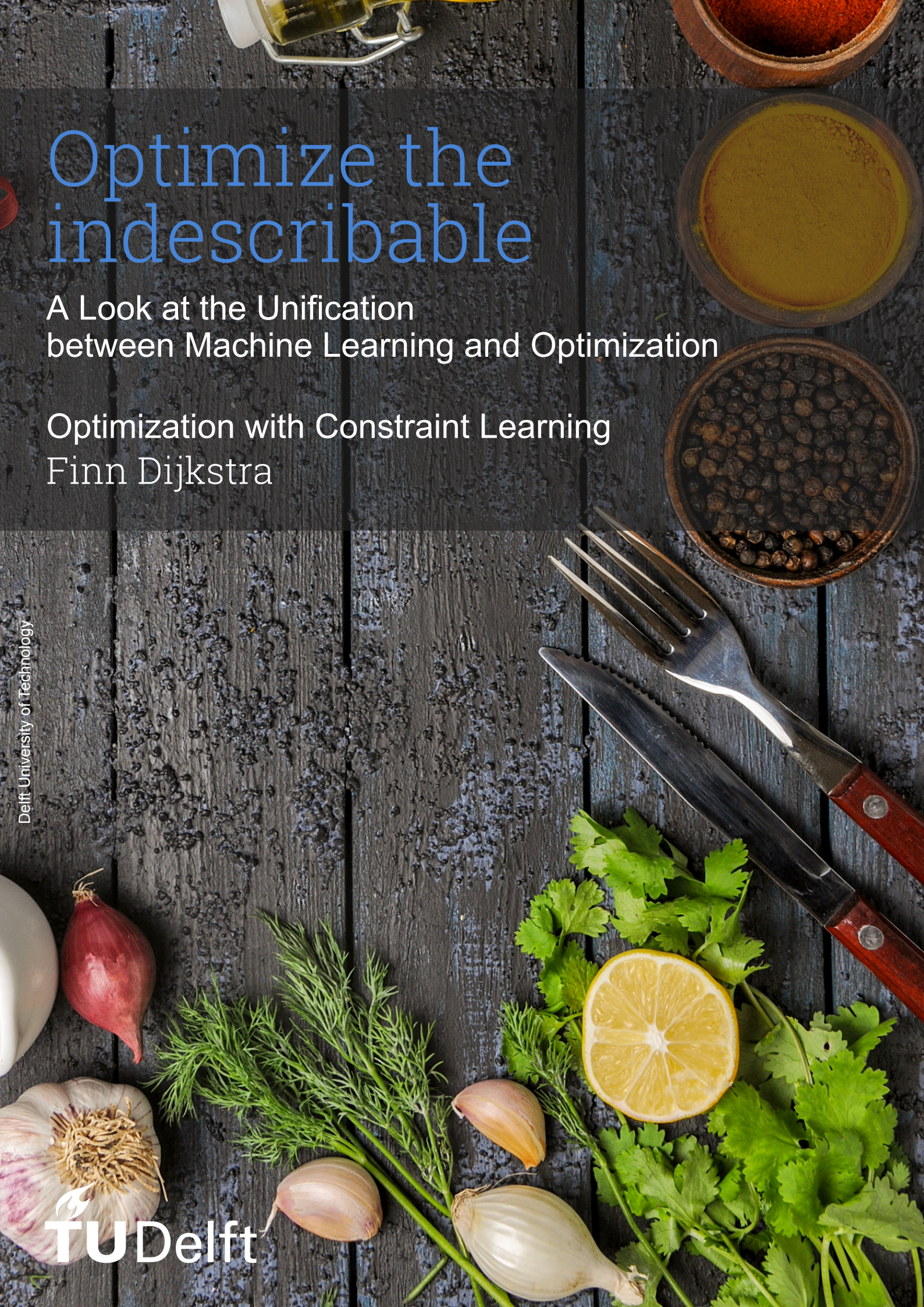

Optimize the indescribable

A Look at the Unification between Machine Learning and Optimization

Optimization with Constraint Learning

Finn Dijkstra

Delft University of Technology

TU Delft

Optimize the indescribable

A Look at the Unification between Machine Learning and Optimization

by

Finn Dijkstra

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 21, 2022 at 9:00 AM.

Student number:	4600339	
Project duration:	February 22, 2022 – July 12, 2022	
Thesis committee:	Dr. K. Postek,	TU Delft, supervisor
	Dr. ir. G. F. Nane,	TU Delft

Cover:	Food.Mediterranean food by Ali Raza under CC BY-NC 2.0 (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Abstract

Packages to encode Machine Learned models into optimization problems is an underdeveloped area, despite the advantages is could provide. The main draw of implementing Machine Learned models into optimization models, is that it allows the optimizer to better account for the human experience. Maragno D., Wiberg H. et al. constructed an implementation of the encoding with their package OptiCL. In order to verify their implementation and provide principles for (re)designing packages with similar functions, an amount of components of OptiCL were replicated within this paper. The requirements for the program were first constructed before detailing the implementation process. After the program was implemented, both OptiCL and the found program were tested in order to compare performances. Using the results and an investigation of the two implementations, a framework for encoding similar packages was provided using the insights gained. Using mathematical formulations supplied by Maragno D., Wiberg H. et al., design principles outlined in this report and research into the encoding of other Machine Learned models, other developers could construct robust packages that allow for easy integration of valuable information gained from Machine Learning into optimization problems. This in turn allows for frequently used optimization models to account for more human understanding.

Finn Dijkstra
Delft, July 2022

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
MIOCL	Mixed Integer Optimization with Constraint Learning
WFP	World Food Program
ML	Machine Learning/Learned
LR	Linear Regression
SVR	Support Vector Regression
TR	Trust Region
CH	Convex Hull
MIO	Mixed Integer Optimization
MAE	Mean Squared Error
MSE	Mean Squared Error
V-MSE	Validation Mean Squared Error
OOP	Object Oriented Programming

1

Introduction

In this paper we aim to marry the two worlds of machine learning and optimization.

For those unfamiliar with either or both of these fields, that statement might be unclear in its meaning. These fields, however, are integrated into modern life and will be encountered, unknowingly, every day. The navigation in your car, the videos that get recommended to you online, the stock of your local supermarket, all of these are primarily based on at least one of these fields. To illustrate more clearly what they entail separately and what aim to achieve in this paper, we take a closer look at the first example: the navigation in your car.

Currently, the route you are shown when asking the navigation for help, is entirely based on an optimization problem with concrete goal, variables and constraints. The computer inside calculates the route that takes the least amount of time given a few constraints, such as avoiding ferry's and tollways. While functional and efficient, such optimizers are currently limited to objectives and constraints humans can formulate, i.e. express in a formula with known values, such as total travel time based on travel time of road segments.

Machine Learning, on the other hand, is able to calculate an estimator for things humans can perceive, but not formulate. For example humans could perceive and provide a comfort rating of a route but not express a underlying logic in a formula. Once trained with various routes and comfort ratings, a Machine Learning estimator could give a comfort rating of a given route, even though that exact route has not received a human rating before.

Clearly, it would be immensely useful if we could use the Machine Learned estimators (often called ML models) in an optimizer, however little to no support is currently available. The navigation in your car could for example take into account the comfort of your commute when supplying you with your route to work. It could make sure the delivered route is the fastest possible, while making sure it's up to a certain standard of comfort, or, alternatively, calculate the most comfortable commute, while making sure you still arrive on time. The latter is not only ideal for your daily commute, but could also be used for route suggestions when you plan to make a holiday trip.

As it stands, ML models are often perceived as ”black boxes”; measurements go in, results comes out. Optimizers, however, need transparency in their constraints and objectives to solve their problem. The requirement to solve this mismatch and to enable the combined operation of ML models and optimizers thereby becomes clear: create methods to extract the driving formulas of the ML models and implement them into an optimization solver as constraints used in the optimization calculation. This is exactly what Maragno D., Wiberg H. et al. have done with their package OptiCL [1] as outlined in the paper Mixed Integer Optimization with Constraint Learning (MIOCL) [2] and what we aim to replicate here.

The goal of the paper is to provide feedback on OptiCL and deliver strategies for (re)designing packages aiming to encode ML models into optimization problems. To achieve this, we aim to replicate the function and compare the methods of MIOCL's example for the World Food Program (WFP): the goal of the program is to supply ration packets as cheaply as possible to groups of people in different cities in Syria, while making sure the ration not only supplies the recipients with their daily nutrients, but also makes sure the combination of ingredients is up to a certain standard of taste. The taste of the

rations is given a score estimated by a ML model and, using that score, the optimizer minimizes the cost.

The structure of this paper will be as follows: first the project's goal introduced above will be described further and the requirements for our program will be extrapolated, subsequently the implementation process is described, and finally the results are outlined and compared to those found in the original paper.

2

Requirements

The goal of this paper, is to supply feedback on OptiCL and provide tactics for researchers and developers aiming to (re)designing similar packages. To achieve meaningful feedback, a program will be implemented independent of OptiCL.

In the program, the aim is that it can solve the optimization problem described in the WFP example of MIOCL, ensuring it is compatible with the ML-methods, trust regions and performance measurements as described in MIOCL. All of the components will be explained further in detail in this paper.

Mission statement In the World Food Program example of the paper, the goal is to find the cheapest way to provide groups of inhabitants within different cities with their necessary daily nutrients. We aim to know what the ingredients of the found ration are, where to purchase those ingredients and what the optimum transport routes are.

Normally such an optimizer contains little more than minimizing the cost while making sure every inhabitant gets their daily dose of nutrients, i.e. a classical linear programming model combining a transportation optimization with diet requirement constraints.

The aim here however is to extend our optimizer to ensure the rations we provide are sufficiently tasty. As tastiness is not something that humans can accurately describe with formulas, the tastiness is estimated with a Machine Learned model, which shall feed into the optimizer to ensure that each food basket is sufficiently tasty.

Components Using the description above, we can break down the program into key components. These components are necessary for the program to fulfill our purpose and are divided into separate parts in such a way that implementing them part by part is possible.

The first component that should be included in the program is the basic optimizer, that is the optimizer which finds the cheapest ration delivery plan without accounting for the taste of the rations. The second component of the program will be the ability to train models using different ML methods. Furthermore, the formulas inside of the ML models need to extracted and implemented into the optimizer. In additional, it is essential to the program to make sure the optimization solutions are limited to regions where we trust the ML model, i.e. regions that are close enough to the data on which the ML model is based. Finally we need to add ways to measure the performance in both accuracy and computation time in order to compare this implementation's performance with the performance of the program developed in the paper, called OptiCL.

3

Implementation

This chapter is about the components required to replicate the function of OptiCL as provided in the World Food Program example of MIOCL. As a quick reference, these are the functions the program must fulfill as already found in Chapter 2:

1. Optimize the WFP problem, not yet account for taste.
2. Train models with ML methods.
3. Encode found ML models into the optimizer.
4. Limit the optimizer to regions where the model is trustworthy.
5. Measure the performance of the program

In the first section of this chapter a more detailed description is provided off what each of these components entails and on the characteristics they must have. Further sections describe the implementation process for each component; the sections start with necessary background information, continuing on to mathematical descriptions where necessary, and finish with explaining the process of encoding the components into Python.

3.1. The Components

In this section we will go into the different components, providing information on what it provides to the program and what is necessary to fulfill it. All components are explained in order of implementation, which means we start with the basic optimizer:

The Optimizer needs to find the lowest cost solution for the WFP problem. All optimization problems look mathematically similar: all have a function to optimize (either they aim to minimize it's value or maximize it) and several constraints limiting the amount of possible solutions. While many different sorts of optimization problems and methods to solve them exist, both this paper and MIOCL that it is based on restrict themselves to linear problems where variables can be a combination of integers and continuous variables. This type of problem, called a Mixed Integer Optimization (MIO) or Mixed Integer Programming (MIP) problem, provides a good balance between efficiency and usefulness.

Here the aim is to minimize the cost of the ration delivery plan and the constraints are here to ensure the plan is feasible and provides the recipients with all their nutrients. This optimizer essentially needs to solve two problems: find the best ingredients in a ration and find the best purchasing location and supply route given that ration. Due to the extensive amount of work already done in the optimization field, both of these problems already have worked out examples that need little adaptation. In addition to the examples, plenty of Python packages are available to solve such optimization problems. In the implementation section an explanation will be provided on the mathematical structure of an optimizer and on formula and parameter acquisition, before providing some notes on the process of encoding the optimizer in the program.

The Machine Learning models are necessary to estimate values that humans are unable to formulate. It can be used to estimate a vast amount of values, from incredibly complex phenomenons, such as the flow of visitors through a museum, to more abstract values, such as taste or comfort. Such a model is trained on a large data set using a Machine Learning method. Due to the linearity restriction of an MIO problem, the amount of usable ML methods is limited. These methods all need to be able to be formulated linearly.

Encoding ML models into the optimizer needs additional effort, even if the ML models are linear. The optimizer cannot access the linear formulas that drive the models and as such cannot use the methods that normally solve linear problems. This means that the parameters need to be extracted out of the models and encoded into the optimizer. This process of extraction and encoding is the main function that OptiCL provides and the core functionality we aim to replicate in this paper.

As the aim is to provide the same functionality as OptiCL, the specific methods chosen to be encoded are: Linear Regression, Support Vector Regression, Regression Trees. All of these methods will be described separately in the implementation section, starting with a general description and continuing onto the mathematical descriptions and encoding process.

Trust regions are necessary to keep the solution confined to spaces where the model can accurately predict the modeled value. A common method used to create these regions, and the method used in this paper, is constructing a Convex Hull (CH) around the data set on which the model is trained. A Convex Hull is the smallest region around a set, such that every point of the set is within the region and the region is convex, in other words, the region has no angle exceeding 180 degrees.

While a single Convex Hull already limits the degree the prediction can differ from the true value, a data set will likely still contain areas within the hull, where little to no data has been collected. To combat this a technique called Clustering is often used; clustering creates different groups of data points, such that every group is as densely packed with data points as it can be. After the data set is separated into groups, separate Convex Hulls are created around each of the groups. While this limits the amount of forms our solution can take even further, it is expected to provide a more accurate prediction.

Performance Measurement is necessary to compare the accuracy and the computation time of different ML methods and the use of trust regions. Additionally it is necessary to achieve the main goal of this report: study the efficiency of OptiCL and discover potential oversights or areas of the package with simple yet effective improvements. We measure the accuracy of the predictions and the computation time, with and without trust region of all ML methods using this implementation and OptiCL. To ensure robustness, this step is done iteratively, changing parameters of the optimizer and the subset of the data, on which the ML models train, at each step.

3.2. Implementation Details

3.2.1. The Optimizer

To implement the optimizer, two things are necessary: a mathematical description of the problem and a solver in which the found description is encoded. Luckily plenty of solvers are available for these sorts of problems. We have chosen to use the solver Gurobi, which is the same solver as used MIOCL. In the coming paragraphs the process of finding the mathematical description will be detailed, along with the mathematical description itself. After we arrive at the description of the problem, the key formulas to be encoded into Gurobi will be highlighted.

Mathematical description: As a reminder: the goal of the basic model is to minimize cost of supplying rations to communities across different cities while meeting certain nutrition requirements. The World Food Program requires information on the ingredients and their quantities in the rations, where to purchase them and how to transport these ingredients to their destinations, as well as related costs.

From the mission statement, it can be determined that the solver needs to find the minimal value for the objective function, which is described as the combined cost of purchasing of the ingredients in the rations and the shipping of said ingredients to the destination sites. The key constraint is the minimum required nutritional value of each ration.

As often is the case, larger optimization problems can be broken down into a combination of multiple smaller problems for which descriptions are already readily available. Here problem can be split into two well known problems; a Stigler diet problem and a multi-commodity flow problem.

The Stigler diet problem optimises diet costs (i.e. purchasing price) while ensuring the sum of all nutrients in a daily diet equals or exceeds the recommended daily nutritional requirement. The assumption is that the supplied ration is the main or only nutrition of the recipient. The tastiness or palatability constraint of a ration will be fed into the Stigler diet problem from the ML learned model at a later stage.

The multi-commodity flow problem optimises transport costs based on a description of the possible transportation network, symbolizing every city with a point, commonly called a node. Any path that leads from one city to the next, is described as link between the two respective nodes and is called an edge. The resulting transportation costs over this path then can be described as a number assigned to that edge, called flow. The keep our network manageable we define three types of nodes: starter (where ingredients are purchased), transport and destination nodes.

Two specific constraints for the multi-commodity flow problem are the following: First, the amount of ingredients delivered to a city needs to equal its demand, i.e. ensure a ration for every person for every day. Secondly, there should be no excess remaining so any cities only used for transfer of ingredients need to distribute as much of each ingredient as they receive and all purchased quantities need to leave the starter cities.

A specific realization necessary to join the two separate problems is the fact that the cost of the ingredients could differ depending on the city in which they are purchased. As the objective function is the sum of all transport and ingredient acquisition costs, the transport costs of the multi-commodity flow problem need to be combined with the purchasing costs of Stigler diet problem.

The problem can now be described as in the set of equations (3.1a)-(3.1g), using the sets from Table 3.1 and the variables and parameters from Table 3.2:

Symbol	Set Description
N	Set of all Nodes in the network
N_S	Set of all Starting Nodes (cities where the ingredients are purchased)
N_T	Set of all Transfer Nodes (cities where shipments can travel through)
N_D	Set of all Destination Nodes (cities with the communities that need rations)
I	Set of all ingredients for the rations
V	Set of all nutritional values we measure

Table 3.1: Set descriptions

Symbol	Description	Unit
	Variables	
X_k	Weight of ingredient k in a single ration	g
f_{ijk}	Weight of ingredient k transported from node i to node j	t
	Parameters	
y	Conversion from grams to metric ton	-
t	Days that the shipment of rations should last	d
d_j	Demand (amount of recipients) at node j	-
r_n	Daily requirement of nutrient n	mg
v_{kn}	Value of nutrient n in ingredient k	mg/g
c^p_{ik}	Cost of purchasing a metric ton of ingredient k at node i	$\$/t$
c^t_{ij}	Cost of shipping a metric ton from node i to node j	$\$/t$

Table 3.2: Variable and Parameter descriptions

$$minimize: \sum_{i\in N}\sum_{j\in N}\sum_{k\in I}(f_{ijk}c^T_{ij}) + \sum_{i\in N_D}\sum_{j\in N}\sum_{k\in I}(f_{ijk}c^P_{ik}) \tag{3.1a}$$

$$Given: \sum_{k\in I} X_k \cdot v_{kn} \geq r_n \qquad \forall n \in N \tag{3.1b}$$

$$\sum_{i\in N_S\cup N_T}(f_{ijk}) = t \cdot d_j \cdot X_k \qquad \forall j \in N_D \tag{3.1c}$$

$$\sum_{i\in N_S\cup N_T}(f_{ijk}) = \sum_{i\in N_T\cup N_D}(f_{jik}) \qquad \forall j \in N_T \tag{3.1d}$$

$$X_k \geq 0 \qquad \forall k \in I \tag{3.1e}$$

$$f_{ij} \geq 0 \qquad i,j \in N \tag{3.1f}$$

$$f_{ijk} = 0 \qquad \forall k \ \wedge \ (i = j \vee j \in N_S \vee i \in N_D) \tag{3.1g}$$

Encoding process As mention before, the solver used in this project is Gurobi. In addition to the standalone program, Gurobi has support for multiple programming languages, including the language used in this project: Python. To ensure Gurobi uses methods for linear problems, it is required to encode the problem in the form described (3.1a)-(3.1g), as rearranging an equation likely leads to a non-linear description. After understanding the environment sufficiently, implementation of the problem is straight forward. After testing the optimizer by checking the solution for different parameters, it was concluded that the optimizer was successfully implemented.

3.2.2. Machine Learning models

As explained in earlier paragraphs, machine learning is the process of developing a model that mimics human thought processes, especially useful when it is humanly not possible to express a relation between input and output directly in a formula. There are different methods to develop a model, however, within this paper, all methods used are developed by a process called supervised learning. Supervised learning start with collecting large amounts of data which contains measured output based on a given input. In the WFP problem we require a data set with the surveyed taste score for a given ration composition. After the training data has been acquired, the model is calibrated on that data-set. After the model finished training its prediction for the output, an output can be predicted using the model when supplied with new input. The training data of the WFP problem consists of 5000 measurements, each with a taste score between 0 and 1.

3.2.2.1. Linear Regression

A Linear Regression model can be best visualized starting with two dimensions. To illustrate, we shall use sugar content of a product and taste of a product for input and output respectively. We collect the data from different products by surveying data. Then find a straight line such that the data points are as close as possible to the line (as seen in figure 3.1 and 3.2). The height (the taste score) of the line, at 0 sugar content is called the intercept. The slope of the line is then called the taste coefficient for sugar. Extending to more dimensions now becomes more clear. Say we add protein content as an extra input, then we find a plane such that data points have the least distance, the intercept is found at 0 for both values and the slope in the direction of a dimension is the taste coefficient of that dimension (see figure 3.3). Embedding a minimum on taste then equates to restricting the optimization problem to the area underneath the plane, as seen in figure 3.4. Higher dimensions function the same, with the resulting structure of the model forming a hyperplane.

Mathematical description Coefficients and an intercept describe the hyperplane as in equation (3.2). Those coefficient can be found by finding the values where they attain a minimal Mean Squared Error (MSE) with regards to the data set as described in equation (3.2). This method stimulates a more even distribution of errors in comparison to finding coefficient by their absolute errors, as large singular errors are punished more heavily than multiple smaller errors.

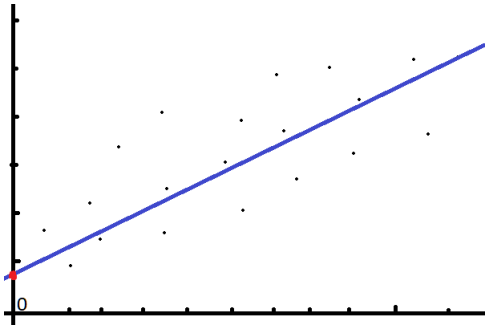

Figure 3.1: A linear regression of a sample data set

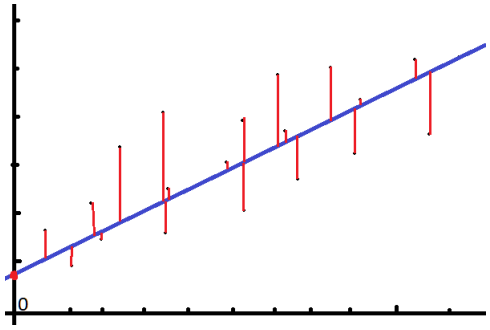

Figure 3.2: The same linear regression with errors in prediction illustrated

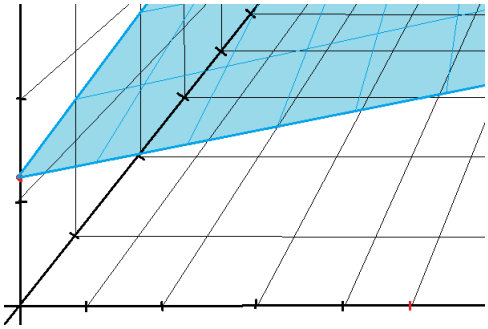

Figure 3.3: A sample 3d linear regression

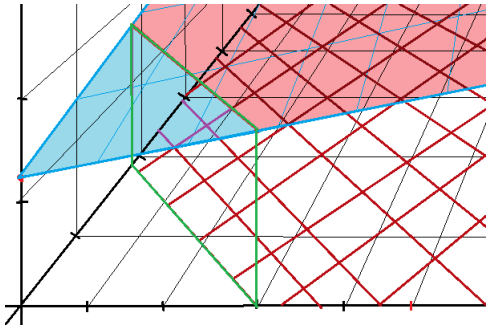

Figure 3.4: Restriction of input to fulfill constraint

$$taste(x) = b + \sum_{i=1}^{n} c_i x_i \tag{3.2}$$

$$Minimize \sum_{(x_i,y_i)\in D} (y_i - taste(x_i))^2 \qquad \text{with } D = \text{the training data} \tag{3.3}$$

Encoding process The encoding process is fairly simple: the equations of (3.4) are encoded into Gurobi.

$$y = b + \sum_{i=1}^{n} c_i x_i \tag{3.4a}$$

$$y \geq taste\_requirement \tag{3.4b}$$

3.2.2.2. Support Vector Regression

The Support Vector Regression, like Linear Regression, results in a hyperplane close to all data points. The difference between the two methods lies with the properties of the resulting hyperplanes. Linear Regression results in a hyperplane as close as possible to all points, while Support Vector Regression aims to be more accurate for most data point and stimulates a more evenly distributed dependency of every input dimension.

Mathematical description and Encoding process An SVRs result in a hyperplane like Linear Regressions, an SVR can be mathematically described as in (3.2) and encoded with the constraint in (3.4). To achieve the desired properties of the SVR, the coefficients are found by minimizing the equation (3.5). D is the data set again, the other unfamiliar terms are n, which symbolizes the dimension of our input, and C, which is a variable determined when encoding. The SVR prioritizes minimizing errors for large value for C and prioritizes even distribution of input dimension dependency. Note that using absolute errors punishes singular outliers less heavily, as mentioned earlier.

$$Minimize \sum_{i=1}^{n} c_i^2 + C \sum_{(x_i,y_i)\in D} |y_i - taste(x_i)| \tag{3.5}$$

3.2.2.3. Regression Tree

The final model, like a tree, starts at a base point and keeps splitting the further in the tree you go. At every branching point an inequality is evaluated, if the input is greater or equal to the splitting value it will go to the right, if its less then the splitting point than it goes to the left. Once the end of the tree, called a leaf, is reached, we predict the score of the input to equal the value associated with the found leaf. A diagram illustrating the sorting process is shown at figure 3.5. Because the tree sorts the input into different groups, assigning a value to each of the resulting groups, it can be visualized as in figure 3.6. This implies adding Regression Tree constraints in an optimizer is the same as restricting the input to lie within the leaf groups, where the associated value fulfills the constraint.

Mathematical description The mathematical description of a Regression Tree revolves around indicator functions, functions that result in 1 if the input fulfills a condition and 0 if the input does not. These functions are incredibly useful to show what leaf an input belongs to and to show if in input fulfills the inequality at a splitting point. While an indicator function is normally denoted by $1_A(x)$ or $I_A(x)$, the following paragraphs use $i_a(x)$ for the inequality at a splitting point a, while keeping $I_n(x)$ to symbolize if x belongs to leaf n. As all inequalities must be linear functions and our input can be represented as a vector, the inequality at a splitting point can be described as a matrix multiplication (see (3.6a)) and the indicator function i_a follows from that description (see (3.6b)).

$$M_a \cdot x \geq b_a \tag{3.6a}$$

$$i(x)_a = \begin{cases} 1 & M_a \cdot x \geq b_a \\ 0 & M_a \cdot x < b_a \end{cases} \tag{3.6b}$$

Following the description for the inequalities a unique path can be constructed to reach each leaf. An example path has been provided in equation (3.7a) to reach leaf_2 of figure 3.5. Using the paths found, a description for $I_n(x)$ can be found as shown in (3.7b).

$$p(l_2) = \{i(l_2)_{f(x)\geq a} = 0,\ i(l_2)_{g(x)\geq b} = 0,\ i(l_2)_{\mathrm{i}(x)\geq d} = 1\} \tag{3.7a}$$

$$I(x)_n = \begin{cases} 1 & \forall i(n)_a \in p(n) : i(n)_a = i(x)_a \\ 0 & else \end{cases} \tag{3.7b}$$

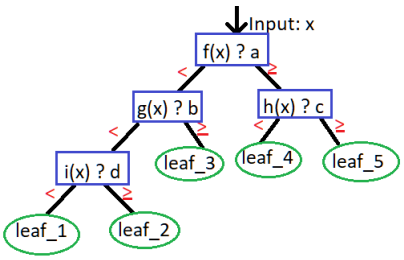

Figure 3.5: A diagram of how a tree sorts input onto a leaf

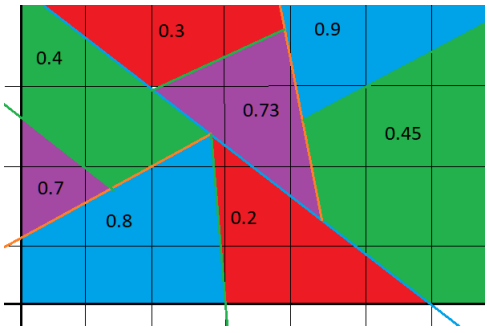

Figure 3.6: An example on how a Regression Tree classifies input and assigns values

As any input can only be sorted into a single leaf, the Regression Tree is described with the equations in (3.8), with L the set of all leaves and c_n the value of leaf n.

$$taste(x) = \sum_{n \in L} c_n I(x)_n \tag{3.8a}$$

$$with \sum_{n \in L} c_n = 1 \tag{3.8b}$$

Encoding process Using the mathematical description of a Regression Tree, there exist essentially 3 methods to implement the model. The first method is the simplest, yet least efficient; Gurobi has support for binary variables, therefore all equations from (3.6b) till (3.8b) can be implemented directly into the model, adding $taste(x) \geq taste\_requirement$ as well.

The second method is to remove all leaves, that are insufficient for the taste requirement, before implementing into the model. This eliminates the need to add (3.8a) and the taste constraint directly into the model. Removing insufficient leaves is done quickly and results in the solver containing less variables and having to calculate less equations. Due to the solver calculating every equation a large number of times, this saves a significant amount of time, especially for stricter taste constraints.

The final method follows a similar logic to the second, except the optimization model is solved for every feasible leaf separately. After the optimal solutions are found for each leaf, the costs of the solutions are compared resulting in the cheapest becoming the final solution.

In the implementation, the second method has been used to ensure the encoding of the ML model into the optimization problem remains separated for adaptability of solving methods.

3.2.3. Trust Region

Constructing Convex Hulls to define trust regions is a non-trivial task for this data set. The data set, more specifically a set where each data point is a ration description with an accompanying taste score, has an input with 25 dimensions, each a potential ingredient of the ration. Efficiency of Convex Hull descriptions should be considered, as higher dimensional data sets often create CHs with a large amount of descriptive vertices. A traditional description of a hull would be dependant by describing the interior by a convex combination of all its border points, resulting in an equation that becomes explosively complex as dimensions of the data set and border points of the hull increase.

The use of packages available online to compute the CH is, to the best of our research and understanding, unavailable. Either the package is only able to calculate data that is 2 or 3 dimensional, or the package (such as the one provided by SciPy) relies on Qhull to calculate the hull. After researching the topic, a paper by A. Siegel [3] was found. In that paper, it was found that Qhull is unable to calculate hulls of large dimensional data sets; the program ran out of memory when calculating a 10 dimensional hull with 1000 supplied data points. The 25 dimensions with a similar amount of data points given by the WFP problem would have the same result, which our testing confirmed. While Siegel did provide an alternative algorithm to calculate higher dimensional hulls, no package has been provided and we were unable to implement the found algorithm as significant developmental resources had been invested into developing a function independently before finding the paper.

Failure to calculate the Convex Hull After discovering that standard packages were unsuitable for the needs of the problem, development of a function began that was able to calculate higher dimensional Convex Hulls. While constructing the function, several methods were tested, all aiming to supply a CH formulated such that the optimizer could process the hull more efficiently. After several attempts where roadblocks were met, a final approach was formulated: the data set would be repeatedly reduced in dimensions by translating the entire set over two vertices, that are guaranteed to be border points. After the entire set was translated, the two translating vertices for every dimension would be used to iteratively increase dimensions back to full. At every dimension, points would be added that lie outside of the incomplete description of the hull, resulting in a complete hull when the dimension was back to full. The border points were stored into triangular facets. As these facets are triangular in nature, it should allow the optimizer to restrict itself to the given facets by relatively uncomplicated matrix multiplication (see equation (3.9)).

$$F = \{\mathbf{v}_0, \mathbf{v}_1, ..., \mathbf{v}_n\} \tag{3.9a}$$

$$\mathbf{w}_i = \mathbf{v}_i - \mathbf{v}_0 \qquad \forall 1 \leq i \leq n \tag{3.9b}$$

$$M = \begin{pmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ . \\ . \\ \mathbf{w}_n^T \end{pmatrix} \tag{3.9c}$$

$$\mathbf{x} \in F \iff \mathbf{x} - \mathbf{v}_0 = M \cdot \begin{pmatrix} c_1 \\ c_2 \\ . \\ . \\ c_n \end{pmatrix} \iff M^{-1}(\mathbf{x} - \mathbf{v}_0) = \begin{pmatrix} c_1 \\ c_2 \\ . \\ . \\ c_n \end{pmatrix} \tag{3.9d}$$

$$with\ c_0 = \sum_{i=1}^{n} c_i \tag{3.9e}$$

$$0 \leq c_i \leq 1 \qquad \forall 0 \leq i \leq n \tag{3.9f}$$

While ultimately unable to implement this method due to lack of resources, interest in pursuing this problem has not reduced and development will likely continue after this paper has been published.

Alternative restriction of the optimizer While unable to implement the Convex Hull, the need to restrict the optimizer close to the data set remains. To ensure sufficient time remained to implement remaining features, a simple restriction has been chosen. In this implementation, the optimizer will be restricted to the minimum and maximum value that an ingredient attains on the given data set.

3.2.4. Performance Measurement

The main goal of this paper is to replicate as much features of OptiCL as feasible and afterwards compare the two implementations. Without measurements, comparisons have no qualitative evidence to back them up. To deliver such evidence, two things need to be discussed: setup and measures.

Setup To ensure the measurements are robust, many different data sets and model descriptions will be tested. Each taste score and trust region method will be measured over 1000 iterations, every iteration will be supplied with a subset of the Machine Learning training data and parameters that lie between 0.8 and 1.2 times their original value, with the exception of nutrient requirements and nutrient values of the food. The exceptions have been decided as it could deliver a skewed taste score. Additionally, Python multiprocessing is used in order to speed up data collection. Lastly two control groups are constructed, one where no taste is accounted for and another patchwork solution, where the taste score is substituted the percentage that sugar accounts for all calories in order to simulate human estimation.

Measures The two properties, that are important when considering wider use of constraint learning, are efficiency and accuracy. To measure efficiency, all computation times are measured. Accuracy we measure two fold: once against the complete training set to measure accuracy within expected input and the predicted taste of the final solution will be compared to the ”true” taste score (determined by the function that generated the training data set) to measure the necessity of the trust region.

As the project setup has now been detailed, the results can be found on the following page.

4

Results and Discussions

In this chapter the results and observations about the performance of all different methods and implementations are outlined. After the specific results are discussed, a larger look at the implementation of OptiCL will be taken in comparison using the knowledge gained as result of implementing the replication. All points of improvement that are provided relate specifically towards the implementation of OptiCL, but many revelations are applicable to any implementations that aim to integrate ML-methods into optimization models.

4.1. Results

The results of the tests can be found be found in Table 4.1 and Table 4.2.

Test setup description: All test are done on a desktop computer running Windows 10. The computer has 16GB of RAM and an AMD Ryzen 5 5600X. While Python multiprocessing was used to increase the speed of testing, no virtual memory shortage was observed, which could lead to inaccurate results. The test consists of 1000 iterations, at each step all combinations of trust region and ML methods are tested. As a control the model is solved at each step without accounting for taste and with accounting for taste by a patchwork formulation of taste created by a human. In order to test the implementations under differing circumstances, every iteration step is supplied with 1000 point sample of the larger 5000 point data set for training the taste score. Additionally at every step, all parameters, with the exception of nutritional requirements and nutritional values of food, are varied from 0.8 to 1.2 times their original value. The Mean Squared Error with regards to the 5000 point data set, the absolute error between the predicted and true taste score of the optimization and the computation time of the entire program is measured and the mean and standard deviation are displayed in Table 4.1 and Table 4.2 respectively.

Before addressing other ML methods, a small note on the OptiCL Linear Regression: an inspection has been done on both OptiCLs implementation and the implementation of OptiCL into the performance measurement function as both the MSE and the absolute error with regards to the solution are of such a large quantity, that it becomes unusable as the true taste score attains value between 0 and 1. The inspection, however, was not able to deliver the reason for the inaccuracy of the ML model.

ML method	Mean V-MSE	MAE	MAE-TR
Without Machine Learning			
No Taste	*NaN*	*NaN*	*NaN*
Patchwork	0.105 (NaN)	0.767 (0.954)	0.289 (0.036)
This Implementation			
LR	0.019 ($1.41 \cdot 10^{-4}$))	0.047 (0.080)	0.053 (0.050)
SVR	0.020 ($2.35 \cdot 10^{-4}$))	0.048 (0.069)	0.044 (0.052)
Tree	0.016 ($6.99 \cdot 10^{-4}$))	0.607 (0.873)	0.186 (0.089)
OptiCL			
LR	3.009 (0.274)	1.627 (0.115)	1.672 (0.096)
SVR	0.298 ($2.02 \cdot 10^{-4}$))	0.618 (0.481)	0.537 (0.050)
Tree	0.108 (0.026)	0.567 (0.736)	0.194 (0.075)

Table 4.1: Taste score performance (Standard deviation) of 1000 iteration and 1000 elements in the sub set

ML method	Time	Time-TR
Without Machine Learning		
None	0.044 (0.004)	0.047 (0.006)
Patchwork	0.045 (0.006)	0.048 (0.007))
This Implementation		
LR	0.052 (0.007)	0.055 (0.008)
SVR	0.188 (0.014)	0.191 (0.014)
Tree	0.369 (0.028)	0.366 (0.029)
OptiCL		
LR	1.248 (0.090)	1.249 (0.090)
SVR	0.287 (0.020)	0.288 (0.018)
Tree	5.098 (0.391)	5.115 (0.397)

Table 4.2: Computations times (Standard deviation) of 1000 iteration and 1000 elements in the sub set

4.1.1. ML method performances

The best two performing ML methods are this papers implementation of Linear Regression and Support Vector Regressions. Both performing similar in accuracy, with the SVR specifically performing better than Linear Regression in providing a solution with accurate taste score when restricting the solution to a trust region. This is balanced by the computational speed that the Linear Regression provides, only slightly slower than no taste constraint and patchwork taste constraint. While our implementation of a Regression Tree and OptiCLs implementation for Support Vector Regression and Regression Trees have errors less than 0.65, it is important to remember that the true taste score only attains values between 0 and 1. In this context absolute errors greater than 0.1 already disrupt the usefulness of the optimized solution, due to unreliability of taste.

Min/Max constraint vs Convex Hull. In order to measure the difference between this papers implementation of a trust region and OptiCLs implementation of a CH as trust region, another, smaller, test has been done. In this test OptiCL was tasked to implement a CH using 50 first data sets and parameter variations of the larger test. The results can be seen in Table 4.3. When comparing the results of the two trust region methods, the hull on average adds an extra 0.5 seconds computation time and performs slightly worse in accuracy, which should be attributable the selection of parameters and training sub sets. The reasoning as to why the accuracy should not be worse is that any region defined convex

combination of data points, regardless of formulation or elements removed from an original data set, at minimum restricts the region the minima and maxima of the data set on each dimension.

OptiCL with CH		
ML method	**MAE**	**Time**
LR	1.630 (0.087)	1.824 (0.040)
SVR	0.585 (0.044)	0.864 (0.030)
Tree	0.228 (0.109)	5.662 (0.248)

Table 4.3: Measurements of OptiCLs Convex Hull (Standard deviation) of 50 iteration and 1000 elements in the sub set

OptiCL vs this Implementation. As seen in Table 4.1 and Table 4.2, OptiCL currently performs worse in both accuracy and computational time in comparison to this implementation. After inspection of the OptiCL code, the main difference between the two implementations was found. This paper uses base parameters provided by SciPy, while OptiCL uses a parameter grid to find the optimal parameters to train the model. This logically causes longer computation times, yet should deliver a more accurate solution. A possible cause for the inaccuracy is that OptiCL might overfit the model to the data set; the data points within the sample training sub set could be predicted more accurately, however input needs to remain incredibly close to those data points to remain accurate. This line of reasoning is supported by the models performance on the larger data set. The Validation MSE tests the models performance on a predetermined set of reasonable inputs outside of the training set and OptiCL achieves lower accurate than this implementation where no parameters have been optimized.

4.2. Discussion

The discussion is split into two part: one part evaluating model specific performance of different taste constraint implementations and the second part evaluating the implementation of OptiCL.

4.2.1. Model evaluation

While both Linear Regression and Support Vector Regression perform sufficiently well on the model, Regression Trees grow to be useless due to their accuracy when input is not within areas with dense training data. This is further supported by the improvement in accuracy found when evaluating increasingly similar data (from solution error, to restricted error, to MSE of the original training data).

Human performance Surprisingly, the patchwork human estimation performs similarly to a Regression Tree in accuracy, without the need for models training. The respectable performance, despite the lack of parameter/formula optimization, could stem from a combination of two characteristics. The formulation was created by a human with, like all other humans, years of amateur experience on judging food. This characteristic leads to a passively pre-trained estimation, regardless of how primitive the formulation was. The other characteristic is that the formulation does not directly rely on the ingredients, but the relation of underlying nutritional values: the taste score is corresponded 1 to 1 with the ratio of calories that originate from sugar. Regardless of the cause of the performance, it highlights the contribution human interaction to improve Machine Learned models, which could consist of supplying different input formulations and supplying adequately accurate starting parameters. The validity of this observation is an area that is suitable for additional research.

4.2.2. Implementation evaluation

While the mathematical background of OptiCL is sound, the results of the tests, along with the implementation process, highlight some points of improvements for OptiCL as a package. This advice can be used as implementation policies for any other programs aiming to proved integration of machine learning into optimization for public use. Before they are discussed, the validity of the OptiCL results found in this paper is discussed.

4.2.2.1. Validity of OptiCL results.

With regards to the results found in MIOCL, the absolute solution errors found here are in line with the results in their tests, when accounting for the expected difference by squaring the error instead of

taking the absolute error. The difference in accuracy of validation MSEs is due to this paper measuring the MSE on the entire training set and training it on a 1000 data point sub set. As a result, the two are non comparable. Lastly the difference in computation time is due to this paper measuring the time to construct the optimization model, train the ML models, implement the models and finally solve the optimization model. The time measured in MIOCL, in contrast, only measures computation time to solve the final optimization model. As a result, the assumption can be made that, with the exception of Linear Regression, the same version of the program has been used in the measurements and the measurements are suitably valid.

4.2.2.2. New Implementation Strategies

As discussed, all points of improvements for OptiCL mentioned in the following paragraphs can be used as design principles for anyone developing packages to integrate ML models into optimization models. The key takeaways can be split up in the following categories: clarity on input and use, accessibility of functions and variables, adaptability and goal fulfillment.

Clarity on input and use. While implementing OptiCL into the model, several issues were encountered on which functions needed to be used to achieve the desired goal. In addition to finding the correct Python function to call, many of the functions required inputs that had little explanation on what data types it accepted and how the types should be constructed. The simplest solution to create clarity on program usage is **more documentation.**

Accessibility of functions and variables. During the testing phase, the Machine Learned model was required to test the accuracy of its predictions. The structure of OptiCL, however, did not allow access directly to the model, only the coefficients could be extracted. Similarly, several functions contain sub-functions, which were inaccessible from the user, and had to be called indirectly and inefficiently through the main function. Both of these issues could solved by **either defining all functions independently or using an OOP approach with classes**, as both allow access to all functions and the former does not have shared variables while the latter allows shared variables to be accessed.

Adaptability. To the best of our understanding, in order to implement an ML model into an optimization problem, OptiCL requires that all steps are taken within the functions supplied by themselves, including the final solving of the optimization problem. The rigidity of use necessitated alterations within the package to accommodate for the requirements of this papers use. Likewise the user can opt to not save results, however the implementation attempts and fails to access paths to file locations regardless of the choice made. The path to improving adaptability is more challenging than the previous: **all modules should be creating to work independently and accept and supply common data structures**. Alternatively, new variants of existing modules can be created that provide additional support for those structures.

Goal fulfillment. Lastly we urge to **keep in mind what the goal of the implementation is**. OptiCL contains incongruences that restrain the usefulness of the program. For example, it requires model training using methods supplied by the package implying start to finish encoding, yet in the manuscript of OptiCL only measurements of computation time for solving the final optimization model are supplied, implying a piecewise independent package. When constructing a start to finish package, it is useful to add the ability to train and encode the ML models and solve the optimization problem in a single function call. The goal of parameter optimization is to increase accuracy of the ML-model. Inferring implementation steps from the goal, it is necessary to ensure that the accuracy does indeed increase.

5

Conclusion

The goal of this project was to compare the performance of OptiCL to an independently programmed implementation, in order to discover implementation strategies for (re)designing packages to fulfill similar functions. While the implementation was successful implementing most functions, we were unable to implement half of the Machine Learning methods and only implemented a simplified version of a trust region.

During testing, several observation about specific Constraint learning practices were made. The two main observations were that ML models similar to Regression Trees require strict trust regions and that additional human interaction could lead to better results, which can be studied further.

Additionally, the following strategies for providing packages to users were provided:

- Increase detail in documentation to improve clarity on use.
- Convert to an Object Oriented Programming approach or define every function independently to allow users access to useful variables and functions.
- Ensure models are independent and accept industry standard data types to increase adaptability of the package to more use cases.
- Formulate the goal of package in order to decrease incongruence between goals and package functionality.

Additional areas that have the possibility for future research and development are:

- Constructing a publicly available implementation for finding higher dimensional convex hulls.
- Finding description methods for Convex Hulls to increase efficiency of constraining an optimizer to the interior of the hull.
- Finding more best practices for implementing Machine Learning models into optimization problems, such as the comparatively high benefit found when restricting the Regression Tree in this implementation.
- Research into the encoding of additional Machine Learned models, beyond those found in MIOCL.

References

[1] M. Donato and W. Holly. *OptiCL*. 2022. URL: https://github.com/hwiberg/OptiCL.git (visited on 04/11/2022).

[2] M. Donato et al. *Mixed-Integer Optimization with Constraint Learning*. 2021. DOI: 10.48550/ARXIV.2111.04469. URL: https://arxiv.org/abs/2111.04469.

[3] A. Siegel. "A parallel algorithm for understanding design spaces and performing convex hull computations". In: *Journal of Computational Mathematics and Data Science* 2 (2022), p. 100021. ISSN: 2772-4158. DOI: https://doi.org/10.1016/j.jcmds.2021.100021. URL: https://www.sciencedirect.com/science/article/pii/S2772415821000110.