which needs to be minimized:

$$C_{\text{total}} = \sum_{i=1}^{K} C^i$$

To compute the gradients of the weight matrices and bias vectors with respect to the cost function is a very notationally heavy undertaking. We try to give a concise explanation and give the reader a feeling of how this is done. We say a weight $w^{(l)}_{kj}$ is a weight from node k of layer $l-1$ to node j of layer l. Similarly, we say the bias value $b^{(l)}_j$ is the bias of node j in layer l. Furthermore, we say that $z^{(l)}_j$ is the summation of the weights multiplied by the input and the bias value for node j in layer l. This is the summation that is inserted into the activation function. This leads to the following computation for node $a^{(l)}_j$, shown in Figure 2.10.

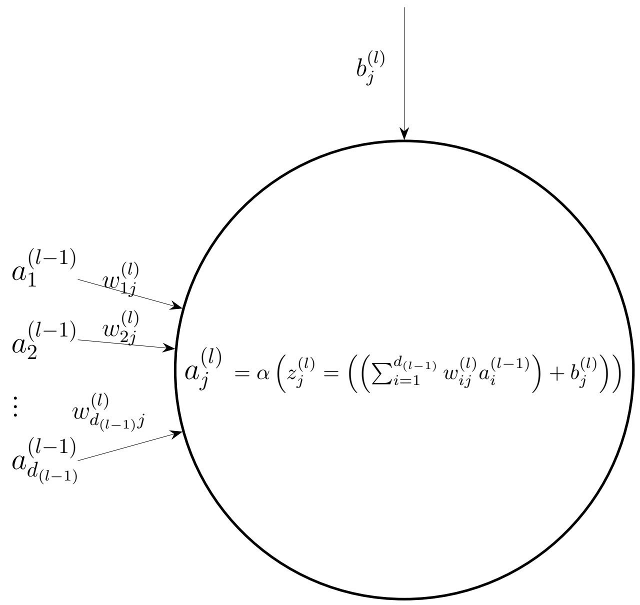

$b_j^{(l)}$
$a_1^{(l-1)}$
$w_{1j}^{(l)}$
$a_2^{(l-1)}$
$w_{2j}^{(l)}$
$\vdots$
$w_{d_{(l-1)}j}^{(l)}$
$a_{d_{(l-1)}}^{(l-1)}$
$a_j^{(l)} = \alpha\left(z_j^{(l)} = \left(\left(\sum_{i=1}^{d_{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)}\right) + b_j^{(l)}\right)\right)$

Figure 2.10: Processing input for one node in a NN, based on Vieira et al. [2017], using different notation

Computing the gradients of the weight and bias values with respect to the cost function can be done using the chain rule. We can divide it into three parts:

$$\frac{\delta C^i}{\delta w_{kj}^{(l)}} = \frac{\delta z_j^{(l)}}{\delta w_{kj}^{(l)}} \cdot \frac{\delta a_j^{(l)}}{\delta z_j^{(l)}} \cdot \frac{\delta C^i}{\delta a_j^{(l)}}$$

$$\frac{\delta C^i}{\delta b_j^{(l)}} = \frac{\delta z_j^{(l)}}{\delta b_j^{(l)}} \cdot \frac{\delta a_j^{(l)}}{\delta z_j^{(l)}} \cdot \frac{\delta C^i}{\delta a_j^{(l)}}$$

The first two parts in each chain rule equation are easy to compute by looking at the equations shown in Figure 2.10. The derivative of the activation function is, of course, dependent on the choice of activation function. Therefore, an activation function with an easily computable derivative is chosen.

$$\frac{\delta z_j^{(l)}}{\delta w_{kj}^{(l)}} = a_k^{(l-1)}$$

$$\frac{\delta z_j^{(l)}}{\delta b_j^{(l)}} = 1$$

$$\frac{\delta a_j^{(l)}}{\delta z_j^{(l)}} = \alpha'(z_j^{(l)})$$

The last part in each chain rule, the computation of the derivative of the cost function with respect to the j-th neuron, is a bit more difficult, as the node $a_j^{(l)}$ influences all nodes of the following layer. Therefore, we refer to Appendix A for this derivation. Note that the derivatives only need information we already have, so it is easily computable. Computing the overall direction the parameters should go in, is done by averaging the specific gradient over all the data points:

$$\frac{\delta C}{\delta w_{kj}^{(l)}} = \frac{1}{K}\sum_{i=1}^{K}\frac{\delta C^i}{\delta w_{kj}^{(l)}}$$

$$\frac{\delta C}{\delta b_{j}^{(l)}} = \frac{1}{K}\sum_{i=1}^{K}\frac{\delta C^i}{\delta b_{j}^{(l)}}$$

This method of updating the weight matrices and bias vectors with gradients, called backpropagation, was inspired by gradient descent. Gradient descent is a well-known optimization method (Boyd and Vandenberghe [2004]). It is used to solve the problem of minimizing a given function $f(\boldsymbol{x})$, which cannot be done exact. This well-evolved method takes a starting point $\boldsymbol{x}$ and updates it using the gradient $\triangle\boldsymbol{x} = -\nabla f(\boldsymbol{x})$. In this way, the value $\boldsymbol{x}$ is updated in the following way: $\boldsymbol{x} = \boldsymbol{x} + t\triangle\boldsymbol{x}$ with learning rate t. This is also what is done in backpropagation:

$$w_{kj}^{(l)} = w_{kj}^{(l)} + t \cdot \frac{\delta C}{\delta w_{kj}^{(l)}}$$

$$b_{j}^{(l)} = b_{j}^{(l)} + t \cdot \frac{\delta C}{\delta b_{j}^{(l)}}$$

In continuous optimization, a lot of different learning rates t have been investigated (Boyd and Vandenberghe [2004]). The one that is often used for NNs is the fixed step size, which is a constant between 0 and 1. This means another parameter needs to be chosen before training the model: learning rate t. Often, this parameter is set equal to 0.001.

After the weights and biases are updated, all the training data is put into the model again, the cost function is computed, and the weights and biases are updated with backpropagation again. This is repeated until a certain stopping criterion is met, further discussed in Section 2.5.3.

2.5.3 Characteristics of the neural network

In Section 2.5.2, an explanation of the generic NN model is given. There, a few characteristics are mentioned, which we discuss in more detail here. These are the stopping criteria, the activation function, parameter initialization, the cost function, and overfitting. Furthermore, we discuss one more concept, which is important for our research and to improve the accuracy of the model: batch normalization.

Stopping criteria

When training a NN, all data points are fed to the model, and forward and backward propagation is performed. This process repeats until a stopping criterion is met. This stopping criterion can be a predefined number of iterations or a predefined threshold for the cost function with respect to the training data. But, the most common method is early stopping. Before training the NN, the data is always split into a training and a test set. With early stopping, the training data is then split into a training and a validation set. The training set is used to train the model. After each iteration, the validation set is evaluated in the model. Once the loss of the validation set does not diminish for a few iterations, the training stops. For our models, we stop the training after 30 iterations without improvement.

Activation functions

The activation function α is used for computing the value of each neuron and accounts for the non-linearity of the problem. Activation functions were designed as a decision boundary on whether the neuron will or will not transfer information to the next layer (Santosh et al. [2022]). In other words, whether the neuron is activated or not, hence the name activation function. The most simple activation function shows this best:

$$\alpha(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

This activation function defines each neuron to be either one or zero. As one can imagine, this often does not lead to a very accurate model. Of course, many other non-linear functions exist, but some additional function properties are essential when choosing a good activation function. First of all, the function must be easy to evaluate to reduce the runtime of training the model. Furthermore, we have seen in Section 2.5.2 that the derivative of the activation function is used for training the model. Therefore, this derivative must exist, be computable, and be easily evaluated. Two of the most common activation functions are sigmoid and ReLU.

The sigmoid function and its derivative are:

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

$$\sigma'(z) = \sigma(z)\left(1 - \sigma(z)\right)$$

A property of the sigmoid function is that it outputs numbers between 0 and 1. Furthermore, its gradient is very steep near 0. In the backpropagation method, this leads to the neurons being steered towards 0 or 1 (Santosh et al. [2022]). The main problem with sigmoid is that it has vanishing gradients: the gradients are smaller than one, so with the method of training, they will become small, very fast. This leads to only minor updates of the parameters in the right direction (Wang [2019]).

This vanishing gradients problem is the reason that the Rectified Linear Unit (ReLU) function is also a popular choice, as it does not have this problem. The derivative of the ReLU function is either 0 or 1, and therefore there are no vanishing gradients. The ReLU function and its derivative are:

$$ReLU(z) = \max\{0, z\}$$

$$ReLU'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Following mathematical rules, the derivative of $z = 0$ does not exist. However, if we want the model to work, we need to define it a value. The value of the ReLU derivative at $z = 0$ is defined as 0 (Santosh et al. [2022]). Another problem encountered with the ReLU activation function is the dying ReLU: once a neuron reaches the state 0, it can never recover, and it will never again contribute to the network (Leung [2021]). To account for both problems, variations of the ReLU function have been generated, for instance, the Scaled Exponential Linear Units (SELU).

$$SELU(z) = \begin{cases} \lambda z & \text{if } z > 0 \\ \lambda a\left(e^z - 1\right) & \text{if } z \leq 0 \end{cases}$$

$$SELU'(z) = \begin{cases} \lambda & \text{if } z > 0 \\ \lambda a e^z & \text{if } z \leq 0 \end{cases}$$

with $\lambda \approx 1.05$ and $a \approx 1.67$. More information on SELU and a computation of the values of these constants can be found in Klambauer et al. [2017]. More information on other variations of the SELU function can be found in Goodfellow et al. [2016]. Moreover, information on the sigmoid function and general ReLU functions is given in Section 6.3 of Goodfellow et al. [2016] or Section 2.2.2 of Santosh et al. [2022].

In our multi-class regression problem, the goal is to predict with which probabilities an instance belongs to a certain class. The output layer outputs these probabilities approximately but often deviates slightly from the correct values. These deviated numbers should not be interpreted as probabilities. Sometimes, the model can output numbers bigger than one, and there is no guarantee the summation of the entries of the output vector is equal to one, as is required when representing a probability. To make the output of the model more interpretable and have it represent a probability, an activation function is used in the output layer. This activation function can be a different function than used in the rest of the network. In multi-class regression, the softmax function can determine these probabilities. This function uses the values of the entire output layer but each time only determines the value of one neuron i. Therefore, the softmax function is used for every entry i of the output vector $A^{(L+1)}$, and notated as follows:

$$softmax(\boldsymbol{A}^{(L+1)})_i = \frac{e^{a_i^{(L+1)}}}{\sum_{j=1}^{m} e^{a_j^{(L+1)}}}$$

The softmax function transforms the output into a probability (Kotu and Deshpande [2019]). Furthermore, the softmax function preserves the relative ordering of the classes. Apart from using this function in the output layer, this function can also be used as an activation function for the entire network.

Parameter initialization

There are many methods for parameter and weight initialization (Goodfellow et al. [2016]). The most common method is randomly initializing the weight values. In this way, when repeating the process, the NN can end up in a different local minimum, which might improve the solution. Another method of randomly assigning weights is normalized initialization (Goodfellow et al. [2016], Heinlein and Postek [2022], Glorot and Bengio [2010]): the weights are set randomly using a uniform distribution based on the depths of the layer and its previous layer:

$$w_{ij}^{(l)} \sim U\left(-\sqrt{\frac{6}{d_l + d_{l-1}}}, \sqrt{\frac{6}{d_l + d_{l-1}}}\right)$$

When building a model, this randomness is generated by a computer, and therefore, it is pseudo-random. For our research to be reproducible, we fix this randomness, such that the model assigns the same random weights each time when it is trained under the same circumstances.
For setting the bias, the consensus is to initialize them all at zero (Goodfellow et al. [2016]). Another method is to assign all biases the same constant. This prevents the network from dying when using a ReLU activation function. However, there is no reason to choose this method when we use a different activation function.

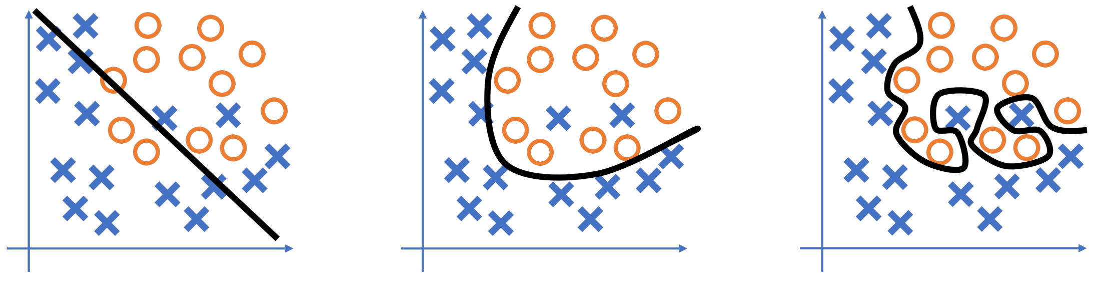
The other parameters (number of hidden layers, the depth of the hidden layers, and the learning rate) are based on the type of NN used. Often they are tried to be improved by doing a grid search. In grid search, for each parameter, a set of options is constructed. For each combination of parameters, a model is trained and evaluated. Based on this, the most favorable parameter combination is determined.

Cost function

The goal of the cost function is to determine the lack of performance of the model and help steer the parameters in the right direction. In other words, we try to minimize the cost. We have to correct the previous terminology. The cost function C_i is called a loss function because it only considers the data point i. The cost function C is indeed called a cost function, as it uses all data points. Loss functions have been studied a lot, especially in the field of continuous optimization. Some examples of loss functions are L2-loss, L1-loss, quadratic loss, and cross-entropy. More on these loss functions can be found in Kordonsky [2021] and Van der Meulen [2022]. The loss functions can be divided into two groups: classification and regression loss. As our model is a regression model, we use a regression loss. Examples of regression loss are L1-loss, L2-loss, and mean squared error loss.

Overfitting

When training a NN, we divide the available data into two subsets: training data and test data. With the help of the training data, we build and train the model. With the help of test data, we determine the model performance by evaluating data that the model has never seen before. However, the problem can occur in that the model performs extremely well on the training data but poorly on the test data. This problem is called overfitting: the model can find an excellent relation between the training data and training labels, but this relation generally does not hold. This results in bad performance on the test data. An example of overfitting is shown in Figure 2.11, in which a model tries to predict whether a data point will be a circle or a cross based on dividing the data points into two groups.



(a) Underfitting (b) Best fitting (c) Overfitting

Figure 2.11: Overfitting and underfitting by Bhande [2018]

There are several reasons why overfitting can occur. First of all, the NN can have too many parameters for the number of data points. Another reason is that the data is noisy, which means the data points include a lot of extra information that is unimportant but does negatively influence the model. Lastly, one can perform backpropagation too many times, such that the model will focus more on perfectly predicting the training data than on finding the general relation between input and output. If the overfitting is caused by reason one, adding extra data when training and testing might help. Unfortunately, this data is not always available. Reducing the number of parameters might also work in this case. One method to reduce the number of parameters is called Dropout (Srivastava et al. [2014]). In this method, some units of neurons are randomly dropped from the network during training to see if it results in more accurate models. But also lowering the number of hidden layers might be helpful. If the overfitting is caused by one of the other two reasons, the way to reduce overfitting is by regularization. Regularization methods diminish the error of the test data at the cost of the error of the training data. The advantage of regularization is that it also takes underfitting into account, in which the model's predictions are too general, causing the model to be inaccurate. There are many regularization techniques, even some specially made for deep learning. Many are elaborately described in the study of Goodfellow et al. [2016]. One of the best-known techniques, coming from linear algebra, is L2-regularization, also known as ridge regression. One of the easiest ways to prevent overfitting is early stopping, discussed in the stopping criteria paragraph in this section. Another method is only training for a low preset number of iterations. Setting this number too low might lead to underfitting, so setting this preset number right is a challenge alone. More information on these two methods can also be found in Goodfellow et al. [2016] and AssemblyAI [2021a].

Batch normalization

When training a NN, some input features might attain values on an extremely different scale. If we want all input features to influence the next neuron, the corresponding weights must differ quite in size. These highly varying weights can lead to the unstable gradient problem: the gradients of the weights can either explode or vanish (which also shows up with some activation functions). Furthermore, this causes very slow convergence (Heinlein and Postek [2022]). To prevent the gradients from exploding or vanishing, the data should be normalized.

Standard normalization of the data helps to prevent these previously mentioned shortcomings. However, the new method called batch normalization is even more powerful. Besides normalizing the input data, batch normalization also normalizes the output of each layer. In other words, normalization is performed after each layer in the network. This method has shown that it solves the unstable gradient method. Furthermore, it reduces the need for regularization methods because it also helps prevent overfitting (AssemblyAI [2021b]). Batch normalization works as follows. First, all the input features (or the values of the neurons) are normalized. After, the values are scaled by a parameter γ and the offset β is added. These parameters γ and β are trained by the model itself. And so, we have more parameters to train in the training phase. Exactly one γ and one β per batch normalization, resulting in $2(L+1)$ extra parameters. Batch normalization, therefore, causes one iteration of the training to take longer than it would without batch normalization. However, it causes the network to train more accurately, which leads to fewer iterations needed to reach the same accuracy (AssemblyAI [2021b]). In the end, batch normalization will save time and make the training faster.

2.5.4 Types of NN

Over the years, different structures of NNs have been developed. The structure of the layers can be used, to improve the models based on different purposes. In this section, we discuss three well-known types of NNs and their purpose. More types of NNs can be found in Goodfellow et al. [2016].

Convolutional neural network

The Convolution Neural Network (CNN) might be the most well-known type of NN. CNNs have given impressive results in image analysis: object detection, digit recognition, and image recognition (Datta [2020]). It has now even found its way from simple image analysis, like recognizing hand-written images, to analyzing images in the medical field (Chen et al. [2018], Shajun Nisha and Nagoor Meeral [2021]). The main assumption of CNNs is that the data instances have a grid-like topology (Goodfellow et al. [2016]). Obviously, we do not, as we work with for the most part unrelated input features. Therefore, we do not use CNNs.

Recurrent neural network

Another well-known type of NN is a Recurrent Neural Network (RNN). In a RNN, not only the values of the neurons of the previous layer are used, but also the values of other neurons. This can be the previous value of the neuron itself, or the value of a neuron in one of the next layers, as shown in Figure 2.12. RNNs work well for time-dependent data (Heinlein and Postek [2022]) or for processing a sequence of data (Goodfellow et al. [2016]). This does not apply to our problem, so we do not use RNNs.

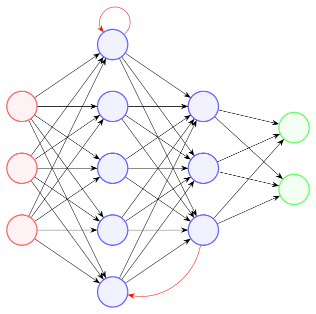


Figure 2.12: Recurrent neural network with in red the recurrent connections

Multi-Layer perceptron

The type of NN we use is the Multi-Layer Perceptron (MLP), as introduced at the beginning of Section 2.5.2. In literature, MLPs and Feed Forward Neural Networks (FFNNs) are often used interchangeably. Officially, MLPs are a subclass of FFNNs. The MLP is one of the first NNs designed, and it follows the easy FFNN structure: only forward connections. The differences between MLPs and FFNNs are the following: MLPs only have connections to the next layer, while FFNNs can also have ones that skip a layer, as long as the connections are forward. Furthermore, MLPs have at least one hidden layer, while for FFNNs this does not have to be true. MLPs' ability to classify correctly is shown by its use and the results in the medical field (Marques et al. [2022], Merjulah and Chandra [2019]). Despite us not having a classification problem but a regression problem, we do border on classification as we also focus on the best performing algorithm. Therefore, we decide MLPs are a good choice.

2.6 Imbalanced data

A NN performance is not only dependent on the choice of network and parameters but also on the data. In general, the consensus is that the more data a network is provided with, the better it trains. However, the quality of the data is also important. In machine learning, there is often the issue of imbalanced data. Imbalanced data occurs when data from one class (classification) or data from a certain range of values (regression) is underrepresented. The model wants to perform well in general, and so, focuses on the majority of the data points and not on the underrepresented ones. This focus causes the model to predict this last group very badly. There are two main ideas in literature to prevent this problem: downsampling and upweighting. In this section, we discuss the general idea of both and give a few examples of specific methods. We discuss our method in more depth in Section 2.6.3.

2.6.1 Downsampling

With downsampling, only a subset of the original training data is selected and used as training data. This data is selected such that each class, or each range of values, is represented by approximately the same number of data points. There are different ways to select such a subset, for instance, by uniform sampling. In uniform sampling, data points from the majority class are removed at random, until the minority and majority classes are represented evenly. This explanation immediately introduces the main problem of downsampling: reducing the amount of data. By removing many data points from the majority class, bad results can occur because the model does not have enough data to train on anymore. However, if downsampling does not lead to this problem, it performs really well. Tyagi and Mittal [2020] state that downsampling methods reduce the skewness of a data set and are well-performing over all kinds of machine learning methods, and so should be considered when balancing a data set.

Other examples of downsampling methods are edited nearest neighbor and the neighborhood cleaning rule. These methods have in common that they use information about neighboring points to remove certain data points. However, the downfall of this, is that it highly increases the computational time. Furthermore, it needs to be established on what basis two points are called neighbors. Also, almost all neighbor methods are designed for classification problems. The details of these methods can be found in Lee and Seo [2022].

Furthermore, problem-specific algorithms have been developed for selecting data points, especially in the medical field. Ren et al. [2022] perform research on imbalanced scRNA-seq data. They discuss studies in which both uniform sampling and thresholding techniques prove promising. Moreover, they show that their own algorithm MURP improves on the original results. The idea of this algorithm is more problem based and depends on the information about the scRNA cells. However promising, these very specific algorithms do not apply to our problem.

If we take a look at the problem at hand, none of these methods fits. No problem-specific algorithms apply to our problem as far as we know. We also do not use the mentioned nearest neighbor methods, as they are designed for classification rather than regression. Furthermore, many nearest-neighbor methods create synthetic data points. We only use existing data points, as we know for sure these exist in practice. Because we have a multi-class problem, it is hard to sample data points uniformly. Data points that contribute to the majority of one algorithm can be part of the minority group for another algorithm. Therefore, it is harder to select which data points to remove by uniform sampling. To solve this problem, we formulate our problem as an Integer Linear Program (ILP). This method removes data points from the majority while keeping in mind the influence on the other classes. More details on ILPs and the exact formulation of downsampling can be found in Section 2.6.3 and Section 3.6.1, respectively. To the best of our knowledge, no research on downsampling using an ILP has been performed yet.

2.6.2 Upweighting

The other option to deal with imbalanced data is upweighting. In upweighting, the focus is not on removing data such that the data set is balanced, like in downsampling, but on prioritizing the underrepresented data points such that the model thinks it is dealing with a balanced data set. This prioritizing can be done in several ways. The most basic method is the one of class weights.

In this method, a weight is assigned to each data point. The underrepresented data points are assigned high weights, while the data points of the majority class are assigned low weights. This weight is used in the loss function: the data points with a high weight contribute more to the loss than those with a low weight. The model minimizes the loss and, therefore, prioritizes the data points with a high weight. The upside of this method is that it does not diminish the number of data points in the training set. However, Tyagi and Mittal [2020] show that upweighting does not necessarily perform better than downsampling.

There are two main methods for upweighting. One is literally assigning weights to each data point, as described in the last paragraph. Usually, this is done for classification problems. The most common method to assign weights is the following. The scale between the majority and minority class is determined, for example, 1 : 15. Then the weights of the minority class are set to 15, while the weights of the majority class are set to 1. The other method for upweighting is called oversampling. In this method, extra data for the minority class is generated. The easiest way is to include the existing data points of the minority class several times in the data set. However, other methods use synthetic data to increase the minority class. One of the most known methods is called SMOTE, in which we take a convex combination of two data points of the minority class, which creates a new combination of input features, and so, a new data point. Details about SMOTE are found in Elreedy and Atiya [2019].

However, like in downsampling, we want to stay clear of synthetic data points to make the data set as realistic as possible. Therefore, we do not consider SMOTE and other synthetic oversampling methods. Furthermore, we cannot weight the data points using class weights, as we are working with a multi-class regression problem. Therefore, the only option is to oversample data points by including existing data points in the training set several times. Again, this is a bit more difficult as we are working with a multi-class regression problem. Luckily, this problem can be formulated as an ILP. More information on ILPs and the exact formulation of the oversampling problem can be found in Section 2.6.3 and Section 3.6.2, respectively. To the best of our knowledge, no research on oversampling using an ILP has been performed yet.

2.6.3 Integer linear program

In discrete optimization, we categorize all optimization problems, each group having its own corresponding solution methods. One of these categories contains the maximization and minimization problems with integer variables. These problems are referred to as integer programming problems. This category can be divided into several subcategories. One of them is known as the integer linear programming problems: the maximization or minimization problems that are formulated only using linear formulas. Such a formulation is known as an Integer Linear Program (ILP). An ILP consists of an objective function, which is minimized or maximized, and a set of constraints. These constraints dictate the feasible set of values for the variables. The format of a general ILP is as follows:

$$\min \boldsymbol{c}^T \boldsymbol{x} \tag{2.2}$$

$$A\boldsymbol{x} \leq \boldsymbol{b}, \tag{2.3}$$

$$\boldsymbol{x} \in \mathbb{N}^n_{\geq 0} \tag{2.4}$$

where $A \in \mathbb{R}^{m \times n}$ is a known matrix, $\boldsymbol{b} \in \mathbb{R}^m$ and $\boldsymbol{c} \in \mathbb{R}^n$ are known vectors and $x \in \mathbb{N}^n_{\geq 0}$ is an unknown vector consisting of nonnegative integers. ILPs have been used to solve planning and strategic problems for decades (Wolsey [1998]). Difficult problems must be modeled so they can be solved by computers. Modeling the problem as an ILP is preferable, as there are algorithms, like branch&bound, which solve the problem exactly. Solving to optimality, however, might be hard, as many problems are NP-hard. If optimality cannot be reached in a reasonable time, we can consider the best solution found till then. Because ILP is an exact method, we know the optimality gap between the optimal solution and the current solution. Therefore, we know how good the current solution is. More on solution methods and different applications can be found in Wolsey [1998]. The problem of balancing the data set with downsampling or oversampling is a minimization problem. These problems can be described only using linear constraints. The ILP formulation of downsampling and oversampling can be found in Section 3.6.1 and Section 3.6.2, respectively.

2.7 Feature analysis

An important question for machine learning methods is how the input features contribute to the output. Feature analysis can give insights into the most influential features, which is valuable information. First of all, it makes the model easier to interpret. By performing such an analysis, rules of thumb can be devised and redundant features removed, which reduces the number of parameters that need to be trained. Furthermore, new input features can be deduced. Secondly, in some practical cases, the choices made by a model might be seen as unethical. For instance, in predictive policing, where potential criminal activity is identified. For those models, ethnicity could play an important role. Understanding the model's choices is highly necessary to corroborate that they are ethical. For our research, however, ethical issues are less explicitly present.

The analysis of feature influence is not included in research a lot of the time. Only the model performance is considered important, unlike the reason the model behaves as it does. This is also shown by the limited amount of literature on feature analysis. Because of this lack of literature, Christoph Molnar, a statistician and machine learner, started to combine the little knowledge there was. This section is based primarily on his work (Molnar [2022]).

Unfortunately, NNs are one of the machine learning methods in which the relation between the input and the output is the least clear. In NNs, this relation is represented by the many weights and bias matrices, which are not easily interpretable. Therefore, other techniques have to be used. In this section, we discuss three of them: SHAP, partial dependence plots, and permutation feature importance. All explanation models make certain assumptions about the data to be able to analyze the relation between the input and output. Because those assumptions might not be correct, Molnar [2022] recommends using several methods so there is more evidence for the drawn conclusions. We investigate the model using three different methods. All three methods are model-agnostic, which means they do not use any specific characteristics of the model but can be applied to any machine learning model. There exist feature analysis methods specific to NNs, but to the best of our knowledge, these can only be used for CNNs.

Between the different feature analysis methods, a division is made between local and global explanation methods. Global methods use all data to draw their conclusions, while local methods only use one data point. Partial dependence plots and permutation feature importance are global methods. For SHAP, both a global and local variation exists.

2.7.1 SHAP

The first explanation method is called SHapley Additive exPlanations (SHAP). In this section, we discuss a local and global model-agnostic variation of SHAP. SHAP is one of the most used feature analysis tools, and therefore, there are a lot of different variations, most of them being model-specific. The local SHAP method we discuss is known as KernelSHAP, while the global SHAP is referred to as SHAP Feature Importance. In this section, we first discuss the origin of SHAP, followed by an explanation of KernelSHAP, and concluded with a short paragraph on SHAP Feature Importance. In both methods, SHAP tries to explain the prediction of a data point $\boldsymbol{x}$ by computing the contribution of each feature to the deviation from the average. This contribution is also known as the Shapley value, a concept from game theory. Inspired by this concept, the SHAP method was designed. To understand SHAP, first, understanding the Shapley values is in order.

In game theory, we have a set of players (input features) which influence the result of the game (output). One way to determine the quantity of this influence is by calculating the Shapley value. To determine the Shapley value of an input feature i, we use an already trained model $\hat{f}$. The Shapley value of input feature i is determined by looking at all possible combinations of input features and determining the output of the model both with and without input feature i. The difference between the two values is considered the influence of input feature i. Averaging the difference of all possible combinations leads to the average influence of input feature i: the Shapley value. In more mathematical terms, consider a set of input features N, with one of the features being input feature i and coalitions S. Let $\hat{f}(S)$ be the output of model $\hat{f}$ for coalition S, then the Shapley value ϕ_i is calculated by:

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} \left(\hat{f}(S \cup \{i\}) - \hat{f}(S)\right)$$

In theory, this is a good way to determine the influence of an input feature, but in practice, there are many possible coalitions, so it takes a lot of runtime to determine a Shapley value. This problem was solved by Štrumbelj and Kononenko [2013], who proposed an approximation of the Shapley value by not looking at all coalitions but sampling a subset. Unfortunately, there are still many other disadvantages to using Shapley values in practical cases. More on this can be found in Molnar [2022].

To overcome some of these disadvantages, KernelSHAP was designed. KernelSHAP is a local method, so only computes the Shapley value using one specific data point $\boldsymbol{x} \in X$. It determines the Shapley value for every input feature by trying to fit a linear regression model that explains the output of the model. To train this linear regression model, the following steps are taken, considering data point $\boldsymbol{x}$ of length n:

1. Sample K random coalition vectors $\boldsymbol{z}^k \in \{0,1\}^n, k \in \{1, \cdots, K\}$
   Each of these vectors $\boldsymbol{z}^k$ represents a different coalition using the input feature values of the data point $\boldsymbol{x}$. Each coalition vector is the blueprint for a coalition. How a coalition is built from a coalition vector is explained in step 2.

2. Establish the coalition based on the coalition vector $\boldsymbol{z}^k$ and apply the model $\hat{f}$ to determine the corresponding output value.
   To extract the coalition from the coalition vector, a mapping h is used from $\boldsymbol{z}^k$ to the original feature space. The map h translates the 0's and 1's to feasible feature values. The 1's are mapped unto the same feature value as the feature has for data point $\boldsymbol{x}$. The 0's are mapped unto a random feasible value. The feasible values consist of the values this feature obtains for other data points, but a different value than the feature has for data point $\boldsymbol{x}$. Eventually, each coalition is evaluated in the model: $\hat{f}(h(\boldsymbol{z}^k))$.

3. Weight the different coalition vectors $\boldsymbol{z}^k$
   By evaluating a coalition vector $\boldsymbol{z}^k$ in model $\hat{f}$ as described in Step 2, we only can conclude how much the entire coalition deviates from the original output. Our goal is to determine the

influence per input feature, so a logical choice would be to look at the coalition in which only one input feature is different. The deviation of that coalition is entirely dependent on this one input feature. However, feature interference would not be accounted for at all. Therefore, all other coalitions must also be considered. But, for the coalition vectors with only one or a few input features set to zero, it is clearer which input feature causes the deviation, so those coalitions are more insightful. To account for this, the coalitions are weighted. Coalition vectors with only a few 0's or 1's get a large weight, while the other coalitions get assigned small weights. To determine this weight, Lundberg and Lee [2017] proposed the SHAPkernel. Because some coalitions have much higher weights than others, which means they have a lot more influence, SHAP implementations often sample random coalitions in a smart way: they first sample coalitions with the higher weights, and only if there is enough memory and time left, they sample coalitions with lower weights. In this way, the approximation of the Shapley values becomes more accurate.

4. Fit a weighted linear model
   Lundberg and Lee [2017] show that linear regression with the SHAPkernel leads to Shapley values indeed. Therefore, such a linear regression model is fitted using the coalitions $\boldsymbol{z}^k$:

$$\hat{f}\left(h(\boldsymbol{z}^k)\right) \approx \phi_0 + \sum_{i=1}^{n} \phi_i^{(\boldsymbol{x})} z_i^k$$

   In this formula, ϕ_0 is equal to the average prediction over all data points in X: $\phi_0 = \frac{1}{|X|}\sum_{\boldsymbol{x}\in X}\hat{f}(\boldsymbol{x})$. Training this model with an appropriate loss function leads ϕ_i to approximate the Shapley values, which can then be extracted. The Shapley value ϕ_i for data point $\boldsymbol{x}$, is also more explicitly denoted as $\phi_i^{(\boldsymbol{x})}$. More on this can be found in Molnar [2022].

The most important thing to remember about Shapley values is their interpretation. The Shapley values do not say how much an input feature contributes to the output, but how much an input feature contributes to deviating from the average output.

Local Shapley values can be used to determine the global Shapley values. For each individual data point $\boldsymbol{x}$, we can calculate the influence of the input feature i. The overall influence of the input feature, called the absolute Shapley values, is the average of the influence over all data points:

$$I_i = \frac{1}{|X|}\sum_{\boldsymbol{x}\in X} |\phi_i^{(\boldsymbol{x})}|$$

This is also known as SHAP Feature Importance.

Shapley values have a ton of advantages. The most important is that SHAP has a strong theoretical foundation. There has been a lot of research on Shapley values, their application in game theory, and their application in economics. Moreover, many model-specific variations exist, which are very fast. Therefore, they can easily compute the global Shapley values. Unfortunately, there has not been a variation developed for NNs. There are only two main disadvantages. The general method KernelSHAP is quite slow, so computing the global Shapley values is not always possible. Furthermore, feature dependence is not highly taken into consideration. This is very hard to do when looking at feature influence. We encounter this problem in the other explanation models as well.

2.7.2 Partial dependence plots

The second explanation method is called partial dependence plots. Partial dependence plots are used for global explanation. In a partial dependence plot, one or two features are plotted against the output of a specific neuron of the output layer. This is done by computing the marginal effect of the input feature on the output. An example of a partial dependence plot is shown in Figure 2.13a.

The marginal effect of a coalition S of input features is computed by the following steps.

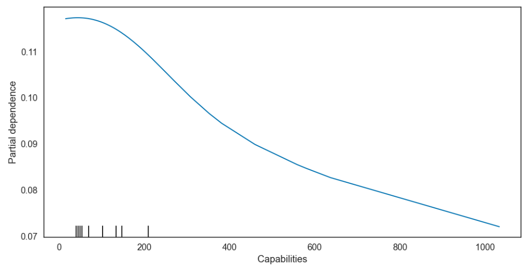
1. Sample M random data points $\boldsymbol{x}^{(i)}, i = 1, \ldots, M$. From each of these data points, we remove the input features in coalition S, resulting in the vector $\boldsymbol{x}_C^{(i)}$, which contains the values of the remaining input features for data point i. In partial dependence plots, the number of input features in coalition S can only be one or two. The reason for this is the visualization of the outcome.

2. Determine the average output for various values of coalition S
   The features in coalition S can take on different combinations of values. For a finite number K of combinations, we investigate the influence of the values on the output. We do this by first considering one combination of values, called $\boldsymbol{x}_S$. This combination of values is evaluated in combination with the vector of remaining features $\boldsymbol{x}_C^{(i)}$ for every data point:

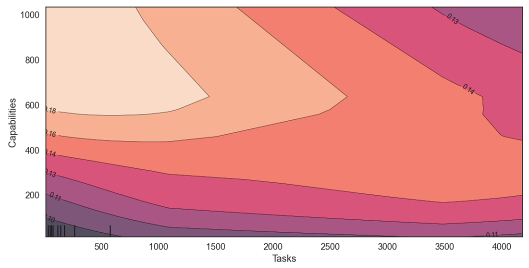
$$\text{average output for values } \boldsymbol{x}_S = \frac{1}{M} \sum_{i=1}^{M} \hat{f}\left(\boldsymbol{x}_S, \boldsymbol{x}_C^{(i)}\right)$$

   In words, each data point is evaluated in the model, whereby the original values for the input features of coalition S are replaced by $\boldsymbol{x}_S$. This repeats for a finite number K of values for coalition S. Hereby it is assumed that the input features are independent of one another. This is highly unlikely, and therefore, a major disadvantage of this method.

3. Plot the partial dependence
   In step 2, we determined the average output for K different combinations of the coalition S. For each combination, this average output can be plot against the values of the features of the coalition S. When the coalition has one input feature, this leads to a graph, as seen in Figure 2.13a. When the coalition has two input features, this leads to a 3D-colormap as seen in Figure 2.13b.



(a) Feature capabilities



(b) Features capabilities and tasks

Figure 2.13: Partial dependence plots

The main issue with partial dependence plots is, that it is assumed that the feature values are independent of each other. Of course, this is often not the case. Therefore, partial dependence plots are not infallible. This is partly solved by looking at a coalition with two input features, in which the dependency of the two input features is shown, as seen in Figure 2.13b. Unfortunately, this is the maximum number of features that is comparable simultaneously. The influence of the other features is not taken into account. The advantage of partial dependence plots is that it is easy to implement and interpret.

But how do we determine which feature has the most influence on the output? Greenwell et al. [2018] proposed a logical way to look at the curve of the plot. He applied the following logic: when the curve is flat, the influence of the feature on the output is low, while if it varies, the influence is high. In other words, he defined importance in terms of how much the values deviate from the average value: the standard deviation. This value can be computed by the formula for the standard deviation for coalition S, in which we have K different combinations of feature values:

$$I(S) = \sqrt{\frac{1}{K-1}\sum_{k=1}^{K}\left(\hat{f}_S\left(x_S^{(k)}\right) - \frac{1}{K}\sum_{k=1}^{K}\hat{f}_S\left(x_S^{(k)}\right)\right)^2}$$

In this formula, the sum on the right computes the average of the output for all K different coalitions. Thereafter, in the subtraction, the difference between this average and the output for a specific combination is computed. This leads to the standard deviation of coalition S.

2.7.3 Permutation feature importance

The last method is called permutation feature importance and is a global model-agnostic feature analysis method. This method is easy to understand but does not have a strong mathematical background. The method works as follows: we start with a trained model $\hat{f}$ and training data X. For this method, we use each data point $\boldsymbol{x} \in X$. If we want to determine the feature importance of feature i, we permute the values of this feature within the data points. After, the newly constructed data points are evaluated in the model $\hat{f}$, and the loss is determined. If shuffling the feature results in a big change in the loss, we say that the feature is important, while if there is only a minor change, we say the feature is not influential. How much influence a feature has, called the Feature Importance (FI), is calculated by taking the absolute value of subtracting the new loss from the original loss. The higher this value, the more important the feature is. Repeating this process for every feature results in a measure in which we can order the features from the most important to least important.

2.8 Our contribution

The VRP and its heuristics have been widely researched in the last decades, as discussed in Section 2.2 and Section 2.3. With the rise of machine learning, some new literature on the algorithm selection problem in combination with the VRP has arisen. Unfortunately, as discussed in Section 2.4, few research papers discuss their feature selection. One article that does discuss their feature selection, albeit for the crew pairing problem, is Guo [2019]. Unfortunately, they did not find a solid relation between their features and the performance of the algorithms. They recommend looking into more complex machine learning methods, like Neural Networks.

In this thesis, we take the advice of Guo [2019] to heart, and we investigate whether Neural Networks can help identify the features which help distinguish between the performance of the different algorithms. We do not do this by using a greedy heuristic on the input features. which has been done before, but by performing an elaborate feature analysis of several models. By looking at what input features the model thinks are important, we build a new model with the input features which help the model distinguish between the different algorithms. This might also help find relations that we were not expecting to find.

Furthermore, we not only focus on the input features and their influence, but we also look at data balancing. Because we are working with a multi-labeled regression problem, not many data balancing methods have been developed and used, as most of the data balancing methods are designed for classification problems. Therefore, we investigate something new: formulating the data balancing problems as an ILP and finding a good solution within a reasonable amount of time. In this thesis, we investigate whether this leads to good results.

Chapter 3

Research setup

This chapter includes the details of our research setup. We start with a short introduction of the proposed question from ORTEC, followed by a methodology and general data description. The details of the methodology, the training of the NN, and the choices surrounding labeling data, input features, and construction algorithms are explained in depth.

For each OFS customer, ORTEC designs an algorithm. As seen in Section 2.3, many different heuristics exist with corresponding parameter settings. Right now, decisions are made based on expert knowledge. Exploring these decisions takes a lot of time. The question from ORTEC is whether data can help make decisions easier. A good prediction model on the algorithm performance will save time, as it helps the experts look in the right direction. Also, evaluating how the data leads to a prediction gives insight into why particular heuristics perform better than others. These insights can also be applied to future data. In this research, we build a prediction method, predicting which construction algorithm works best. Also, an elaborate feature analysis is performed to understand the prediction model. With that information, the model can be improved. This leads to the following research question:

How to find the most accurate Neural Network model, which predicts the best-suited construction algorithm for a given data instance, and what is the influence of the choice of input features on the performance of the model?

This research question is answered by performing the following steps:

1. Selecting a set of input features based on the data and selecting a set of construction algorithms (Section 3.2 and Section 3.3, respectively)
2. Retrieving, labeling, and analyzing data (Section 3.1, Section 3.4, and Section 3.5, respectively)
3. Building the prediction model and analyzing its performance (Section 3.7 and Section 4.2.1)
4. Performing elaborate feature analysis on this model (Section 4.2.2)
5. Rebuilding and re-analyzing various prediction models using different data sets and various (sub)sets of input features (Section 3.6, Section 4.3, Section 4.4, Section 4.5, Section 4.6 and Section 4.7)

The research question might indicate a classification problem, as we want to determine which algorithm works best. However, we do not approach the problem as a classification problem but rather a regression problem. As discussed in Section 2.5.1, we build a regression model that predicts the probabilities of each algorithm. A high probability indicates a well-performing algorithm, while a low probability indicates a bad-performing algorithm. From these probabilities, we can still conclude whether the best algorithm is found by looking at the algorithm with the highest probability. We do this approach because it gives us even more information. It also shows whether one algorithm works exceptionally well or whether some algorithms work equally well. If the last is true, these algorithms can be combined into one large algorithm by the ORTEC experts, leading to better results. In short, turning the problem into a regression problem, instead of a classification problem provides us with more valuable information.

In step 4, an elaborate feature analysis is performed after the model is trained. This analysis shows the most important features when predicting the best working construction algorithm. Also, it is good to find out if any features are redundant, as these can be removed from the model. This leads to training fewer parameters. The feature analysis is done using several methods, to strengthen our conclusions. The three methods are SHAP, partial dependence plots, and permutation feature importance. More on these methods can be found in Section 2.7.

3.1 Data description

The data for our research consists of JSON files containing information on tasks and vehicles from OFS customers from ORTEC. Because we focus on construction, we only consider data in which none of the tasks are already planned: no input data. Otherwise, when we have input data, we already have a starting point, so construction is not needed anymore. We picked seven customers that work without input data. We refer from now on to them as customer A, customer B, ..., customer G. For each of these customers, both the JSON files and their tailored algorithms are used. We refer to the algorithms as AlgA, AlgB, ..., AlgF. We only have six algorithms compared to seven customers, as two customers have identical construction algorithms. More information on the algorithms can be found in Section 3.3. An elaborate data analysis on the evaluation of the data in the various construction algorithms can be found in Section 3.5.

3.2 Input features

In this section, we discuss the input features we have selected for our NN. Selecting the best input features for a machine learning model has been a difficult question. Guo and Hsu [2003] stated that the choice of input features highly depends on expert experiences. Of course, it is also highly dependent on the characteristics of the problem. For instance, the number of vehicles and tasks are two very logical numbers to select, even without having any expert knowledge. More on the problem of feature selection was discussed in Section 2.4.

The selection of input features is based on several sources. We selected input features based on our own merit, literature, and the suggestions of the optimization experts of ORTEC. The literature used are the studies from Mocking [2017], Smith-Miles et al. [2014], and Desrochers et al. [1990]. The complete list of input features can be found below. The number of capabilities refers, in short, to the number of skills the drivers can have.

1. Number of vehicles
2. Number of tasks
3. Number of capabilities
4. Binary: heterogeneous/homogeneous vehicles

5. Number of depots
6. Start of time window (average)
7. Start of time window (standard deviation)
8. Start of time window (skewness)
9. Start of time window (kurtosis)
10. End of time window (average)
11. End of time window (standard deviation)
12. End of time window (skewness)
13. End of time window (kurtosis)
14. Length of time window (average)
15. Length of time window (standard deviation)
16. Length of time window (skewness)
17. Length of time window (kurtosis)
18. Percentage of tasks in which all time windows are identical
19. Task duration (average)
20. Task duration (standard deviation)
21. Task duration (skewness)
22. Task duration (kurtosis)
23. Costs per route
24. Costs per kilometer
25. Costs per task
26. Costs per hour
27. Distance from task to closest 3 vehicles (average)
28. Distance from task to closest 3 vehicles (standard deviation)

3.3 Construction algorithms

A predetermined set of algorithms is used to predict the best construction algorithm. This set needs to be well picked, as it needs to be kept simple, but cover as many possible different algorithms. Eventually, we made a selection of 14 different construction algorithms. These 14 algorithms consist of the algorithms AlgA, $\cdots$, AlgF mentioned in Section 3.1, straight-forward algorithms, and some algorithms proposed by experts of ORTEC.

1. AlgA
2. AlgB
3. AlgC
4. AlgD
5. AlgE
6. AlgF
7. Farthest to depot (parallel)
8. Farthest to depot (sequential)
9. Nearest to depot (parallel)
10. Nearest to depot (sequential)
11. Parallel cheapest insertion
12. Regret insertion
13. Smallest time window (parallel)
14. Smallest time window (sequential)

Algorithms 1-6 are customer algorithms, and so confidential. An explanation of the construction algorithms 7-14 can be found below.

Farthest to depot
In the algorithms called ‘Farthest to depot’, first a seed task is planned, followed by the rest of the tasks. The selected seed task is the one that is located farthest from the depot. After the seed task is planned, the rest of the tasks are planned. The task that is closest to the current route, is selected and inserted. In the sequential version, only one seed task is selected, and the rest

of the tasks are added for that particular route until it is full. After the route is completed, the process repeats, and the next route is built. For the parallel algorithm, all seed tasks are selected simultaneously. Thereafter, the route with the shortest driving time is selected, and on that route, a new task is inserted. Inserting this task causes the drive time to go up. To plan the next task, the route with the shortest driving time is selected again. This can be another or the same route as previously selected. This is one of the ways in which parallel insertion can be performed, and it is used in practice, as it takes less time to be performed than parallel cheapest insertion.

Nearest to depot
The algorithms called ‘Nearest to depot’ are similar to the algorithms ‘Farthest to depot’, except for the selection of seed tasks. In ‘Nearest to depot’ the task that is nearest to the depot is selected as seed task. The other parts of the algorithms are identical to their ‘Farthest to depot’ counterparts.

Parallel cheapest insertion
In the algorithm ‘Parallel cheapest insertion’ all tasks are considered for all vehicles each time. The combination of the task and the vehicle that has the least amount of driving time is selected, and the task is planned. Then, the process repeats until no more tasks are feasible.

Regret insertion
In the algorithm ‘Regret insertion’ all tasks are considered for all vehicles each time. The task with the highest regret value, see Section 2.3, is planned on its best vehicle. After this, the process repeats until no more tasks are feasible.

Smallest time window
The algorithms called ‘Smallest time window’ are similar to the algorithm ‘Farthest to depot’ except for the selection of seed tasks. In ‘Smallest time window’, the task with the smallest time window is selected as the seed task. The other parts of the algorithms are identical to their ‘Farthest to depot’ counterparts.

At the end of each of the algorithms 1-14, after the construction, a local search and parallel cheapest insertion (as described above) is performed. This local search and parallel cheapest insertion are identical for all algorithms and try to reorder or add new tasks to the routes. However, they do not remove any tasks or build any new routes. Adding this local search and parallel cheapest insertion contributes to finding the best construction algorithm because of the following reason. Solutions given by some construction algorithms can easily be improved by local search, while other algorithms might have a solution that is mainly set. Therefore, checking if the solution can be easily improved, which makes room for new tasks to be planned, is something that needs to be considered when appointing which construction algorithm works best.

The downside of adding local search is that it can take some runtime. Therefore, we incorporated a fast local search, which only includes 2-OPT. In the algorithms of ORTEC, next to local search, also ruin & recreate is used. Despite ruin & recreate improving a solution, we do not include it in our research, as it takes up a lot of time. Furthermore, finding a good construction algorithm without ruin & recreate is still useful, as in practice, a good initial solution diminishes the runtime of ruin & recreate. Furthermore, in ruin & recreate, only 5 or 10 percent of the nodes is deleted. Therefore, a lot of the initial solution stays intact,so it is important to find a good-performing construction algorithm.

3.4 Labeling data

To train the NN, the data needs to be labeled. The label of each data point needs to represent how well the algorithm has performed compared to the other algorithms. Furthermore, different data points from different customers also need to be comparable. This leads us to label the data using probabilities. We use the softmax function, as described in Section 2.5.3, which assigns a high probability to an algorithm that has performed relatively well, while it assigns a low probability to an algorithm that performs relatively badly. In this way, we have a natural order of the algorithms. Moreover, we can appoint which algorithm works the best and how much better it performs than the other algorithms.

To assign these probabilities, the data point is evaluated in every algorithm. Determining how well the algorithm has performed, is done by assigning a number to each algorithm. The details can be found in the next paragraph. After this, these numbers are mapped to probabilities using the softmax function. Repeating this process for every data point, we label all training and test data. Using these probabilities allows for finding the best working algorithm and offers insight into how well different algorithms work. This is exactly the goal, as the prediction of the model is only a starting point for the experts of ORTEC, which might want to combine the concepts of several well-performing algorithms. An overview of the labeling steps is shown in the example in Figure 3.1.

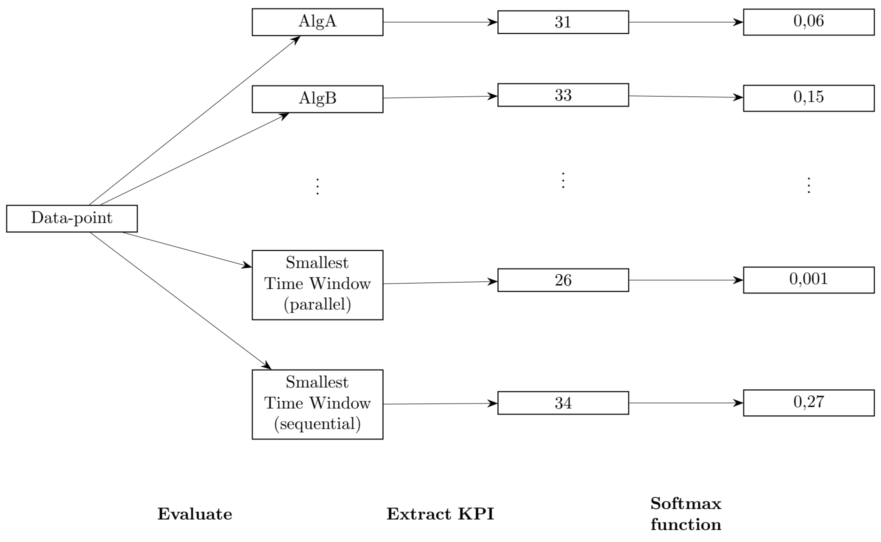
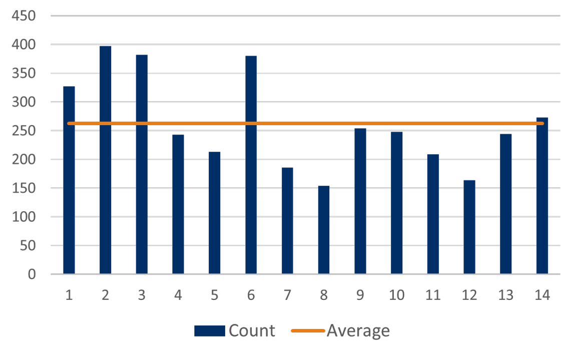


Figure 3.1: Process of labeling data

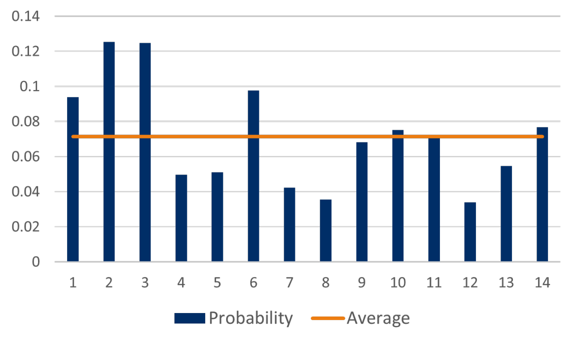
To determine the best-working algorithm, ORTEC looks at Key Points of Interest (KPIs). These KPIs are important features that are minimized or maximized during the optimization. Examples of KPIs are driving time and costs. The most important KPI in general, is the number of planned tasks. Most of the time, there are too many tasks to be able to plan them all. Therefore, to maximize profit and to provide customer satisfaction, as many tasks as possible need to be planned. Therefore, we base our labels on this KPI.

3.5 Data analysis

Before training and evaluating NNs, a data analysis is required. We do this by looking at the results after labeling the data. We consider how often an algorithm performs the best and how well an algorithm performs on average. The results per algorithm are shown in Figure 3.2a and Figure 3.2b, respectively. In both cases, algorithms 1, 2, 3, and 6 perform much better than average, while algorithms 4, 5, 7, 8, 12, and 13 perform quite much worse than average.



(a) Number of times a construction algorithm performs best



(b) Average label for each construction algorithm

Figure 3.2: Performance for all data evaluated in every construction algorithm

3.6 Balancing data

If we take a look at the data analysis in the previous section, we conclude that the data set is quite imbalanced. This could cause problems for the performance of the model, as this imbalanced data can cause overfitting. In Section 2.6, we discuss the problem of imbalanced data and several ways to solve this. Two of those methods are called downsampling and oversampling, which we also discuss in more detail in Section 2.6.1 and Section 2.6.2, respectively. We can formulate downsampling and oversampling as an Integer Linear Program (ILP) to balance the data. In this section, we derive the ILP formulation for both downsampling and oversampling the data.

3.6.1 Downsampling

The problem of downsampling can be formulated as follows. We first describe it in a format following directly from the problem. Then, we rewrite the constraints in a linear form. We introduce set I containing all the data points, set J containing all algorithms, and the parameters p_{ij}, which represent the labeled probability of data point $i \in I$ for algorithm $j \in J$. The goal of balancing the data is to have the average performance of each algorithm $j \in J$ approximately equal. We do this by including or excluding each data point $i \in I$ in the data set. We define binary variable $X_i \in \{0,1\}$ for each data point $i \in I$ to represent whether or not the data point is included in the data set. We define:

$$X_i = \begin{cases} 1, & \text{if data point } i \text{ is included} \\ 0, & \text{otherwise} \end{cases}$$

We then compute the average probability Y_j for each algorithm $j \in J$ in the following way:

$$Y_j = \frac{\sum\limits_{i \in I} p_{ij} X_i}{\sum\limits_{i \in I} X_i}, \quad \forall j \in J.$$

This average probability must be approximately equal for all algorithms $j \in J$. Therefore, the difference between the maximum Y_j and the minimum Y_j needs to be minimal. This leads to the following objective function and constraints:

$$\min \left(Y_{max} - Y_{min}\right)$$

$$Y_{max} \geq Y_j, \quad \forall j \in J$$
$$Y_{min} \leq Y_j, \quad \forall j \in J$$

Because the objective function forces Y_{max} to be as small as possible, while the constraint forces Y_{max} to be bigger than all Y_j, it will exactly take on the maximum value over all Y_j. The same logic applies to Y_{min}. This leads to the following IP:

$$\min \left(Y_{max} - Y_{min}\right) \tag{3.1}$$
$$Y_j = \frac{\sum\limits_{i \in I} p_{ij} X_i}{\sum\limits_{i \in I} X_i}, \quad \forall j \in J \tag{3.2}$$
$$Y_{max} \geq Y_j, \quad \forall j \in J \tag{3.3}$$
$$Y_{min} \leq Y_j, \quad \forall j \in J \tag{3.4}$$
$$X_i \in \{0,1\}, \quad \forall i \in I \tag{3.5}$$

This formulation is not yet an ILP, as Constraints 3.2 are not linear. When we rewrite Constraints 3.2 to $\sum_{i \in I} Y_j X_i = \sum_{i \in I} p_{ij} X_i$, it contains a product of binary variable X_i and nonnegative variable Y_j. Now, we can rewrite the constraints in a linear form by defining $Z_{ij} = Y_j X_i$. Replacing this in our problem leads to the following ILP formulation:

$$\min\left(Y_{max} - Y_{min}\right) \tag{3.6}$$
$$\sum_{i\in I} Z_{ij} = \sum_{i\in I} p_{ij}X_i, \quad \forall j \in J \tag{3.7}$$
$$Z_{ij} \leq X_i, \quad \forall i \in I, j \in J \tag{3.8}$$
$$Z_{ij} \leq Y_j, \quad \forall i \in I, j \in J \tag{3.9}$$
$$Z_{ij} \geq Y_j - (1 - X_i), \quad \forall i \in I, j \in J \tag{3.10}$$
$$Y_{max} \geq Y_j, \quad \forall j \in J \tag{3.11}$$
$$Y_{min} \leq Y_j, \quad \forall j \in J \tag{3.12}$$
$$Z_{ij} \geq 0, \quad \forall i \in I, j \in J \tag{3.13}$$
$$X_i \in \{0,1\}, \quad \forall i \in I \tag{3.14}$$

Although our problem is now correctly formulated as an ILP, one type of constraint needs to be added. Our data comes from different companies. We do not want the ILP to be able to exclude all the data points from one specific company. To prevent this, we set a minimum threshold for the number of data points included for each company. We define C as the set of all companies, and I_c is the set of data points from company $c \in C$. Furthermore, we define D_c as the minimum number of data points from company $c \in C$ that must be included. This leads to the following extra constraints:

$$\sum_{i\in I_c} X_i \geq D_c, \quad \forall c \in C. \tag{3.15}$$

3.6.2 Oversampling

The problem of oversampling can be formulated as follows. We first describe it in a format following directly from the problem. Then, we rewrite it in a linear form. We again have set I containing all the data points, set J containing all algorithms, and the parameters p_{ij}, which represent the labeled probability of data point $i \in I$ for algorithm $j \in J$. The goal of balancing the data is to have the average performance of each algorithm $j \in J$ approximately equal. We do this by including data points $i \in I$ several times in the data set. We define the positive integer $X_i \in \mathbb{N}_{>0}$ for each data point $i \in I$ to represent the number of times the data point i is included. This means we do not exclude any data points. Then, just like in the previous section, we again compute this average probability Y_j for each algorithm $j \in J$ and define Y_{max} and Y_{min}. This leads to the following IP:

$$\min\left(Y_{max} - Y_{min}\right) \tag{3.16}$$
$$Y_j = \frac{\sum\limits_{i\in I} p_{ij}X_i}{\sum\limits_{i\in I} X_i}, \quad \forall j \in J \tag{3.17}$$
$$Y_{max} \geq Y_j, \quad \forall j \in J \tag{3.18}$$
$$Y_{min} \leq Y_j, \quad \forall j \in J \tag{3.19}$$
$$X_i \in \mathbb{N}_{>0}, \quad \forall i \in I \tag{3.20}$$

We add one extra constraint to this IP. We want each data point to be included in the data set at least once, which is guaranteed by $X_i \in \mathbb{N}_{>0}$. But, we also want to avoid symmetric solutions, and therefore, at least one data point should be included exactly once. Therefore, this last constraint is:

$$\min_{i\in I} X_i = 1. \tag{3.21}$$

This formulation is not yet an ILP, as Constraints 3.17 and Constraint 3.21 are not linear. We can make Constraints 3.17 linear, as it contains a product between an integer variable and a non-negative continuous variable. We first rewrite the integer variables X_i to a binary representation using the binary variables W_{ik}: $X_i = \sum_{k=0}^{6} 2^k W_{ik}$, for $W_{ik} \in \{0,1\}$. This leads to an upper bound of 126 on how often a data-point can be included in the training set. The updated IP is as follows:

$$\min\left(Y_{max} - Y_{min}\right) \tag{3.22}$$

$$\sum_{i\in I} Y_j \left(\sum_{k=0}^{6} 2^k W_{ik}\right) = \sum_{i\in I} p_{ij} \left(\sum_{k=0}^{6} 2^k W_{ik}\right), \quad \forall j \in J \tag{3.23}$$

$$\min_{i\in I} \left(\sum_{k=0}^{6} 2^k W_{ik}\right) = 1 \tag{3.24}$$

$$Y_{max} \geq Y_j, \quad \forall j \in J \tag{3.25}$$

$$Y_{min} \leq Y_j, \quad \forall j \in J \tag{3.26}$$

$$W_{ik} \in \{0,1\}, \quad \forall i \in I, k = 0, \cdots, 6 \tag{3.27}$$

Constraints 3.23 and Constraint 3.24 are not linear. We first focus on Constraints 3.23. Constraints 3.23 contain a product of a binary variable W_{ik} and a non-negative variable Y_j. We again define $Z_{ijk} = Y_j W_{ik}$. Now, we can rewrite Constraints 3.23 in a linear form, as follows:

$$\sum_{i\in I} \sum_{k=0}^{6} 2^k Z_{ijk} = \sum_{i\in I} p_{ij} \sum_{k=0}^{6} 2^k W_{ik}, \quad \forall j \in J \tag{3.28}$$

$$Z_{ijk} \leq W_{ik}, \quad \forall i \in I, j \in J, k = 0, \cdots, 6 \tag{3.29}$$

$$Z_{ijk} \leq Y_j, \quad \forall i \in I, j \in J, k = 0, \cdots, 6 \tag{3.30}$$

$$Z_{ijk} \geq Y_j - (1 - W_{ik}), \quad \forall i \in I, j \in J, k = 0, \cdots, 6 \tag{3.31}$$

$$Z_{ijk} \geq 0, \quad \forall i \in I, j \in J, k = 0, \cdots, 6 \tag{3.32}$$

Constraint 3.24 fixes at least one t to be included only once in the data set. We define binary variables V_i, which are equal to one, when $\sum_{k=0}^{6} 2^k W_{ik}$ is equal to one, and zero otherwise. Now, we can rewrite Constraint 3.24 in a linear form, as follows:

$$\sum_{k=0}^{6} 2^k W_{ik} \leq 1 + 125(1 - V_i), \quad \forall i \in I \tag{3.33}$$

$$\sum_{k=0}^{6} 2^k W_{ik} \geq 1, \quad \forall i \in I \tag{3.34}$$

$$\sum_{i\in I} V_i \geq 1 \tag{3.35}$$

$$V_i \in \{0,1\}, \quad \forall i \in I \tag{3.36}$$

Constraints 3.33 put an upper bound on the summation, so how many times a data point can be included in the set is restricted. If V_i is equal to zero, this upper bound is 126, which is also the maximum number the sum can obtain by construction. And so, this upper bound is in line with the problem.

This leads to the final ILP formulation:

$$\min\left(Y_{max} - Y_{min}\right) \tag{3.37}$$

$$\sum_{i\in I}\sum_{k=0}^{6} 2^k Z_{ijk} = \sum_{i\in I} p_{ij} \sum_{k=0}^{6} 2^k W_{ik}, \quad \forall j \in J \tag{3.38}$$

$$Z_{ijk} \leq W_{ik}, \quad \forall i \in I, j \in J, k = 0, \cdots, 6 \tag{3.39}$$

$$Z_{ijk} \leq Y_j, \quad \forall i \in I, j \in J, k = 0, \cdots, 6 \tag{3.40}$$

$$Z_{ijk} \geq Y_j - (1 - W_{ik}), \quad \forall i \in I, j \in J, k = 0, \cdots, 6 \tag{3.41}$$

$$\sum_{k=0}^{6} 2^k W_{ik} \geq 1, \quad \forall i \in I \tag{3.42}$$

$$\sum_{k=0}^{6} 2^k W_{ik} \leq 1 + 125(1 - V_i), \quad \forall i \in I \tag{3.43}$$

$$\sum_{i\in I} V_i \geq 1 \tag{3.44}$$

$$Y_{max} \geq Y_j, \quad \forall j \in J \tag{3.45}$$

$$Y_{min} \leq Y_j, \quad \forall j \in J \tag{3.46}$$

$$V_i \in \{0, 1\}, \quad \forall i \in I \tag{3.47}$$

$$W_{ik} \in \{0, 1\}, \quad \forall i \in I, k = 0, \ldots, 6 \tag{3.48}$$

$$Z_{ijk} \geq 0, \quad \forall i \in I, j \in J, k = 0, \ldots, 6 \tag{3.49}$$

3.7 Loss, accuracy, and grid search

When building a NN, there are many choices to be made. To find the best-performing model, we try different settings regarding hidden layers, the width of the layers, and the type of activation function. We do this using grid search: for each parameter, we determine a set of values we want to investigate, and we train and evaluate a model using every combination of parameter values. This leads to many different models, from which we want to pick the best. What is considered the best model can be investigated using different measures.

First of all, the test loss can be considered. When training the model, a random part of the data is left out of the training: the test data. After the model is trained, this test data is evaluated in the model, and the loss is determined. Both in training and testing the model, we use the mean squared error loss. This shows how well the test data is predicted in terms of how close the predicted probabilities are to the correct labels.

However, the loss does not incorporate all information. We are also interested in whether the model can predict the best-performing algorithm. Correctly determining what the top-performing algorithms are, is more important than determining the order of the worst-performing algorithms correctly. A method that helps to evaluate this, is accuracy. We calculate how many times the model predicts the best working algorithm correctly. Representing this as a percentage is known as the accuracy of a model. We determine both the accuracy using all data points and using the data points per algorithm. The data points can be divided into two groups: data points in which several algorithms perform equally best, and data points in which one algorithm outperforms the others. For the accuracy, we look at all data points and whether the model can predict one of the best-performing algorithms correctly. When we look at the accuracy per algorithm, we only consider the data points with one uniquely best-performing algorithm. Grouping them per algorithm and determining the accuracy per group gives extra information about how well each algorithm can be identified.

Chapter 4

Results

In this chapter, we discuss the results of the several investigated models. We start in Section 4.1 with an introduction to our research and the establishment of a baseline. This is followed by the results and feature analysis of our original model in Section 4.2. We try to improve this model using different methods. The first method is by balancing the data. We discuss the results of the models built using downsampled and oversampled data in Section 4.3 and Section 4.4, respectively. The second method is improving the choice of input features. We add more input features to the model in Section 4.6. In Section 4.7, we build smaller models, based on the feature analysis of previous models. We conclude this chapter with Section 4.8, in which we build models for each algorithm individually.

4.1 Introduction

In this introduction, we explain the different performance measures, how we deal with randomness, and the baseline for our model. In this chapter, we built various models using grid search for the activation function, the hidden layers, and the width of the layers. We train each model until the stopping criterion is met. We use the method early stopping, as explained in Section 2.5.3. After training, we evaluate the performance of each model on the test data using mean squared error loss, accuracy, and accuracy per algorithm, as described in Section 3.7.

Furthermore, in our research, 4-fold cross-validation is used when training and evaluating the model. This means the data is split into four evenly distributed groups. Each time, one of the groups is considered as test data, while the other three groups are used as training data. To make the correct conclusion under which settings the best model is trained, we must consider all combinations of training and test data. The results over the different cross-validations are averaged. Also, when training the model, the data is shuffled. This is controlled by a random seed, allowing to reproduce results. To minimize the influence of the choice of seed, each model setting is re-evaluated using three different seeds. Averaging the results from all cross-validations and all random seeds leads to the results portrayed in this chapter.

We want to determine whether the loss and accuracy results in this chapter are good. Therefore, we establish a baseline by building a simple model: always predict one specific algorithm to be the best with a probability of 1. We want to design a model that can differentiate between the different algorithms, so the model should achieve a lower loss and a higher accuracy than the simple models. We build 14 different baseline models, each model saying that its specific algorithm is always the best with a probability of 1. The loss and accuracy results for the different models are shown in

Table 4.1. This table shows, for instance, that if algorithm 1 is always predicted to be the best algorithm, the model accuracy is 45%. The lowest loss, and therefore the loss baseline, is 0.28. The highest accuracy, and therefore the accuracy baseline, is 54.7%. Note, the percentages in Table 4.1 do not add up to 100% because we have multi-labeled data: for some data points multiple algorithms perform equally well. The model which achieves the highest accuracy also gives the lowest loss, which is 0.28. This is, therefore, the baseline of the loss.

Table 4.1: Loss and accuracy baseline

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Loss	0.29	0.28	0.28	0.30	0.30	0.29	0.30	0.30	0.29	0.29	0.29	0.30	0.30	0.29
Accuracy	45.0	54.7	52.6	33.5	29.3	52.3	25.6	21.2	35.0	34.2	28.8	22.6	33.6	37.6

4.2 Original model

In this section, we determine the best settings for our original model with a grid search in Section 4.2.1. After that, we perform an elaborate feature analysis on the best-performing model in Section 4.2.2, followed by an analysis of the improvement of the KPIs in Section 4.2.3. In Section 4.2.4, an overview on how to improve the model based on the results is included.

4.2.1 Grid search

For our first model, we use all the data provided to us by ORTEC and labeled as discussed in Section 3.4. We start with a large grid search in which we consider two different activation functions: softmax and SELU. We build models with 2, 5, and 10 hidden layers and 20, 40, and 60 nodes per layer. This leads to the results in Table 4.2. For 7 out of 9 settings the softmax function performs better than the SELU function in terms of accuracy. The other 2 settings do not yield very accurate models. The loss, however, is slightly lower for the SELU function. But, the accuracy indicates whether the few best-performing models are ordered correctly, while the loss focuses on ordering all algorithms correctly. Of course, we are more interested in getting the best algorithms right. Therefore, we conclude that the softmax function performs better than the SELU function, and so, from now on, we use the softmax function.

The accuracy and loss performance indicate different settings to lead to a best-performing algorithm. In terms of accuracy, 5 hidden layers and a width of 60 is ideal, while in terms of loss, 2 hidden layers with a width of 20 is best. To gain more insight into the models, we look at the accuracy per algorithm, shown in Table 4.3. We conclude that the softmax function, 5 hidden layers, and width 40 gives the most accurate model on average for uniquely best algorithms. Furthermore, the model with 2 hidden layers and a width of 20 contains 0s for five algorithms, which means that none of the data points are predicted correctly. This is evidence that the model overfits. For 5 hidden layers and a width of 40, only one algorithm has a percentage of 0. Therefore, we favor this last setting.

Table 4.2: Loss and accuracy results for large grid search

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	20	0.135	57.6
		40	0.141	56.1
		60	0.142	57.3
	5	20	0.147	55.5
		40	0.149	56.7
		60	0.149	58.3
	10	20	0.150	53.8
		40	0.151	57.3
		60	0.145	55.2
SELU	2	20	0.131	57.3
		40	0.135	56.7
		60	0.136	56.2
	5	20	0.148	54.0
		40	0.146	55.8
		60	0.142	57.7
	10	20	0.145	53.1
		40	0.144	54.0
		60	0.143	56.4

Table 4.3: Accuracy results for large grid search per algorithm for unique labeled data points

Activation function	HL	Width	1	2	3	4	5	6	7	8	9	10	11	12	13	14	average
Softmax	2	20	8.9	77.0	60.4	0	3.5	39.4	0	0	13.3	45.8	11.0	0	0	12.4	19.4
		40	4.6	73.5	60.1	0	8.2	30.5	8.3	0	12.2	37.6	29.9	0	0	13.0	19.9
		60	5.0	72.9	60.6	2.8	0	24.6	11.1	4.2	18.6	42.2	28.8	0	0	18.5	20.7
	5	20	6.9	67.4	44.1	0	14.6	28.4	2.8	4.2	11.6	30.9	25.9	5.6	0	6.9	17.8
		40	7.8	64.0	60.1	8.3	2.1	27.4	5.6	11.1	18.1	39.5	33.2	16.7	0	12.0	21.8
		60	10.0	72.5	54.7	2.8	14.6	33.4	2.8	0	21.5	34.4	30.4	3.7	0	18.9	21.4
	10	20	3.8	61.3	45.5	0	7.6	21.3	4.2	15.3	18.2	28.1	12.4	16.7	0	19.2	18.1
		40	14.5	68.1	47.8	5.6	9.0	34.0	2.8	0	16.0	26.1	27.6	11.1	0	13.8	19.7
		60	3.0	53.1	43.4	0	8.9	26.7	11.1	6.9	14.9	23.7	15.0	0	0	3.5	15.0
SELU	2	20	7.9	82.8	73.1	0	21.5	23.4	0	4.2	7.3	38.8	24.1	0	0	9.9	20.9
		40	9.1	83.1	59.2	6.9	13.2	26.7	2.8	0	14.0	33.5	22.0	0	0	9.9	20.0
		60	5.2	78.9	64.4	2.8	3.5	16.8	18.1	0	20.0	30.1	20.0	3.7	0	6.9	19.3
	5	20	13.5	81.0	54.0	0	12.5	14.1	12.5	0	7.6	46.9	18.5	0	0	9.9	19.3
		40	9.7	72.1	64.5	0	4.2	16.1	8.3	8.3	18.1	32.1	30.0	0	0	5.8	19.2
		60	11.5	81.0	67.0	0	10	21.9	2.8	0	20.6	45.8	33.5	0	0	7.2	21.5
	10	20	14.1	88.0	41.7	0	8.3	21.4	8.3	4.2	14.0	23.3	31.0	0	0	7.6	18.7
		40	11.4	77.3	55.1	0	5.6	18.7	0	0	21.5	24.2	22.4	0	0	10.2	17.6
		60	15.1	70.2	59.0	0	4.2	26.2	0	0	19.6	24.9	26.6	0	0	7.2	18.1
average			9.0	73.6	56.4	1.6	8.4	25.1	5.6	3.3	16.0	33.8	24.6	3.2	0	10.7	

Considering all results, we conclude that a model with the softmax function, 5 hidden layers, and a width of 40 or 60 gives the best results. We perform a small grid search in this region of numbers: we build new models with 4, 5, and 6 hidden layers and a width of 30, 35, $\cdots$, 70. Their performance is shown in Table 4.4 and Table 4.5.

Table 4.4: Loss and accuracy results for small grid search

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	4	30	0.147	57.1
		35	0.147	58.6
		40	0.149	57.7
		45	0.150	57.5
		50	0.150	58.0
		55	0.151	57.9
		60	0.151	57.8
		65	0.150	58.5
		70	0.151	57.3
	5	30	0.150	56.9
		35	0.149	56.7
		40	0.149	56.7
		45	0.151	56.6
		50	0.151	56.3
		55	0.154	56.6
		60	0.149	58.3
		65	0.150	59.3
		70	0.151	57.3
	6	30	0.151	56.7
		35	0.152	56.5
		40	0.155	57.2
		45	0.151	56.7
		50	0.152	59.0
		55	0.153	57.2
		60	0.149	57.5
		65	0.150	58.7
		70	0.152	56.5

When we look at the loss and accuracy in Figure 4.4, there are a few interesting findings. First, 6 hidden layers is outperformed by 4 or 5 hidden layers, for almost every setting, for both loss and accuracy. Therefore, 6 hidden layers does not yield the best-performing model. Secondly, two settings might lead to the best model. The first is 5 hidden layers and a width of 65, as it has the best accuracy of 59.3%. The other setting is 4 hidden layers and a width of 35. This model has the lowest loss of 0.147 and a high accuracy of 58.6%. In Table 4.5, the model with 4 hidden layers and a width of 35 performs best by far in terms of accuracy. This leads us to believe that 4 hidden layers and a width of 35 is the best-performing model, as it performs well both in terms of loss and accuracy. From now on, we refer to this model as our original model.

If we compare our original model to the baseline established in Section 4.1, we conclude the following. First, the loss has almost halved in comparison to the baseline loss. This shows that our original model predicts probabilities much closer to the labeled probabilities than the hot-one encoded vectors in the baseline models. Furthermore, the accuracy is also 3.9% better than the accuracy baseline. Concluding, our model improves the baseline.

Table 4.5: Accuracy results for small grid search per algorithm for unique labeled data points

Activation function	HL	Width	1	2	3	4	5	6	7	8	9	10	11	12	13	14	average
Softmax	4	35	9.6	75.4	60.1	0	19.0	30.9	11.1	11.1	15.4	46.5	33.8	0	0	11.0	23.2
		40	8.9	63.9	63.0	0	9.0	35.9	2.8	8.3	17.5	46.5	33.4	5.6	0	8.5	21.7
		45	5.8	68.8	51.3	4.2	16.7	36.6	8.3	12.5	23.1	43.3	26.7	0	0	8.5	21.8
		50	10.7	74.9	53.9	0	12.5	30.5	11.1	23.6	9.0	34.4	27.1	0	0	8.5	21.2
		55	8.3	67.8	50.4	0	0	29.7	0	9.7	17.2	33.6	39.9	11.1	0	14.3	20.1
		60	6.7	72.6	57.4	0	4.2	28.4	11.1	0	16.3	46.5	34.1	0	0	11.0	20.6
		65	11.6	66.5	56.3	0	1.4	24.5	8.3	4.2	15.7	49.1	43.2	0	0	11.9	20.9
		70	7.4	73.1	57.7	0	3.5	30.6	11.1	12.5	25.2	28.8	31.3	0	0	7.9	20.7
	5	35	10.3	70.2	50.5	0	3.5	32.1	2.8	2.8	11.1	28.1	28.5	0	0	14.4	18.2
		40	7.8	64.0	60.1	8.3	2.1	27.4	5.6	11.1	18.1	39.5	33.2	16.7	0	12.0	21.8
		45	8.3	68.7	48.9	0	9.0	36.7	6.9	16.7	15.8	33.1	24.8	0	0	11.0	20.0
		50	5.3	67.4	48.9	6.9	7.9	30.5	2.8	12.5	24.3	29.5	32.7	0	0	14.3	20.2
		55	6.9	60.3	56.2	6.9	4.2	25.0	15.3	12.5	7.8	25.8	33.1	5.6	0	14.0	19.5
		60	10.0	72.5	54.7	2.8	14.6	33.4	2.8	0	21.5	34.4	30.4	3.7	0	18.9	21.4
		65	8.5	75.9	54.5	0	16.7	36.2	0	20.8	7.1	36.5	46.4	0	0	11.6	22.4
		70	7.1	68.4	56.2	4.2	8.2	34.6	13.9	8.3	21.8	39.0	33.9	5.6	0	11.3	22.3
	6	35	11.7	67.0	48.3	4.2	4.7	35.0	6.9	0	19.6	35.8	25.0	0	0	2.7	18.6
		40	10.6	62.8	55.1	4.2	0	26.6	8.3	4.2	20.8	29.4	33.2	0	0	15.9	19.4
		45	11.1	70.1	40.1	11.1	8.3	24.3	15.3	8.3	17.5	37.3	22.8	0	0	16.8	20.2
		50	13.3	61.3	55.7	2.8	5.6	37.4	5.6	0	14.0	42.6	40.8	0	0	16.7	21.1
		55	5.0	65.5	52.6	0	2.8	33.4	0	8.3	21.2	31.2	18.1	0	0	15.4	18.1
		60	6.3	62.4	50.2	0	15.3	29.8	4.2	8.3	21.4	39.5	41.8	3.7	0	15.8	21.3
		65	5.0	65.9	55.7	2.8	10.4	27.4	9.7	12.5	15.1	43.5	35.0	0	0	12.4	21.1
		70	5.3	65.8	52.2	0	6.9	36.8	12.5	12.5	8.3	40.1	27.9	0	0	9.3	19.8
average			8.4	68.0	53.8	2.4	7.8	31.4	7.4	9.2	16.9	37.3	32.4	2.2	0	12.3	

4.2.2 Feature analysis

We perform an elaborate feature analysis on our original model of 4 hidden layers and a width of 35. We analyze the model using three different feature analysis methods: permutation feature importance, SHAP, and partial dependence plots, as described in Section 2.7.

First, permutation feature importance is considered, as it immediately gives a general overview of the important and unimportant features, shown in Figure 4.1. The higher the bar, the more important the feature is. As can be seen in Figure 4.1, the feature ‘Start of time window (standard deviation)’ is the most important input feature. Furthermore, the features ‘Heterogeneous vehicles’, ‘Costs per kilometer’, ‘Costs per task’, and ‘Costs per hour’ do not have any bar because these features have the same value for every data point. Therefore, these features are redundant. These features could be of influence if we would acquire data in which their value does vary. Based on the results in Figure 4.1, we make up the following list of the 5 most important input features:

1. Start of time window (standard deviation)
2. End of time window (standard deviation)
3. Number of vehicles
4. Start of time window (average)
5. Length of time window (standard deviation)

Unfortunately, we cannot conclude yet that these are indeed the 5 most important features in the model. Permutation feature importance is not foolproof, so for such a list more evidence is needed. This evidence is provided by other feature analysis methods. For the second method, we look at the overall SHAP values, shown in Figure 4.2. The input features are ordered from having the most influence to having the smallest influence over all algorithms. The features ‘Start of time window (standard deviation)’, ‘End of time window (standard deviation)’, and ‘Length of time window (standard deviation)’ are included in the top 5, again. Furthermore, there are no bars for the features ‘Heterogeneous vehicles’, ‘Costs per kilometer’, ‘Costs per task’, and ‘Costs per hour’, as they are redundant. We construct a list of the 5 most important features based on the SHAP values:

1. Length of time window (standard deviation)
2. Number of depots
3. Start of time window (standard deviation)
4. End of time window (standard deviation)
5. Length of time window (average)

There is some overlap between the list from the permutation feature importance and SHAP. As previously mentioned, the three time window features for standard deviation are included in both lists. Furthermore, the two remaining features on the permutation feature importance list, ‘Number of Vehicles’ and ‘Start of time window (average)’, are in places 9 and 19 for the SHAP feature analysis. This is the reason we use several methods, as we now know more surely that 4 of the 5 input features of permutation feature importance are indeed important, while ‘Number of vehicles’ might not be. The two remaining input features of SHAP, ‘Number of depots’ and ‘Length of time window (average)’, have high bars for the permutation feature importance and are just outside the top 5. Therefore, those two features are probably also important for the model.

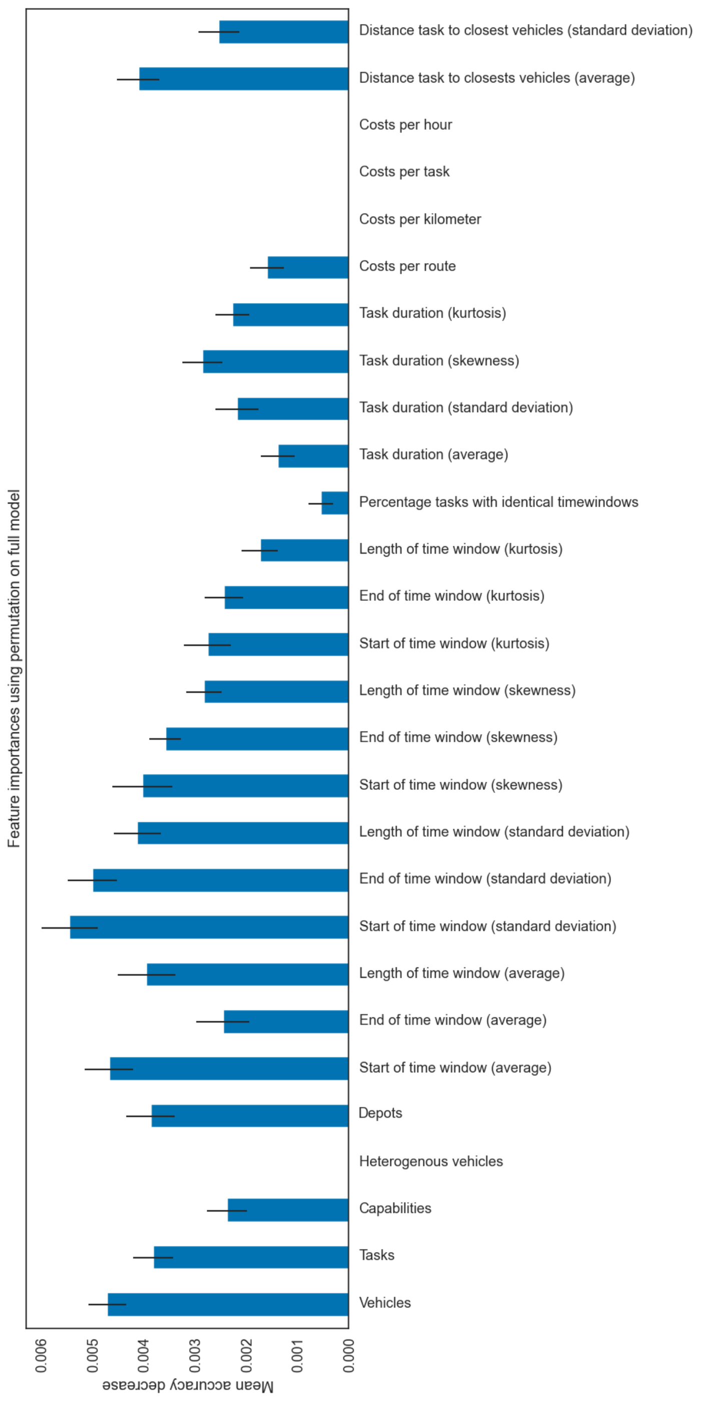


Figure 4.1: Permutation feature importance for the original model

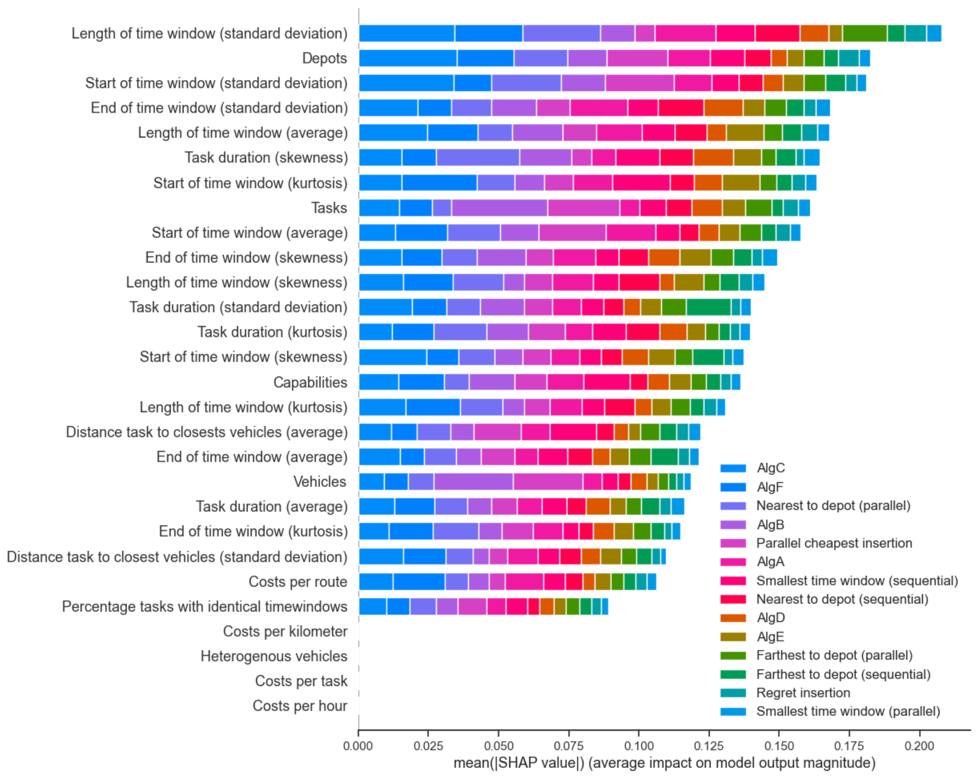


Figure 4.2: SHAP values for every algorithm

As the last method, we consider partial dependence plots, which show the relation between the value of a feature and the prediction of one specific algorithm. Per feature and algorithm, a plot is made, resulting in 392 different plots. For a better overview, we do not compute all the plots but the Greenwell number, as discussed in Section 2.7.2. The Greenwell number shows how much the output changes for different input feature values and so how influential a feature is. The Greenwell number for all features and all algorithms is shown in Table 4.6. We again see an odd result for 'Heterogenous vehicles', 'Costs per kilometer', 'Costs per task', and 'Costs per hour': all Greenwell values are equal to zero. This means that these input features have absolutely no influence on the outcome, so they are redundant. Furthermore, the algorithms 'Farthest to depot (sequential)' and 'Shortest time window (parallel)' have a low total Greenwell number of 13.1 and 13.7, respectively. For these algorithms, each input feature has a low Greenwell number, and therefore, varying these input features has almost no influence on the output of the model. This leads to these algorithms not being trained correctly. This could be due to the lack of data, or due to the input features not being representative of the problem.

We again design a top 5 of the most important input features based on the Greenwell numbers. We do this by finding the 5 input features with the highest Greenwell number. This top 5 is quite different than the other two, caused by the fact that we now consider the influence per algorithm, while for the other two methods, we looked at the overall influence. Computing the Greenwell numbers leads to the following list:

1. Percentage tasks with identical time windows

2. Number of vehicles
3. Number of depots
4. Costs per route
5. Length of time window (kurtosis)

We now construct a final top 5, considering all methods. None of the input features are included in all three lists. There are precisely 5 input features that are included in two of the lists, so we conclude these are important. This leads to the following final list:

1. Start of time window (standard deviation)
2. End of time window (standard deviation)
3. Length of time window (standard deviation)
4. Number of depots
5. Number of vehicles

Table 4.6: Greenwell numbers for all features and algorithms

(a) $*10^-3$

	AlgA	AlgB	AlgC	AlgD	AlgE	AlgF	FTD (par)	FTD (seq)	NTD (par)	NTD (seq)	PCI	RI	STW (par)	STW (seq)
Number of vehicles	2.4	1.1	1.8	5.9	1.4	5.3	1.5	0.7	0.2	1.3	3.5	0.6	1.7	1.8
Number of tasks	0.7	3.7	1.4	0.8	0.6	1.0	0.9	0.2	0.2	1.1	2.6	0.7	1.1	1.2
Number of capabilities	1.1	3.3	1.8	0.2	0.4	1.9	0.8	0.8	0.4	0.5	0.8	0.5	0.2	2.5
Heterogeneous vehicles	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Number of depots	2.4	0.6	5.4	0.5	0.3	3.1	1.3	0.7	3.8	1.7	4.5	1.3	1.5	3.7
Start time window (Av)	2.1	0.3	0.3	0.5	0.2	2.4	0.9	0.5	1.9	0.3	2.2	0.3	0.4	1.1
End time window (Av)	0.2	0.3	0.5	0.6	0.6	0.2	0.8	0.7	0.9	0.3	0.7	0.4	0.2	0.6
Length time window (Av)	1.6	2.2	2.5	0.5	0.8	1.3	0.3	0.7	0.5	1.1	1.2	0.2	0.6	0.9
Start time window (SD)	0.2	1.3	3.9	0.2	0.3	0.7	0.3	0.2	2.1	0.1	2.6	0.3	0.3	0.5
End time window (SD)	2.7	1.6	1.3	4.0	0.7	1.0	0.9	0.4	0.5	1.5	0.2	0.3	0.3	0.8
Length time window (SD)	2.5	1.0	2.6	0.8	0.4	2.5	2.0	0.6	2.7	1.9	0.4	0.9	0.4	1.1
Start time window (skewness)	0.8	1.0	1.5	0.9	0.7	0.5	0.8	1.3	0.3	0.4	0.7	0.2	0.2	0.4
End time window (skewness)	2.6	2.3	1.0	1.0	1.2	2.0	0.3	0.4	0.5	1.3	1.7	0.3	1.0	0.6
Length time window (skewness)	2.5	0.6	3.1	0.7	0.9	3.0	0.4	0.6	2.1	2.7	0.9	1.1	0.3	0.5
Start time window (kurtosis)	2.1	1.1	0.9	0.9	3.2	2.5	0.3	0.3	1.1	0.2	0.2	0.1	0.5	3.8
End time window (kurtosis)	1.9	0.7	0.8	0.6	0.2	1.9	0.5	0.4	1.8	0.2	1.4	0.2	0.4	0.5
Length time window (kurtosis)	0.7	0.4	1.0	0.2	0.8	4.2	0.3	0.2	0.9	0.6	0.3	0.8	0.8	0.4
Percentage tasks identical TWs	3.5	3.9	2.5	0.9	1.3	3.4	0.5	0.3	0.7	2.4	1.1	8.8	0.6	0.6
Task duration (Av)	1.5	0.5	1.7	0.7	0.3	1.7	0.4	0.2	0.7	0.3	0.2	0.4	0.5	0.7
Task duration (SD)	0.2	2.3	1.8	0.2	1.0	0.8	0.7	2.3	0.7	0.5	0.5	0.2	0.4	0.4
Task duration (skewness)	0.7	2.2	0.1	1.7	0.7	0.9	0.4	0.3	2.1	0.8	0.4	0.2	0.5	2.2
Task duration (kurtosis)	0.6	0.9	0.6	1.0	0.6	1.0	0.2	0.2	0.5	1.3	0.6	0.4	0.2	0.7
Costs per route	3.2	4.6	3.8	0.8	1.8	3.9	1.7	0.4	1.6	2.4	1.2	1.2	0.7	1.8
Costs per kilometer	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Costs per task	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Costs per hour	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Distance task to closest vehicles (Av)	0.6	1.2	2.0	0.6	0.5	1.4	0.9	0.4	0.5	0.6	3.1	0.1	0.5	1.6
Distance task to closest vehicles (SD)	0.8	0.6	3.1	0.3	0.6	3.1	0.3	0.5	0.2	1.2	0.2	0.3	0.4	0.2
Total	37.5	37.9	45.3	24.3	19.3	49.7	17.3	13.1	26.7	24.7	31.1	19.8	13.7	28.8

4.2.3 Improvement in KPIs

In Section 3.4, we introduced the labels of each data point, based on the optimization criterion number of planned tasks. Optimization criteria are also known as Key Points of Interest (KPIs). Since we have built the original model, we can now investigate if and how much the model improves the KPI number of planned tasks. We do this by grouping the test data per company. For each company, we look at the number of planned tasks for the algorithm that is now in use, the algorithm that is predicted by the model, and the best-performing algorithm. With this information, we compute the Mean Percentage Error (MPE). The MPE shows how much the results improve when using another algorithm in percentages. We compare the algorithm now in use with both the algorithm predicted by the model and the best-performing algorithm. The results are shown in Table 4.7.

Table 4.7: Mean percentage error per company

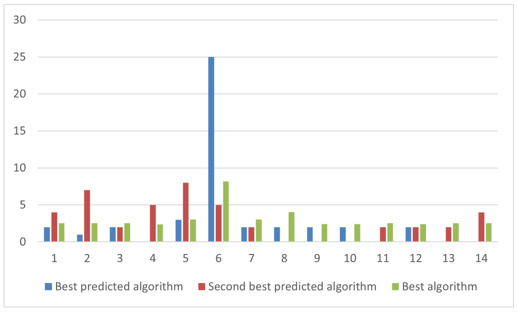
	MPE predicted algorithm (%)	MPE best algorithm (%)
Company A	0.30	1.34
Company B	0.08	1.83
Company C	3.66	15.3
Company D	-0.01	0.67
Company E	0.77	1.16
Company F	0.28	0.28

The results in Table 4.7 show that the solution for almost every company has better KPIs if we follow the predictions of our model. Only for company D, this is false, but the results are only 0.01% worse than the results of the algorithm now in use. Also, for every company, there is still room to grow. By comparing the algorithm in use to the best-performing algorithm, we see that the KPIs can still improve by some percentages. Of course, this is an average, so, we also look at how often a prediction improves the KPI and how often the algorithm in use performs better. These results are shown in Table 4.8. We do not consider the size of the improvement. From Table 4.8, we conclude that for every company, except company B, the predicted algorithm more often improves the KPI than reduces the results compared to the algorithm in use. Therefore, based on these findings and the MPE, we conclude that our model does improve the algorithms that are now in use.

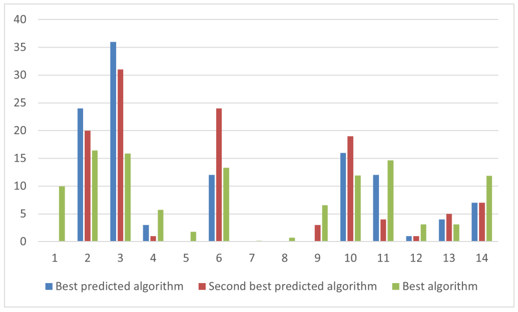
Table 4.8: Comparison algorithm in use and predicted algorithm

	% alg. in use better	% equal	% predicted alg. better
Company A	6.5	73.6	19.9
Company B	16.8	70.5	12.7
Company C	24.7	47.3	28.0
Company D	10.3	69.4	20.4
Company E	19.9	28.8	51.3
Company F	18.2	3.0	78.7

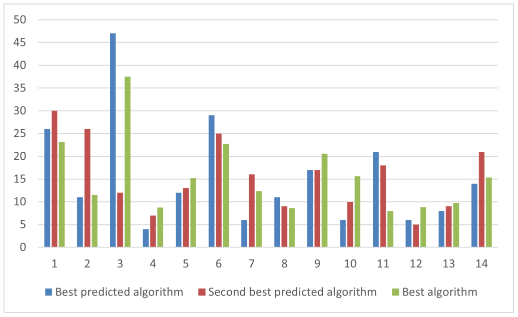
The results above are based on determining the best algorithm per data point: the algorithms might vary between different data points. In practice, only one algorithm is used per company. Therefore, we investigate which algorithm is predicted to be the best most often, and whether replacing the algorithm now in use with this algorithm improves the KPIs. We first determine the distribution of the predicted algorithms in Figure 4.3. From these graphs, we can conclude what algorithm is predicted best most often. Furthermore, we can conclude whether the distribution of the predicted algorithms is approximately equal to the distribution of the best-performing algorithm. For all companies except company B, the algorithm most often predicted to be the best is indeed the algorithm that is the best most often. For all companies, except company C, the predicted best algorithm is not the algorithm now in use. Therefore, we are interested in finding out whether replacing the algorithm now in use, improves the KPIs. This results in Table 4.9, which shows that replacing the algorithm for companies A, D, E, and F improves the KPIs.



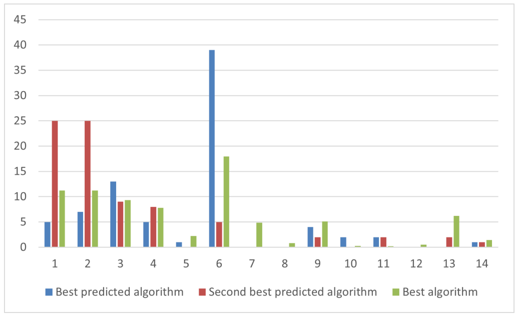
(a) Company A



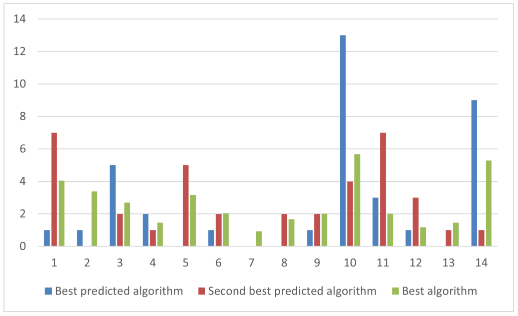
(b) Company B



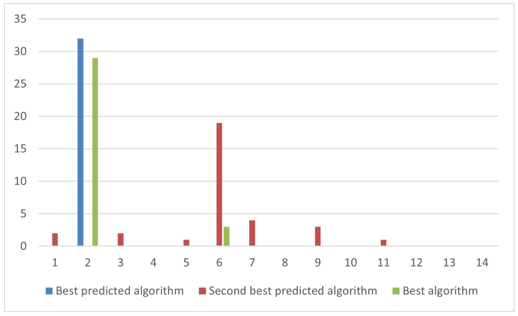
(c) Company C



(d) Company D



(e) Company E



(f) Company F

Figure 4.3: Distribution showing the number of times an algorithm is selected per company

Table 4.9: MPE predicted best algorithm

	MPE predicted best algorithm (%)
Company A	0.76
Company B	-0.22
Company C	0.0
Company D	0.39
Company E	1.33
Company F	0.26

4.2.4 Improving the model

If we look at the baseline in Table 4.1 and compare it with the accuracy results of the original model, we see that our model performs only a few percent better. The loss is improved a lot, as the loss of our original model is half of the loss of the baseline. However, we want to improve even further, especially the accuracy. Based on our observations, we can improve in a few directions.

First, if we look at the data analysis in Section 3.5, we conclude that the data is quite imbalanced. The data analysis shows that especially algorithms 7, 8, and 13 are underrepresented, while algorithms 2 and 3 are overrepresented. We try to solve this by using downsampling and oversampling, as described in Section 2.6. We use the ILP formulation in Section 2.6.3, to acquire the new downsampled and oversampled training data. We evaluate the loss and accuracy of the new models in Section 4.3 and Section 4.4, respectively.

Another interesting finding is that in Table 4.3 and Table 4.5, algorithm 13, ‘Smallest time window (parallel)’, is never predicted correctly while having similar performance to algorithms 7 and 8. Since algorithms 7 and 8 do not have a high percentage but are predicted correctly sometimes, one can wonder why that is not the case for algorithm ‘Smallest time window (parallel)’. This might be because the model cannot differentiate between algorithm 13 and the other algorithms due to the input features not representing the differences. This is verified by the fact that ‘Smallest time window (parallel)’ has a low total Greenwell number of 13.7. Therefore, adding more input features should improve the model. We add a new set of input features and leave out redundant features in Section 4.6.

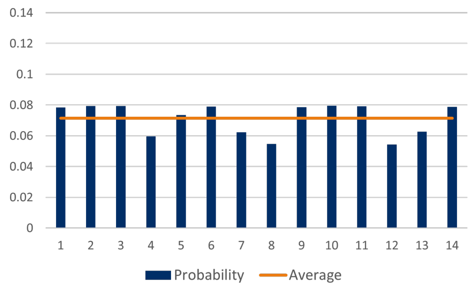
4.3 Downsampling

In Section 3.5, we concluded that we have imbalanced data, which could be why our model does not perform perfectly. If we acquire a balanced data set, it might improve our model. One of the methods to get balanced data is downsampling: we select a subset of data points such that the data set is more balanced. More general information on downsampling can be found in Section 2.6.1. In this section, we perform a data analysis and grid search on the downsampled data and discuss the results.

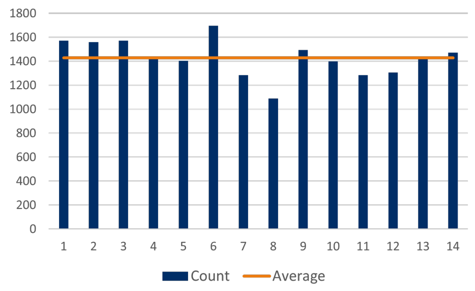
In downsampling, we exclude specific data points from the training data set. Determining which data points to exclude is hard because we are dealing with a multi-label regression problem. A data point might have a high probability for an overperforming algorithm, so could potentially be removed. However, it could be the case that this data point has a high probability for an underperforming algorithm as well. Removing this data point lowers the average performance of this algorithm even further. Therefore, we might not want to exclude this data point from the data set. We have to find the right balance between several algorithms at the same time. Luckily, this problem can be formulated as an Integer Linear Program (ILP), introduced in Section 2.6.3 and modeled specifically for downsampling in Section 3.6.1. The ILP keeps in mind the influence of the data points on several algorithms simultaneously.

4.3.1 Data analysis

Solving the ILP in Section 3.6.1 is done by an ILP-solver. Unfortunately, solving an ILP is NP-hard, so we cut off the solver after 24 hours. We use the best solution found within these 24 hours. We solve the ILP this way for every of the 12 training data sets, which results in new training sets. Combining these new training sets and performing a data analysis, leads to Figure 4.4. A data analysis on each individual training set, rendered similar results. Figure 4.4a shows that the data set is not perfectly balanced, although we tried to balance the average performance of all construction algorithms. Algorithms 4, 7, 8, 12, and 13 are still underperforming, albeit less than in the original data set. This is due to the constraints forcing a number of data points from each company to be included in each training set. Furthermore, Figure 4.4b shows that the number of times an algorithm performs best is not perfectly balanced. This is not surprising, as this criterion was not considered when balancing the data.



(a) Average label for each construction algorithm



(b) Number of times a construction algorithm performs best

Figure 4.4: Performance for all data evaluated in every construction algorithm over all training data sets

4.3.2 Grid search

For each split and random seed, the mean squared error loss and accuracy are computed, as shown in Table 4.10. We again construct models with 2, 5, and 10 hidden layers, with a width of 20, 40, and 60. We immediately conclude that the loss results are similar, or even better, than those of the original model but that the accuracy results are way worse. The accuracy results are even more than 10% worse than the baseline. The model cannot distinguish what algorithm performs best at all, as it does not have enough data points to train correctly, which results in bad accuracy results. The best accuracy result is given for 2 hidden layers and a width of 60. The best loss results are found for any setting of 5 hidden layers, and 10 hidden layers with a width of 60. If we weigh the balance between loss and accuracy, we conclude that 10 hidden layers with a width of 60 yields the best model. We perform a small grid search around this setting.

Table 4.10: Loss and accuracy results for large grid search downsampling

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	20	0.142	41.4
		40	0.146	41.9
		60	0.148	43.3
	5	20	0.139	37.5
		40	0.139	39.5
		60	0.139	35.6
	10	20	0.141	38.4
		40	0.140	35.8
		60	0.139	40.6

Interestingly, in Table 4.11, for every setting the loss is identical, while the accuracy varies. For the accuracy, 11 hidden layers with a width of 70 performs best. However, these accuracy results are nowhere close to those of the original model and the baseline. Therefore, downsampling does not improve our model, and we should not consider it any further for this data set.

Table 4.11: Loss and accuracy results for small grid search downsampling

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	9	50	0.139	37.8
		60	0.139	36.9
		70	0.139	42.1
		80	0.139	39.4
	10	50	0.139	37.8
		60	0.139	40.1
		70	0.139	38.9
		80	0.139	38.7
	11	50	0.139	38.0
		60	0.139	40.5
		70	0.139	42.6
		80	0.139	36.1

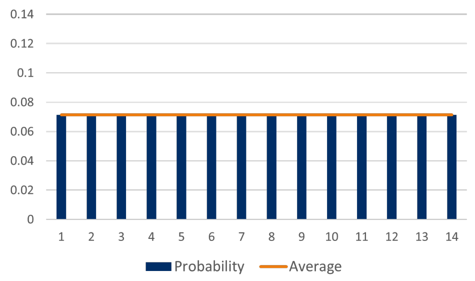
4.4 Oversampling

Another method that deals with imbalanced data, is oversampling, as discussed in Section 2.6.2. The data set is balanced by duplicating data points that perform well for low-performing algorithms. It is hard to determine which data points need to be duplicated because we are dealing with a multi-label regression problem. Luckily, this exact problem can be formulated as an ILP, as discussed in Section 2.6.3. The ILP keeps the influence of the data points for different algorithms in mind and can be found in Section 3.6.2. It cannot be solved to optimality in polynomial time because the problem is NP-hard. Therefore, we cut off the solver after 24 hours and use the best-found solution within these 24 hours. In this section, we look at the data analysis of this best solution and perform a grid search on this newly acquired data.

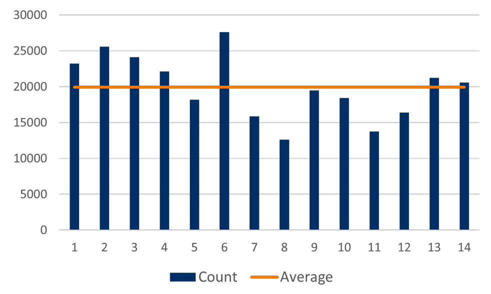
Before we discuss the data analysis, we explain the correct way to perform oversampling. First, the data set is divided into a training, validation, and test set. After, oversampling is performed only on the training set. Otherwise, the following problem occurs. If we would oversample before splitting the data, duplicate data points could end up in both the training and test set. This means that the model is partly trained on test data, which leads to invalid results. Therefore, first, the splitting is performed, followed by the oversampling of the training set. In downsampling, we also first split the training and test data and then perform the downsampling, but not the same problem can occur there. Because we only remove data points in downsampling, no duplicate data points can end up in both the test and training set.

4.4.1 Data analysis

We balance all 12 training sets, as discussed in Section 4.1, by letting an optimizer solve the proposed ILP in Section 3.6.2 with a cut-off after 24 hours. After, we perform a data analysis on the training sets altogether, shown in Figure 4.5. We also analyzed each oversampled data set independently, which led to similar results. The ILP was designed to balance the average label for each construction algorithm. As shown in Figure 4.5a, it performed well. However, the number of times an algorithm performs best is not automatically balanced as well, as shown in Figure 4.5b.



(a) Average label for each construction algorithm



(b) Number of times a construction algorithm performs best

Figure 4.5: Performance for all data evaluated in every construction algorithm over all training data sets

4.4.2 Grid search

For each split and random seed, the mean squared error loss and accuracy are computed. In Table 4.12, we see the results for the different settings. We again construct models with 2, 5, and 10 hidden layers, with a width of 20, 40, and 60. We immediately conclude that 5 hidden layers does not yield the best-performing models anymore, as 2 hidden layers always performs better for both loss and accuracy. Furthermore, for 10 hidden layers, it is clear that the wider the layers, the better the performance. Two settings yield good results: 10 hidden layers with a width of 60 and 2 hidden layers with a width of 20. The first has the lowest loss and second highest accuracy, while the other has the highest accuracy and the second lowest loss. Therefore, we perform a small grid search in both areas, which leads to Table 4.14 and Table 4.15. Furthermore, we conclude that for two hidden layers, a width of 60 performs better than a width of 40, so we check whether a higher width results in a good model. This leads to Table 4.13.

Table 4.12: Loss and accuracy results for large grid search oversampling

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	20	0.181	50.9
		40	0.198	47.6
		60	0.198	49.4
	5	20	0.206	42.4
		40	0.213	44.6
		60	0.206	47.2
	10	20	0.196	42.5
		40	0.202	45.2
		60	0.173	49.7

From Table 4.13, we conclude that increasing the width for 2 hidden layers does not result in improvements. Moreover, 3 hidden layers underperforms in comparison with 2 hidden layers. From Table 4.14, we conclude that reducing the width even further improves the loss but also deteriorates the accuracy by 1.4%. We decide this trade-off is too big and determine a width of 20 to be the best setting for 2 hidden layers. Again, 3 hidden layers is outperformed by 2 hidden layers. From Table 4.15, we conclude that the models with a high accuracy, unfortunately, also have a high loss, while models with a low loss, unfortunately, have a low accuracy. Therefore, there is no immediate clear choice under what settings the best model is built.

Table 4.13: Loss and accuracy results for small grid search oversampling part 1

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	60	0.198	49.4
		70	0.200	48.9
		80	0.201	47.0
	3	60	0.209	46.4
		70	0.203	48.4
		80	0.201	48.7

Table 4.14: Loss and accuracy results for small grid search oversampling part 2

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	15	0.174	49.5
		20	0.181	50.9
		25	0.190	49.1
	3	15	0.186	48.2
		20	0.203	45.0
		25	0.200	49.0

Table 4.15: Loss and accuracy results for small grid search oversampling part 3

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	9	60	0.181	48.1
		70	0.165	50.7
		80	0.158	47.3
	10	60	0.152	49.4
		70	0.173	49.7
		80	0.159	49.1
	11	60	0.153	47.3
		70	0.153	46.2
		80	0.163	46.0

If we consider all tables, we conclude that 2 hidden layers and a width of 20 reaches the highest accuracy of 50.9%, and the model with 10 hidden layers and a width of 60 has the lowest loss of 0.152. In both cases, these results are worse than the original model. Furthermore, only the loss is an improvement on the baseline. Therefore, we conclude that oversampling this way does not help to improve our model. Further investigation into what causes this problem is needed. It is hard to draw any conclusions based on the NN because it is a black-box method. Therefore, we look at the oversampled data. For instance, one data point is included in the training data set 68 times. Its labeled probabilities are:

$$\begin{bmatrix} 0.189 & 0.189 & 0.189 & 0.189 & 0.010 & 0.189 & 0.000 \\ 0.000 & 0.026 & 0.000 & 0.009 & 0.000 & 0.009 & 0.000 \end{bmatrix}$$

Studying this data point, we suggest the following. We know from the data analysis in Section 3.5 that algorithms 2, 3, and 6 are well-performing overall, which is also true for this specific data point. However, algorithm 1 and algorithm 4 are performing equally well. We suspect the oversampling includes this data point multiple times to increase the average probability of algorithm 4 because in general algorithm 4 is a bad-performing algorithm. However, this data instance does not exclusively work well for algorithm 4. Therefore, the model might not learn the relationship between the input features and algorithm 4, but it focuses more on algorithms 2, 3, and 6. To solve this problem, only data points that work exclusively best for algorithm 4 should be included multiple times. This leads to oversampling on data points that have a unique best algorithm, which is described in Section 4.5.

4.5 Oversampling unique best algorithms

In Section 4.4, we concluded that oversampling the entire training set regarding the average probability did not lead to good results. There might be several reasons for this, for instance, the ILP focuses too much on data points in which multiple algorithms perform best. An example of this was shown in Section 4.4. Therefore, we decide to reperform oversampling, but only regarding the data points in the training set that have a unique best algorithm. Furthermore, it might be more logical for each algorithm to have the same amount of data points for which the algorithm is best performing, in contrast to looking at how much better they perform. For our ILP, this results in the following change in input:

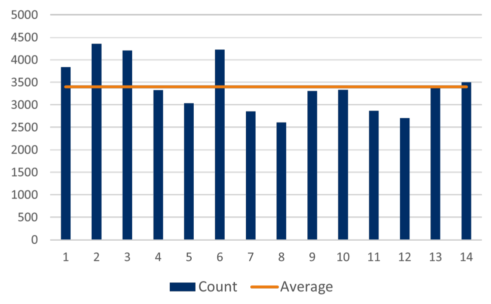
$$p_{ij} = \begin{cases} 1, & \text{if } j \text{ is the best-performing algorithm} \\ 0, & \text{otherwise} \end{cases}$$

Furthermore, we perform the oversampling only on data points with a unique best algorithm. All other data points from the training data set are included exactly once.

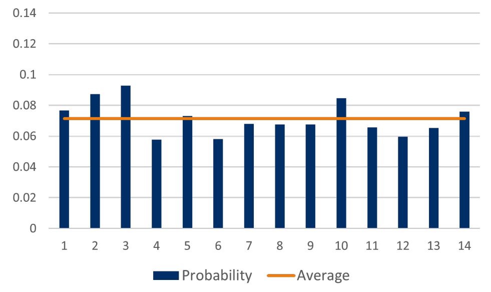
We let an optimizer solve the ILP formulated in Section 3.6.2 with the newly assigned probabilities for the 12 filtered training sets. All ILPs are solved to optimality within 24 hours. In Figure 4.6, we analyzed all the training data sets together. The data analysis for each individual training set is similar. Although the ILPs are solved to optimality, we do not get as balanced results as for the previous oversampling because, after the oversampling, we add the data points in which multiple algorithms perform best. This causes the data to be slightly imbalanced again.

Table 4.16: Accuracy results for large grid search for oversampling per algorithm for unique labeled data points

Activation function	HL	Width	1	2	3	4	5	6	7	8	9	10	11	12	13	14	average
Softmax	2	20	16.4	67.8	51.4	2.1	5.6	10.7	8.3	23.6	8.3	20.8	18.75	0	0	9.2	17.4
		40	13.6	45.7	58.8	4.2	8.3	7.8	0	0	9.7	22.2	34.0	0	0	6.3	15.1
		60	8.5	52.9	58.6	0	2.8	9.7	16.7	6.9	6.9	18.1	19.1	0	0	11.3	15.1
	5	20	11.7	39.5	29.0	2.1	6.9	6.1	0	0	4.2	11.1	5.6	0	7.7	6.3	9.3
		40	12.1	35.1	42.4	2.8	13.9	8.0	8.3	6.9	6.9	9.7	11.5	0	0	10	12.0
		60	2.1	57.1	42.8	0	6.9	5.5	0	15.3	13.9	9.7	18.8	3.8	0	7.5	13.1
	10	20	16.1	22.9	19.7	0	16.0	7.0	8.3	12.5	8.3	11.1	12.8	0	0	7.5	10.2
		40	13.2	29.8	29.5	8.3	5.6	10.9	0	16.7	20.8	9.7	10.4	0	0	3.8	11.3
		60	7.9	62.9	45.5	8.3	0	9.2	0	6.9	8.3	11.1	26.4	3.8	0	9.6	14.3
average			11.3	46.0	42.0	3.1	7.3	8.3	4.6	9.9	9.7	13.7	17.5	0.8	0.9	7.9	



(a) Number of times a construction algorithm performs best



(b) Average label for each construction algorithm

Figure 4.6: Performance for all data evaluated in every construction algorithm over all training data sets

4.5.1 Grid search

We again perform a large and small grid search on the new oversampled training sets. For the large grid search, we look at 2, 5, and 10 hidden layers in combination with a width of 20, 40, and 60. This gives the results shown in Table 4.17. First of all, for each hidden layer, we conclude that the higher the width, the better the accuracy, and often the lower the loss as well. Therefore, we can assume that we have to perform an even larger grid search and look at higher values for the width, shown in Figure 4.18.

Table 4.17: Loss and accuracy results for large grid search oversampling based on unique best algorithms

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	20	0.173	48.6
		40	0.183	48.4
		60	0.183	51.0
	5	20	0.192	46.9
		40	0.194	47.8
		60	0.191	48.8
	10	20	0.186	44.8
		40	0.189	47.4
		60	0.164	49.0

Table 4.18: Loss and accuracy results for large grid search part 2 oversampling based on unique best algorithms

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	50	0.186	49.1
		100	0.183	52.2
		150	0.178	53.7
		200	0.178	53.1
		250	0.177	54.9
		300	0.175	53.5
	5	50	0.195	46.5
		100	0.186	49.1
		150	0.172	51.2
		200	0.167	49.9
		250	0.159	50.9
		300	0.156	51.4
	10	50	0.183	48.5
		100	0.142	42.2
		150	0.138	53.1
		200	0.138	53.1
		250	0.138	53.8
		300	0.138	53.1

From Table 4.18, we conclude that 2 hidden layers always outperforms 5 and 10 hidden layers in terms of accuracy. However, this is not the case for the loss because 10 hidden layers has the lowest loss. Furthermore, in contrast to all the previous models, the width for the best-performing models has highly increased. This best-performing model in terms of accuracy, has 2 hidden layers, a width of 250, and an accuracy of 54.9%. Therefore, it has slightly improved the baseline. In terms of loss, the best-performing model has 10 hidden layers, a width of 150-300 with a loss of 0.138, which is a real improvement on both the baseline and the original model. We conclude that this way of oversampling performs better than the one in Section 4.4, but much can still be improved, as the accuracy is nowhere near the results of the original model.

4.6 Additional input features

The performance of a model is not only dependent on the data but also on the choice of input features. For the model to distinguish between different algorithms, it needs enough input features to find this distinction. Therefore, the model might also improve by adding new input features. Moreover, we can remove redundant input features, as they do not contribute to the output of the model. The new selection of input features is based on several sources, just like the original set of input features. Some follow from the feature analysis in Section 4.2.2, some are a logical follow-up from the original features, and some have been suggested by the people of ORTEC. This leads to the following list. Features 1-24 were already included in the original model, while 25-63 are new features.

1. Number of vehicles
2. Number of tasks
3. Number of capabilities
4. Number of depots
5. Start of task time window (average)
6. Start of task time window (standard deviation)
7. Start of task time window (skewness)
8. Start of task time window (kurtosis)
9. End of task time window (average)
10. End of task time window (standard deviation)
11. End of task time window (skewness)
12. End of task time window (kurtosis)
13. Length of task time window (average)
14. Length of task time window (standard deviation)
15. Length of task time window (skewness)
16. Length of task time window (kurtosis)
17. Percentage of tasks with identical time windows
18. Task duration (average)
19. Task duration (standard deviation)
20. Task duration (skewness)
21. Task duration (kurtosis)
22. Costs per route
23. Distance from task to closest 3 vehicles (average)
24. Distance from task to closest 3 vehicles (standard deviation)
25. Distance from task to closest other task (average)
26. Distance from task to closest other task (standard deviation)
27. Percentage of tasks with identical address to another task
28. Start of vehicle time window (average)
29. Start of vehicle time window (standard deviation)
30. Start of vehicle time window (skewness)
31. Start of vehicle time window (kurtosis)
32. End of vehicle time window (average)
33. End of vehicle time window (standard deviation)
34. End of vehicle time window (skewness)
35. End of vehicle time window (kurtosis)
36. Length of vehicle time window (average)
37. Length of vehicle time window (standard deviation)
38. Length of vehicle time window (skewness)
39. Length of vehicle time window (kurtosis)
40. Vehicle allowed travel time before route (average)
41. Vehicle allowed travel time before route (standard deviation)
42. Vehicle allowed travel time after route (average)
43. Vehicle allowed travel time after route (standard deviation)
44. Breakrule ‘Earliest Start’ minimum time (average)
45. Breakrule ‘Earliest Start’ minimum time (standard deviation)
46. Breakrule ‘Earliest Start’ maximum time (average)
47. Breakrule ‘Earliest Start’ maximum time (standard deviation)
48. Breakrule ‘First Activity’ minimum time (average)
49. Breakrule ‘First Activity’ minimum time (standard deviation)

50. Breakrule 'First Activity' maximum time (average)
51. Breakrule 'First Activity' maximum time (standard deviation)
52. Break duration (average)
53. Break duration (standard deviation)
54. Percentage of available tasks per vehicle (average)
55. Percentage of available tasks per vehicle (standard deviation)
56. Percentage of tasks with identical address and overlapping time windows
57. Percentage of tasks in closest range to the centre
58. Percentage of tasks in the two closest ranges to the centre
59. Percentage of tasks in the three closest ranges to the centre
60. Maximum distance to centre
61. Maximum task duration
62. Maximum distance between task and closest three vehicles
63. Maximum distance between tasks and their closest neighbor

A few input features might not be self-explanatory, so we discuss them here. Regarding input features 44-51, there are two types of rules about when a break should be planned for the driver. Some companies prefer to plan a break regarding the earliest start time of the driver. This earliest start time is the time a driver could start working, but he possibly starts later. Other companies prefer to plan a break regarding the first activity that is performed by the driver. Hence, the input features 'Breakrule Earliest Start' and 'Breakrule First Activity'. To gain more insight into the geographical location of the tasks, we compute the centre of the tasks, which is equal to the average longitude and latitude of all tasks. We then compute the distance from the centre to the furthest task and divide this distance into four equal parts. This results in four radi for four circles. These are the different ranges that are used in the input features 57-59. Furthermore, no input features regarding workload balance are incorporated, as those were identical for nearly all data.

4.6.1 Grid search

Like in previous sections, we start the analysis of the performance with a large grid search. We investigate the accuracy and loss of the model under different settings. We look at 2, 5, and 10 hidden layers in combination with a width of 20, 40, and 60 nodes. This leads to the results portrayed in Table 4.19. This table shows that the models with 2 hidden layers almost always outperform 5 and 10 hidden layers, both in terms of loss and accuracy. Furthermore, a width of 20 and 40 yield good results for 2 hidden layers, with an accuracy of 58.7%. Therefore, we perform a small grid search with 2 and 3 hidden layers for a width of 15, 20, . . . , 45. The results are shown in Table 4.20.

From Table 4.18, we conclude that 2 hidden layers outperforms 3 hidden layers most of the time. The settings in which 3 hidden layers gives a better result, the result is relatively bad. Therefore, we conclude that 2 hidden layers is the best setting. Furthermore, we see that the width of 20 and 40 we found in Table 4.19, again yield the best-performing models in terms of accuracy. However, in terms of loss, the smaller the width, the better. For a width of 15, the accuracy drops by 3 percent, so we conclude the trade-off is too big, and so we would have concluded that the best-performing model is that with 2 hidden layers and a width of 20. However, the results in Table 4.21 lead to the conclusion that the model with a width of 20 overfits on algorithm 2: 5 algorithms are never predicted correctly, and algorithm 2 has a very high percentage. Furthermore, for a width of 40, the table shows that only 2 algorithms are never predicted correctly. Therefore, we decide

Table 4.19: Loss and accuracy results for large grid search for a model with additional input features

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	20	0.137	58.7
		40	0.144	58.7
		60	0.145	57.6
	5	20	0.145	55.8
		40	0.150	57.5
		60	0.145	56.9
	10	20	0.146	55.3
		40	0.149	54.6
		60	0.140	52.8

that the model with 2 hidden layers and a width of 40 yields the best results. This gives the final additional input features model with an accuracy of 58.7% and a loss of 0.144. Both the loss and accuracy are an improvement on the baseline, but only the loss is an improvement on our original model.

Table 4.20: Loss and accuracy results for small grid search for the model with additional input features

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	15	0.133	55.7
		20	0.137	58.7
		25	0.139	58.0
		30	0.143	58.5
		35	0.144	56.6
		40	0.144	58.7
		45	0.146	57.5
	3	15	0.137	56.7
		20	0.143	56.5
		25	0.146	57.2
		30	0.146	57.0
		35	0.148	57.4
		40	0.147	57.8
		45	0.147	57.6

Table 4.21: Accuracy results for large grid search for the model with additional input features

Activation function	HL	Width	1	2	3	4	5	6	7	8	9	10	11	12	13	14	average
Softmax	2	20	23.5	83.2	60.9	0	2.1	23.7	0	0	14.8	19.7	22.3	0	0	6.1	18.3
		40	20.9	80.2	61.2	0	5.6	25.5	3.7	4.2	19.6	22.0	29.3	5.6	0	21.6	21.4
		60	18.1	85.3	56.7	0	4.9	20.4	0	0	10	14.6	37.8	0	0	21.0	19.2
	5	20	12.6	82.1	58.6	0	9.5	24.3	4.2	8.3	16.7	10.9	14.0	0	0	10.2	18.0
		40	15.1	80.8	59.5	0	6.3	20.7	0	11.1	7.9	24.6	35.7	0	0	24.0	20.4
		60	17.2	80.9	59.5	0	6.5	22.2	0	30.6	14.2	14.4	31.2	0	0	17.6	21.0
	10	20	17.6	82.1	58.0	0	14.4	23.5	2.8	0	11.0	8.8	14.8	0	0	10.9	17.4
		40	10.0	60.5	50.8	0	6.25	30.1	0	4.2	14.2	16.9	26.0	0	16.7	14.2	17.8
		60	21.8	53.5	28.3	0	7.6	24.8	0	3.7	8.1	1.7	11.4	0	0	6.7	12.0
average			17.4	76.5	54.8	0	63.2	23.9	1.2	6.9	12.9	14.8	24.7	0.6	1.9	14.7	

4.6.2 Feature analysis

We perform an elaborate feature analysis on our additional input features model of 2 hidden layers and a width of 40. We again perform this analysis using the three methods: permutation feature importance, SHAP, and partial dependence plots. Based on the results of these methods, we construct a list of the important input features. Because we have approximately twice as many input features as in the original model, we also compute a list twice as long: the 10 most important input features.

First, we look at the permutation plot for all the input features, shown in Figure 4.7. Unlike in the permutation plot of the original model, we do not see any redundant input features. However, the 'Breakrule Duration SD', has almost no influence. Based on Figure 4.7, we establish a top 10 of the most important input features, consisting of the input features with the highest means:

1. Percentage of available tasks per vehicle (average)
2. End of time window (standard deviation)
3. Length of time window (standard deviation)
4. End of time window (skewness)
5. Length of vehicle time window (average)
6. Breakrule FA maximum time (average)
7. Length of time window (average)
8. Number of tasks
9. Start of time window (average)
10. Number of capabilities

Secondly, we compute and investigate the SHAP values, as shown in Figure 4.8. Figure 4.8 only includes the top 30 input features for convenience. The figure that includes all 63 input features is included in Appendix B. From Figure B.1, we again conclude that 'Breakrule duration (standard deviation)' is almost redundant, as it is the input feature with the lowest total influence. Based on the results of Figure 4.8, we design a top 10 of the most important input features:

1. End of time window (average)
2. Length of time window (average)
3. Percentage of available tasks per vehicle (average)
4. End of time window (standard deviation)
5. Length of vehicle time window (average)
6. Number of depots
7. Length of time window (skewness)
8. Start of time window (standard deviation)
9. Percentage of available tasks per vehicle (standard deviation)

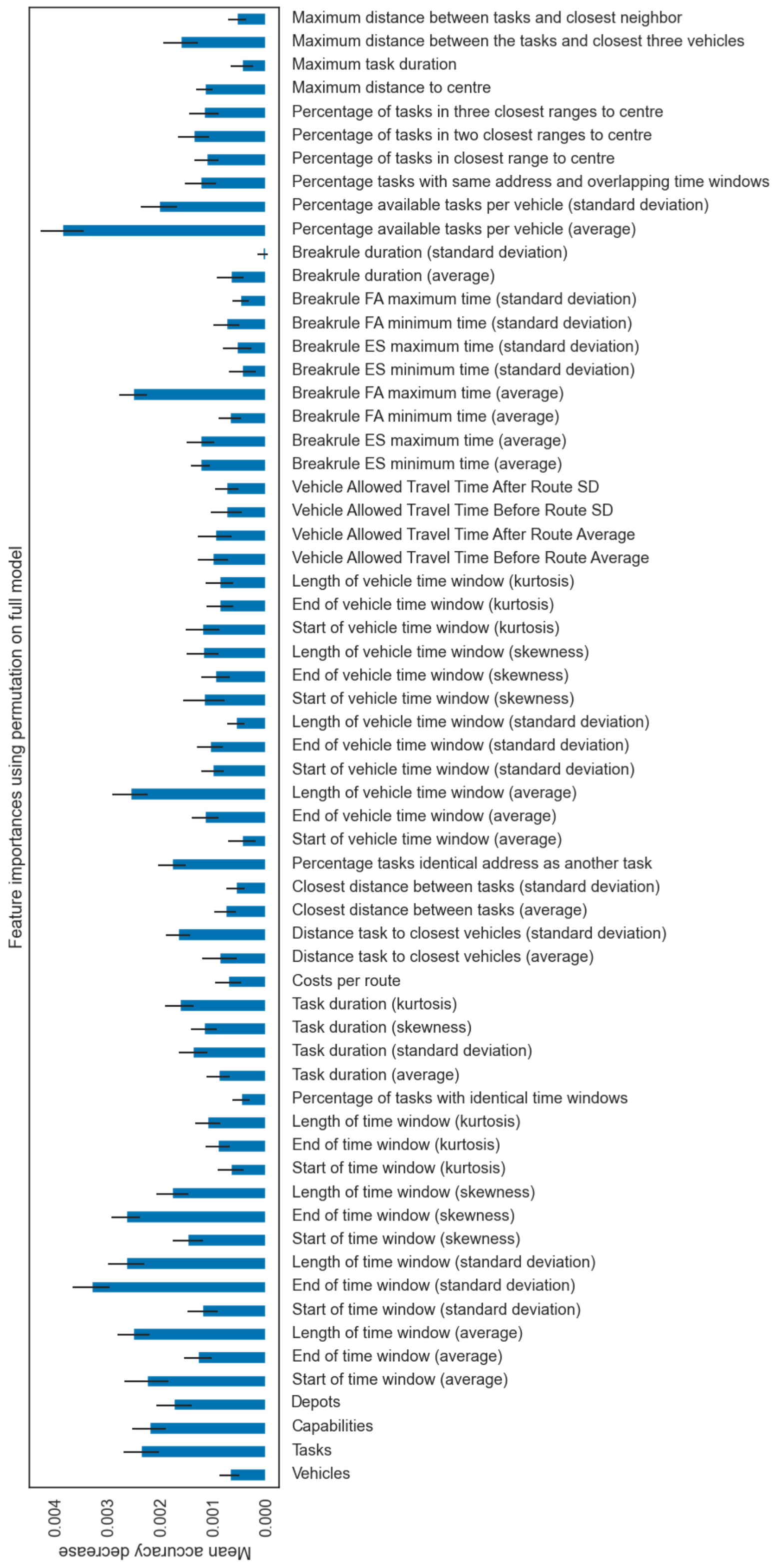


Figure 4.7: Permutation feature importance for the model with additional input features

10. Length of time window (standard deviation)

We compare the list of permutation feature importance and SHAP. Exactly half of the lists overlap shortly. Exactly half of the lists overlap. The remaining features of the permutation feature importance list are all in the top 30 of the SHAP. However, the most influential are 'Breakrule FA maximum time (average)' and 'Start of time window (average)' in places 11 and 14. Those features are still very influential in SHAP, so they probably influence the outcome of the model a lot as well. From the remaining features on the list of SHAP, only 'Number of depots', 'Length of time window (skewness)', and especially 'Percentage available tasks per vehicle (standard deviation)' have some influence according to the feature permutation.

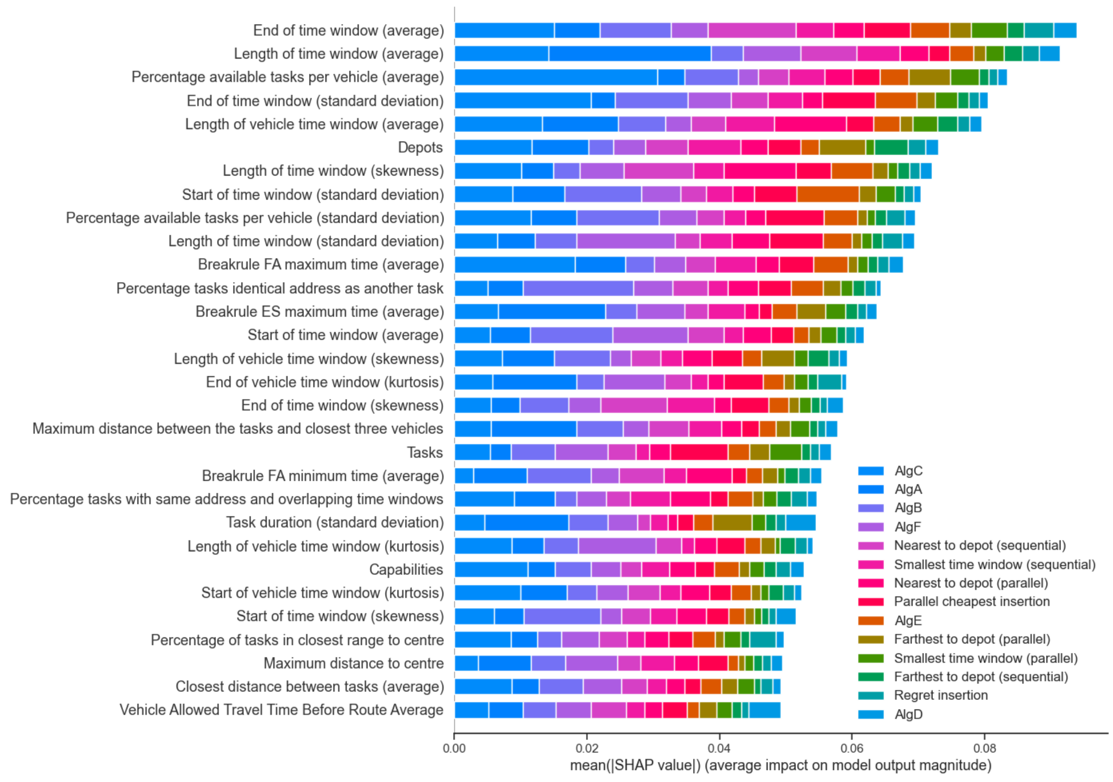


Figure 4.8: SHAP values for every algorithm

Last, we compute the Greenwell numbers. We now have a combination of 63 input features with 14 algorithms, so the table is included in Appendix B. We discuss the most interesting results here. In contrast to the results of the other methods, the 'Breakrule Duration SD' does not badly. However, it is still not a feature of note. Furthermore, two algorithms have a total Greenwell number below 20: 'Farthest to depot (sequential)' and 'Regret insertion'. This means that varying the input features does not influence the output a lot. Compared to the original model, the lowest total Greenwell number has increased from 13.1 to 16.8, so the input features do have more influence. From the rest of the results, we construct a list of the 10 most influential input features based on the highest Greenwell number in the table.

1. Percentage of tasks with identical time windows

2. Costs per Route
3. Percentage of available tasks per vehicle (average)
4. Breakrule FA minimum time (standard deviation)
5. Number of depots
6. Maximum distance between task and closest three vehicles
7. Number of tasks
8. Length of time window (average)
9. Breakrule FA maximum time (average)
10. Number of vehicles

Based on the results of the three methods, we construct an overall top 10. First, one input feature is included in all three lists: ‘Percentage available tasks per vehicle (average)’. Furthermore, seven input features are included in two of the lists and are included in our overall top 10 as well. We only have to include two more for a top 10. We include ‘Start of time window (average)’ as well, as it is included in the top 10 of permutation feature importance and is in place 14 for SHAP. We also include ‘Percentage of available tasks per vehicle (standard deviation)’, as it is in the top 10 of SHAP and performs well for the permutation feature importance. Moreover, both features perform well based on the Greenwell numbers. This leads to the final list:

1. Percentage of available tasks per vehicle (average)
2. Length of time window (average)
3. End of time window (standard deviation)
4. Length of vehicle time window (average)
5. Length of time window (standard deviation)
6. Breakrule FA maximum time (average)
7. Number of tasks
8. Number of depots
9. Start of time window (average)
10. Percentage of available tasks per vehicle (standard deviation)

4.6.3 Improvement in KPIs

The new model with additional input features has a similar performance to our original model. Therefore, we analyze whether this is also true regarding the improvement of the KPI. We group the test data per company and compare the number of planned tasks of the algorithm in use with both the algorithm predicted by our model and the best-performing algorithm. This leads to the results in Table 4.22, in which the mean percentage error is presented. Adhering to the predictions of our model, the number of planned tasks of each company improves, except for Company D. We can also see that there is still some room for improvement, as following the best algorithm leads to an even higher improvement.

These improvements are averages, so it is also interesting to look at how many times the predicted algorithm improves the algorithm now in use. Table 4.23 shows for what percentage of the data points the algorithm in use gives better results, for what percentage the predicted algorithm works best, and for what percentage their KPIs are equal. We conclude that for companies A, D, E, and F, the predicted algorithm performs better more often than the algorithm in use. However, most of the time, the predicted model and the algorithm in use perform equally well. These results are similar to those of the original model in Section 4.2.3.

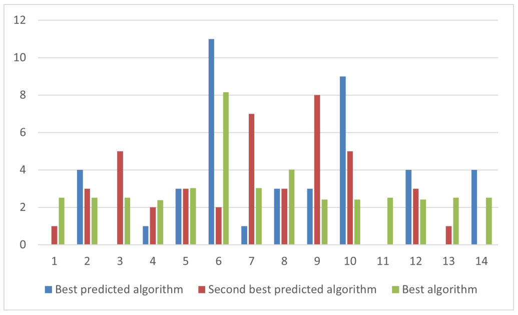
Table 4.22: Mean percentage error predicted and best algorithm

	MPE predicted algorithm (%)	MPE best algorithm (%)
Company A	0.18	1.34
Company B	0.04	1.83
Company C	1.73	15.3
Company D	-0.06	0.67
Company E	0.37	1.16
Company F	0.17	0.28

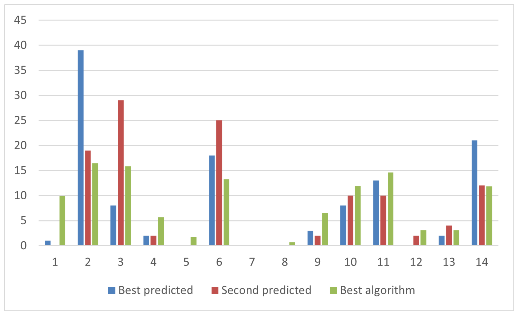
Table 4.23: Comparison algorithm in use and predicted algorithm

	% alg. in use better	% equal	% predicted alg. better
Company A	0.0	88.5	11.5
Company B	22.4	61.6	16.0
Company C	26.1	48.6	25.3
Company D	19.8	58.3	22.0
Company E	14.1	48.2	37.7
Company F	0.0	45.0	55.0

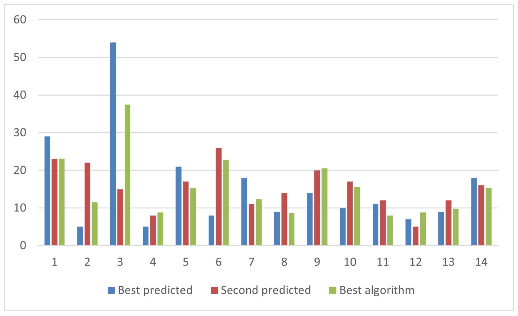
The results above are based on determining the best algorithm per data point, so the algorithms might vary between different data points. In practice, only one algorithm is used for all the data points for one company. Therefore, we investigate which algorithm is predicted to be the best most of the time and whether replacing the algorithm now in use with this algorithm improves the KPIs. We first determine the distribution of the predicted algorithms in Figure 4.9. From these graphs, we can conclude which algorithm is predicted best most often and if the distribution of the predicted algorithms somewhat represents the distribution of the best-performing algorithm. This shows that for all companies, the algorithm that is predicted to be the best most often is indeed most often the best. Only for company B and C, this is also the algorithm that is now in use. For the other companies, we can investigate whether replacing the algorithm in use improves the KPIs. This results in Table 4.24, which shows that replacing the algorithm for all of the four companies A, D, E, and F, improves the KPIs.



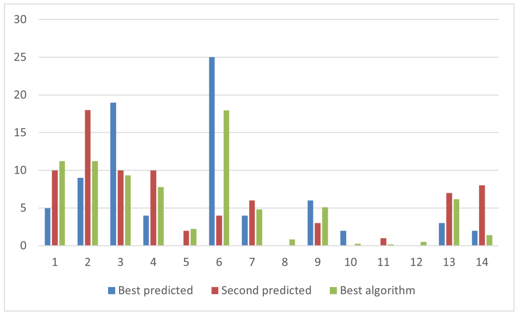
(a) Company A



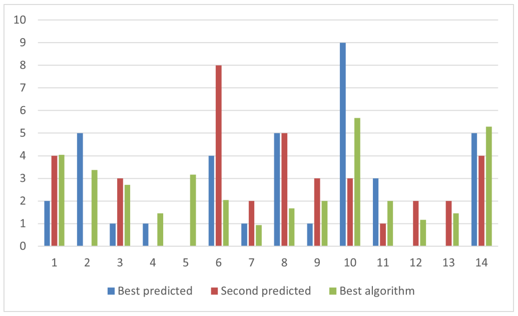
(b) Company B



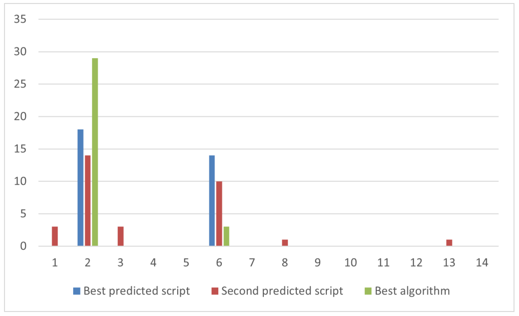
(c) Company C



(d) Company D



(e) Company E



(f) Company F

Figure 4.9: Distribution showing the number of times an algorithm is selected per company

Table 4.24: MPE best predicted algorithm

	MPE best predicted algorithm (%)
Company A	0.76
Company B	0.0
Company C	0.0
Company D	0.39
Company E	1.33
Company F	0.26

4.7 Reducing the number of input features

In Section 4.6, we tried to improve the original model by adding input features to provide the model with more information about the problem. However, adding extra input features also causes the model to have to train more parameters. In this section, we build a new model with a lower number of input features, based on the feature analysis in Section 4.2.2 and Section 4.6.2. By reducing the number of input features based on feature analysis, we hope to contain enough information for the model to distinguish between the various construction algorithms while reducing the number of parameters.

4.7.1 12 input features

If we merge the most important input features found in Section 4.2.2 and Section 4.6.2, we find 12 important input features, as there is some overlap between the lists. This is a large reduction from the original model and the new input features model, with 28 and 63 input features, respectively. The input features of our reduced model are the following:

1. Start of time window (standard deviation)
2. End of time window (standard deviation)
3. Length of time window (standard deviation)
4. Number of depots
5. Number of vehicles
6. Percentage of available tasks per vehicle (average)
7. Length of time window (average)
8. Length of vehicle time window (average)
9. Breakrule FA maximum time (average)
10. Number of tasks
11. Start of time window (average)
12. Percentage of available tasks per vehicle (standard deviation)

With these input features, we again perform a large grid search for 2, 5, and 10 hidden layers with a width of 20, 40, and 60. The results are shown in Table 4.25. For 2 hidden layers and a width of 20, we finally improve both the loss and the accuracy results of the original model, with a new loss of 0.125 and an accuracy of 61.7%. Furthermore, we conclude that for loss and accuracy, 2 hidden layers outperforms 5 and 10 hidden layers. To investigate the results further, we perform a small grid search shown in Table 4.26. We again conclude that 2 hidden layers outperforms 3 hidden layers every time. The model with a width of 20 remains the best setting with a loss of 0.125 and an accuracy of 61.7%.

To determine whether the results in Table 4.25 are due to selecting the most important input features or only due to reducing the number of parameters, we also select 12 random input features and build models under the same settings. This leads to the results in Table 4.27. This table shows that having 12 random input features leads to loss and accuracy results similar to those of the original model. However, none of the models attains the loss and accuracy results using the 12 most important input features. Therefore, selecting the most important features does have an influence.

Table 4.25: Loss and accuracy results for large grid search for the model with the 12 most important input features

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	20	0.125	61.7
		40	0.129	61.5
		60	0.132	61.5
	5	20	0.141	58.6
		40	0.148	58.0
		60	0.144	57.6
	10	20	0.149	55.6
		40	0.149	56.4
		60	0.139	53.6

Table 4.26: Loss and accuracy results for small grid search for the model with the 12 most important input features

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	15	0.125	60.2
		20	0.125	61.7
		25	0.127	59.9
		30	0.128	61.0
		35	0.129	60.8
		40	0.129	61.5
	3	15	0.129	59.5
		20	0.133	59.3
		25	0.134	59.7
		30	0.137	59.9
		35	0.137	60.4
		40	0.139	59.8

Table 4.27: Loss and accuracy results for large grid search for the model with 12 random input features

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	20	0.129	59.3
		40	0.134	59.0
		60	0.137	56.3
	5	20	0.145	56.1
		40	0.148	58.0
		60	0.149	56.4
	10	20	0.146	56.3
		40	0.148	56.4
		60	0.138	52.9

4.7.2 7 input features

In the previous section, we concluded that reducing the number of input features based on the feature analysis improved the model. Therefore, we try reducing the number of input features even more. Based on the knowledge of the ORTEC experts, we constructed a list of 7 input features from the list of 12:

1. Start of time window (standard deviation)
2. End of time window (standard deviation)
3. Length of time window (standard deviation)
4. Percentage available tasks per vehicle (average)
5. Percentage available tasks per vehicle (standard deviation)
6. Length of time window (average)
7. Length of vehicle time window (average)

We perform a small grid search, whose results are shown in Table 4.28. This table shows that reducing the number of input features even further does not result in better models.

Table 4.28: Loss and accuracy results for small grid search for the model with 7 input features

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	20	0.129	59.0
		40	0.132	58.8
		60	0.133	58.7
	3	20	0.133	57.5
		40	0.139	58.5
		60	0.141	58.8

4.7.3 18 input features

Because reducing the number of input features to only seven does not improve the results, we investigate whether having a slightly larger group of input features improves the results. We determine the 18 most important input features similar to determining the 12 most important features. For the original model, we constructed a list of the 9 most important input features, similar to Section 4.2.2. For the model with additional input features, we constructed a list of the 15 most important input features based on the feature analysis results found in Section 4.6.2. These lists are included in Appendix C. Together, they formed a list of 18 most important input features:

1. Start of time window (standard deviation)
2. End of time window (standard deviation)
3. Length of time window (standard deviation)
4. Number of depots
5. Number of vehicles
6. Percentage of available tasks per vehicle (average)
7. Length of time window (average)
8. Length of vehicle time window (average)
9. Breakrule FA maximum time (average)
10. Number of tasks
11. Start of time window (average)

12. Percentage of available tasks per vehicle (standard deviation)
13. Start of time window (kurtosis)
14. Length of time window (skewness)
15. Percentage of tasks with identical address as another task
16. Number of capabilities
17. End of time window (average)
18. Maximum distance between the tasks and closest three vehicles

Performing a large grid search using these input features results in Table 4.29. This table shows that adding even more important features does not yield better results. Therefore, we conclude that determining the right amount of input features can help improve the model. The right balance between noise, number of parameters, and enough information for the model needs to be found.

Table 4.29: Loss and accuracy results for large grid search for the model with 18 input features

Activation function	Hidden layers	Width	Loss	Accuracy
Softmax	2	20	0.128	60.2
		40	0.133	59.3
		60	0.136	60.0
	5	20	0.143	58.0
		40	0.148	57.6
		60	0.142	57.3
	10	20	0.146	55.4
		40	0.148	56.3
		60	0.141	53.5

4.7.4 Improvement in KPIs

The model with the 12 most important input features with 2 hidden layers and a width of 20, results in the best-performing model overall in this thesis. Therefore, we also investigate whether the KPIs improve based on the predictions of this model. In Table 4.30, the mean percentage error is shown. The predicted algorithm and the best algorithm are both compared to the algorithm now in use for each company. For all companies, we see an improvement in KPIs. In previous analyses of the KPIs, in Section 4.2.3 and Section 4.6.3, we concluded that the predictions of our model led to a reduction for company D, but for our model with 12 input features, this is not the case. Furthermore, there is still room for improvement as the percentual improvement for the best algorithm is still higher for each company. From Table 4.31, we conclude that for companies A, D, E, and F, the predicted algorithm performs better most of the time. For all four companies, the differences in percentage are also quite high.

The results above are based on determining the best algorithm per data point. The algorithms might vary between different data points. In practice, only one algorithm is used for all the data points for one company. Therefore, we investigate which algorithm is predicted to be the best the most often and whether replacing the algorithm now in use with this algorithm improves the number of planned tasks. We first determine the distribution of the predicted algorithms in Figure 4.10. For companies A, C, D, and F, the algorithm that is predicted to be the best most often, is indeed the best most often. For companies B and E, this is not true, but rather the algorithm that is predicted to be the second best most often, is the best algorithm the most often. Therefore, the

Table 4.30: Mean percentage error

	MPE predicted algorithm (%)	MPE best algorithm (%)
Company A	0.91	1.34
Company B	0.39	1.83
Company C	1.77	15.3
Company D	0.26	0.67
Company E	0.95	1.16
Company F	0.09	0.28

Table 4.31: Comparison algorithm in use and predicted algorithm

	% alg. in use better	% equal	% predicted algorithm
Company A	6.9	72.4	20.7
Company B	21.3	59.4	19.3
Company C	26.2	50.7	23.1
Company D	6.3	66.4	27.3
Company E	7.0	48.2	44.8
Company F	5.9	9.6	84.5

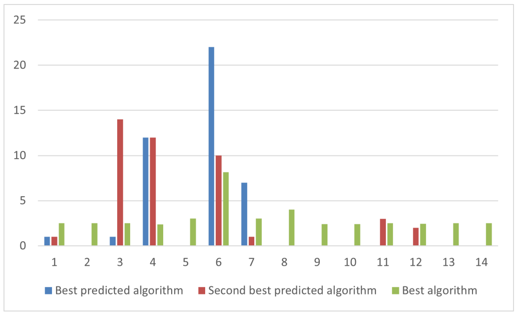
predictions are somewhat correct but can be improved. For company C, the best algorithm is the algorithm that is now in use. For the other algorithms, another algorithm is predicted to be the best. Therefore, it is interesting to see whether this algorithm improves the KPI on average for all test data of that particular company. Table 4.32 shows that replacing the algorithms of companies A, B, D, and F indeed results in a better KPI. However, replacing algorithm E does not improve the number of planned tasks but reduces it.

Table 4.32: MPE best predicted algorithm

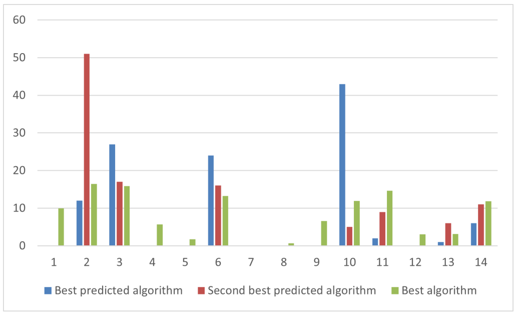
	MPE best predicted algorithm (%)
Company A	0.76
Company B	0.52
Company C	0.0
Company D	0.39
Company E	-0.69
Company F	0.26

4.8 Single construction algorithm models

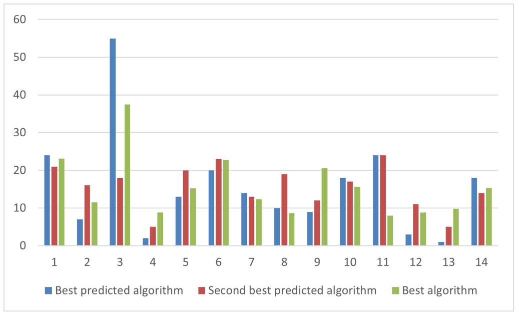
The results in the previous sections show that the models cannot predict the best algorithm in approximately 40% of the cases. Therefore, we investigate whether the probability of one specific algorithm can be predicted correctly. By building a model for one specific algorithm, all the parameters are tuned to the preferences of that particular algorithm, and the model does not need to combine the preferences of all algorithms simultaneously. This might lead to a better prediction. Also, we investigate whether the probabilities of some algorithms can be predicted more precisely



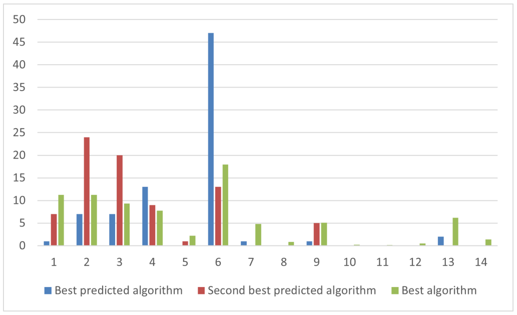
(a) Company A



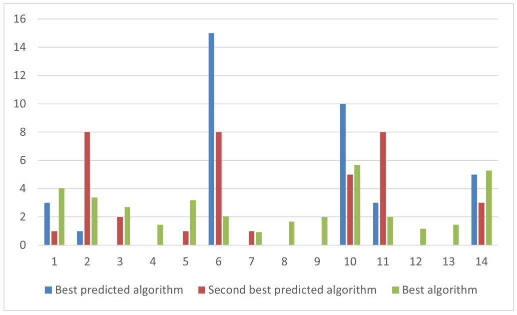
(b) Company B



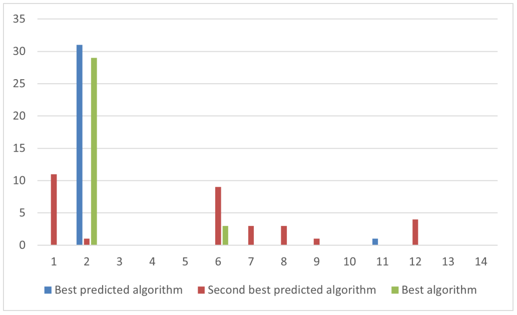
(c) Company C



(d) Company D



(e) Company E



(f) Company F

Figure 4.10: Distribution showing the number of times an algorithm is selected per company

than others.

For each algorithm independently, we build a new NN model. This time, we do not have an output of 14 different probabilities, but the output layer now only consists of 1 node. To build such a model, a few minor changes to the original setup need to be made. First, the softmax function can only be used for layers with a width of 2 or more. Therefore, we cannot use the softmax function in the output layer anymore. So, we decide to switch to the origin of the softmax function: the sigmoid function, see Section 2.5.3. After a small exploration of different activation functions, we decided to use the sigmoid function for every hidden layer, just like we used the softmax function for every layer in the original model. Secondly, we cannot determine the accuracy, as by only having one output, we cannot compute anymore whether the best algorithm also has the highest predicted probability. Therefore, in this section, we do not consider accuracy anymore. Last, we change the cost function. We do not use the mean squared error but rather the mean absolute error. In this way, by computing the loss, we can more easily interpret the range in which the probabilities are predicted on average.

With these changes, we build two different types of models for each algorithm: one with the 12 most important input features and one with all the input features. By doing this, we can investigate again whether reducing the number of input features gives better individual results. We again perform a large grid search, but now with 2 and 5 hidden layers and a width of 20, 40, and 60. Because we repeat this process for every algorithm, this results in 14 tables with results. For convenience, they are included in Appendix D. As an example, the results of the algorithm 'Regret insertion' are shown in Table 4.33.

Table 4.33: Mean absolute value loss for 'Regret insertion'

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.030	0.090
		40	0.033	0.074
		60	0.026	0.070
	5	20	0.060	0.067
		40	0.043	0.067
		60	0.048	0.069

If we look at the results in Appendix D, we conclude the following. For every algorithm and every setting, the model with 12 input features has a lower loss than the model with all the input features. The models with only 12 input features perform way better, which we concluded in Section 4.7 as well. For some algorithms, the loss is even halved, just like for the algorithm 'Regret insertion' in Table 4.33.

If we take a closer look at the results of the models with 12 input features, we conclude that for almost every algorithm and width, 2 hidden layers outperforms 5 hidden layers. This is in line with the accuracy and loss results we found in Section 4.6 and Section 4.7. Furthermore, we notice a small difference in performance between the different algorithms. The loss results for 2 hidden layers for the various algorithms range from 0.026 - 0.092, although most models have a loss of around 0.045. These results might explain why the accuracy of our model is not any higher. If, for example, a data point has a probability of 0.10, it could be predicted on average in the range of 0.055 - 0.145. Because this is true for every algorithm, the order of the algorithms is quite easily predicted incorrectly. Unless one algorithm is more than 0.20 better than all the other algorithms, we have quite a chance that the best-performing algorithm is predicted incorrectly.

Chapter 5

Conclusions and further research

In this chapter, we summarize the results of Section 4 and answer the research question:

How to find the most accurate Neural Network model, which predicts the best-suited construction algorithm for a given data instance, and what is the influence of the choice of input features on the performance of the model?

5.1 Interpretation of the results

In this thesis, we have built several NN models to try and solve the algorithm selection problem for real-life VRPs. In the VRP, vehicles have to be assigned a set of tasks such that all constraints are met. ORTEC solves these problems using different construction algorithms, from which beforehand it is hard to determine which algorithm performs best. Therefore, the machine learning method NN can help predict the best working construction algorithm based on the features of the problem.

We have proposed an original model, which we tried to improve on in several ways. Before we built our original model, we started by establishing a baseline, which yielded a loss of 0.28 and an accuracy of 54.7%. We then build our original model by performing a grid search to find the number of layers, the width of each layer, and the activation function that led to the best-performing model. This yielded a model with a loss of 0.147 and an accuracy of 58.6%. This is an improvement on the baseline, but the model is not very well-performing. The loss has almost halved, as expected. In our baseline, we predicted each data point to work well for one algorithm solely, so we hot-one-encoded the probabilities. If we look at the labels of the data points, this is seldom true. This means that the prediction and loss in the baseline were terrible, and much could be improved, as we did.

To improve our model further, we headed in two directions: data balancing and input feature analysis. The data balancing, which we did by downsampling and oversampling data, did not yield any better results. Downsampling did not give better results because the amount of data was reduced, so the model had too little data to train on. Using oversampled data, the model probably found relations in the training data, which were incorrect for the test data. This is caused by the fact that for some algorithms, only a few data points performed well. By copying these data points into the set several times, no new information became available to the model but it only helped focus the model more on the information at hand. This led to relations found which are correct for the few data points in the training set but not in general, and so they were incorrect for the test set too. Adding new data should improve the results.

Secondly, we tried to improve the model by looking at the input features. The model needs enough features which describe the problem, without adding too many features that cause noise. Furthermore, the more features, the more parameters need to be trained, for which you need a lot of data. This is also shown in our results. Our original model had a loss of 0.147 and an accuracy of 58.6%. We first tried adding more input features for the model to have more information to distinguish between the different algorithms by increasing the number of input features from 28 to 63. This yielded a model with a loss of 0.144 and an accuracy of 58.7%, which is a small improvement on our original model. Based on later results, we conclude it did not improve much as the model contained too many parameters. We then proceeded by doing an elaborate feature analysis of the original model and the model with the additional input features. Eventually, this resulted in a list of 12 most important input features: a large reduction from the original number of input features. We indeed concluded that using the most important features to build a smaller model yielded good results. We found the best working model to have a loss of 0.125 and an accuracy of 61.7%, which is a big improvement on our original results. Therefore, we can conclude that performing an elaborate feature analysis leads to better results. The choice of input features is, therefore, of utmost importance.

5.2 Application for ORTEC

The goal for ORTEC was to find a model that predicts what algorithms work well for certain real-life data. Based on the distribution graphs shown in the results, we can conclude that the optimal distribution is not found yet, so algorithms that work well are missed sometimes. However, for almost all models, the distribution correctly predicts which algorithm performs best based on how often it performs best for individual cases. For some companies, this leads to a suggested replacement of the algorithm, which in almost all cases does improve the KPIs. Therefore, our model can predict a better algorithm and give a better understanding of the workings of the different algorithms based on the distributions. Therefore, we conclude that the main goal of ORTEC has been accomplished.

5.3 Limitations

In this thesis, we made some assumptions and decisions to be able to perform the experiments. Because of these decisions, we cannot be absolutely sure of the results. The most important limitations are discussed in the section.

First of all, we selected a set of input features and construction algorithms. Many different input features and construction algorithms exist, of which we only used a subset. Therefore, other input features and construction algorithms might lead to different results.

Secondly, we only had a limited amount of data, which we concluded that it was highly likely to be the reason some results were different than expected. Although, whether this is true, cannot be said with absolute certainty.

Thirdly, for the feature analysis, assumptions were made to determine which input features are important. For permutation feature importance and partial dependence plot, we assumed that all feature values are independent of each other. For SHAP, only a little feature dependence is taken into consideration. The different methods deal differently with the features, which leads to different results. By combining the results of the various methods, we found more evidence for our conclusions, but we cannot say with certainty which input features the model considers important.

5.4 Recommendations for future research

In this section, we discuss the recommendations to improve and elaborate on our research. First, we give some recommendations for improving the results of the models. Secondly, we give some recommendations that are specific to ORTEC.

First of all, to improve the performance of the models, we recommend gathering more data, especially new data on the underrepresented algorithms. We tried to solve the lack of data with oversampling, but this only copies existing information and does not create new information. We think that by adding more versatile data, the model will distinguish between the different algorithms better. Another way to create more versatile data is to use SMOTE, an oversampling technique that creates synthetic data points. However, it might not be straightforward how to do this on multi-label regression data. Secondly, other multi-label regression machine learning models could be investigated. It would be interesting to find out whether we find the same results regarding the feature selection, namely that feature selection is important and does improve the results. In our research, we specifically chose for NNs based on other research and limited time, but to strengthen the conclusion on the influence of the features, the research could be repeated on different machine learning methods, or different data while using a NN. Thirdly, the models in this thesis were trained by mean squared error loss. Inspired by the distribution graphs, we could also investigate whether using cross-entropy loss yields better results. Cross-entropy loss looks at whether the predicted distribution of the algorithms is similar to that of the label. Last of all, we concluded in this thesis that reducing the number of input features leads to a better model, as fewer parameters need to be trained. This could also be the case regarding the number of construction algorithms. Further research could determine whether increasing or reducing the number of construction algorithms would benefit the performance of the model.

For ORTEC, we recommend first investigating whether more data will improve the results. Furthermore, we only based the labels on the number of planned tasks, while the ORTEC algorithms work with several KPI. New models could keep in mind several KPIs. Furthermore, we focused only on construction algorithms, while ORTEC also uses local search and ruin & recreate. Algorithms including local search and ruin & recreate, could also be used to determine the labels.

Bibliography

Jan Karel Lenstra and Alexander H. G. Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11:221–227, 1981. doi: 10.1002/net.3230110211.

S.C. Ho and D. Haugland. A tabu search heuristic for the vehicle routing problem with time windows and split deliveries. *Computers & Operations Research*, 31(12):1947–1964, 2004. doi: 10.1016/S0305-0548(03)00155-2.

Gilbert Laporte, Stefan Ropke, and Thibaut Vidal. Chapter 4: Heuristics for the vehicle routing problem. *Vehicle Routing: Problems, Methods, and Applications, Second Edition*, pages 87–116, 11 2014. doi: 10.1137/1.9781611973594.ch4.

John R. Rice. The algorithm selection problem. In Morris Rubinoff and Marshall C. Yovits, editors, *Advances in Computers*, volume 15, pages 65–118. Elsevier, 1976. doi: 10.1016/S0065-2458(08)60520-3.

Alessio Guerri and Michela Milano. Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 475–479. IOS Press, 01 2004. doi: 10.5555/3000001.3000101.

Jin Guo. Searching for relevant features to classify crew pairing problems. Master's thesis, Chalmers University of Technology, University of Gothenburg, 2019.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, pages 507–523. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-25566-3_40.

Christos H. Papadimitriou. The Euclidean travelling salesman problem is NP-complete. *Theoretical Computer Science*, 4(3):237–244, 1977. doi: 10.1016/0304-3975(77)90012-3.

George B. Dantzig and John Hubert Ramser. The truck dispatching problem. *Management Science*, 6:80–91, 1959. doi: 10.1287/MNSC.6.1.80.

Bruce Golden, Arjang Assad, Larry Levy, and Filip Gheysens. The fleet size and mix vehicle routing problem. *Computers & Operations Research*, 11(1):49–66, 1984. ISSN 0305-0548. doi: 10.1016/0305-0548(84)90007-8.

Roberto Baldacci, Maria Battarra, and Daniele Vigo. *Routing a Heterogeneous Fleet of Vehicles*, pages 3–27. Springer Verlag, 01 2008. doi: 10.1007/978-0-387-77778-8_1.

Soonpracha Kusuma, Anan Mungwattana, Gerrit Janssens, and Tharinee Manisri. Heterogeneous VRP review and conceptual framework. In *Proceedings of the International MultiConference of Engineers and Computer Scientists (IMECS 2014) vol. 2*, volume 2210, pages 1052–1059, 03 2014.

Jairo R. Montoya-Torres, Julián López Franco, Santiago Nieto Isaza, Heriberto Felizzola Jiménez, and Nilson Herazo-Padilla. A literature review on the vehicle routing problem with multiple depots. *Computers & Industrial Engineering*, 79:115–129, 2015. ISSN 0360-8352. doi: 10.1016/j.cie.2014.10.029.

Martin W. P. Savelsbergh. Local search in routing problems with time windows. *Annals of Operations Research*, 4:285–305, 1985. doi: 10.1007/BF02022044.

Nasser A. El-Sherbeny. Vehicle routing with time windows: an overview of exact, heuristic and metaheuristic methods. *Journal of King Saud University - Science*, 22(3):123–131, 2010. ISSN 1018-3647. doi: 10.1016/j.jksus.2010.03.002.

M. Mourgaya and F. Vanderbeck. The periodic vehicle routing problem: classification and heuristic. *RAIRO - Operations Research*, 40(2):169–194, 2006. doi: 10.1051/ro:2006015.

Peter M. Francis, Karen R. Smilowitz, and Michal Tzur. *The Period Vehicle Routing Problem and its Extensions*, pages 73–102. Springer US, Boston, MA, 2008. doi: 10.1007/978-0-387-77778-8_4.

Inmaculada Rodríguez-Martín, Juan-José Salazar-González, and Hande Yaman. The periodic vehicle routing problem with driver consistency. *European Journal of Operational Research*, 273 (2):575–584, 2019. ISSN 0377-2217. doi: 10.1016/j.ejor.2018.08.032.

Manolis N. Kritikos and George Ioannou. The balanced cargo vehicle routing problem with time windows. *International Journal of Production Economics*, 123(1):42–51, 2010. ISSN 0925-5273. doi: 10.1016/j.ijpe.2009.07.006.

S. Shahnejat-Bushehri, A. Kermani, O. Arslan, J.-F. Cordeau, and R. Jans. A vehicle routing problem with time windows and workload balancing for Covid-19 testers: A case study. *IFAC-PapersOnLine*, 55(10):2920–2925, 2022. ISSN 2405-8963. doi: 10.1016/j.ifacol.2022.10.175. 10th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2022.

Simona Mancini, Margaretha Gansterer, and Richard F. Hartl. The collaborative consistent vehicle routing problem with workload balance. *European Journal of Operational Research*, 293(3):955–965, 2021. ISSN 0377-2217. doi: 10.1016/j.ejor.2020.12.064.

Damon Gulczynski, Bruce Golden, and Edward Wasil. The period vehicle routing problem: new heuristics and real-world variants. *Transportation Research Part E: Logistics and Transportation Review*, 47(5):648–668, 2011. ISSN 1366-5545. doi: 10.1016/j.tre.2011.02.002.

Nicolas Jozefowiez, Frédéric Semet, and El-Ghazali Talbi. Multi-objective vehicle routing problems. *European Journal of Operational Research*, 189(2):293–309, 2008. ISSN 0377-2217. doi: 10.1016/j.ejor.2007.05.055.

Sandhya Bansal and Rajeev Goel. Multi objective vehicle routing problem: A survey. *Asian Journal of Computer Science and Technology*, pages 1–6, 11 2018.

Oren E. Nahum and Yuval Hadas. A framework for solving real-time multi-objective VRP. In Jacek Żak, Yuval Hadas, and Riccardo Rossi, editors, *Advanced Concepts, Methodologies and Technologies for Transportation and Logistics*, pages 103–120, Cham, 2018. Springer International Publishing. doi: 10.1007/978-3-319-57105-8_5.

Gilbert Laporte. The vehicle routing problem: an overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(3):345–358, 1992. doi: 10.1016/0377-2217(92)90192-C.

Teodor Gabriel Crainic. *Parallel Solution Methods for Vehicle Routing Problems*, pages 171–198. Springer US, Boston, MA, 2008. ISBN 978-0-387-77778-8. doi: 10.1007/978-0-387-77778-8_8.

Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, 1987. ISSN 0030364X, 15265463. doi: 10.1287/opre.35.2.254.

Johannes Josef Schneider and Scott Kirkpatrick. *Construction Heuristics*, pages 59–61. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006a. ISBN 978-3-540-34560-2. doi: 10.1007/978-3-540-34560-2_11.

Marco Diana and Maged M. Dessouky. A new regret insertion heuristic for solving large-scale dial-a-ride problems with time windows. *Transportation Research Part B: Methodological*, 38(6): 539–557, 2004. ISSN 0191-2615. doi: 10.1016/j.trb.2003.07.001.

Refael Hassin and Ariel Keinan. Greedy heuristics with regret, with application to the cheapest insertion algorithm for the TSP. *Operations Research Letters*, 36(2):243–246, 2008. ISSN 0167-6377. doi: 10.1016/j.orl.2007.05.001.

Manfred Gilli and Enrico Schumann. Chapter 9 - portfolio optimization with “threshold accepting”: a practical guide. In Stephen Satchell, editor, *Optimizing Optimization*, pages 201–223. Elsevier, Boston, 2010. ISBN 978-0-12-374952-9. doi: 10.1016/B978-0-12-374952-9.00009-9.

Miguel-Angel Gil-Rios, Ivan Cruz-Aceves, Fernando Cervantes-Sanchez, Igor Guryev, and Juan-Manuel López-Hernández. Chapter 8 - Automatic enhancement of coronary arteries using convolutional gray-level templates and path-based metaheuristics. In Siddhartha Bhattacharyya, Paramartha Dutta, Debabrata Samanta, Anirban Mukherjee, and Indrajit Pan, editors, *Recent Trends in Computational Intelligence Enabled Research*, pages 129–153. Academic Press, 2021. ISBN 978-0-12-822844-9. doi: 10.1016/B978-0-12-822844-9.00005-0.

Rina Dechter. Chapter 7 - Stochastic greedy local search. In Rina Dechter, editor, *Constraint Processing*, The Morgan Kaufmann Series in Artificial Intelligence, pages 191–208. Morgan Kaufmann, San Francisco, 2003. ISBN 978-1-55860-890-0. doi: 10.1016/B978-155860890-0/50008-6.

Johannes Josef Schneider and Scott Kirkpatrick. *Local Search*, pages 69–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006b. ISBN 978-3-540-34560-2. doi: 10.1007/978-3-540-34560-2_11.

Gerhard Schrimpf, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000. ISSN 0021-9991. doi: 10.1006/jcph.1999.6413.

Haipeng Guo and William H. Hsu. *Algorithm Selection for Sorting and Probabilistic Inference: A Machine Learning-Based Approach.* PhD thesis, Kansas State University, USA, 2003. AAI3100557.

Mamman Salisu, Salisu Abdulrahman, Alhassan Adamu, Yazid Ado, and Akilu Rilwan. An overview of the algorithm selection problem. *International Journal of Computer (IJC)*, 01 2017. doi: 10.1016/S0065-2458(08)60520-3.

Kate Smith-Miles, Jano van Hemert, and Xin Yu Lim. Understanding TSP difficulty by learning from evolved instances. In Christian Blum and Roberto Battiti, editors, *Learning and Intelligent Optimization*, pages 266–280, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13800-3. doi: 10.1007/978-3-642-13800-3_29.

Daniel T.G. Mocking. Online matching of algorithm configuration. Master’s thesis, Vrije Universiteit Amsterdam, 2017.

Kate Smith-Miles, Davaatseren Baatar, Brendan Wreford, and Rhyd Lewis. Towards objective measures of algorithm performance across instance space. *Computers & Operations Research*, 45:12–24, 2014. ISSN 0305-0548. doi: 10.1016/j.cor.2013.11.015.

Sri Harsha Tanamala. Solve machine learning problems: Feature selection (part 4). *Analytics Vidhya*, April 2021.

Jack Tan. Feature selection for machine learning in Python - wrapper methods. *Towards Data Science*, October 2020.

Joon Ian Wong and Nikhil Sonnad. Google's AI won the game Go by defying millennia of basic human instinct. *Quartz*, March 2016.

Y. C. A. Padmanabha Reddy and N. Mohan Krishna Varma. Review on supervised learning techniques. In P. Venkata Krishna and Mohammad S. Obaidat, editors, *Emerging Research in Data Engineering Systems and Computer Communications*, pages 577–587, Singapore, 2020. Springer Singapore. ISBN 978-981-15-0135-7. doi: 10.1007/978-981-15-0135-7_53.

Xiangdong Wu, Xiaoyan Liu, and Yimin Zhou. Review of unsupervised learning techniques. In Yingmin Jia, Weicun Zhang, Yongling Fu, Zhiyuan Yu, and Song Zheng, editors, *Proceedings of 2021 Chinese Intelligent Systems Conference*, pages 576–590, Singapore, 2022. Springer Singapore. ISBN 978-981-16-6324-6. doi: 10.1007/978-981-16-6324-6_59.

Rui Nian, Jinfeng Liu, and Biao Huang. A review on reinforcement learning: Introduction and applications in industrial process control. *Computers & Chemical Engineering*, 139:106886, 2020. ISSN 0098-1354. doi: 10.1016/j.compchemeng.2020.106886.

Ajaykumar Kadam, Vasant Wagh, Aniket Muley, Bhavana Umrikar, and R Sankhua. Prediction of water quality index using artificial neural network and multiple linear regression modelling approach in Shivganga River basin, India. *Modeling Earth Systems and Environment*, 5:3, 09 2019. doi: 10.1007/s40808-019-00581-3.

Amanpreet Singh, Narina Thakur, and Aakanksha Sharma. A review of supervised machine learning algorithms. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1310–1315, 2016.

Alexander Heinlein and Krzysztof Postek. Lecture notes of linear algebra and optimization for machine learning, June 2022.

Grant Sanderson and Josh Pullen. Neural networks. *3Blue1Brown*, October 2017. URL `https://www.3blue1brown.com/topics/neural-networks`.

Sandra Vieira, Walter H.L. Pinaya, and Andrea Mechelli. Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications. *Neuroscience & Biobehavioral Reviews*, 74:58–75, 2017. ISSN 0149-7634. doi: 10.1016/j.neubiorev.2017.01.002.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization, Part 1.* Cambridge University Press, 2004.

KC Santosh, Nibaran Das, and Swarnendu Ghosh. Chapter 2 - Deep learning: a review. In KC Santosh, Nibaran Das, and Swarnendu Ghosh, editors, *Deep Learning Models for Medical Imaging*, Primers in Biomedical Imaging Devices and Systems, pages 29–63. Academic Press, 2022. ISBN 978-0-12-823504-1. doi: 10.1016/B978-0-12-823504-1.00012-X.

Chi-Feng Wang. The vanishing gradient problem. *Towards Data Science*, January 2019.

Kenneth Leung. The dying ReLU problem, clearly explained. *Towards Data Science*, March 2021.

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *Cornell University*, 2017. doi: 10.48550/ARXIV.1706.02515.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Vijay Kotu and Bala Deshpande. Chapter 10 - Deep learning. In Vijay Kotu and Bala Deshpande, editors, *Data Science (Second Edition)*, pages 307–342. Morgan Kaufmann, second edition edition, 2019. ISBN 978-0-12-814761-0. doi: 10.1016/B978-0-12-814761-0.00010-1.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, 2010.

Tomer Kordonsky. Loss functions. *Artificialis*, August 2021.

Frank van der Meulen. Lecture notes of statistical inference, February 2022.

Anup Bhande. What is underfitting and overfitting in machine learning and how to deal with it. *GreyAtom*, March 2018.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 06 2014. doi: 10.5555/2627435.2670313.

AssemblyAI. Regularization in a neural network — dealing with overfitting, November 2021a.

AssemblyAI. Batch normalization — what it is and how to implement it, November 2021b.

Shubhajit Datta. A review on convolutional neural networks. In Rabindranath Bera, Prashant Chandra Pradhan, Chuan-Ming Liu, Sourav Dhar, and Samarendra Nath Sur, editors, *Advances in Communication, Devices and Networking*, pages 445–452, Singapore, 2020. Springer Singapore. ISBN 978-981-15-4932-8. doi: 10.1109/ic-ETITE47903.2020.049.

Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, and Thomas Blaschke. The rise of deep learning in drug discovery. *Drug Discovery Today*, 23(6):1241–1250, 2018. ISSN 1359-6446. doi: 10.1016/j.drudis.2018.01.039.

S. Shajun Nisha and M. Nagoor Meeral. 9 - Applications of deep learning in biomedical engineering. In Valentina Emilia Balas, Brojo Kishore Mishra, and Raghvendra Kumar, editors, *Handbook of Deep Learning in Biomedical Engineering*, pages 245–270. Academic Press, 2021. ISBN 978-0-12-823014-5. doi: 10.1016/B978-0-12-823014-5.00008-9.

João Alexandre Lôbo Marques, Francisco Nauber Bernardo Gois, João Paulo do Vale Madeiro, Tengyue Li, and Simon James Fong. Chapter 4 - Artificial neural network-based approaches for computer-aided disease diagnosis and treatment. In Akash Kumar Bhoi, Victor Hugo C. de Albuquerque, Parvathaneni Naga Srinivasu, and Gonçalo Marques, editors, *Cognitive and Soft Computing Techniques for the Analysis of Healthcare Data*, Intelligent Data-Centric Systems, pages 79–99. Academic Press, 2022. ISBN 978-0-323-85751-2. doi: 10.1016/B978-0-323-85751-2.00008-6.

R. Merjulah and J. Chandra. Chapter 10 - Classification of Myocardial Ischemia in delayed contrast enhancement using machine learning. In D. Jude Hemanth, Deepak Gupta, and Valentina Emilia Balas, editors, *Intelligent Data Analysis for Biomedical Applications*, Intelligent Data-Centric Systems, pages 209–235. Academic Press, 2019. ISBN 978-0-12-815553-0. doi: 10.1016/B978-0-12-815553-0.00011-2.

Shivani Tyagi and Sangeeta Mittal. Sampling approaches for imbalanced data classification problem in machine learning. In Pradeep Kumar Singh, Arpan Kumar Kar, Yashwant Singh, Maheshkumar H. Kolekar, and Sudeep Tanwar, editors, *Proceedings of ICRIC 2019*, pages 209–221, Cham, 2020. Springer International Publishing. doi: 10.1007/978-3-030-29407-6_17.

Wonjae Lee and Kangwon Seo. Downsampling for binary classification with a highly imbalanced dataset using active learning. *Big Data Research*, 28:100314, 2022. ISSN 2214-5796. doi: 10.1016/j.bdr.2022.100314.

Jun Ren, Quan Zhang, Ying Zhou, Yudi Hu, Xuejing Lyu, Hongkun Fang, Jing Yang, Rongshan Yu, Xiaodong Shi, and Qiyuan Li. A downsampling method enables robust clustering and integration of single-cell transcriptome data. *Journal of Biomedical Informatics*, 130:104093, 2022. ISSN 1532-0464. doi: 10.1016/j.jbi.2022.104093.

Dina Elreedy and Amir F. Atiya. A comprehensive analysis of synthetic minority oversampling technique (smote) for handling class imbalance. *Information Sciences*, 505:32–64, 2019. ISSN 0020-0255. doi: 10.1016/j.ins.2019.07.070.

L.A. Wolsey. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 1998. ISBN 9780471283669. doi: 10.1002/9781119606475.

Christoph Molnar. *Interpretable Machine Learning: a guide for making black box models explainable*. Lulu, December 2022. ISBN 9780244768522.

Erik Štrumbelj and Igor Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowledge and Information Systems*, 41:647–665, 12 2013. doi: 10.1007/s10115-013-0679-x.

Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, 12 2017. doi: 10.48550/arXiv.1705.07874.

Brandon M. Greenwell, Bradley C. Boehmke, and Andrew J. McCarthy. A simple and effective model-based variable importance measure. *ArXiv*, abs/1805.04755, 2018. doi: 10.48550/arXiv.1805.04755.

M. Desrochers, J.K. Lenstra, and M.W.P. Savelsbergh. A classification scheme for vehicle routing and scheduling problems. *European Journal of Operational Research*, 46(3):322–332, 1990. ISSN 0377-2217. doi: 10.1016/0377-2217(90)90007-X.

Appendices

Appendix A

Gradients of a NN

This appendix is based on the works of Sanderson and Pullen [2017] and is a continuance of the work on computing the gradients in the backpropagation of a NN, started in Section 2.5.2.

To remind the reader what we are working on: after we fed the model the training data and the cost is determined, we try to adjust the weight matrices and bias vectors in such a way, that the model will predict the training data better. This is done, by computing the local gradients for each node j in layer l: $a_j^{(l)}$. Computing the gradients of the weight and bias values with respect to the cost function can be done using the chain rule. We can divide the gradients into three parts:

$$\frac{\delta C^i}{\delta w_{kj}^{(l)}} = \frac{\delta z_j^{(l)}}{\delta w_{kj}^{(l)}} \cdot \frac{\delta a_j^{(l)}}{\delta z_j^{(l)}} \cdot \frac{\delta C^i}{\delta a_j^{(l)}}$$

$$\frac{\delta C^i}{\delta b_j^{(l)}} = \frac{\delta z_j^{(l)}}{\delta b_j^{(l)}} \cdot \frac{\delta a_j^{(l)}}{\delta z_j^{(l)}} \cdot \frac{\delta C^i}{\delta a_j^{(l)}}$$

The first two parts in each chain rule equation are easy to compute by taking a good look at the equations shown in Figure 2.10. The derivative of the activation function is of course dependant on the choice of activation function. Therefore, an activation function is chosen of which the derivative is easily determined.

$$\frac{\delta z_j^{(l)}}{\delta w_{kj}^{(l)}} = a_k^{(l-1)}$$

$$\frac{\delta z_j^{(l)}}{\delta b_j^{(l)}} = 1$$

$$\frac{\delta a_j^{(l)}}{\delta z_j^{(l)}} = \alpha'(z_j^{(l)})$$

Now we only have to determine $\frac{\delta C}{\delta a_j^{(l)}}$. When $l = L + 1$, or in other words, when l is the output layer, then we can easily compute this last derivative. Remember $a_j^{(l)}$ is the value of the j-th node in layer l. If this is the output layer, we also used the notation $\hat{y_j}$, the j-th entry of the prediction of the model. This gives, together with the MSE loss function:

$$\frac{\delta C}{\delta a_j^{(l)}} = \frac{\delta C}{\delta \hat{y_j}} = 2(\hat{y_j} - y_j)$$

in which y_j is the j-th entry of the label of this data instance. And we can again see that this is easy to be computed with the data we have.

However, if we look at any other layers, we see the following happen. A neuron of any other layer then the output layer, influences the cost function in many different ways. If we only take a look at the L-th layer of the NN, we see that one neuron there, has a connection to all the output layer neurons, and therefore influences all the output layer neurons. This makes the derivative equal to:

$$\frac{\delta C_0}{\delta a_j^{(L)}} = \sum_{j=1}^{d_L} \frac{\delta z_j^{(L+1)}}{\delta a_j^{(L}} \frac{\delta a_j^{(L+1)}}{\delta z_j^{(L+1)}} \frac{\delta C_0}{\delta a_j^{(L+1)}}$$

In which we already know the last term. This is an iterative process. Each neuron in the network influences all neurons in the next layer and therefore influences the cost function. In general this makes the derivative of a neuron of layer l equal to:

$$\frac{\delta C_0}{\delta a_j^{(l)}} = \sum_{j=1}^{d_l} \frac{\delta z_j^{(l+1)}}{\delta a_j^{(l}} \frac{\delta a_j^{(l+1)}}{\delta z_j^{(l+1)}} \frac{\delta C_0}{\delta a_j^{(l+1)}}$$

And this is also why it is called backpropagation. You start computing at the last layer: the back. And from there, you work back layer by layer, because in this way all the information needed is already at hand.

Appendix B

Feature analysis additional input features model

In this appendix we include the figure containing all the SHAP-values and the table containing all the Greenwell numbers for the model with additional input features, as described in Section 4.6. For convenience, they are not included in Section 4.6.2, but are shown here.

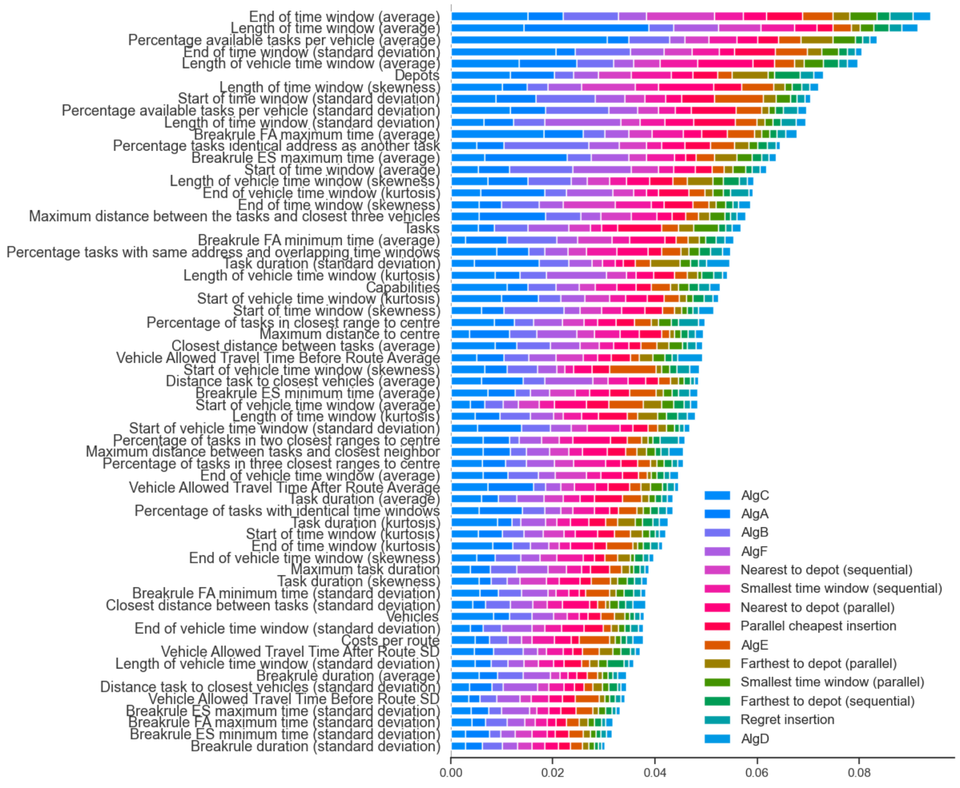


Figure B.1: SHAP values for every algorithm for the additional input features model

Table B.1: Greenwell numbers for all features and algorithms for additional input features model

(a) $*10^-3$

	AlgA	AlgB	AlgC	AlgD	AlgE	AlgF	FTD (par)	FTD (seq)	NTD (par)	NTD (seq)	PCI	RI	STW (par)	STW (seq)	Av
Number of vehicles	0.2	0.9	3.5	0.4	0.9	0.9	2.6	0.0	2.5	1.1	3.3	0.4	0.1	1.9	1.3
Number of tasks	0.8	0.7	2.3	0.9	1.4	2.0	0.8	0.1	1.2	1.3	3.8	0.1	1.1	0.3	1.2
Number of capabilities	1.5	1.2	3.2	0.6	0.8	0.6	0.5	0.4	0.5	0.7	0.2	0.6	0.7	0.4	0.8
Number of depots	1.7	0.9	3.9	0.1	0.2	0.5	1.7	1.8	0.4	1.4	0.9	0.1	0.2	1.6	1.1
Start of time window (Av)	0.3	2.4	0.5	0.1	0.3	1.3	0.4	0.3	0.5	1.3	0.8	0.3	0.6	0.2	0.7
End of time window (Av)	1.0	1.7	3.1	0.6	0.6	0.7	0.6	0.5	0.6	1.8	0.5	0.7	0.7	1.0	1.0
Length of time window (Av)	3.7	1.3	2.2	0.6	0.3	0.7	0.2	0.3	0.6	1.2	0.2	0.4	0.5	1.3	1.0
Start of time window (SD)	1.2	1.4	1.3	0.2	1.1	0.6	0.3	0.0	0.0	0.4	0.8	0.1	0.4	0.3	0.6
End of time window (SD)	0.1	0.8	3.1	0.2	0.9	0.9	0.3	0.1	0.2	1.0	0.8	0.1	0.4	0.5	0.7
Length of time window (SD)	0.5	1.2	0.2	0.3	0.4	2.1	0.0	0.2	0.5	0.5	0.6	0.4	0.0	0.6	0.5
Start of time window (skewness)	0.4	3.0	1.6	0.5	0.6	0.2	0.2	0.1	1.1	0.9	0.5	0.1	0.1	0.9	0.7
End of time window (skewness)	0.9	1.6	1.6	0.2	0.5	0.7	0.1	0.3	0.3	1.7	1.5	0.3	0.4	1.9	0.9
Length of time window (skewness)	0.7	0.4	2.9	0.2	0.9	1.4	0.3	0.2	2.2	2.2	1.0	0.3	0.2	0.2	0.9
Start of time window (kurtosis)	0.5	0.3	1.8	0.0	0.5	0.2	0.4	0.1	1.1	0.6	0.0	0.1	0.4	0.2	0.4
End of time window (kurtosis)	0.3	0.4	1.5	0.2	1.1	0.3	0.1	0.2	0.6	0.1	0.2	0.3	0.1	0.0	0.4
Length of time window (kurtosis)	0.7	1.0	1.2	0.2	0.2	0.5	0.9	0.6	0.3	0.3	1.1	0.3	0.1	0.2	0.6
% of tasks identical time windows	7.8	4.2	4.5	1.2	4.6	2.6	1.1	0.3	3.5	2.1	1.6	0.8	1.2	1.2	2.6
Task duration (Av)	0.7	0.4	0.6	0.2	0.5	1.2	0.1	0.4	0.5	0.9	0.9	0.3	0.1	0.3	0.5
Task duration (SD)	1.8	0.7	0.4	1.2	0.4	0.7	1.1	0.1	0.0	0.1	0.3	0.2	0.3	0.2	0.5
Task duration (skewness)	0.2	0.1	1.6	0.1	0.6	0.0	0.4	0.2	1.1	1.0	0.1	0.3	0.0	0.1	0.4
Task duration (kurtosis)	0.3	0.4	2.8	0.2	0.2	0.4	0.9	0.3	1.1	0.1	0.5	0.2	0.4	0.3	0.6
Costs per route	0.5	1.7	2.3	1.5	5.5	1.8	0.5	0.0	1.3	1.1	0.0	0.2	0.7	2.3	1.4
Distance task to closest vehicles (Av)	2.4	0.3	1.6	0.1	0.1	2.2	0.3	0.2	0.2	0.4	0.5	0.2	0.1	0.8	0.7
Distance task to closest vehicles (SD)	0.6	0.1	0.3	0.3	0.6	2.4	0.2	0.2	0.4	0.7	0.3	0.1	0.1	0.8	0.5
Closest distance between tasks (Av)	0.6	1.8	2.6	0.3	0.7	0.7	0.4	0.1	0.4	1.3	0.2	0.6	0.7	0.3	0.8
Closest distance between tasks (SD)	0.2	0.2	0.4	0.1	0.1	0.3	0.1	0.0	0.1	0.3	0.4	0.3	0.1	0.3	0.2
% tasks identical address as another	0.4	1.8	0.1	0.2	0.3	0.2	0.4	0.2	0.3	1.0	0.0	0.3	0.1	0.1	0.4
Start vehicle time window (Av)	0.6	0.4	0.1	0.2	1.0	0.1	0.7	0.5	0.9	0.6	0.1	0.1	0.5	0.2	0.4
End vehicle time window (Av)	0.5	0.8	0.4	0.4	0.1	0.1	0.2	0.1	0.2	1.0	0.2	0.1	0.0	0.4	0.3
Length vehicle time window (Av)	1.8	0.4	1.9	0.5	0.3	0.5	0.1	0.7	1.7	0.3	0.7	0.2	0.6	1.6	0.8
Start vehicle time window (SD)	1.7	0.8	0.2	0.0	0.1	0.5	0.3	0.1	0.8	0.3	0.5	0.1	0.2	1.7	0.5
End vehicle time window (SD)	0.5	0.1	1.2	0.2	0.1	0.5	0.3	0.1	1.0	0.1	1.0	0.3	0.5	0.4	0.5
Length vehicle time window (SD)	0.2	0.1	0.0	0.1	0.1	0.3	0.2	0.3	0.1	0.3	0.3	0.2	0.2	0.4	0.2
Start vehicle time window (skewness)	1.0	0.9	2.1	0.6	2.2	0.3	0.1	0.3	0.2	0.1	0.3	0.4	0.1	0.6	0.7
End vehicle time window (skewness)	0.5	0.3	1.5	0.5	0.1	0.2	0.5	0.2	0.5	0.2	0.2	0.2	0.1	1.5	0.5

Table B.2: Number of Greenwell for all features and algorithms for additional input features model

(a) $*10^-3$

	AlgA	AlgB	AlgC	AlgD	AlgE	AlgF	FTD (par)	FTD (seq)	NTD (par)	NTD (seq)	PCI	RI	STW (par)	STW (seq)	Av
Length vehicle time window (skewness)	1.4	1.2	1.2	0.2	0.1	0.4	0.7	0.4	1.3	0.8	0.4	0.3	0.4	0.6	0.7
Start vehicle time window (kurtosis)	1.0	0.4	1.6	0.1	0.2	0.5	0.1	0.2	0.2	0.7	0.4	0.2	0.2	0.1	0.4
End vehicle time window (kurtosis)	1.6	0.1	0.9	0.1	0.1	0.9	0.1	0.1	0.3	0.3	0.7	0.5	0.3	0.2	0.4
Length vehicle time window (kurtosis)	0.4	0.6	1.2	0.0	0.2	1.5	0.4	0.4	0.5	0.3	1.0	0.1	0.0	0.1	0.5
Vehicle travel time before route (Av)	1.3	0.3	0.3	0.7	0.4	0.3	0.6	0.3	0.2	1.4	0.8	0.3	0.6	0.6	0.6
Vehicle travel time after route (Av)	1.9	0.2	1.8	0.2	0.4	0.2	0.3	0.1	1.0	0.3	0.5	0.4	0.4	0.5	0.6
Vehicle travel time before route (SD)	0.5	1.9	1.1	0.4	3.0	0.7	0.1	0.4	0.9	0.7	1.4	0.7	0.1	2.2	1.0
Vehicle travel time after route (SD)	1.9	2.2	0.8	0.4	0.7	0.1	0.6	0.3	0.5	0.4	0.3	0.2	0.5	0.9	0.7
Breakrule ES minimum time (Av)	0.7	0.2	1.3	0.2	0.5	0.4	0.2	0.1	0.1	0.9	0.5	0.4	0.0	0.2	0.4
Breakrule ES maximum time (Av)	2.1	0.7	0.5	0.2	0.4	0.9	0.6	0.1	0.3	0.2	0.2	0.0	0.4	0.6	0.5
Breakrule FA minimum time (Av)	1.2	1.7	0.4	0.2	0.2	0.4	0.5	0.3	0.7	1.4	0.2	0.1	0.1	0.2	0.5
Breakrule FA maximum time (Av)	1.4	0.1	3.5	0.4	0.8	0.6	0.4	0.2	0.1	1.0	0.7	0.3	0.1	1.1	0.8
Breakrule ES minimum time (SD)	1.2	0.4	0.9	0.1	0.6	0.3	0.4	0.1	0.3	1.1	0.7	0.3	0.3	0.7	0.5
Breakrule ES maximum time (SD)	0.1	1.1	0.3	0.1	0.8	1.2	0.1	0.1	0.5	0.3	1.6	0.1	0.3	0.4	0.5
Breakrule FA minimum time (SD)	2.4	0.9	2.1	0.1	1.4	4.3	0.2	1.0	1.7	1.5	0.4	1.1	0.2	0.1	1.2
Breakrule FA maximum time (SD)	1.8	2.0	1.1	0.0	1.2	0.2	0.8	0.6	0.6	0.0	1.0	0.2	0.5	2.2	0.9
Breakrule duration (Av)	1.9	1.9	0.3	0.5	0.4	0.8	0.1	0.1	0.9	0.9	1.4	0.5	0.6	0.3	0.8
Breakrule duration (SD)	2.2	0.1	1.0	0.1	1.5	0.5	0.3	0.3	0.5	0.6	0.8	0.3	0.1	1.1	0.7
Percentage available tasks per vehicle (Av)	0.9	0.8	5.4	0.3	0.9	0.3	1.1	0.2	1.3	0.9	0.8	0.1	0.8	0.8	1.0
Percentage available tasks per vehicle (SD)	1.0	1.3	1.4	0.4	0.8	0.0	0.2	0.0	0.2	0.0	2.2	0.4	0.1	0.1	0.6
% tasks identical address overlapping TW	0.6	0.3	0.9	0.3	0.1	0.6	0.2	0.3	0.2	0.1	0.1	0.4	0.2	0.9	0.4
% of tasks in closest range to centre	0.5	0.4	1.3	0.2	0.2	0.7	0.3	0.1	0.7	0.9	0.5	0.7	0.4	0.1	0.5
% of tasks in two closest ranges to centre	0.1	0.1	1.0	0.1	0.2	0.6	0.3	0.3	1.2	0.3	0.4	0.8	0.1	0.2	0.4
% of tasks in three closest ranges to centre	0.3	0.2	0.6	0.3	0.2	0.7	0.1	0.4	0.3	0.3	0.2	0.0	0.1	1.0	0.3
Maximum distance to centre	1.2	0.8	0.4	0.1	0.2	1.2	0.3	0.1	0.1	1.2	0.3	0.0	0.4	1.6	0.6
Maximum task duration	0.3	0.2	0.4	0.1	0.2	1.0	0.3	0.1	0.2	0.1	0.4	0.0	0.1	0.2	0.3
Max. distance tasks and closest 3 vehicles	3.9	2.9	0.6	0.8	0.3	0.9	0.8	0.3	1.5	1.7	0.3	0.2	1.0	1.6	1.2
Max. distance tasks and closest neighbor	0.3	0.1	0.5	0.2	0.2	0.2	0.2	0.1	0.5	0.3	0.0	0.1	0.1	0.1	0.2
Total	71.0	57.5	91.0	20.2	44.4	49.0	27.3	16.8	43.7	47.1	42.8	18.2	20.3	43.8	

Appendix C

18 most important input features analysis

In this appendix, we explain in short how we designed the list of 18 most important input features mentioned in Section 4.7.3. We constructed this list in the same way we constructed the list with 12 most important input features, found in Section 4.7.1. We first computed a list of 9 most important input features for each method of the feature analysis for the original model, whose results are shown in Section 4.2.2. Thereafter, we compute a list of 15 most important input features for the model with additional input features, which feature analysis is included in Section 4.6.2 and Appendix B. This led to the following lists.

We start by determining the 10 most important input features for the original model for each feature analysis method. Eventually, this results in a total of 9 most important input features. For the permutation feature importance for the original model, the 10 most important features are:

1. Start of time window (standard deviation)
2. End of time window (standard deviation)
3. Number of vehicles
4. Start of time window (average)
5. Length of time window (standard deviation)
6. Number of tasks
7. Number of depots
8. Length of time window (average)
9. Distance task to closest vehicles (average)
10. Start of time window (skewness)

For SHAP, the 10 most important features are:

1. Length of time window (standard deviation)
2. Number of depots

3. Start of time window (standard deviation)
4. End of time window (standard deviation)
5. Length of time window (average)
6. Task duration (skewness)
7. Start of time window (kurtosis)
8. Number of tasks
9. Start of time window (average)
10. End of time window (skewness)

Based on the Greenwell numbers, the 10 most important features are:

1. Number of vehicles
2. Number of tasks
3. Number of capabilities
4. Number of depots
5. Start of time window (standard deviation)
6. End of time window (standard deviation)
7. Start of time window (kurtosis)
8. Length of time window (kurtosis)
9. Percentage tasks with identical timewindows
10. Costs per route

If we compare the above three lists, we draw the following conclusion. There are 4 input features which appear on every list, and therefore, must be very important: ‘Start of time window (standard deviation)’, ‘End of time window (standard deviation)’, ‘Number of tasks’ and ‘Number of depots’. Furthermore, there are 5 input features which are included in two of the lists. Therefore, we decide to leave the list at the 9 most important input features, resulting in:

1. Start of time window (standard deviation)
2. End of time window (standard deviation)
3. Number of tasks
4. Number of depots
5. Length of time window (standard deviation)
6. Length of time window (average)
7. Start of time window (average)

8. Start of time window (kurtosis)
9. Number of vehicles

We repeat the same for the model with additional input features, but only now we determine the 15 most important input features for each method, which eventually results in a total list of 15 most important input features. For permutation feature importance, the constructed list is:

1. Percentage of available tasks per vehicle (average)
2. End of time window (standard deviation)
3. Length of time window (standard deviation)
4. End of time window (skewness)
5. Length of vehicle time window (average)
6. Breakrule FA maximum time (average)
7. Length of time window (average)
8. Number of tasks
9. Start of time window (average)
10. Number of capabilities
11. Percentage of available tasks per vehicle (standard deviation)
12. Number of depots
13. Length of time window (skewness)
14. Percentage tasks identical address as another task
15. Distance task to closest vehicles (standard deviation)

For SHAP, this results in the following list:

1. End of time window (average)
2. Length of time window (average)
3. Percentage available tasks per vehicle (average)
4. End of time window (standard deviation)
5. Length of vehicle time window (average)
6. Number of depots
7. Length of time window (skewness)
8. Start of time window (standard deviation)
9. Percentage available tasks per vehicle (standard deviation)
10. Length of time window (standard deviation)

11. Breakrule FA maximum time (average)
12. Percentage tasks identical address as another task
13. Breakrule ES maximum time (average)
14. Start of time window (average)
15. Length of vehicle time window (skewness)

Based on the Greenwell numbers, we construct the following list:

1. Number of vehicles
2. Number of tasks
3. Number of capabilities
4. Number of depots
5. End of time window (average)
6. Length of time window (average)
7. End of time window (standard deviation)
8. Start of time window (skewness)
9. Percentage of tasks with identical time windows
10. Costs per route
11. Vehicle allowed travel time before route (standard deviation)
12. Breakrule FA maximum time (average)
13. Breakrule FA minimum time (standard deviation)
14. Percentage available tasks per vehicle (average)
15. Maximum distance between the tasks and closest three vehicles

If we compare all three lists, we determine that 6 of the input features are included in all three lists. Furthermore, there are 7 input features which appear on two of the lists. If we look a bit further than the top 10, we see that 'Length of time window (skewness)' is included in the top 15 of permutation feature importance, and is in place 16 for the Greenwell number. Because this feature is somewhat important for two methods, we include it in the final top 15 as well. The same goes for 'Maximum distance between tasks and closest three vehicles', which is included for the Greenwell numbers, and is in place 16 for the permutation feature importance. This leads to a final list of 15 most important input features based on the additional input features model:

1. Percentage available tasks per vehicle (average)
2. Number of depots
3. End of time window (standard deviation)
4. Length of time window (average)
5. Breakrule FA maximum time (average)

6. Length of vehicle time window (average)
7. Percentage available tasks per vehicle (standard deviation)
8. Length of time window (standard deviation)
9. Percentage tasks identical address as another task
10. Start of time window (average)
11. Number of tasks
12. Number of capabilities
13. End of time window (average)
14. Length of time window (skewness)
15. Maximum distance between tasks and closest three vehicles

If we combine the list of 9 input feature from the original model and the list of 15 input features from the model with additional input features, this leads to a final list of 18 most important input features, as there is some overlap.

1. End of time window (standard deviation)
2. Length of time window (standard deviation)
3. Length of time window (average)
4. Start of time window (average)
5. Number of depots
6. Number of tasks
7. Start of time window (standard deviation)
8. Start of time window (kurtosis)
9. Number of vehicles
10. Percentage available tasks per vehicle (average)
11. Breakrule FA maximum time (average)
12. Length of vehicle time window (average)
13. Length of time window (skewness)
14. Percentage available tasks per vehicle (standard deviation)
15. Percentage tasks identical address as another task
16. Number of capabilities
17. End of time window (average)
18. Maximum distance between the tasks and closest three vehicles

Appendix D

Loss results per construction algorithm

In this appendix, we include the results of the grid search for each single construction algorithm, as discussed in Section 4.8.

Table D.1: Mean absolute value loss for ‘AlgA’

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.071	0.090
		40	0.072	0.083
		60	0.072	0.083
	5	20	0.074	0.091
		40	0.080	0.091
		60	0.077	0.088

Table D.2: Mean absolute value loss for ‘AlgB’

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.057	0.065
		40	0.057	0.067
		60	0.058	0.062
	5	20	0.060	0.064
		40	0.060	0.064
		60	0.060	0.065

Table D.3: Mean absolute value loss for ‘AlgC’

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.092	0.100
		40	0.090	0.094
		60	0.090	0.093
	5	20	0.092	0.104
		40	0.092	0.097
		60	0.089	0.092

Table D.4: Mean absolute value loss for ‘AlgD’

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.035	0.079
		40	0.035	0.074
		60	0.035	0.069
	5	20	0.059	0.073
		40	0.038	0.088
		60	0.038	0.072

Table D.5: Mean absolute value loss for ‘AlgE’

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.043	0.093
		40	0.043	0.082
		60	0.043	0.085
	5	20	0.074	0.074
		40	0.056	0.095
		60	0.047	0.065

Table D.6: Mean absolute value loss for ‘AlgF’

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.069	0.087
		40	0.070	0.092
		60	0.071	0.102
	5	20	0.074	0.085
		40	0.076	0.087
		60	0.076	0.078

Table D.7: Mean absolute value loss for 'Farthest to depot (parallel)'

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.034	0.083
		40	0.036	0.070
		60	0.036	0.081
	5	20	0.062	0.069
		40	0.046	0.077
		60	0.036	0.079

Table D.8: Mean absolute value loss for 'Farthest to depot (sequential)'

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.036	0.108
		40	0.031	0.074
		60	0.031	0.082
	5	20	0.066	0.075
		40	0.049	0.088
		60	0.047	0.093

Table D.9: Mean absolute value loss for 'Nearest to depot (parallel)'

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.052	0.072
		40	0.051	0.081
		60	0.052	0.065
	5	20	0.081	0.082
		40	0.056	0.073
		60	0.058	0.080

Table D.10: Mean absolute value loss for 'Nearest to depot (sequential)'

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.054	0.091
		40	0.058	0.102
		60	0.055	0.069
	5	20	0.078	0.092
		40	0.064	0.072
		60	0.063	0.065

Table D.11: Mean absolute value loss for ‘Parallel cheapest insertion’

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.057	0.092
		40	0.067	0.108
		60	0.057	0.079
	5	20	0.075	0.088
		40	0.060	0.068
		60	0.060	0.089

Table D.12: Mean absolute value loss for ‘Regret insertion’

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.030	0.090
		40	0.033	0.074
		60	0.026	0.070
	5	20	0.060	0.067
		40	0.043	0.067
		60	0.048	0.069

Table D.13: Mean absolute value loss for ‘Smallest time window (parallel)’

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.035	0.061
		40	0.036	0.067
		60	0.036	0.071
	5	20	0.054	0.071
		40	0.042	0.071
		60	0.038	0.079

Table D.14: Mean absolute value loss for ‘Smallest time window (sequential)’

Activation function	Hidden layers	Width	Loss 12 features	Loss all features
Softmax	2	20	0.056	0.075
		40	0.060	0.097
		60	0.057	0.079
	5	20	0.081	0.089
		40	0.064	0.078
		60	0.064	0.079