

École polytechnique de Louvain

DL8.5 Decision tree used in a Predict-then-Optimize framework

Decision tree learning using SPO loss error
function

Author: **Delphine LECLERCQ**
Supervisor: **Siegfried NIJSSEN**
Readers: **Gaël AGLIN, Charles THOMAS**
Academic year 2022–2023
Master [120] in Data Sciences Engineering

Acknowledgments

This endeavor would not have been possible without the help and guidance of Professor Siegfried Nijssen. His supervision during the elaboration of this thesis was essential. I would also like to express my deepest appreciation to Gaël Aglin for providing his help and expertise throughout my work. I would like to extend my sincere thanks to Charles Thomas for accepting the role of reader.

Lastly, I would like to mention my family, especially my parents, for their love and support. Diane and Pierre Leclercq have provided guidance and encouragements and they kept me motivated and focused. I would also like to mention Scoop, the family Labrador, for being the best puppy in the world.

Abstract

This thesis explores previous research done in the *Predict-then-Optimize* framework. More specifically, it looks at decision trees whose output is used as input in an optimization problem. In this case, the optimization problem will be a shortest path optimization. Four models are tested: the CART classifier, the CART regression tree, the DL8.5 decision tree with MSE loss function and finally, the DL8.5 decision tree with the Smart-Predict-then-Optimize loss (SPO). They are tested on situations of varying complexity. In these situations, the behavior of each method was quite similar no matter the complexity with the exception of the CART classifier. This leads to the conclusion that the CART classifier is quite accurate in simpler situations, but is not capable of handling more complexity. The DL8.5 regression tree used with MSE is more accurate than the CART regression tree when observing the MSE. On the other hand, these two methods have comparable results when it comes to the extra travel time. Finally, the DL8.5 decision tree with the SPO loss function is pretty systematically the better performing method when observing the decision error. We conclude that DL8.5 and SPO loss perform well together and give consistently better results than other combinations.

Contents

Introduction	1
1 Decision Trees - Basic concepts	3
1.1 Classification trees	4
1.2 Regression trees	4
1.3 Multi-output trees	4
1.4 Decision tree learning	4
2 Previous Work	5
2.1 Article summary	5
2.1.1 Framework	5
2.1.2 Decision trees for shortest path	7
2.1.3 Methodology for decision tree learning using SPO loss	7
2.1.4 Experimental Process	8
2.2 Article follow-up	9
2.3 Experimental process used	10
3 CART decision trees	12
3.1 Algorithm overview	12
3.2 Tree structure	12
3.3 Algorithm accuracy	14
3.3.1 Mean square error	14
3.3.2 Extra travel time	15
4 DL8.5 decision tree with MSE loss function	17
4.1 Algorithm overview	17
4.2 Algorithm accuracy	18
4.2.1 Mean square error	18
4.2.2 Extra travel time	18
4.2.3 Relationship between MSE and Extra travel time	19
5 DL8.5 decision tree with SPO loss function	22
5.1 Algorithm overview	22
5.2 Algorithm accuracy	23
5.2.1 Mean square error	23
5.2.2 Extra travel time	24
5.2.3 Relationship between MSE and Extra travel time	25

6	Comparison of the different methods	26
6.1	Path distribution	26
6.2	Shortest path accuracy	28
6.3	Computational time	28
6.4	Conclusion	29
7	Noisy data	30
7.1	MSE and Average Extra travel time	30
7.2	Shortest path accuracy	32
7.3	Path distribution	33
7.4	Conclusion	33
8	Increased complexity in the optimization problem	35
8.1	Path distribution	35
8.2	Shortest path accuracy	36
8.3	Mean square error analysis	36
8.4	Extra travel time analysis	37
8.5	Relationship between MSE and Extra travel time	38
8.6	Computational time	40
8.6.1	Use of an upper bound on the fitting process	41
8.7	Noisy Data	42
	Conclusion	45
	Bibliography	46
	Appendices	47
A	Data generation Python Code	47
B	Model fitting and prediction Python Code	49

Introduction

Optimization problems are often encountered in every day life: filling the car trunk the best way possible to still see out of the rear-view mirror, going grocery shopping while trying to spend as little as possible... On a larger scale, companies look at asset management and capital management when trying to optimize processes or expenditures. But there are many unknown variables when trying to optimize these decisions. Are sales going to crash? Are clients going to back out of orders? Often optimization is done with predicted variables that aren't 100% sure. The classic way of dealing with this uncertainty is to first predict the variables and then to use them in the optimization problem. A new method is tested where the variable prediction is done with the optimization problem in mind.

This thesis will focus on the use of decision trees to predict the input of the shortest path problem. A possible application for this type of algorithm would be in the context of supply chain cost optimization when delivering goods to retail stores. At first, observations would be collected such as:

- the weather : Sun, rain or snow could impact the road conditions and driving speed
- the cost of gas : Fuel prices carry a large impact on the travel cost
- the date of the year : Depending on the day, roadways can be crowded, or traffic can be fluid. Traffic patterns differ based on the day of the week. Some holiday week-ends are notorious for their traffic jams in the direction of holiday destinations

With these known observations, a decision tree could predict the cost of all possible paths to get from the factory to the store. The predicted cost of all these paths are now used as input for a shortest path problem from the factory to the store in order to optimize the costs incurred on delivering to the store.

This thesis will study this problem in a more theoretical setting. The data is artificially generated and the shortest path problem happens on a square 4×4 grid. Different types of decision trees with different types of cost functions will be tested in the following pages. After presenting the basic concepts of decision trees, the article written by Elmachtoub, Liang and McNellis [7] is summarized. This paper explains how the new decision tree learning process can include the decision error of the optimization problem. This loss function know as the Smart-Predict-then-Optimize

loss (SPO loss) is used to improve the overall accuracy.

Firstly, the CART algorithm will be used and will serve as a baseline since it is a fairly common machine learning tool. The DL8.5 decision tree will be tested next. Finally, the SPO loss method explained in the article will be adapted to fit the DL8.5 algorithm. Before concluding, the theoretical setting will be rendered more complex in order to see if the initial results withstand additional complexity. The objective is to see whether or not the DL8.5 decision tree and the SPO loss function can effectively work together.

This thesis covers situations where the optimization problem that needs solving is a shortest path problem. But note that it is possible to change the optimization problem and adapt these methods to be reused. A few constraints later explained will need to be respected. For example, if a retailer needs to predict sales in a certain area to optimize supply delivery and avoid selling at a loss, these methods could be adapted. This works for many other problems.

Chapter 1

Decision Trees - Basic concepts

In decision making problems, many tools exist to find solutions depending on the problem's context. If there is available training data, supervised machine learning tools like linear regression, support vector machines (SVM) but also decision trees can be utilized. This thesis will focus on decision trees so this first chapter is dedicated to explaining some key basic concepts.

Decision trees have a hierarchical tree-like structure with a root and branches. All input data is entered into the root node. The root is split into multiple branches. Usually each split leads to two new nodes, which is the type of tree commonly used. However, a split may also lead to more than two nodes in special cases. For each split of a node into new branches, a special criteria is used. Nodes will stop splitting into new ones once the stopping criteria is met. These nodes become final nodes called leaves which determine a specific output. The stopping criteria could be a maximum depth, a minimum number of input data found in each leaf...

The main advantage of the decision tree is its interpretability. Each split and decision have a criteria linked to it, explaining why this decision was made. However, the deeper the tree gets (depth being determined by the longest path from the root to a leaf), the harder it becomes to interpret.

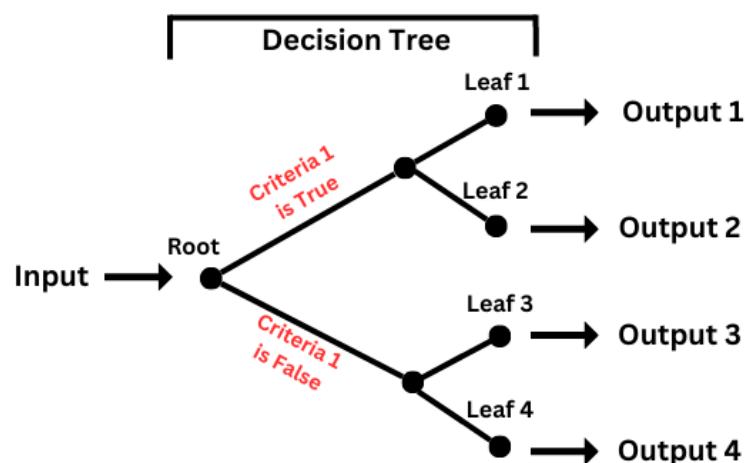


Figure 1.1: Example of a decision tree of depth 2

There are different types of decision trees such as ID3, CART (Classification and Regression Trees) and Chi-Square. Not all will be explored here, nonetheless, CART will be explained later (see chapter 3).

Supervised machine learning algorithms can solve classification problems or regression problems. Decision trees can be separated into classification trees and regression trees and can be used to solve different types of problems.

1.1 Classification trees

After analyzing the input, the classifier tree can sort the input into a specific category, meaning it will associate a value of a discrete variable to the input. If the decision tree is meant to determine what animal is in a picture, the possible outputs from figure 1.1 could look like {cat, dog, fish, horse}.

1.2 Regression trees

A regression tree will predict the value of a continuous variable. So, if the tree is meant to determine the grade of students' homework, the outputs could be {11.3/20, 7.25/20, 14.22/20, 17.96/20}. The decision tree can only predict as many different values as the tree has leaves since each prediction comes from a leaf, so only certain values of the continuous variable can be predicted.

1.3 Multi-output trees

The most common case for decision trees are trees where the leaves each return one value, like a type of animal or a grade. In this thesis, multi-output trees are used. These decision trees can, for each output, give multiple values at a time. If the tree is meant to determine a spatial location with (x, y, z) positions, the outputs could look like {(1,3,5), (2,2,1), (4,1,2), (1,1,3)}.

1.4 Decision tree learning

In supervised learning, in order to predict all these different outputs, the decision tree must learn from existing data. Decision tree learning will result in a tree that is properly fitted to the data it was given for training. In regression trees, this process is usually done by minimizing the mean square error (MSE) between the predicted output and the real output for each observed input in the training set. This is the default method used in Python's Scikit-learn DecisionTreeRegressor.

Chapter 2

Previous Work

The initial idea from which this thesis came to be originates from the 2020 paper "Decision Trees for Decision-Making under the Predict-then-Optimize Framework" written by Elmachoub, Liang and McNellis [7]. Hereunder is a summary of the sections of the article relating to this thesis.

2.1 Article summary

2.1.1 Framework

In this article [7], the decision-making problem is solved under the *predict-then-optimize* framework. This specific framework consists in two steps:

1. Observations are used to predict unknown parameters thanks to a machine learning (ML) algorithm, more specifically, a decision tree in this case.
2. The predicted parameters will then be used as the input of the optimization problem. The solution to this problem will be the final decision.

Typically, the ML algorithm is trained using a function that minimizes the prediction error like MSE. Instead, the authors focus on using a method that directly minimizes the decision error. The difference between the two types of errors is illustrated in figure 2.1. Such a loss function was proposed by Elmouchtoub and Grigas in 2017 [6] called Smart Predict-then-Optimize loss (SPO loss). In spite of this loss function being non convex, discontinuous and so not differentiable, it can be greatly simplified when considering decision trees and their structural properties. Trees trained on minimizing SPO loss are called SPO Trees (SPOTs).

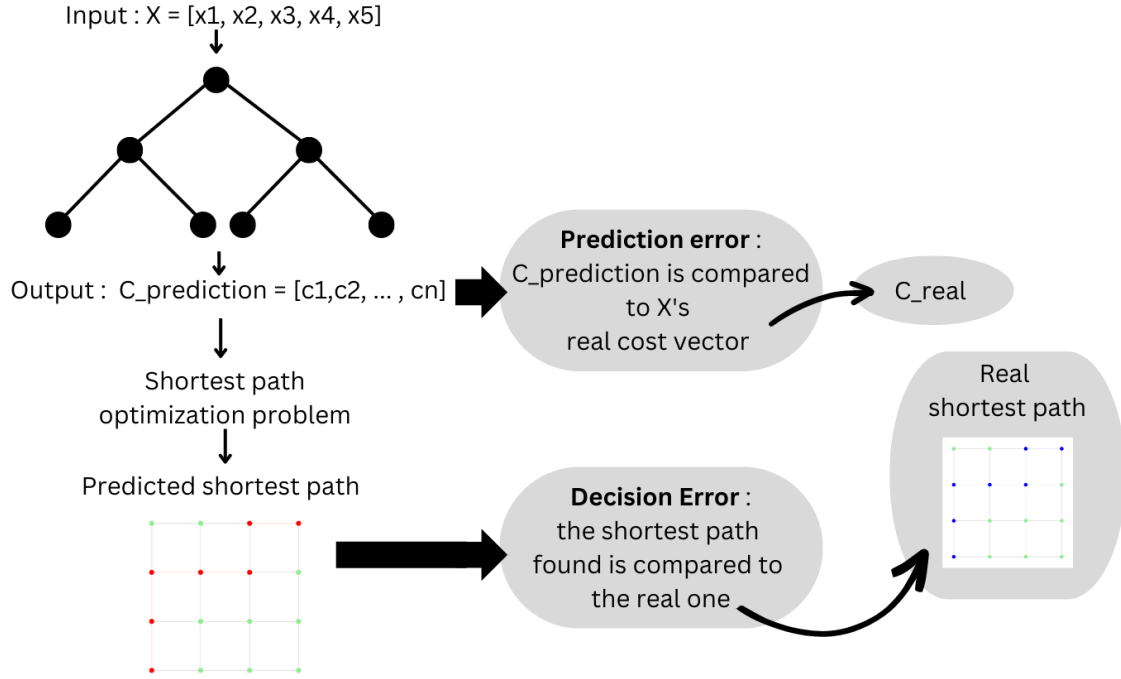


Figure 2.1: Illustration of the prediction error vs. the decision error if the decision making problem is a shortest path problem

The article focuses on decision-making problems with known constraints and an unknown linear objective function that can be predicted from the initial input data. With :

- d : the dimension of the decision space
- $S \subseteq \mathbb{R}^d$: the feasible region for the decisions
- $c \subseteq \mathbb{R}^d$: the cost vector of the optimization problem
- $w \subseteq \mathbb{R}^d$: the vector of decision variables

The decision-making problem can be defined as $z^*(c) = \min_{w \in S}(c^T w)$ and the set of optimal decisions corresponding to $z^*(c)$ can be defined as $W^*(c) = \arg \min_{w \in S}(c^T w)$. When trying to solve individual members of $W^*(c)$, noted $w^*(c)$, c is still unknown and is approximated by its prediction $\hat{c}$ which comes from a ML model that was trained on a given dataset $\{(x_1, c_1), (x_2, c_2), \dots, (x_n, c_n)\}$ assumed to be independent and identically distributed. $x \subseteq \mathbb{R}^d$ denotes a vector of p features available for the prediction of c .

With :

- H : the hypothesis class of ML model candidates
- $f : \mathbb{R}^p \rightarrow \mathbb{R}^d$: a ML model candidate for predicting cost vectors from feature vectors
- $l(.,.) : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}_+$: the loss function used to train the ML models

So, $\hat{c} = f(x)$ is the predicted cost vector resulting from the feature vector x with the model f . And $l(\hat{c}, c)$ is the loss resulting from a prediction $\hat{c}$ when the real cost vector is c . The optimal trained model is the model in the hypothesis class H that scores the smallest average loss on the training data with respect to the loss function $l(.,.)$: $f^* = \arg \min_{f \in H} \frac{1}{n} \sum_{i=1}^n l(f(x_i), c_i)$.

Instead of using a more common loss function like MSE: $l_{MSE}(\hat{c}, c) := \|\hat{c} - c\|_2^2$, using SPO loss will determine the extra cost of making a sub-optimal decision. SPO loss is defined according to the worst possible decision from the predicted cost vector $\hat{c}$ [6]: $l_{SPO}(\hat{c}, c) := \max_{w \in W^*(\hat{c})} \{c^T w\} - z^*(c)$.

2.1.2 Decision trees for shortest path

In a small illustrative example, the authors use a 2-node graph linked together by two edges. Costs associated to those edges were a linear equation and a quadratic equation of the single feature variable. With decision trees, each leaf will assign a single predicted vector $\hat{c}$ and an associated decision $w^*(\hat{c})$. When testing SPOT versus CART algorithm, SPOT returns high quality decisions while staying significantly less complex than the CART tree that results in similar quality decisions. To minimize tree depth, each split in the tree must have a maximized contribution to reduce complexity which will in turn reduce the risk of overfitting.

2.1.3 Methodology for decision tree learning using SPO loss

Training decision trees using the SPO loss function produces SPO Trees (SPOTs). For all leaves $R_1, R_2, \dots, R_L$ of the SPOT, their respective prediction must minimize the SPO loss function:

$$\min_{R_{1:L} \in T} \frac{1}{n} \sum_{l=1}^L \left(\min_{\hat{c}_l} \sum_{i \in R_l} l_{SPO}(\hat{c}_l, c_i) \right) \quad (2.1)$$

where $R_{1:L} \in T$ specifies that the association of observations to leaves must follow a decision tree structure.

The theorem used to simplify training states that the average of the cost vectors corresponding to a leaf node minimizes the SPO loss in that leaf node [7]. To utilize this theorem, uniqueness of the optimal solution to the decision-making problem must be guaranteed. This can easily be accomplished by adding a small noise term to every cost vector of the training dataset. So 2.1 becomes:

$$\min_{R_{1:L} \in T} \frac{1}{n} \sum_{l=1}^L \sum_{i \in R_l} (c_i^T w^*(\bar{c}_l) - z^*(c_i)) \quad (2.2)$$

Recursive partitioning approach

A tree split is represented by (j, s) , where j is the splitting feature component and s is the split point that separates the observations into two leaves $R_1(j, s)$ and

$R_2(j, s)$. In this method, the first split will be determined by the (j, s) pair that minimizes the following problem:

$$\min_{j,s} \frac{1}{n} \left(\sum_{i \in R_1(j,s)} (c^T w^*(\bar{c}_l) - z^*(c_i)) + \sum_{i \in R_2(j,s)} (c^T w^*(\bar{c}_l) - z^*(c_i)) \right) \quad (2.3)$$

After the first split, a greedy algorithm will recursively apply the same split selection process on the data relevant to that branch of the SPOT until meeting a stopping criteria. A greedy algorithm is a problem solving approach that selects the best option available (in this case the best split) without worrying if the end result is the most optimal one or not.

2.1.4 Experimental Process

The shortest path problem was considered on a 4×4 grid graph with each edge representing a road that could only be used going upwards or to the right in order to go from the lower-left corner to the upper-right corner of the grid.

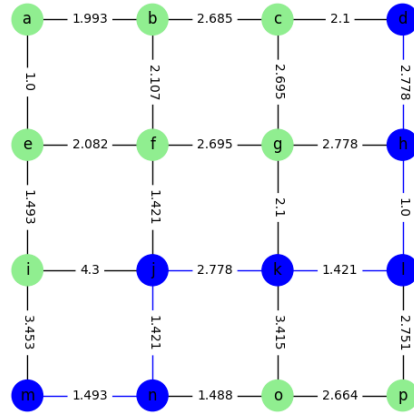


Figure 2.2: Example grid graph with shortest path

Data generation

The feature-cost pairs were generated by:

1. Sampling n feature vectors $x_1, x_2, \dots, x_n$, each from a uniform distribution $Uniform(0, 1)^p$ with $p = 5$, the number of features per feature vector
2. Sampling a matrix $B \in 0, 1^{d \times p}$ by sampling every entry $B_{k,j}$ from $Bernoulli(1, 0.5)$ knowing that $d = 24$ is the dimension of the decision space so the number of edges on the grid
3. Computing each feature vector x_i 's associated cost vector c_i with $c_{ik} = (\frac{1}{\sqrt{p}}(Bx_i)_k + 1)^{deg} \epsilon_i^k$ where:
 - $(Bx_i)_k$ is the k th component of Bx_i

- deg is a fixed positive integer that controls the amount of nonlinearity in the feature-cost mapping
- ϵ_i^k are noise terms sampled from $Uniform([1 - \bar{\epsilon}, 1 + \bar{\epsilon}])$ for some $\bar{\epsilon} \geq 0$

Decision tree training

The trees were all trained using a minimum leaf size of 20 observations. This constraint ensures sufficient accuracy in the leaf's cost vector prediction. The trees were also pruned using the pruning technique described in [4] in order to avoid any overfitting on a validation set (20% of the training set). With a $n = 200$, the authors tested parameters $deg \in 2, 10$ and $\epsilon \in 0, 0.25$ and maximum tree depths were limited to 1, 2, 3 and unlimited. With this depth limitation, it must be noted that even with unlimited depth, CART cannot have a depth larger than 4 due to the minimum leaf size constraint and the way the algorithm works.

Results

SPOTs lead to better decisions than CART while keeping a better interpretability and lower complexity. With a depth of 1, SPOTs achieved a lower SPO loss than CART trees with unrestricted depth. When comparing same depth trees together, the recursive partitioning approach to SPO achieves 23.1 to 26.8% better than CART depending on the depth allowed. The authors explain that CART's lack of accuracy is due to the fact that there is limited amounts of training data and CART trains on prediction error rather than decision error like SPOTs.

When the experiments were redone with $n = 10000$, the difference in performance between SPOTs and CART were much less noticeable. The large amount of training data allows CART to obtain much more accurate predictions and thus nearly-optimal decisions. However, CART trees require more than twice the amount of leaves as SPOTs to accomplish a similar accuracy on the final decision in this scenario. Indeed, the large amounts of data allows the CART algorithm to construct a tree with larger depth than 4.

2.2 Article follow-up

With the authors reporting such good accuracy for this method of decision tree learning, it is inevitable for the reader to want to dig deeper. Can this method be adapted and used with other decision trees? Will the results still prove to be as good? Do the experiments in this paper suffice to know the effectiveness of the method?

On the basis of this article, this thesis sets out to validate that the recursive partitioning approach to SPO loss can be used in harmony with the optimal decision tree DL8.5 later explained in section 4. The experimental context of the paper is reused, adapted and pushed to its limits.

2.3 Experimental process used

In the following chapters, four different methods will be described and tested: CART classification tree, CART regression tree, DL8.5 regression tree and DL8.5 regression tree with a SPO loss function implementation. All methods will accept the same feature vectors x_i . All tests are done with a training set of size $n = 200$ and a test set of size $n_{test} = 50$.

To further investigate the results of the previously explained article [7], the same data generation process is used in this thesis as described in section 2.1.4. The number of features per observation stays $p = 5$ and the dimension of the decision stays $d = 24$. For the computation of the cost vectors, the deg parameter will vary $deg \in [2, 5, 10, 20]$, as will the noise parameter $\epsilon \in [0, 0.25]$. For the next few chapters, only data with a $\epsilon = 0$ will be shown and noisy data will be presented later on.

The generated data will have values from 0 to 1 in the feature vectors and varying values in the cost vectors as seen in the table below 2.1. The dataset created with parameter $deg = 5$ will be referred to as *the dataset of degree 5* later on. Please note the extremely high values for $deg = 20$ which are bordering on unrealistic. These values were kept to satisfy curiosity more than for useful experimental results.

deg	minimum value	maximum value = $(\sqrt{5} + 1)^{deg}$
2	1	10,5
5	1	354,9
10	1	125 943,7
20	1	15 861 809 080,0

Table 2.1: Possible values in the cost vectors with $\epsilon = 0$

In order to accommodate decision trees allowing only boolean values $\{0, 1\}$ as feature values, the five features of the feature vectors were discretized. Each feature is represented by five new features, allowing five bins of equal size $[0, 0.2[$, $[0.2, 0.4[$, $[0.4, 0.6[$, $[0.6, 0.8[$ and $[0.8, 1]$. So each feature vector is now composed of twenty-five features. An example is given in table 2.2. The python implementation of the data generation process can be found in Appendix A.

original feature value	5 new features
0.351	[0, 1, 0, 0, 0]
0.102	[1, 0, 0, 0, 0]
0.78	[0, 0, 0, 1, 0]
0.99	[0, 0, 0, 0, 1]
0.97	[0, 0, 0, 0, 1]

Table 2.2: Example of the discretization of a feature vector

This next part is the decision-making problem. The same shortest path problem in a 4×4 grid is used. The path must go from the lower-left corner to the upper-right corner of the grid (figure 2.2). Although, it is permitted to take any path in any direction thus allowing more paths than in the above mentioned article [7]. In practice, the shortest path will be calculated with the python *NetworkX* package using Dijkstra's algorithm [5].

Now that the experimental process has been described, the following sections will be dedicated to the different types of decision trees and the results obtained in this experimental context.

Chapter 3

CART decision trees

3.1 Algorithm overview

The CART algorithm works for both classification and regression tasks. So the two first methods tested are the CART classification tree and the CART regression tree.

The CART decision tree is a tree where each split results in two branches. When given the training set, a split is determined by evaluating all the possible splits and choosing the one that results in the smallest cost. This greedy approach continues until a predefined stopping criteria is met. For these experiments, the stopping criteria for the decision tree will be an imposed maximum depth.

Since both the classifier and the regressor are used and compared for the sake of thoroughness, the input data must be adapted. The regression algorithm is given (x_i, c_i) pairs, x_i is the feature vector and c_i is the cost vector. This is the kind of data that comes straight out of the data generation process. For any new feature vector, the algorithm must predict the associated cost vector. For the classifier, the supervised learning is done with the feature vectors x_i but instead of using the cost vectors c_i , the algorithm is directly given the shortest path associated to each cost vector. Using the cost vectors directly would mean that every unique cost vector would constitute a class. This would result in too many different classes due to the large range of values in each vector. Now that the training data has been specified, the CART algorithm must go through the learning process. The cost function used is the Mean Square Error (MSE) which is the Python Scikit-learn default for CART.

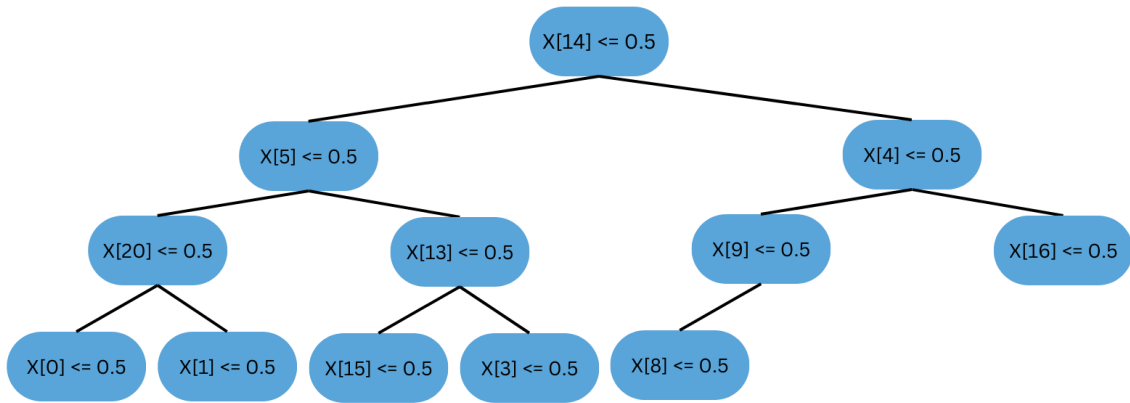
3.2 Tree structure

Due to its greedy method, when looking at the trees with a maximum depth varying from 1 to 4 trained on the same data, it is normal to find that each new tree is built off of the one with a smaller maximum depth. Now it is interesting to note that the splits are not chosen in the same way when comparing both types of trees as shown in figures 3.1. The algorithm works the same way for both the classification and the regression tree. This difference in split choices indicates that training on the

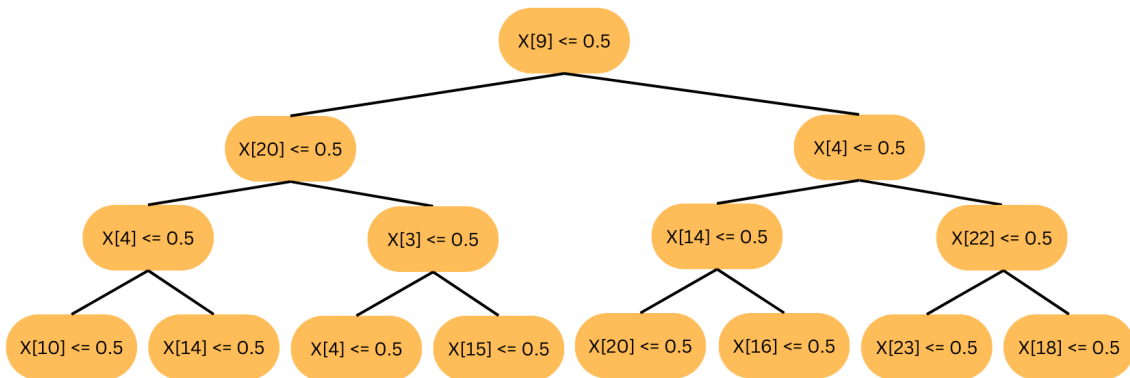
cost vectors and training on the shortest paths calculated from these cost vectors directly results in different decision trees. These trees have multiple leaves and predictions but some of these leaves lead to the same path in the optimization problem. The classifier simply predicts the same path in multiple leaves but the regression tree predicts different cost vector predictions. After looking more closely, these cost vectors lead to the use of the same shortest path even though they are different. The table 3.1 shows the number of unique paths that were predicted from each tree of depth 4. This allows a maximum of 16 different prediction for each tree, but this capacity is underused with only very few unique predictions per tree. The classifier has at least twice the amount of different predictions as the regression tree. It seems to take more advantage of the leaves. Keep in mind that the dataset of degree 2 refers to the dataset generated using the parameter $deg = 2$.

Dataset of :	degree 2	degree 5	degree 10	degree 20
CART classifier tree	2	4	4	8
CART regression tree	1	2	2	3

Table 3.1: Number of unique paths predicted from the decision tree of maximum depth 4



(a) CART classifier tree of maximum depth 4



(b) CART regression tree of maximum depth 4

Figure 3.1: Split comparison of CART classifier and regression trees

3.3 Algorithm accuracy

After having a closer look at the decision trees' structure, the accuracy must be verified. A few accuracy measures for both these decision trees are analyzed in the following sections.

3.3.1 Mean square error

For consistency and interpretability of the graphs throughout this document, each method used in this thesis is assigned a color in order to be easily distinguished.

CART classification tree	blue
CART regression tree	orange
DL8.5 regression tree	green
DL8.5 regression tree with SPO	red

The average MSE with respect to the depth of the tree can be seen in figure 3.2. For all figures representing a type of error, a lower number means lower error and thus more accurate results. Be mindful of the scales on the top left corner of the concerned subgraphs, 1e6 indicates the whole scale must be multiplied by 10^6 .

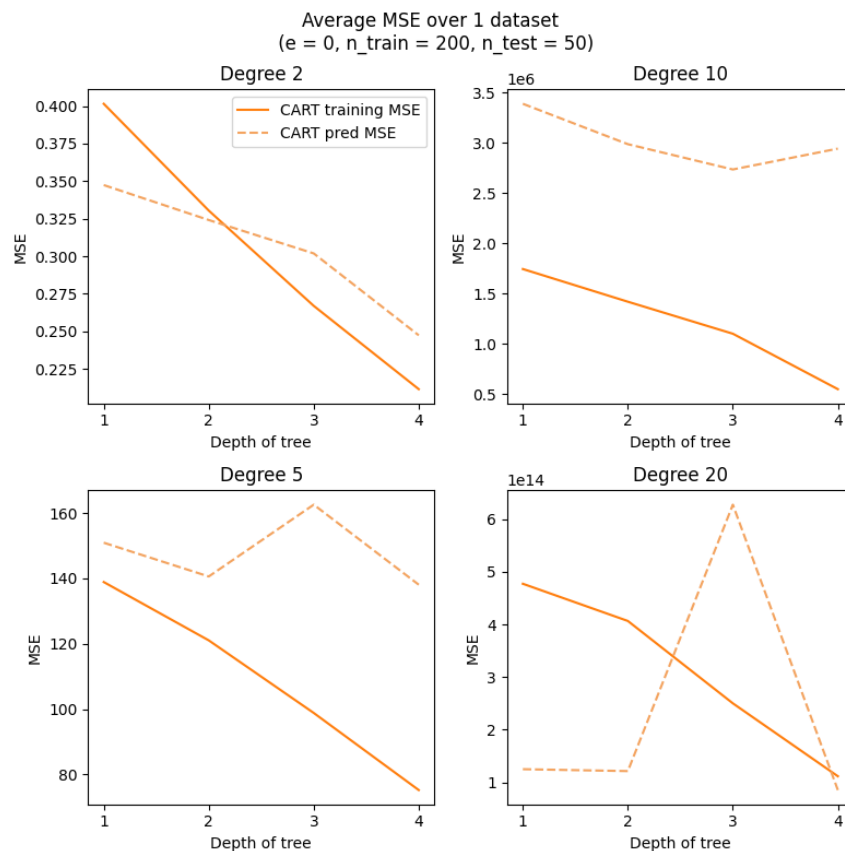


Figure 3.2: Average MSE with respect to the depth of the regression tree

Each element of the cost vector prediction $\hat{c}_i$ is compared to its real value c_i . When using the decision tree on the data it was trained on, we see a constant decrease in MSE as the depth of the tree increases as expected. The more complex the tree can be, the more accurate the results. For the test set, this is not always the case with less predictable MSEs. With the exception of data of degree 2, adding more splits and depth to the tree does not necessarily result in more accurate predictions, this could possibly be caused by an overfitting of the tree on the training data. The CART classification tree does not appear on this figure because it predicts the shortest paths directly so no cost vector predictions are made. Since the MSE is calculated on the difference between its real cost vector and the prediction, no MSE can be computed for the classifier.

3.3.2 Extra travel time

Once the cost vectors are predicted, a shortest path is calculated for each prediction. The travel time of this path is easily found as it is the sum of the costs of all edges used in the graph for the shortest path according to the cost vector.

$$\text{Travel time}(c_i) = \sum_{\forall j \in \text{ShortestPath}(c_i)} c_{i,j} \quad (3.1)$$

where $c_{i,j}$ is the j th element of the cost vector c_i and $\text{ShortestPath}(c_i)$ is the shortest path found using the cost vector c_i . Now the extra travel time is the difference between the travel time of the predicted cost vector $\hat{c}_i$ and the real travel time.

$$\text{Extra travel time}(c_i) = \sum_{\forall j \in \text{ShortestPath}(\hat{c}_i)} c_{i,j} - \sum_{\forall j \in \text{ShortestPath}(c_i)} c_{i,j} \quad (3.2)$$

Since the extra travel time calculation can be done without any cost vector predictions but simply with shortest path predictions and the real cost vector, the regression tree and the classifier tree can be compared.

Looking at figure 3.3, it can easily be observed that the classifier has an overall lower extra travel time in its predictions of the training and test sets. This can be explained by the fact that the classifier is trained with the decision variable (the shortest path) whereas the regression tree is trained only on the prediction error, more precisely the MSE of the predictions. With the exception of the dataset of degree 20, the regression tree seems overfitted for the degree 2, 5 and 10 because it has a less noticeable gain in accuracy as the depth of the tree increases. For degree 20, it is possible for the tree to have the same overfitting issue but the abnormally large range of values pull the extra travel time higher making the overfitting less noticeable for deeper trees.

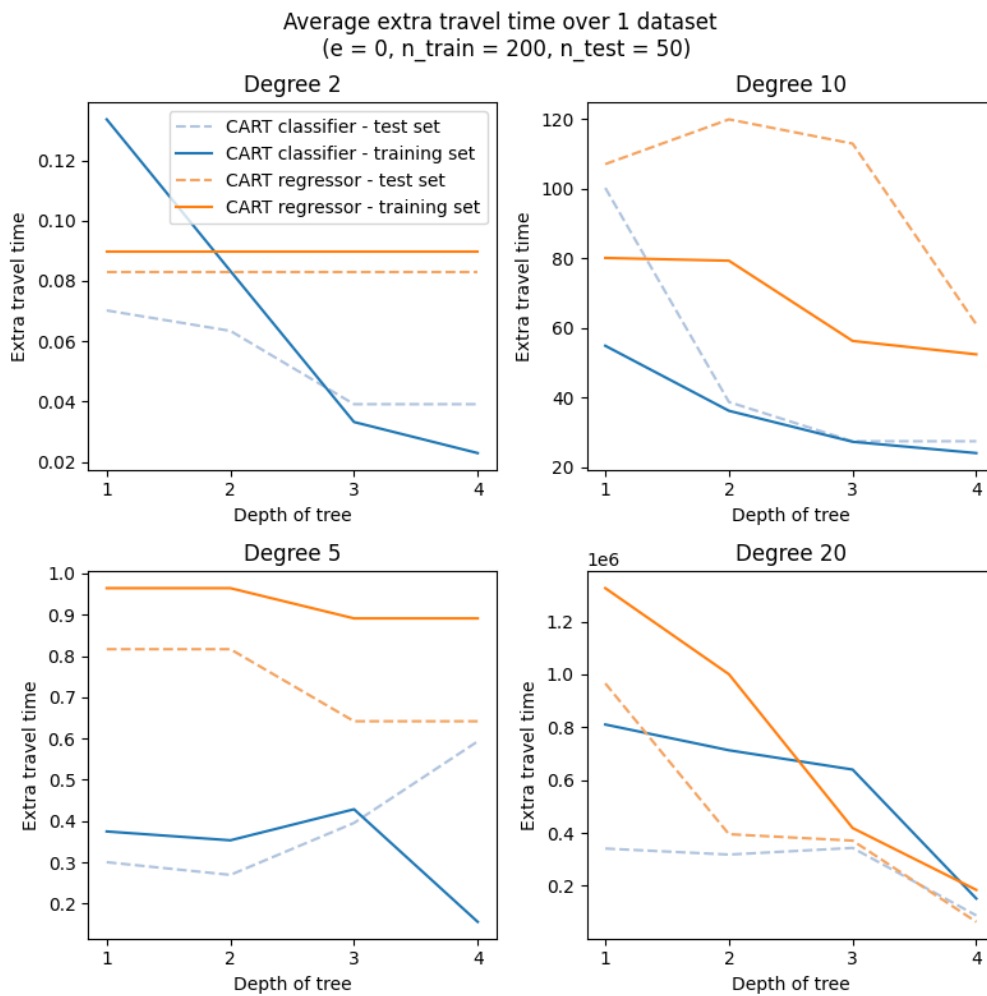


Figure 3.3: Average Extra travel time with respect to the depth of the regression tree

Chapter 4

DL8.5 decision tree with MSE loss function

In this chapter, a new method will be used instead of the CART algorithm. The greedy approach used by CART has been found to be sub-optimal when wanting to limit maximum depth. CART algorithm must compromise between sufficient accuracy and reasonable depth [3]. The DL8.5 algorithm [1] allows the learning of an optimal decision tree (ODTs). These trees reach the lowest possible cost while respecting pre-defined constraints like depth.

4.1 Algorithm overview

Learning OTDs can be done by DL8.5 as long as the optimization criterion is additive which is the case for the Mean square error. This algorithm is based on Branch-and-Bound search as well as Dynamic programming. DL8.5 treats decision tree paths as itemsets where each feature is represented by a positive or negative item and uses itemset mining techniques to efficiently look through the space of possible paths and find the optimal decision tree [1]. All possible splits are recursively explored and a caching system permits reusing results when the same itemset is encountered.

The obligation of an additive optimization criterion and the use of depth and support constraints authorize certain pruning in DL8.5 better explained by its authors [1]. The error calculated for each leaf by the DL8.5 Classifier is, by default, calculated by :

$$leaf_error(I) = |cover(I)| - \max_{t \in T} \{Sup(I, t)\} \quad (4.1)$$

where $cover(I)$ is the set of observations where itemset I is found and $Sup(I, t)$ is the number of examples for a given class t in the leaf with itemset I and T is the set of class labels. The third decision tree is however not the DL8.5 Classifier, but rather the DL8.5 Predictor, a regression tool because this choice allows an output of cost vector predictions.

DL8.5 fortunately comes with a python library PyDL8.5 [2] compatible with

the scikit-learn package that is used in this thesis. This greatly facilitates the modification of the error function in order to match the one used for the CART algorithm, so here, DL8.5 has been implemented to use the following MSE function:

$$leaf_error(I) = \sum_{\forall i: x_i \in cover(I)} \sum_{\forall j} (c_{i,j} - \hat{c}(I)_j)^2 \quad (4.2)$$

where $c_{i,j}$ is the j th element of the cost vector c_i and $\hat{c}(I)_j$ is the j th element of the predicted cost vector for all observations found in the leaf representing itemset I . All feature vectors found in the leaf associated with itemset I will be attributed the same prediction and it is determined by calculating :

$$\hat{c}(I) = \sum_{\forall i: x_i \in cover(I)} c_i * \frac{1}{|cover(I)|} \quad (4.3)$$

This can be interpreted as the average cost vector calculated from the cost vectors associated to the feature vectors included in the leaf representing itemset I . To incorporate this in the implementation, the *leaf_value_function* parameter of the DL8.5 Predictor was written to match the equation 4.3.

4.2 Algorithm accuracy

Now that the DL8.5 algorithm has been better explained, its results will be analyzed and compared to the previous methods.

4.2.1 Mean square error

Both the CART algorithm and DL8.5 use the mean square error to train the decision trees. When observing the results of such ODTs in figure 4.1, the predictions made by the DL8.5 algorithm on the training data are systematically a little better but still very similar. It can be interpreted that the CART decision tree already uses an almost optimal decision tree thus using the DL8.5 algorithm does not greatly improve results. Predictions made on the test set seem to have varying results where adding complexity to the tree does not necessarily make for more accurate results. Again, overfitting could be the cause for this.

4.2.2 Extra travel time

The average extra travel time in figure 4.2 visually represents the error made by the optimization problem's decision. The CART classifier still achieves better results than the other decision trees on the training set but also on the test set, which causes the belief that training on the decision error directly rather than the prediction error of the cost vectors leads to more accurate final decisions. Keep in mind that DL8.5 and the CART regression have been trained on another cost function than the one illustrated here. The DL8.5 algorithm shows rather poor results sometimes performing worse than the CART regression. There may be overfitting of the tree on the training set. Again, like for the CART regression algorithm, the fast decreasing extra travel time for the dataset of degree 20 may be due to the extremely large range of possible values as explained in 2.1.

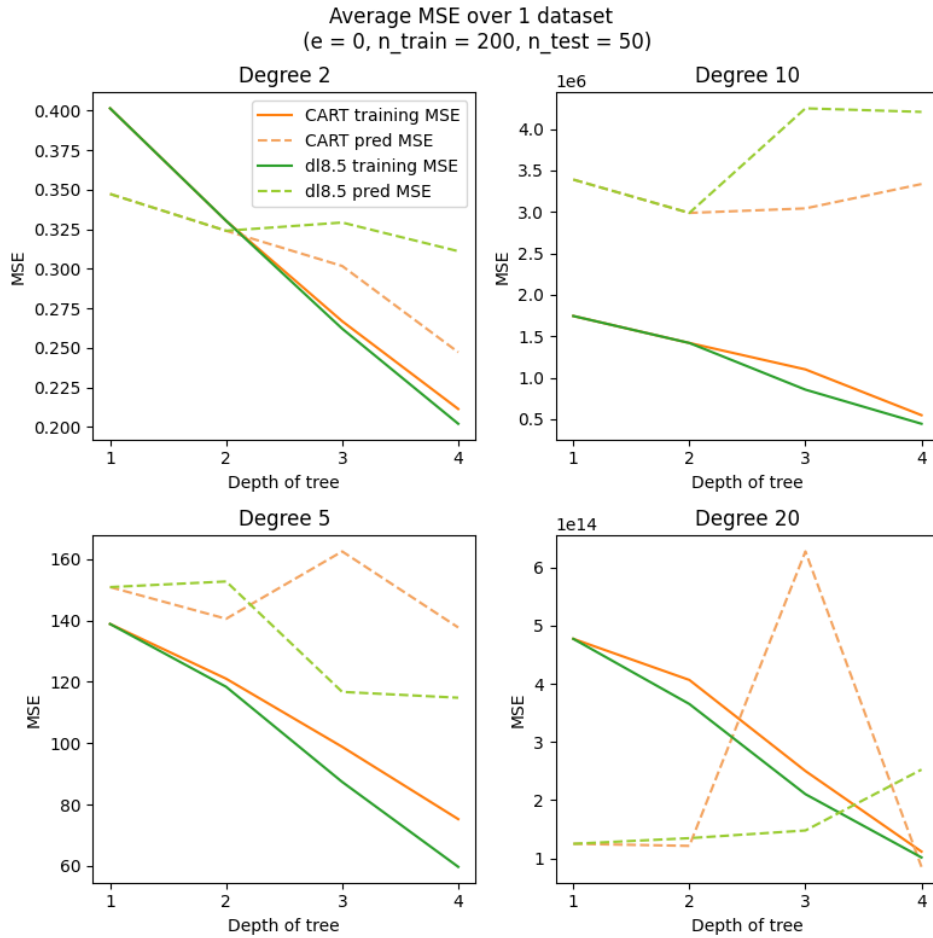


Figure 4.1: Average MSE with respect to the depth of the regression tree

4.2.3 Relationship between MSE and Extra travel time

A question that emerges when looking at the average mean square error and the extra travel time is whether these two error evaluations have a connection. In figure 4.3, both error evaluations are represented. The logarithmic scale for the MSE on the y-axis allows a different view of the data. The DL8.5 predictions appear more prevalent in the visualization due to their superposition on top of the CART data points. The chosen colors are semi-transparent for the multiple layers to shine through. A clear concentration of points appear aligned on the left. These are all the points with 0 extra travel time indicating they took the correct shortest path despite an error in the cost vector predictions. Points aligned vertically are most likely observations for which the shortest path predicted was the same but the cost vector predictions were different and ended up resulting in different MSEs. No other particular pattern can be discerned.

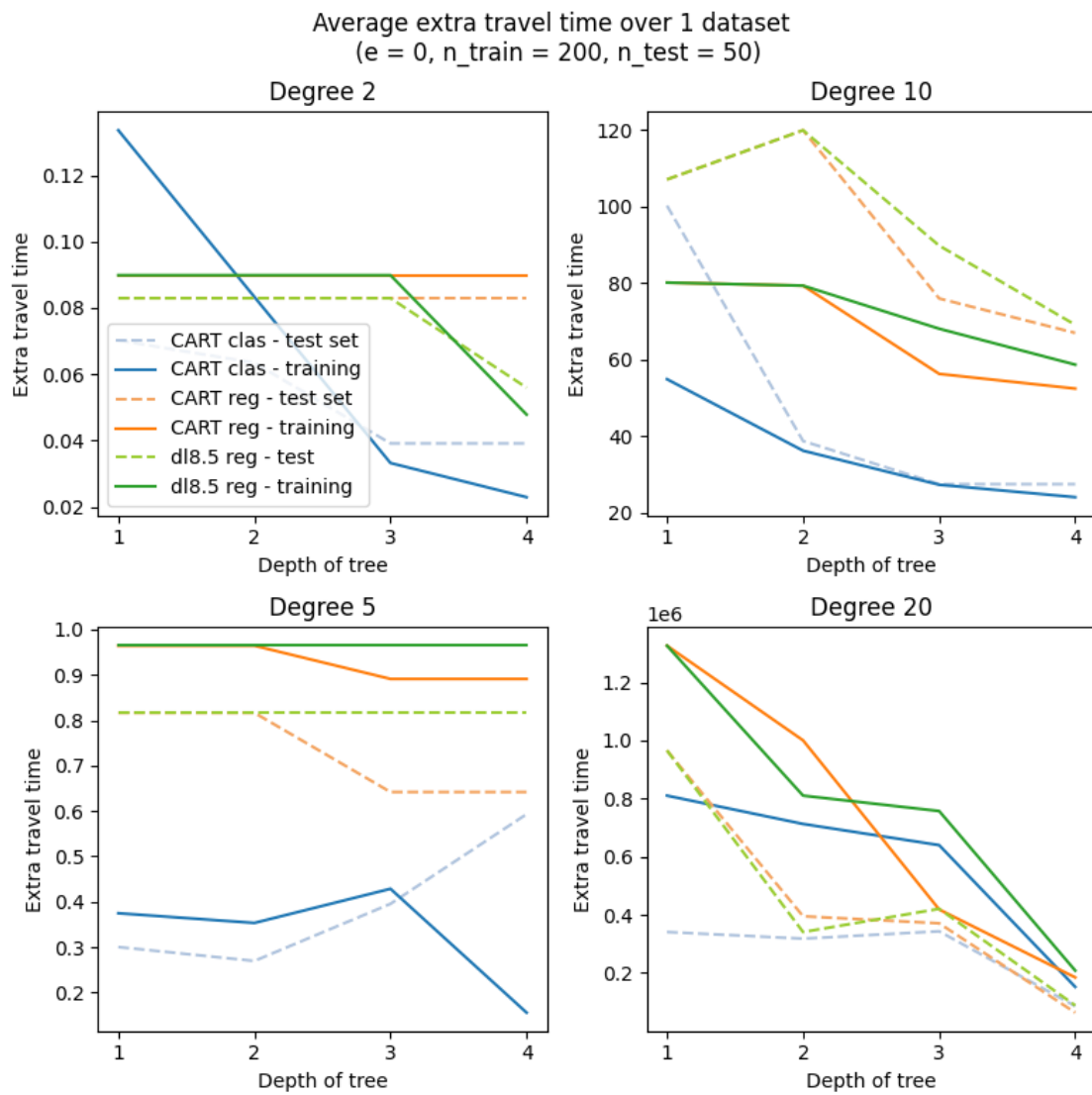


Figure 4.2: Average Extra travel time with respect to the depth of the regression tree

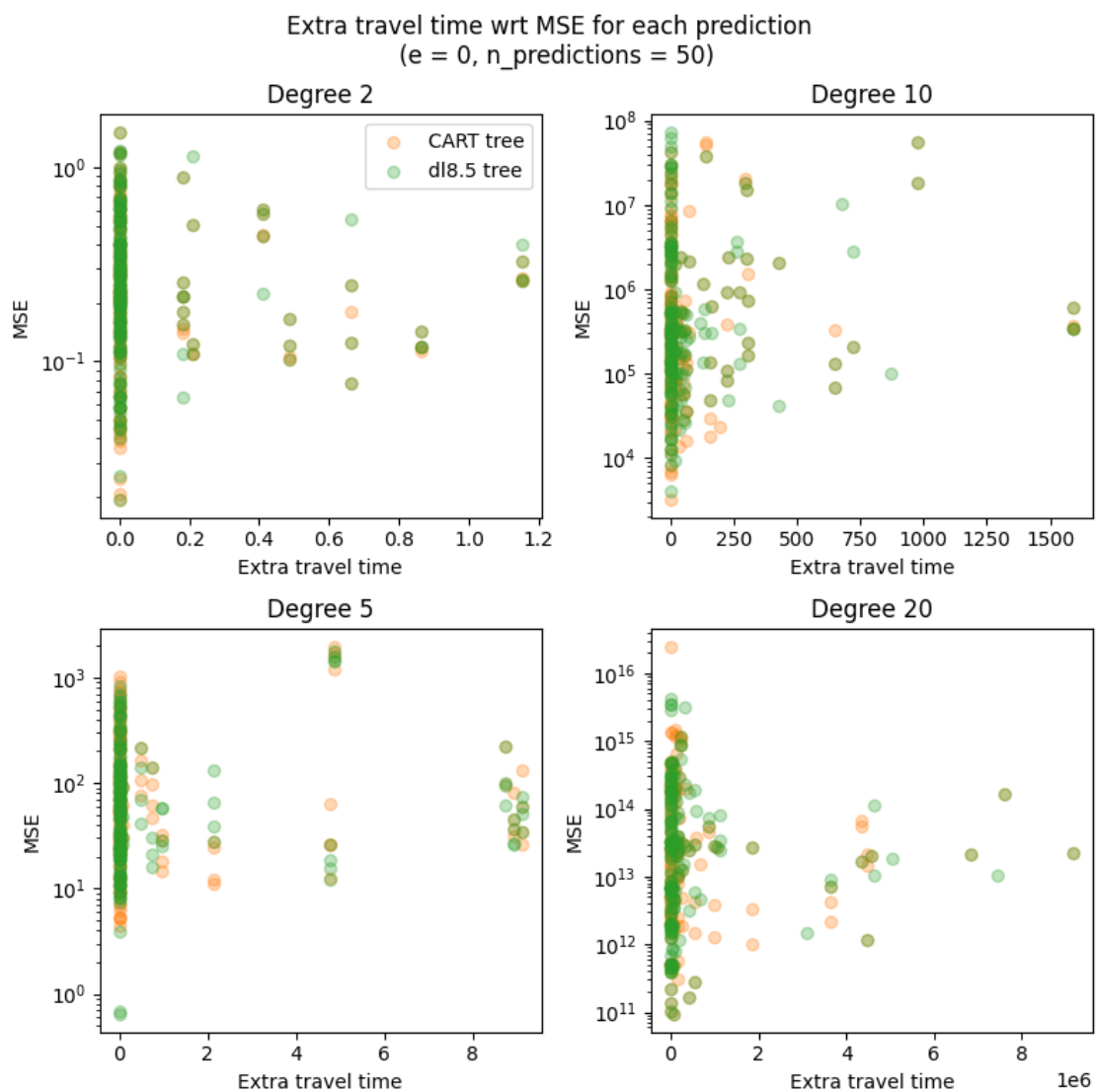


Figure 4.3: Extra travel time with respect to MSE for every prediction of the test set by trees of depth 1 to 4

Chapter 5

DL8.5 decision tree with SPO loss function

In the previous section, the DL8.5 decision tree results in more accurate predictions when compared to the CART algorithm in terms of MSE. However, the results in terms of extra travel time are quite poor because it was not trained on this parameter. When seeing the CART classifier accuracy in predicting the shortest path, it seems like allowing the DL8.5 decision tree to train on the decision error and not on the prediction error could lead to promising results. This introduces the last and final type of decision tree that will be used, where the DL8.5 tree will be trained according to the SPO loss function presented in the article in chapter 2.

5.1 Algorithm overview

For SPO loss to be adapted to the DL8.5 algorithm, the loss function is adapted from the cost function 2.1 to count for only one leaf R at a time to put the recursive partitioning approach into practice:

$$\min_R \frac{1}{n} (\min_{\hat{c}_R} \sum_{i: x_i \in R} l_{SPO}(\hat{c}_i, c_i)) \quad (5.1)$$

with $x_i \in R$ representing the feature vectors that respect the conditions leading to leaf R . Now, thanks to the Theorem used to simplify the SPO loss calculation described in section 2.1.3 the previous function becomes:

$$\min_R \frac{1}{n} \sum_{i: x_i \in R} (c_i^T w^*(\bar{c}) - z^*(c_i)) \quad (5.2)$$

with $\bar{c}$ being the average cost vector found from all cost vectors in the data subset in the leaf R . Once adapted to the shortest path optimization problem, it becomes:

$$\min_R \frac{1}{n} \sum_{i: x_i \in R} \left(\sum_{j \in \text{ShortestPath}(\bar{c})} c_{i,j} - \sum_{j \in \text{ShortestPath}(c_i)} c_{i,j} \right) \quad (5.3)$$

Using the same notations as before, $\text{ShortestPath}(\bar{c})$ is the shortest path found using the cost vector $\bar{c}$. In the actual code implementation, the constant factor $\frac{1}{n}$ is dropped for simplification. This finalizes the loss function to be:

$$\min_R \sum_{i: x_i \in R} \left(\sum_{j \in \text{ShortestPath}(\bar{c})} c_{i,j} - \sum_{j \in \text{ShortestPath}(c_i)} c_{i,j} \right) \quad (5.4)$$

In the previous chapter, the *leaf_error* function of the DL8.5 algorithm (equation 4.2) used MSE as its loss function. Now, this function will calculate the SPO loss as so:

$$\text{leaf_error}(R) = \sum_{i: x_i \in R} \left(\sum_{j \in \text{ShortestPath}(\bar{c})} c_{i,j} - \sum_{j \in \text{ShortestPath}(c_i)} c_{i,j} \right) \quad (5.5)$$

The implementation was done to match the *leaf_error* parameter of the DL8.5 Predictor to this last function. So, to calculate the SPO loss error in one leaf, the shortest path must be calculated for every item respecting the leaf's conditions which will take considerable time.

5.2 Algorithm accuracy

This combination of an optimal decision tree trained on the decision error, thanks to the SPO loss function, should give the most accurate results when observing the extra travel time which depends on the decision error.

5.2.1 Mean square error

Starting with the MSE (figure 5.1), the performance of this last method is worse than the others with a higher MSE on the training data. The use of SPO loss is not always favorable to the DL8.5 algorithm with a variable accuracy for the test set. These results come from the modified training function which trains the decision tree on the decision error. The prediction error of the cost vectors is not taken into account during the tree learning process explaining these mediocre results. Nevertheless, this last method seems to have the lowest MSE for the dataset of degree 20. This is quite surprising. It seems like the test set generally has better predictions than the training set for this data. The reason behind this is unclear but again, the large range of possible values for this dataset can throw off the results completely, making this the least constructive dataset.

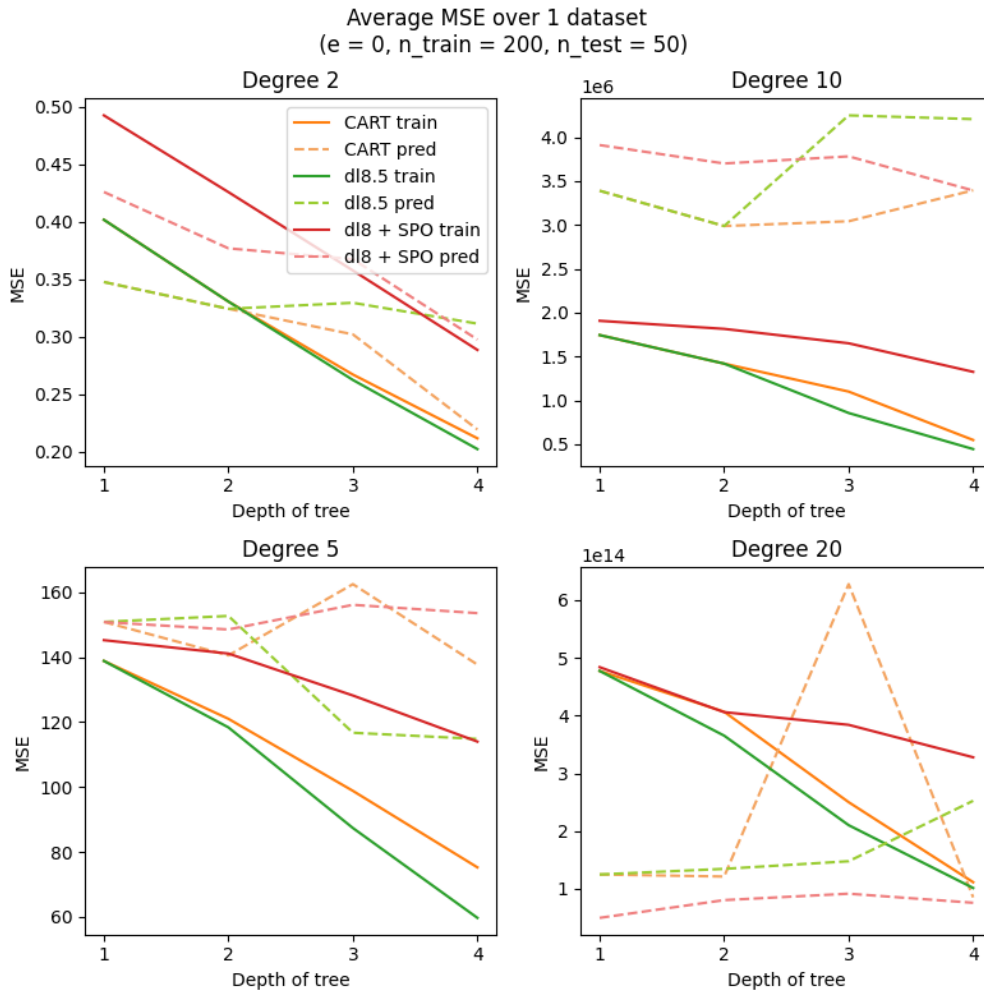


Figure 5.1: Average MSE with respect to the depth of the regression tree

5.2.2 Extra travel time

The results for SPO loss completely turn around when looking at the extra travel time. This final method gives the best predictions on the training set every time and for every depth. For the predictions on the test set, it gives lower extra travel time compared to DL8.5 using an MSE function and CART regression. The results on the test set are similar to the CART classifier results for datasets of degree 2 and 5 but using SPO loss results in lower extra travel time for the datasets of degree 10 and 20 for trees of all 4 depths. This is explained by this new method of training using the decision error that helps the tree find the correct path or at least a path similar to the shortest one.

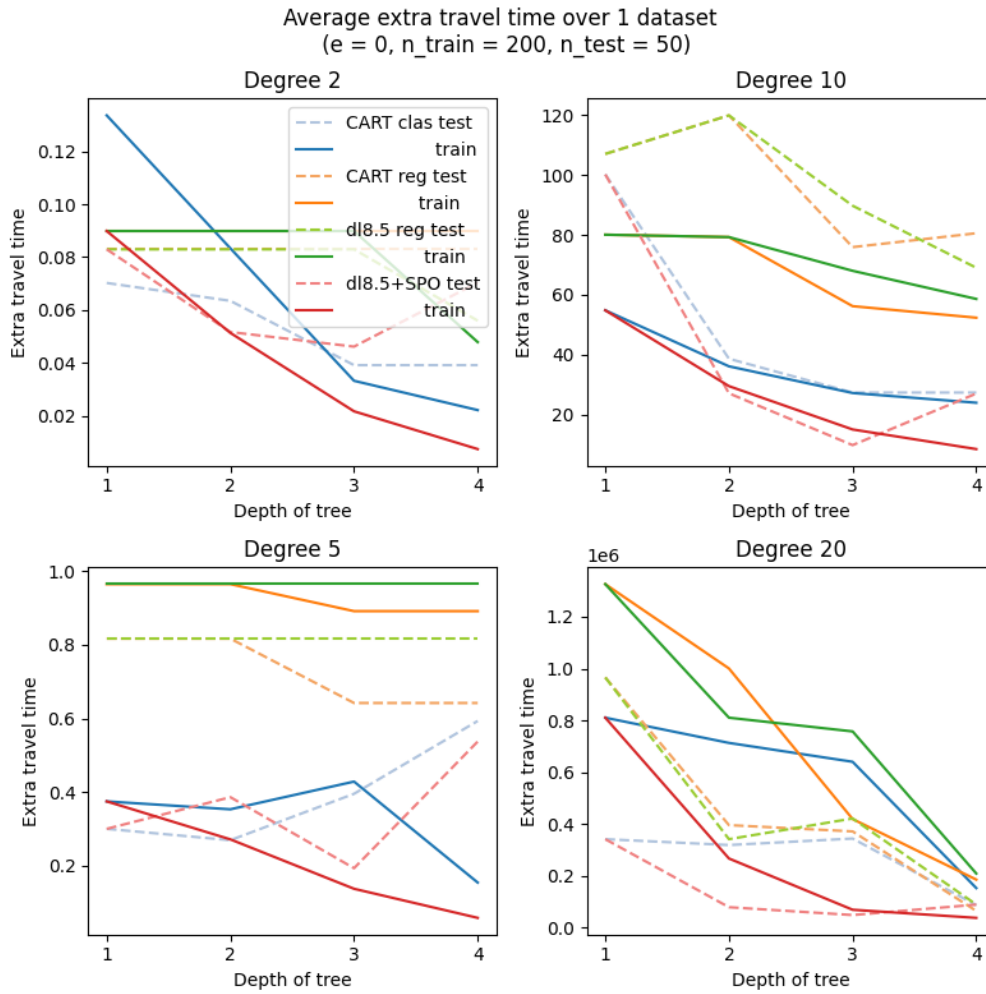


Figure 5.2: Average Extra travel time with respect to the depth of the regression tree

5.2.3 Relationship between MSE and Extra travel time

Again, the extra travel time with respect to MSE for every prediction shows no particular pattern except the same ones explained in section 4.2.3. The visual representation can be found in figure 7.3(a).

Chapter 6

Comparison of the different methods

After testing all four different types of decision trees on the 4×4 grid, the MSE and extra travel time of the different methods have been compared in previous sections. To have thorough evaluations of each method, other aspects still need to be analyzed for a more complete overview.

6.1 Path distribution

The number of shortest paths found and how many times each were used will be explored in more depth. Seeing that the CART classifier gave such accurate results compared to the other methods, this could be due to the use of decision error instead of a prediction error or maybe there simply aren't enough different classes to predict and a classifier with limited depth suffices to predict the paths used. On a 4×4 grid, it would seem like numerous options could be used to get from point A to B. As previously shown in table 3.1, the number of different paths predicted is actually quite low because multiple leaves predict the same paths. This is true for all four methods :

Dataset of :	degree 2	degree 5	degree 10	degree 20
CART classifier tree	2	4	4	8
CART regression tree	1	2	2	3
DL8.5 regression tree	3	3	4	6
DL8.5 + SPO loss regression tree	3	3	4	6

Table 6.1: Number of unique paths predicted from the decision tree of maximum depth 4 on a 4×4 grid

When analyzing the number of paths used, whether it be the optimal paths from the training set or test set, or a predicted path from any of the decision trees, the maximum is around 15. So in a 4×4 grid, around only 15 different paths are used. This number reduces even more when looking at the number of paths any one method can predict. In figure 6.1, when analyzing one color at a time, meaning one type of

decision tree at a time, a maximum of 6 is observed even though the bottom right graph is all trees of depth 4 meaning they could possibly predict up to 16 different paths. This could mean the classification problem is too simple, and the same paths are used over and over again by the training data leading to an overfitted tree that isn't capable of predicting all the possible paths.

In this path distribution diagrams (figure 6.1), the pink crosses and black diamonds indicate how many times the path is the optimal path in the training set and test set respectively. When a method overuses a path, it is very often the ones that are most present in the training set confirming the theory that the decision trees are overfitted. Although, for a tree with a very limited maximum depth like 1, the tree can only predict two different paths so overusing these paths is normal.

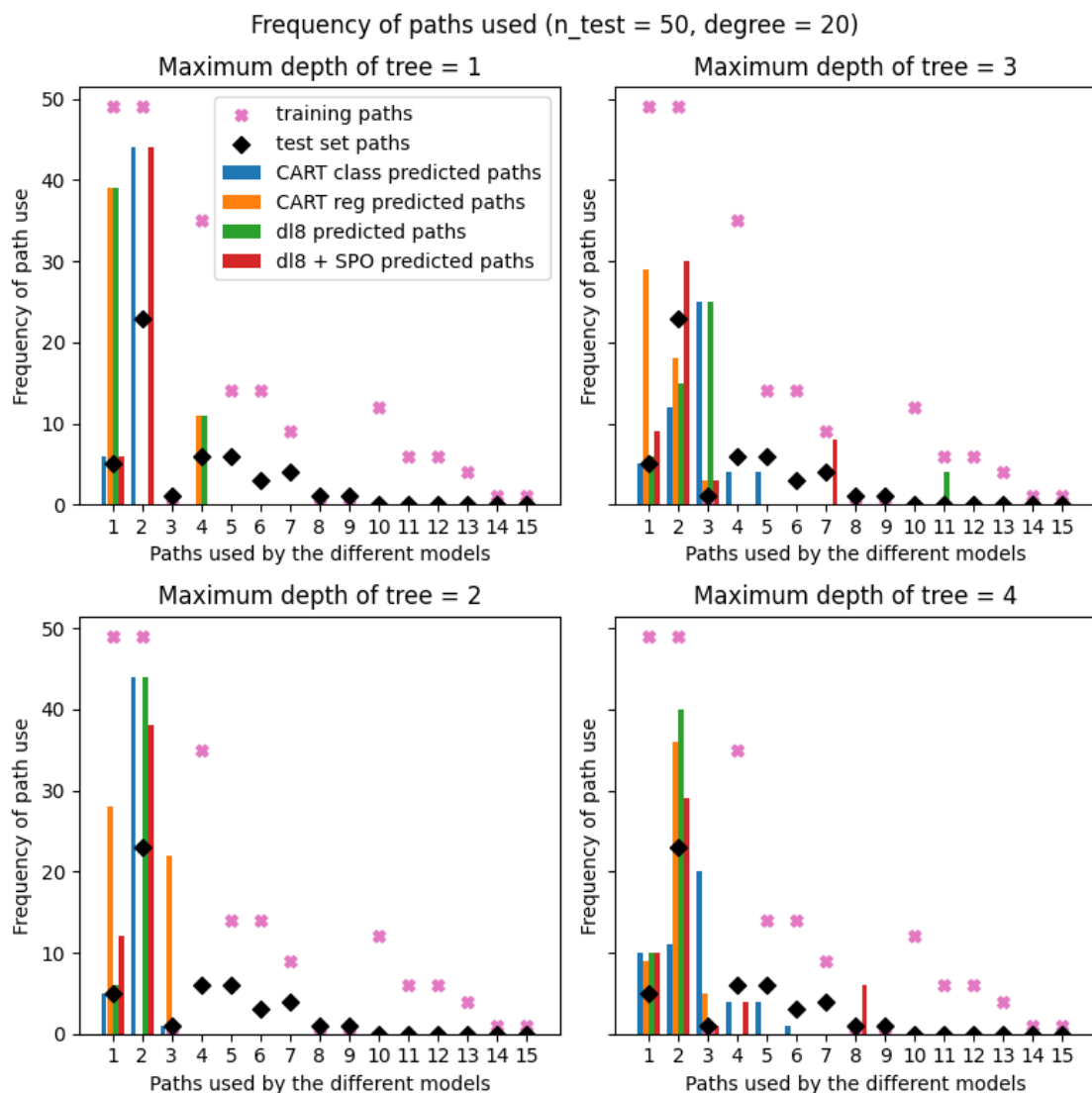


Figure 6.1: Frequency of paths used for data of degree 20 on a 4×4 grid

6.2 Shortest path accuracy

Having seen which paths were predicted and how many times, it is essential to understand if these predictions were correct or not. In table 6.2, the number of correctly predicted paths on the test set are indicated. So every time a prediction matches the real shortest path of the concerned feature vector, the counter is increased by 1. As the datasets become more complex due to the higher value of the *deg* parameter, the decision trees have a harder and harder time finding the exact correct paths. For datasets of degree 10 and 20, the SPO loss function starts to make a difference and achieves equivalent or better results than all the other methods. This confirms the previous observations made from figure 5.2 that the CART classifier and the DL8.5 tree with SPO loss give better results all around than the other two methods for the test set. They do indeed find the shortest path more often.

It is easy to notice that, as observed in the previous sections, increasing the tree depth does not necessarily mean finding the correct path more often. For datasets of degree 2 and 5, the numbers stay almost the same no matter how deep the tree is. For datasets of degree 10 and 20, the way the accuracy evolves as the depth of the tree increases corresponds exactly to what was seen in figure 5.2. This shows, as was expected, that the correct path accuracy is directly correlated to the extra travel time of a decision tree.

	degree 2				degree 5			
Tree of depth :	1	2	3	4	1	2	3	4
CART clas	44	45	45	45	45	46	45	42
CART reg	42	42	42	42	40	40	41	41
DL8.5 reg	42	42	42	44	40	40	40	40
DL8.5 + SPO	42	44	43	40	45	44	45	45
	degree 10				degree 20			
Tree of depth :	1	2	3	4	1	2	3	4
CART clas	32	35	38	38	24	24	18	22
CART reg	29	27	32	32	9	10	17	27
DL8.5 reg	29	27	26	32	9	24	15	26
DL8.5 + SPO	32	39	42	39	24	25	23	23

Table 6.2: Number of correct shortest paths found on a 4×4 grid (/50)

6.3 Computational time

Next is the computational time of all methods. On a 4×4 grid, there are 16 nodes and 24 edges. The time used by any of the methods is reasonable. However, the time required by the DL8.5 algorithm equipped with the SPO loss function is quite higher. Keep in mind that the DL8.5 decision tree explores all possible combinations of features which inevitably takes time. In addition, the SPO loss function requires the calculation of the shortest path for each of the observations multiple times with

real cost vector and predicted cost vector. These are the reasons the fourth method, the DL8.5 decision tree with SPO loss takes so much time.

Decision tree of :	depth 1	depth 2	depth 3	depth 4
CART regression tree	1 ms	1 ms	1 ms	1 ms
DL8.5 regression tree	0.45 s	0.63 s	5.61 s	34.8 s
DL8.5 + SPO loss regression tree	2.18 s	28.75 s	231.75	1275.5 s

Table 6.3: Average computational time (4×4 grid)

6.4 Conclusion

The different methods to train the decision trees do lead to very different results. The CART classifier and the DL8.5 tree trained on SPO loss are more accurate when it comes to extra travel time because they are trained using the shortest path. But, when it comes to mean square error, the methods trained on this specific cost function usually lead to better results. Nevertheless, SPO loss does seem to compete with reasonable accuracy when observing the test set.

Unfortunately, the small amount of optimal paths used does facilitate solving the problem. Simply the fact that decision trees with a maximum depth of 1 can accurately guess the right path more than 80 % of the time on datasets of degree 2 and 5 confirms that the problem is too simple. To fully grasp the abilities of these methods, they should be tested on more complex scenarios. The next two chapters will attempt to do so. Chapter 7 will detail the results on a noisy dataset where the noise parameter ϵ is 0.25, on the same 4×4 grid. Chapter 8 will extend the grid to larger dimensions adding difficulty to the shortest path optimization problem. The experiments were redone on a 25×25 grid, meaning 625 nodes and 1200 edges.

Chapter 7

Noisy data

All the previous chapters explain the four methods' accuracy. However, when generating the data, it is possible to introduce noise into the dataset as explained in section 2.1.4. So far, this noise parameter ϵ has always been set to 0. This chapter will explore the results on a noisy dataset where ϵ is 0.25 on a 4×4 grid.

7.1 MSE and Average Extra travel time

Starting with the MSE, observable in figure 7.1, there was no big surprise when the DL8.5 decision tree using SPO loss gave worse results than the CART regression and the DL8.5 trained with MSE on the training data. This matches the previous observation on data with no induced error. On the test set, again the SPO loss method was overall less effective.

For the extra travel time, results are consistent with what was seen in the previous results. Even though, results are overall worse on the noisy data. This is observed by the change in scale between both figures 5.2 and 7.2. The latest figure shows that the SPO loss and CART classifier are consistently the two best methods for the training set. On the test set, their results are better for the datasets of degree 2 and 5 but they do not perform as well for degrees 10 and 20. The induced noise added onto already complex data due to the high *deg* parameter causes the data to be hard to predict, therefore explaining these results.

Lastly, when comparing the connection between the MSE and extra travel time in figure 7.3, differences are visible when noise is added to the data. For the noisy data, guessing the exact shortest path happens much less often resulting in much less data points aligned vertically over 0.

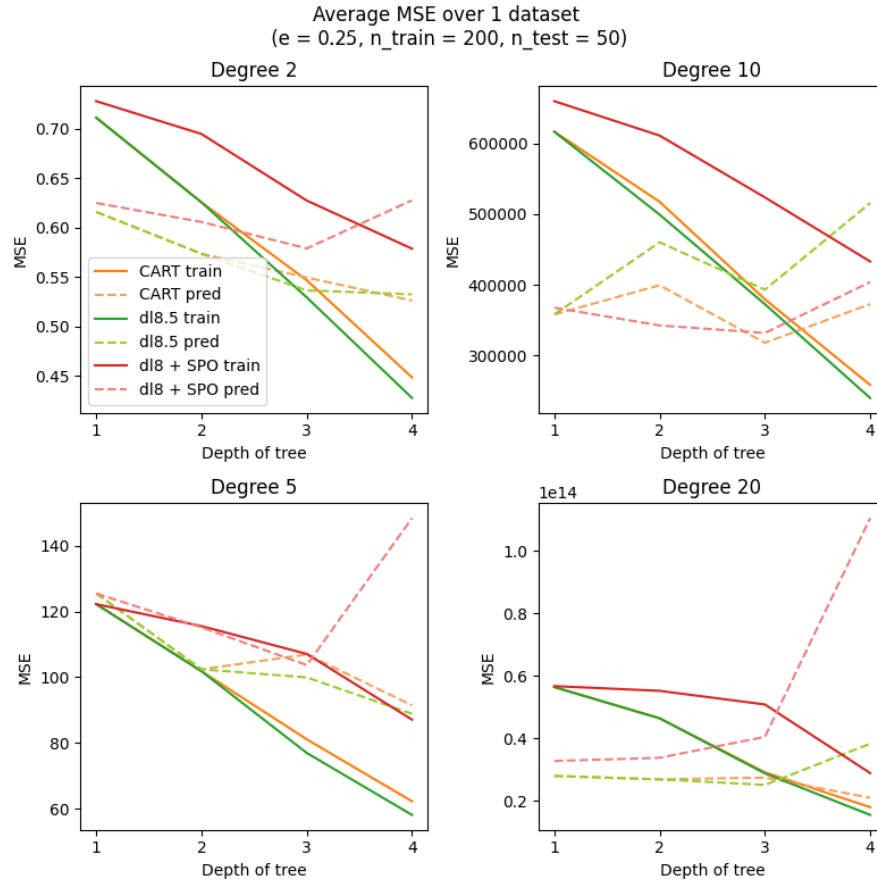


Figure 7.1: Average MSE with respect to the depth of the regression tree

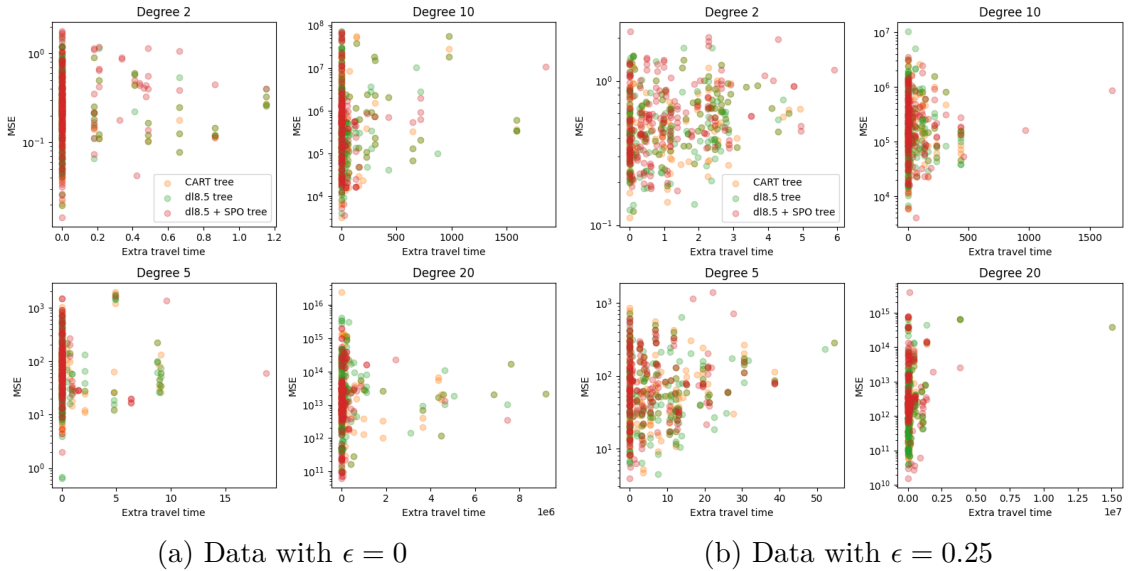


Figure 7.3: Extra travel time with respect to MSE for every prediction of the test set by trees of depth 1 to 4

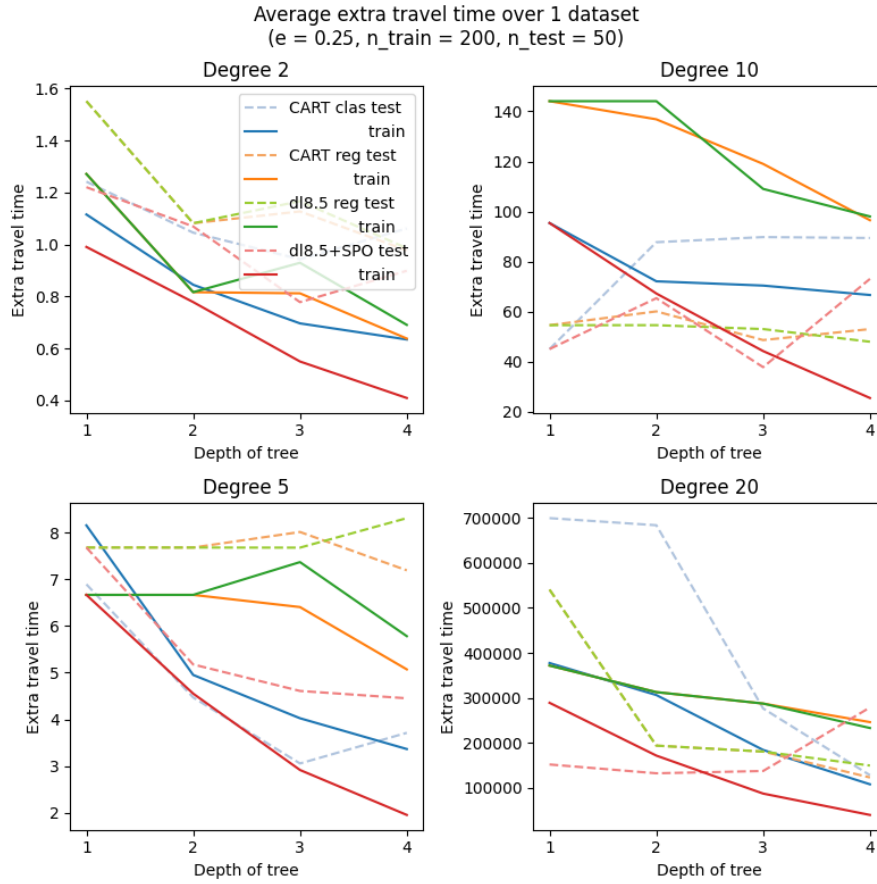


Figure 7.2: Average Extra travel time with respect to the depth of the regression tree

7.2 Shortest path accuracy

Having added a new level of complexity in the data, it has become more difficult to determine the right shortest path as noticed in figure 7.3. The table 7.1 shows a large drop in accuracy for the noisy data. All methods predicted the right path more than 80% of the time for datasets of degree 2 and 5. Now, with the added noise, the ratio drops down to 50% or much less. It seems like the cost vectors are harder to predict with this added difficulty. However, the dataset of degree 10 seems to maintain its level of accuracy compared to the dataset with zero noise.

	degree 2				degree 5			
Tree of depth :	1	2	3	4	1	2	3	4
CART clas	12	12	14	14	19	24	25	25
CART reg	4	13	13	15	14	14	14	16
DL8.5 reg	4	13	8	14	14	14	16	15
DL8.5 + SPO	10	11	14	14	14	19	22	23
	degree 10				degree 20			
Tree of depth :	1	2	3	4	1	2	3	4
CART clas	31	31	29	29	13	12	21	20
CART reg	30	26	27	26	13	19	14	24
DL8.5 reg	30	30	26	28	13	19	14	18
DL8.5 + SPO	31	31	33	28	18	21	21	26

Table 7.1: Number of correct shortest paths found on a 4×4 grid on noisy data (/50)

7.3 Path distribution

The number of used paths is still in the same range with maximum 17 different paths used compared to the 15 encountered for the data without noise. Like before, the trees do not take advantage of all their leaves, even with the added noise. The complete graph can be seen in figure 7.4.

7.4 Conclusion

It is clear that the noise adds some level of complication when trying to predict the cost vectors. The correct shortest paths are found much less often and the extra travel time is overall worse for all four methods. The relative results for one method compared to another seems to remain the same so even if one method performs badly, the methods that performed worse in the data with $\epsilon = 0$ still perform worse on the noisy data.

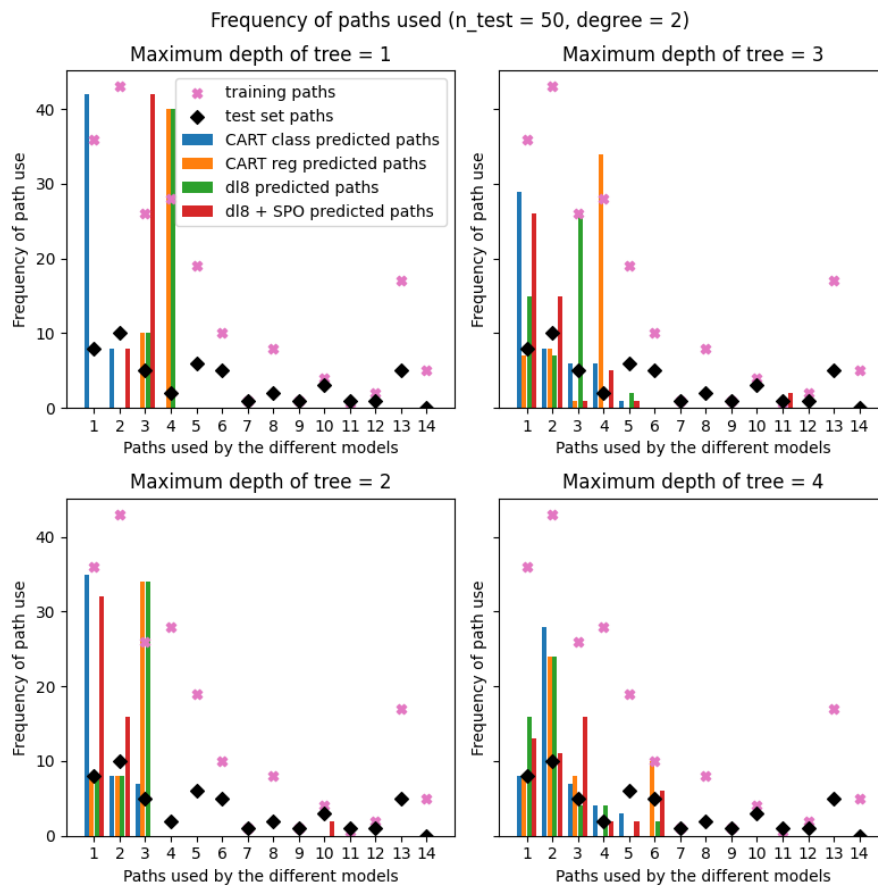


Figure 7.4: Frequency of paths used for a dataset of degree 2 on a 4×4 grid

Chapter 8

Increased complexity in the optimization problem

All previous chapters are dedicated to the study of the methods on a 4×4 grid and as the final test, all methods will now be tested on a much larger grid. The goal is to render the shortest path optimization problem harder to solve by adding more possible paths. The new grid will be of dimension 25×25 totaling in 625 nodes and 1200 edges. The following sections will describe the results for data generated with $\epsilon = 0$. The results for noisy data will be included at the end.

8.1 Path distribution

To first see if the problem is really more complex with these additional nodes and edges, the number of unique paths for each method will be analyzed. In table 8.1, the number of unique predicted paths is much larger and almost maximal every time compared to the smaller 4 by 4 grid. The DL8.5 decision tree trained on the SPO loss does however have less unique leaves with around half of the maximum number allowed. The following sections will reveal if this has a negative influence on its accuracy. With this added complexity to the graph, the number of paths actually used has increased quite a bit. Over 200 paths are taken by data from either the training or the test sets, meaning almost every path is unique due to the 200 samples in the training set and the 50 in the test set. This confirms that the shortest path problem is indeed more complex and that the maximum depth constraint is limiting.

Dataset of :	degree 2	degree 5	degree 10	degree 20
CART classification tree	15	15	14	15
CART regression tree	15	15	15	13
DL8.5 regression tree	16	15	15	16
DL8.5 + SPO loss regression tree	9	9	8	9

Table 8.1: Number of unique paths predicted from the decision tree of maximum depth 4 on a 25×25 grid

8.2 Shortest path accuracy

To understand if any methods stand out at a quick glance, an overview of how many times the correct shortest path was found is shown in table 8.2. Very different results than for the smaller grid (table 6.2) are observed. The numbers have drastically diminished proving that the added complexity of the problem makes it extremely hard to find the exact correct path. At this point the method that returns the path that is closest to the shortest one will stand out, so more details are needed to correctly analyse these methods.

	degree 2				degree 5				degree 10				degree 20			
Tree of depth :	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
CART classification	3	4	2	4	2	2	2	2	0	1	1	1	0	0	0	0
CART regression	0	0	1	0	0	1	2	2	0	0	0	0	0	0	0	0
DL8.5 regression	0	0	0	3	0	1	2	1	0	0	2	0	0	0	0	0
DL8.5 + SPO loss reg	0	1	3	0	0	0	3	1	1	1	1	1	0	0	0	0

Table 8.2: Number of correct shortest paths found on a 25×25 grid (/50)

8.3 Mean square error analysis

Like on the smaller 4×4 grid, multiple accuracy measures can be used. When observing the MSE, the methods trained on this error function are favored compared to the others and are expected to have more accurate results. It is easy to see in figure 8.1 that DL8.5 trained using the SPO loss function has much poorer results on the training set than the other two methods. This is as expected and has been previously observed in the smaller grid's results. However, on the test set, it outperforms the other methods or gives similar results for all but the dataset of degree 2. With the smaller number of unique paths found from the DL8.5 decision tree trained on the SPO loss, these more accurate results could originate from a well fitted tree unlike the other algorithms which have shown signs of overfitting.

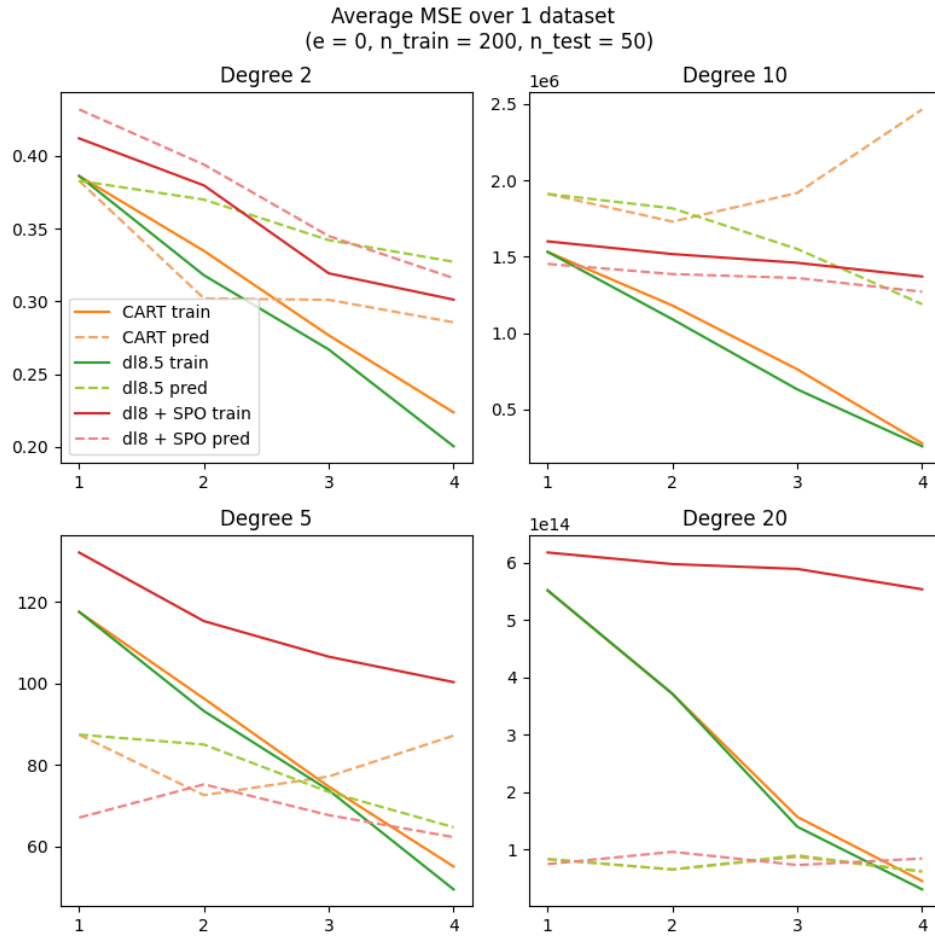


Figure 8.1: Average MSE with respect to the depth of the regression tree on a 25×25 grid

8.4 Extra travel time analysis

The next accuracy measure is extra travel time in figure 8.2. Starting with the CART classifier, let us remember that it outperformed the CART regression algorithm and DL8.5 almost systematically in the smaller grid. It now has the worst accuracy of all for the training and test sets. This shows that with additional possible paths, the classifier no longer has enough leaves to deal with all the possibilities of a bigger grid.

The CART regression and the DL8.5 algorithms share similar and intertwined results leading to the belief that the CART regression still manages to reach a somewhat optimal tree like DL8.5 guarantees. This is similar to what was observed in figure 5.2 for the 4×4 grid.

Finally, the preferred method when using this measure of error is the DL8.5 algorithm trained using the SPO loss function which uses the extra travel time in its calculation of the error. As expected, it almost systematically outperforms the other

methods, both on the training set and on the test set indicating the effectiveness of the SPO loss function. However, for the dataset of degree 2, it seems the SPO loss method has a greater error for the training set when the tree is of degree 4 than when it is of degree 3. This result is surprising due to the error function used. This should not even be theoretically possible and requires further investigation.

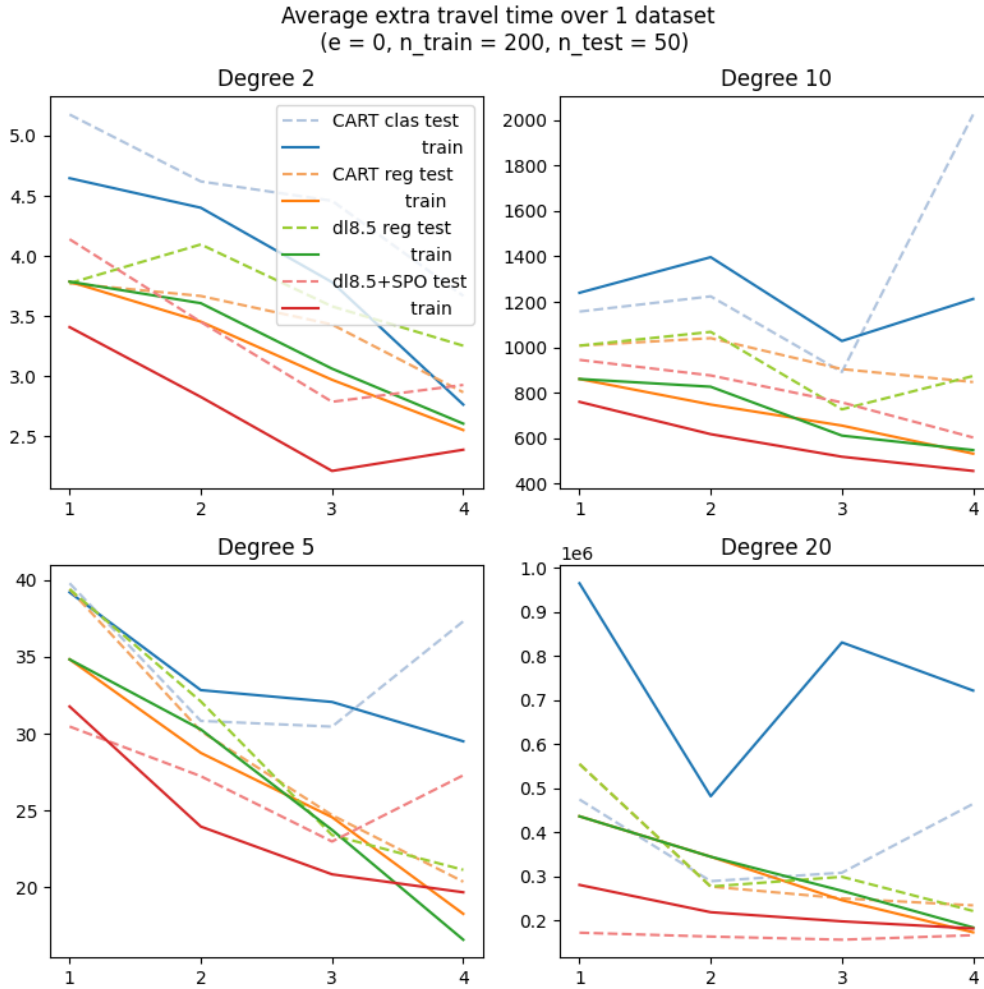


Figure 8.2: Average Extra travel time with respect to the depth of the regression tree on a 25×25 grid

8.5 Relationship between MSE and Extra travel time

With more possible paths, and varying decisions of the shortest path problem, the data looks much more dispersed than for the smaller grid. A clear outline of the bottom right triangle appears totally empty in all sub-graphs of figure 8.3. This seems logical because a larger extra travel time should mean a longer path than the one expected and if the prediction is off, it leads to a larger MSE. For the data of

degree 20, the data points are much closer to the left side with smaller extra travel times like in figure 4.3. With the abnormally large range of values for the data of degree 20, a few outliers with abnormally large extra travel time simply extends the scale too far and compresses the rest of the data.

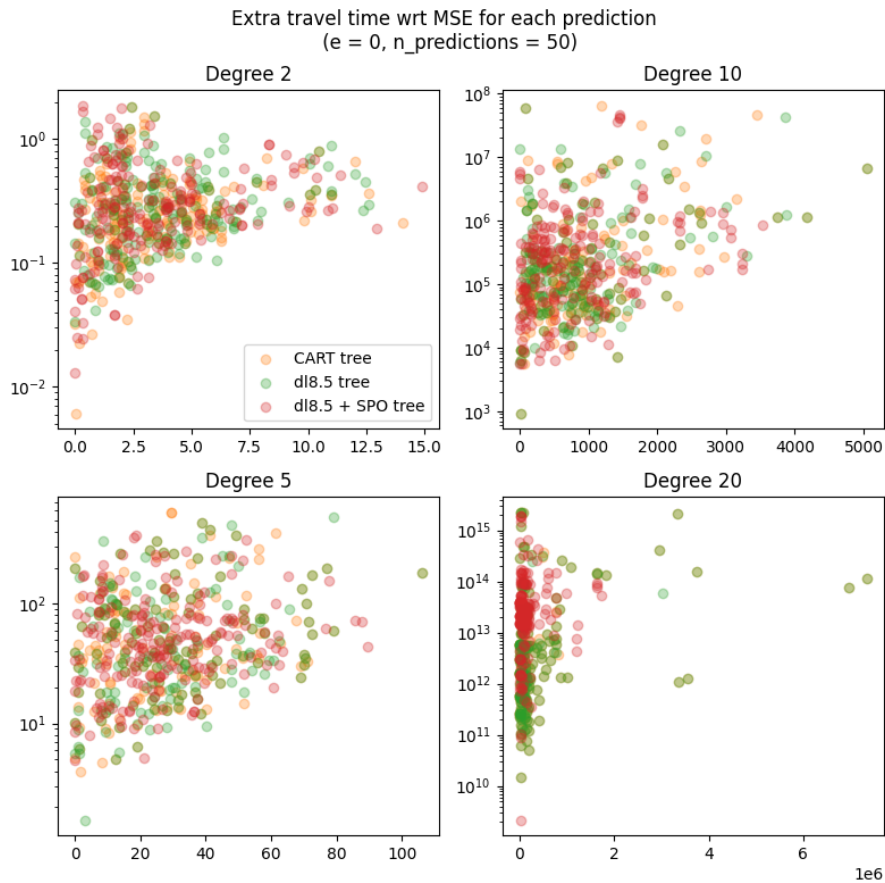


Figure 8.3: Extra travel time with respect to the MSE for every prediction of the test set by trees of depth 1 to 4 on a 25×25 grid

Compared to the smaller grid (figure 4.3), there is much less of a concentration to the left, where values indicate an extra travel time of 0. Due to the added complexity of the problem, the decisions trees can almost never find exactly the right path anymore explaining this observation.

Keep in mind the data of figure 8.3 mixes the predictions of all different depth trees together. When a sub-graph's data is separated by depth of tree, interesting patterns emerge. When comparing the methods with a maximum depth of 1 in figure 8.4, the DL8.5 algorithm with MSE and the CART regression tree have perfectly overlapping predictions. This is the case for all four datasets ($deg \in [2, 5, 10, 20]$). This confirms that the CART regression tree is capable of reaching a tree close to the optimal one like DL8.5, and in this case, the exact optimal tree. To put this into perspective, this case only contains one split, and both methods are trained on the same error function which explains why they reach the same answer.

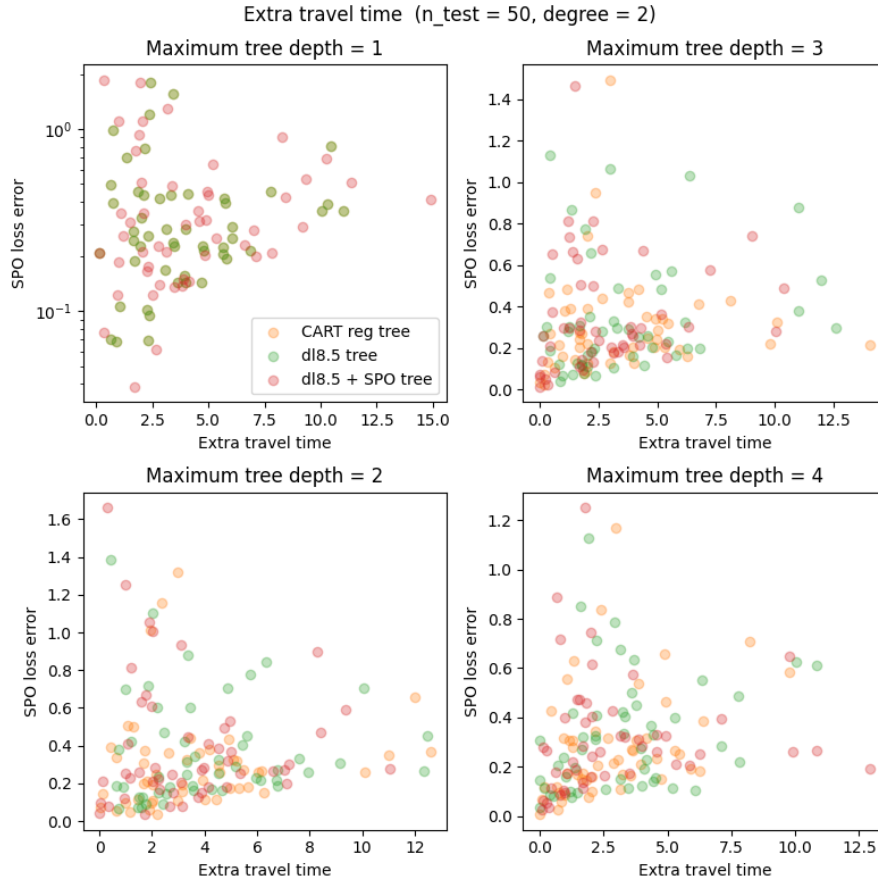


Figure 8.4: Extra travel time with respect to MSE for every prediction of the test set of a dataset of degree 2 on a 25×25 grid

8.6 Computational time

With the enlargement of the grid size comes added complexity and a longer computational time. The advantage of the DL8.5 algorithm is its capacity to give the best tree it has found without having to finish its entire search. However, if the computation times-out, it can no longer guarantee that it has returned the most optimal tree. Both DL8.5 decision trees were given a 4 hour time limit. Table 8.3 shows that the DL8.5 algorithm using MSE always reaches its optimal tree before reaching the time limit. Since the DL8.5 algorithm must search the entire decision space and every combination of features possible, it takes time. When using SPO loss, it must also calculate the shortest path for each of these combinations of features thus slowing the process down. This method consistently reached the time limit giving its best tree found in 4 hours. Looking back at the table 8.1, knowing that the search for the optimal tree times-out every time for DL8.5 with SPO loss, this may explain the lower number of unique paths for this tree.

Decision tree of :	depth 1	depth 2	depth 3	depth 4
CART regression tree	9.4 ms	10.9 ms	12.4 ms	14.1 ms
DL8.5 regression tree	1.74 s	23.44 s	204.99 s	13315.19 s
DL8.5 + SPO loss regression tree	381 s	6321 s	4 h timeout	4 h timeout

Table 8.3: Average computational time on a 25×25 grid

8.6.1 Use of an upper bound on the fitting process

To improve the decision trees without having to extend the time limit, the DL8.5 decision trees were given an upper bound. Using the fitted CART regression tree, its MSE and its SPO loss were determined in order to be used as upper bounds for the DL8.5 trees using MSE and SPO loss respectively. These bounds were calculated by using the error functions 4.2 and 5.5 (re-written here) and summing the value of each leaf together.

$$leaf_error(R) = \sum_{\forall i: x_i \in R} \sum_{\forall j} (c_{i,j} - \hat{c}(R)_j)^2$$

$$leaf_error(R) = \sum_{\forall i: x_i \in R} \left(\sum_{\forall j \in ShortestPath(\bar{c})} c_{i,j} - \sum_{\forall j \in ShortestPath(c_i)} c_{i,j} \right)$$

With T , the CART decision tree and R , a leaf in this tree, the upper-bound is:

$$UB_{MSE}(T) = \sum_{\forall R \in T} \sum_{\forall i: x_i \in R} \sum_{\forall j} (c_{i,j} - \hat{c}(R)_j)^2 \quad (8.1)$$

where $c_{i,j}$ is the j th element of the cost vector c_i and $\hat{c}(R)_j$ is the j th element of the predicted cost vector for all observations found in the leaf R and $x_i \in R$ representing the feature vectors that respect the conditions leading to leaf R . And:

$$UB_{SPO}(T) = \sum_{\forall R \in T} \sum_{\forall i: x_i \in R} \left(\sum_{\forall j \in ShortestPath(\bar{c}_R)} c_{i,j} - \sum_{\forall j \in ShortestPath(c_i)} c_{i,j} \right) \quad (8.2)$$

with $\bar{c}_R$ being the average cost vector found from all cost vectors in the data subset in the leaf R and $ShortestPath(\bar{c}_R)$, the shortest path found using the cost vector $\bar{c}_R$.

When given an upper-bound, the DL8.5 algorithm can quickly prune certain branches of the search tree. If the decision tree considered by DL8.5 has an error above this upper-bound, it is deemed unpromising and is pruned to save time. When a good feasible decision tree is found by the algorithm and it obtains a lower error than the given upper bound, its error will replace the value of the upper-bound to prune more branches faster.

Unfortunately, the use of this upper-bound on the models that timed-out in 4 hours did not increase the trees final accuracy. Since the time limit forces the algorithm to find its best tree so far, it cannot guarantee that the given solution is the optimal one, so an increase in accuracy is possible. However the given tree may be optimal but the algorithm simply has not had enough time to prove it by checking

the other possibilities. It is impossible to know without letting the computation finish entirely. For models that found the optimal tree in the allotted time, this upper bound did not shorten the computation time. The additional pruning from the given bound should have helped. From these observations, it can be deduced that the bound is not good enough and it would have to be smaller to help the decision trees improve.

8.7 Noisy Data

All previous figures for this bigger grid were for data generated with no noise ($\epsilon = 0$). Lastly, the results for the noisy data ($\epsilon = 0.25$) are described here. At this point, finding the exact shortest path is almost a game of luck with both the added noise and the added nodes to the grid. As mentioned before when $\epsilon = 0$, with the noisy data, the number of paths used by the training and test sets is well over 200 so almost every path is unique.

	degree 2				degree 5				degree 10				degree 20			
Tree of depth :	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
CART classification	1	0	0	0	2	2	1	1	0	1	0	0	0	0	0	0
CART regression	1	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0
DL8.5 regression	1	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0
DL8.5 + SPO loss reg	0	1	1	0	1	1	1	2	0	0	0	0	1	1	1	1

Table 8.4: Number of correct shortest paths found (/50)

The MSE of all methods for all datasets follows the pattern of the previous graphs. For the training set, SPO loss results is the least accurate and the CART regression tree and DL8.5 with MSE are neck and neck.

When it comes to extra travel time, results are as expected. On the training set, the CART classifier performs the worst and the SPO tree performs the best with the exception of the trees of depth 4 whom are sometimes outperformed. Now for the test set, these observations still remain true with the exception of the dataset of degree 20. In this last case, all bets are lost with the classifier outperforming the SPO loss function. This may be caused by the large range of values in this dataset. With only a few extremely high values, the whole average could be pulled up.

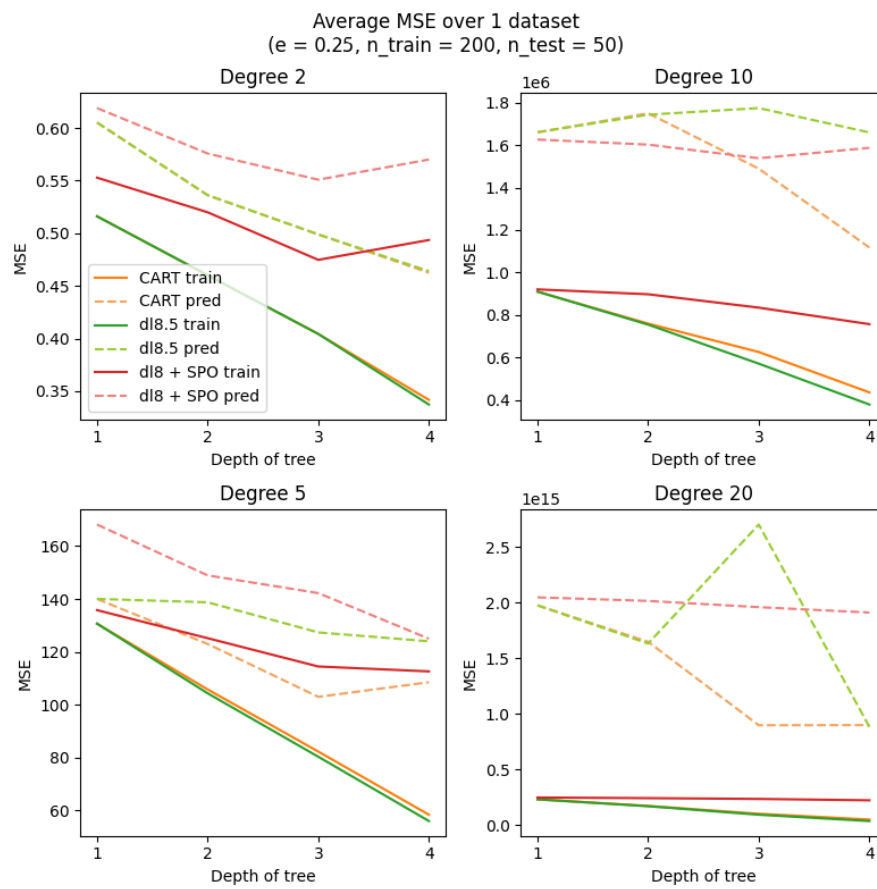


Figure 8.5: Average MSE with respect to the depth of the regression tree on a 25×25 grid with noisy data

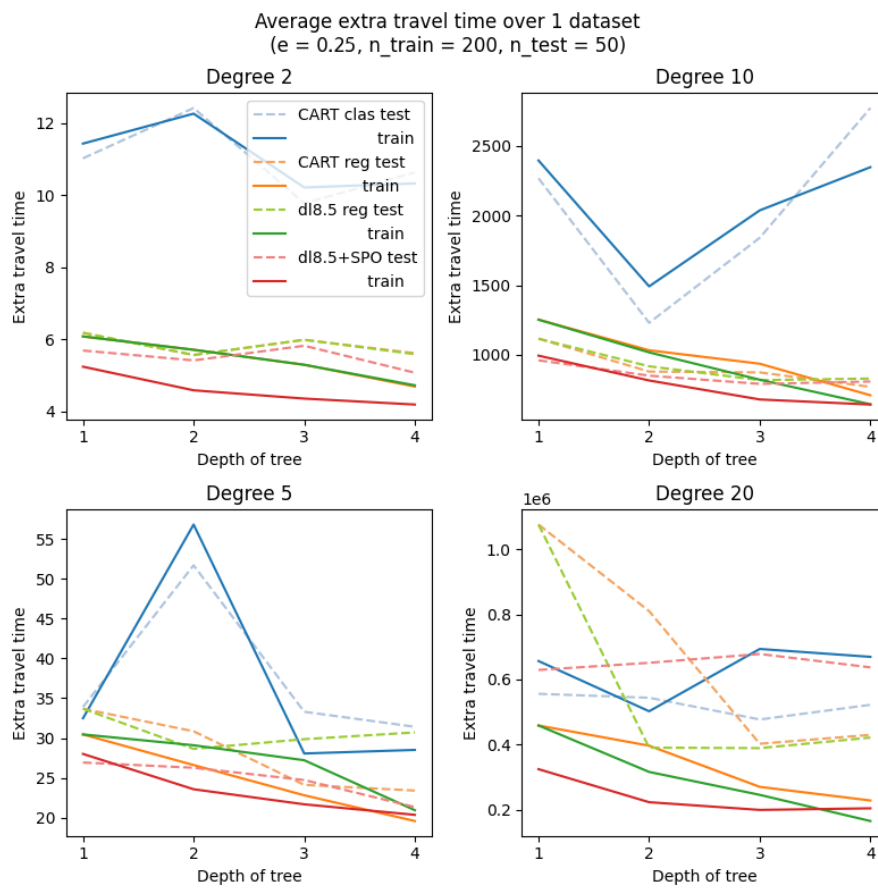


Figure 8.6: Extra travel time with respect to the depth of the regression tree on a 25×25 grid with noisy data

Conclusion

In this context of *predict-then-optimize*, different techniques have been tested here. When using a decision tree for predictions, the learning process depends greatly on the loss function used. More traditional loss functions like MSE are quite effective. Nevertheless, this particular setting is open for improvement. The overall objective of the *predict-then-optimize* framework is to minimize the final error, in this case, identifying the correct shortest path. However wrong the cost vector predictions are, this is of no importance if the final shortest path identified is the best one. This is where the SPO loss function has an advantage over MSE. SPO loss uses the decision error in its learning process. This does come at a certain computational cost but the results do reflect its effectiveness.

If the intermediate predictions require accuracy as well, then SPO loss may not be the best choice. By forcing the decision tree to minimize the final decision error (shortest path), it does overlook trying to minimize the prediction error of the cost vectors. For this reason, using the MSE remains more accurate in this case. As no particular link or pattern connects the MSE and the Extra travel time, it would be unlikely to find a loss function that minimizes both perfectly.

The situations of varying complexity that the four methods have encountered, have allowed the collection of a solid observation dataset. The behavior of each method was quite similar no matter the complexity with the exception of the CART classifier. This leads to the conclusion that the CART classifier is quite accurate in simpler situations, but is not capable of handling additional complexity. The DL8.5 regression tree used with MSE is more accurate than the CART regression tree when observing the MSE. These two have comparable results when it comes to the extra travel time. Finally, the DL8.5 decision tree with the SPO loss function is pretty systematically the best performing method when observing the decision error. It can be concluded DL8.5 and SPO loss perform well together.

With this information in hand, many interesting new paths emerge. Adapting this scenario to other types of optimization problems could be done. Studying SPO more closely could lead to expanding its use beyond decision trees, also in other types of supervised machine learning tools. DL8.5 could also be tested at greater length in simple prediction scenarios which it seems to handle very well.

Bibliography

- [1] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. “Learning Optimal Decision Trees Using Caching Branch-and-Bound Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04 (Apr. 2020), pp. 3146–3153. DOI: 10.1609/aaai.v34i04.5711. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/5711>.
- [2] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. “PyDL8.5: a Library for Learning Optimal Decision Trees”. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. Ed. by Christian Bessiere. Demos. International Joint Conferences on Artificial Intelligence Organization, July 2020, pp. 5222–5224. DOI: 10.24963/ijcai.2020/750. URL: <https://doi.org/10.24963/ijcai.2020/750>.
- [3] Dimitris Bertsimas and Jack Dunn. “Optimal classification trees.” In: *Mach. Learn.* 106.7 (2017), pp. 1039–1082. URL: <http://dblp.uni-trier.de/db/journals/ml/ml106.html#BertsimasD17>.
- [4] L. Breiman et al. *Classification and Regression Trees*. Taylor & Francis, 1984. ISBN: 9780412048418. URL: <https://books.google.be/books?id=JwQx-WOmSyQC>.
- [5] NetworkX Developers. *dijkstra_path*. 2004. URL: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx/algorithms/shortest_paths/weighted/dijkstra_path.html.
- [6] Adam N. Elmachtoub and Paul Grigas. *Smart "Predict, then Optimize"*. 2017. DOI: <https://doi.org/10.48550/arXiv.1710.08005>. arXiv: 1710.08005 [math.OC].
- [7] Adam N. Elmachtoub, Jason Cheuk Nam Liang, and Ryan McNellis. “Decision Trees for Decision-Making under the Predict-then-Optimize Framework”. In: *CoRR* abs/2003.00360 (2020). DOI: <https://doi.org/10.48550/arXiv.2003.00360>. arXiv: 2003.00360. URL: <https://arxiv.org/abs/2003.00360>.

Appendices

A Data generation Python Code

```
1 import numpy as np
2 import math
3 from scipy.stats import bernoulli
4
5 def gen_data(n, n_test_set, p, deg, epsilon):
6     X = np.random.uniform(0, 1, n * p)
7     X = np.reshape(X, (n, p))
8
9     X_test = np.random.uniform(0, 1, n_test_set * p)
10    X_test = np.reshape(X_test, (n_test_set, p))
11
12    p_bernouilli = 0.5
13    n_edges = 1200 #24 or 1200
14    B = bernoulli.rvs(p_bernouilli, size=p * n_edges)
15    B = np.reshape(B, (n_edges, p))
16
17    C = [[0 for _ in range(n_edges)] for _ in range(n)]
18
19    for n in range(n):
20        for k in range(n_edges):
21            B_row = B[k]
22            X_col = X[n]
23            Bx_ik = sum(a * b for a, b in zip(B_row, X_col))
24            base = (((1 / (math.sqrt(p))) * Bx_ik) + 1)
25            C[n][k] = pow(base, deg)
26            noise = np.random.uniform(1 - epsilon, 1 + epsilon, 1)
27            C[n][k] = C[n][k] * noise[0]
28
29    C_test = [[0 for _ in range(n_edges)] for _ in range(n_test_set)]
30    for n in range(n_test_set):
31        for k in range(n_edges):
32            B_row = B[k]
33            X_col = X_test[n]
34            Bx_ik = sum(a * b for a, b in zip(B_row, X_col))
```

```

35         base = (((1 / (math.sqrt(p))) * Bx_ik) + 1)
36         C_test[n][k] = pow(base, deg)
37         noise = np.random.uniform(1 - epsilon, 1 + epsilon, 1)
38         C_test[n][k] = C_test[n][k] * noise[0]
39
40     X_new = []
41     for x in X :
42         x_new = []
43         for feature in x:
44             if feature < 0.2 :
45                 x_new.extend([1, 0, 0, 0, 0])
46             elif feature < 0.4 :
47                 x_new.extend([0, 1, 0, 0, 0])
48             elif feature < 0.6 :
49                 x_new.extend([0, 0, 1, 0, 0])
50             elif feature < 0.8 :
51                 x_new.extend([0, 0, 0, 1, 0])
52             else:
53                 x_new.extend([0, 0, 0, 0, 1])
54         X_new.append(x_new)
55
56     X_test_new = []
57     for x in X_test:
58         x_new = []
59         for feature in x:
60             if feature < 0.2:
61                 x_new.extend([1, 0, 0, 0, 0])
62             elif feature < 0.4:
63                 x_new.extend([0, 1, 0, 0, 0])
64             elif feature < 0.6:
65                 x_new.extend([0, 0, 1, 0, 0])
66             elif feature < 0.8:
67                 x_new.extend([0, 0, 0, 1, 0])
68             else:
69                 x_new.extend([0, 0, 0, 0, 1])
70         X_test_new.append(x_new)
71
72     return X_new, X_test_new, C, C_test

```

B Model fitting and prediction Python Code

```
1 import networkx as nx
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from sklearn import tree
5 from sklearn.metrics import mean_squared_error
6 from sklearn import preprocessing
7 from statistics import mean
8 import collections
9 from pydl85 import DL85Predictor
10
11 # Data in .py file
12 import Bigdatawitherror as Bigdata
13
14 # To be adjusted : 4 or 25
15 grid_size = 25
16
17 # Import data
18 X_2_0 = Bigdata.X_2_0
19 X_test_2_0 = Bigdata.X_test_2_0
20 C_2_0 = Bigdata.C_2_0
21 C_test_2_0 = Bigdata.C_test_2_0
22
23 X_5_0 = Bigdata.X_5_0
24 X_test_5_0 = Bigdata.X_test_5_0
25 C_5_0 = Bigdata.C_5_0
26 C_test_5_0 = Bigdata.C_test_5_0
27
28 X_10_0 = Bigdata.X_10_0
29 X_test_10_0 = Bigdata.X_test_10_0
30 C_10_0 = Bigdata.C_10_0
31 C_test_10_0 = Bigdata.C_test_10_0
32
33 X_20_0 = Bigdata.X_20_0
34 X_test_20_0 = Bigdata.X_test_20_0
35 C_20_0 = Bigdata.C_20_0
36 C_test_20_0 = Bigdata.C_test_20_0
37
38
39 # SPO functions used to calculate the SPO error of a tree-----
40 def SPO_get_mean_cost(tids, C):
41     """
42     Gets average cost of cost vectors with id numbers included in tids
43     :param tids: the index number of the observations included in the leaf
44     :param C: All cost vectors
```

```

45     :return: average cost vector from all those selected by tids
46     """
47     tids = list(tids)
48     leaf_subset = np.array(C)[tids]
49     pred = []
50     for i in range(leaf_subset.shape[1]):
51         column = leaf_subset[:, i]
52         pred.append(np.mean(column))
53     return pred
54
55
56 def SPO_get_training_set_travel_time(C):
57     """ returns a dictionary with as key the index number of
58     the cost vector and as value the travel time for this cost vector"""
59     dict = {}
60     for i in range(len(C)):
61         g = make_grid_graph(C[i])
62         path = nx.dijkstra_path(g, (0, 0), (grid_size - 1, grid_size - 1))
63         tt1 = get_travel_time(g, path)
64         dict[i] = tt1
65     return dict
66
67
68 def SPO_get_leaf_travel_time(C_i, mean_cost):
69     """ returns the travel time of the leaf prediction on the grid of cost vector C_i"""
70     g = make_grid_graph(mean_cost)
71     path = nx.dijkstra_path(g, (0, 0), (grid_size - 1, grid_size - 1))
72     g2 = make_grid_graph(C_i)
73     tt1 = get_travel_time(g2, path)
74     return tt1
75     # -----
76
77     # -----
78 def make_grid_graph(edge_values):
79     """ Make networkx 2D grid with edges weighted according to the edge_values"""
80     graph = nx.grid_2d_graph(grid_size, grid_size)
81     edges = list(graph.edges())
82     for i in range(len(edges)):
83         start, end = edges[i]
84         graph[start][end]['weight'] = edge_values[i]
85     return graph
86
87
88 def format_path(path):
89     """ Shortest path in format ['d', 'h', 'l', 'k', 'j', 'n', 'm'] becomes
90     [('d', 'h'), ('h', 'l'), ('l', 'k'), ('k', 'j'), ('j', 'n'), ('n', 'm')] """

```

```

91     list = []
92     for i in range(len(path) - 1):
93         list.append((path[i], path[i + 1]))
94     return list
95
96
97 def makevisible(edge_values):
98     """ Return edge_values with only 3 first decimals for every value of the vector """
99     int_edge_values = []
100    for i in edge_values:
101        int_edge_values.append(float('% .3f' % (i)))
102    edge_values = int_edge_values
103    return edge_values
104
105
106 def make_visual_grid_graph(edge_values, shortest_path, title, color):
107     """
108     makes a figure with the highlighted shortest path
109     :param edge_values: cost vector as long as number of edges in the grid
110     :param shortest_path: the shortest path used on this grid
111     :param title: title of choice for the figure
112     :param color: color the shortest path will be on the graph
113     :return: save grid with highlighted shortest path
114     """
115     edge_values = makevisible(edge_values)
116     graph = nx.grid_2d_graph(grid_size, grid_size)
117     edges = list(graph.edges())
118
119     color_map = []
120     for node in graph.nodes():
121         if node in shortest_path:
122             color_map.append(color)
123         else:
124             color_map.append('lightgreen')
125     shortest_path = format_path(shortest_path)
126
127     edge_color_map = []
128     for i in range(len(edges)):
129         start, end = edges[i]
130         graph[start][end]['weight'] = edge_values[i]
131         if (start, end) in shortest_path:
132             edge_color_map.append(color)
133         elif (end, start) in shortest_path:
134             edge_color_map.append(color)
135         else:
136             edge_color_map.append('black')

```

```

137
138     # weights = nx.get_edge_attributes(graph, 'weight') #dictionary form
139     # weights = [graph[u][v]['weight'] for u, v in graph.edges()] # list form
140
141     pos = {(x, y): (y, x) for x, y in graph.nodes()}
142     plt.figure(figsize=(30, 30))
143     nx.draw(graph, pos=pos,
144             node_color=color_map,
145             edge_color=edge_color_map,
146             # width = weights,
147             # with_labels=True,
148             node_size=14400 / grid_size)
149     edge_labels = nx.get_edge_attributes(graph, 'weight')
150     # nx.draw_networkx_edge_labels(graph, pos, edge_labels)
151     plt.savefig('ShortestPaths/' + title)
152     plt.close()
153     return
154
155
156 def get_travel_time(graph, path):
157     """ uses the path given and calculates its travel time on the given graph"""
158     travel_time = 0
159     # print(graph.edges())
160     for i in range(len(path) - 1):
161         weights = nx.get_edge_attributes(graph, 'weight')
162         # print(weights)
163         try:
164             weight = weights[(path[i], path[i + 1])]
165             travel_time += weight
166         except KeyError:
167             try:
168                 weight = weights[(path[i + 1], path[i])]
169                 travel_time += weight
170             except KeyError:
171                 print('Really not fixed')
172
173     return travel_time
174
175
176 def get_training_paths(C):
177     """ returns a list of shortest paths, one for each cost vector in C
178     (list of cost vectors)"""
179     paths = []
180     for i in range(len(C)):
181         # Shortest path algo
182         g = make_grid_graph(C[i])

```

```

183         path = nx.dijkstra_path(g, (0, 0), (grid_size - 1, grid_size - 1))
184         paths.append(path)
185     return paths
186
187
188 def update_path_distribution(distrib, paths,
189                             category):
189     # category = training, realdl8, preddl8, realskl, predskl,
190     # realSP0, predSP0, realbaseline, predbaseline
191     """
192     :param distrib: dictionary of paths used
193     :param paths: list of shortest paths
194     :param category: origin of these shortest paths so that they are added to the
195                     correct category on the dictionary
196     :return: an updated path distribution dictionary
197     """
198     for path in paths:
199         if (str(path) in distrib.keys()):
200             distrib[str(path)][category] += 1
201         else:
202             distrib.update({str(path): {"training": 0, "realdl8": 0, "preddl8": 0,
203                                         "realskl": 0, "predskl": 0, "realSP0": 0,
204                                         "predSP0": 0, "realbaseline": 0,
205                                         "predbaseline": 0}})
206             distrib[str(path)][category] += 1
207     return distrib
208
209
210
211 def path_distribution_to_lists(distrib):
212     """changes format of the path distribution dictionary to multiple lists
213     to be used in the plots later"""
214     paths = [[], [], [], []]
215     training = [[], [], [], []]
216     realdl8 = [[], [], [], []]
217     preddl8 = [[], [], [], []]
218     realskl = [[], [], [], []]
219     predskl = [[], [], [], []]
220     realSP0 = [[], [], [], []]
221     predSP0 = [[], [], [], []]
222     realbaseline = [[], [], [], []]
223     predbaseline = [[], [], [], []]
224     for i in range(4): # depth
225         for (k, v) in distrib[i].items():
226             paths[i].append(k)
227             for (k2, v2) in v.items():
228                 if k2 == "training":

```

```

229         training[i].append(v2)
230     elif k2 == "realdl8":
231         realdl8[i].append(v2)
232     elif k2 == "pred dl8":
233         pred dl8[i].append(v2)
234     elif k2 == "realskl":
235         realskl[i].append(v2)
236     elif k2 == "pred skl":
237         pred skl[i].append(v2)
238     elif k2 == "realSP0":
239         realSP0[i].append(v2)
240     elif k2 == "predSP0":
241         predSP0[i].append(v2)
242     elif k2 == "realbaseline":
243         realbaseline[i].append(v2)
244     else:
245         predbaseline[i].append(v2)
246
247     return [paths, training, realdl8, pred dl8, realskl, pred skl,
248            realSP0, predSP0, realbaseline, predbaseline]
249
250
251 def get_tree_predictions(tree, predictions):
252     """ returns the cost vectors that the tree predicts for each leaf
253         only works for dl8.5 trees """
254     if 'value' in tree.keys():
255         predictions.append(tree['value'])
256     if 'left' in tree.keys():
257         get_tree_predictions(tree['left'], predictions)
258     if 'right' in tree.keys():
259         get_tree_predictions(tree['right'], predictions)
260     return predictions
261
262
263 def get_tree_travel_time_and_paths(tree):
264     """ returns the travel time and the path of each prediction in the
265         tree and also a list of the unique predictions """
266     predictions = get_tree_predictions(tree, [])
267     travel_times = []
268     paths = []
269     unique_paths = []
270     for i in range(len(predictions)):
271         g = make_grid_graph(C[i])
272         path = nx.dijkstra_path(g, (0, 0), (grid_size - 1, grid_size - 1))
273         if path not in paths:
274             unique_paths.append(path)

```

```

275         paths.append(path)
276         travel_times.append(get_travel_time(g, path))
277     return (travel_times, paths, unique_paths)
278
279
280 def get_baseline_extra_travel_time(C_test, predictions):
281     """
282     :param C_test: The test set cost vectors
283     :param predictions: The baseline's predictions of these cost vectors
284     :return: list of the the extra travel time normalized, list of the the
285             extra travel time and list of the paths used and the paths it
286             should have taken
287     """
288     list_extra_travel_time = []
289     real_paths = get_training_paths(C_test)
290     paths = []
291
292     for i in range(len(predictions)):
293         g_2 = make_grid_graph(C_test[i])
294         path = predictions[i]
295         tt1 = get_travel_time(g_2, path)
296         tt2 = get_travel_time(g_2, real_paths[i])
297         paths.append(path)
298         list_extra_travel_time.append(abs((tt2 - tt1)))
299     list_extra_travel_time_norm = preprocessing.normalize([list_extra_travel_time])
300     list_extra_travel_time_norm = list_extra_travel_time_norm[0]
301
302     return list_extra_travel_time_norm, list_extra_travel_time, [paths, real_paths]
303
304
305 def get_list_extra_travel_time(C, C_test, predictions, trainingpredictions, clf):
306     """
307     :param C: list of the training cost vectors
308     :param C_test: list of the test cost vectors
309     :param predictions: list of the test cost vector predictions
310     :param trainingpredictions: list of the training cost vector predictions
311     :param clf: the type of model used to do the predictions
312     :return: all the information necessary for the plots after
313     """
314     MSE = mean_squared_error(C_test, predictions)
315     trainingMSE = mean_squared_error(C, trainingpredictions)
316     MSEs = []
317     trainingMSEs = []
318     real_paths = []
319     pred_paths = []
320     predicted_travel_costs = []

```

```

321
322     for i in range(len(C_test)):
323         MSEs.append(mean_squared_error(C_test[i], predictions[i]))
324     for i in range(len(C)):
325         trainingMSEs.append(mean_squared_error(C[i], trainingpredictions[i]))
326
327     list_extra_travel_time = []
328
329     for i in range(len(predictions)):
330         # Shortest path algo
331         g = make_grid_graph(predictions[i])
332         path = nx.dijkstra_path(g, (0, 0), (grid_size - 1, grid_size - 1))
333         g_2 = make_grid_graph(C_test[i])
334         path_2 = nx.dijkstra_path(g_2, (0, 0), (grid_size - 1, grid_size - 1))
335
336         pred_paths.append(path)
337         real_paths.append(path_2)
338
339         tt1 = get_travel_time(g_2, path) # shortest path found with predictions
340         predicted_travel_costs.append(tt1)
341         tt2 = get_travel_time(g_2, path_2) # shortest path found with reel values
342         list_extra_travel_time.append(abs((tt2 - tt1)))
343
344
345     training_list_extra_travel_time = []
346     for i in range(len(trainingpredictions)):
347         # Shortest path algo
348         g = make_grid_graph(trainingpredictions[i])
349         path = nx.dijkstra_path(g, (0, 0), (grid_size - 1, grid_size - 1))
350         g_2 = make_grid_graph(C[i])
351         path_2 = nx.dijkstra_path(g_2, (0, 0), (grid_size - 1, grid_size - 1))
352
353         tt1 = get_travel_time(g_2, path) # shortest path found with predictions
354         tt2 = get_travel_time(g_2, path_2) # shortest path found with reel values
355         training_list_extra_travel_time.append(abs((tt2 - tt1)))
356
357     list_extra_travel_time_norm = preprocessing.normalize([list_extra_travel_time])
358     list_extra_travel_time_norm = list_extra_travel_time_norm[0] # [0] because [[x,x,x]],
359     return list_extra_travel_time_norm, MSE, trainingMSE, list_extra_travel_time, MSEs, \
360         trainingMSEs, [pred_paths, real_paths], predicted_travel_costs, \
361         training_list_extra_travel_time
362
363
364 # SETTING UP THE PARAMETERS
365 n_train = 200 # Number of samples in the training set
366 n_test = 50 # Number of samples in the test set

```

```

367
368 p = 5                # Number of features in the feature vectors
369 n_edges = grid_size * (grid_size - 1) * 2    # Number of edges in the grid
370 epsilon = 0          # Noisy parameter in the data generation process
371 n_datasets = 1
372 visu = None
373 deg = [2, 5, 10, 20]
374 y_scale = 'log'      # Can be linear, log
375 max_depth_list = [1, 2, 3, 4] # List of maximum depths for the predictor models
376 # positions in the subplots later on:
377 positions = [(0, 0), (1, 0), (0, 1), (1, 1)]
378
379 # Setting up the figures
380 fig1, axs1 = plt.subplots(2, 2, figsize=(8, 8))
381 fig2, axs2 = plt.subplots(2, 2, figsize=(8, 8))
382 fig3, axs3 = plt.subplots(2, 2, figsize=(8, 8))
383 fig5, axs5 = plt.subplots(2, 2, figsize=(8, 8))
384
385 # Setting up the data from the import at the top
386 X_full = [X_2_0[:n_train], X_5_0[:n_train], X_10_0[:n_train], X_20_0[:n_train]]
387 X_test_full = [X_test_2_0[:n_test], X_test_5_0[:n_test],
388               X_test_10_0[:n_test], X_test_20_0[:n_test]]
389 C_full = [C_2_0[:n_train], C_5_0[:n_train], C_10_0[:n_train], C_20_0[:n_train]]
390 C_test_full = [C_test_2_0[:n_test], C_test_5_0[:n_test],
391               C_test_10_0[:n_test], C_test_20_0[:n_test]]
392
393 # Do all 4 datasets each with a different degree parameter
394 for i in range(4):
395     print('Bernouilli matrix degree = %s' % (deg[i]))
396     X, X_test, C, C_test = X_full[i], X_test_full[i], C_full[i], C_test_full[i]
397
398     # MSE error function
399     def error(tids):
400         tids = list(tids)
401         C_subset = np.array(C)[tids]
402         pred = leaf_value(tids)
403         error = 0
404         for cost in C_subset:
405             for i in range(len(cost)):
406                 error += pow((cost[i] - pred[i]), 2)
407         return error
408
409     # SPO loss error function
410     def get_SPO_loss(tids):
411         tids = list(tids)

```

```

413     SUM = 0
414     dict = SPO_get_training_set_travel_time(C)
415     mean_cost = SPO_get_mean_cost(tids, C)
416     for id in tids:
417         SUM += abs(SPO_get_leaf_travel_time(C[id], mean_cost) - dict[id])
418     return SUM # /len(X)
419
420
421     # leaf value assignment
422     def leaf_value(tids):
423         tids = list(tids)
424         leaf_subset = np.array(C)[tids]
425         pred = []
426         for i in range(leaf_subset.shape[1]):
427             column = leaf_subset[:, i]
428             pred.append(np.mean(column))
429         return pred
430
431
432     # Set up all the variables for the data that needs to be visualized
433     path_distrib = [{}, {}, {}, {}]
434
435     baseline_extra_travs5 = [[], [], [], []]
436     baseline_training_extra_travs5 = [[], [], [], []]
437
438     baseline_count_correct_paths = []
439     sk_count_correct_paths = []
440     count_correct_paths = []
441     SPO_count_correct_paths = []
442
443     sk_extra_trav_norms = []
444     sk_MSEs = [[], [], [], []]
445     sk_trainingMSEs = [[], [], [], []]
446     sk_extra_travs = []
447     sk_extra_travs5 = [[], [], [], []]
448     sk_training_extra_travs5 = [[], [], [], []]
449     sk_MSE_multiples = []
450     sk_training_MSE_multiples = []
451
452     extra_trav_norms = []
453     MSEs = [[], [], [], []]
454     trainingMSEs = [[], [], [], []]
455     extra_travs = []
456     extra_travs5 = [[], [], [], []]
457     training_extra_travs5 = [[], [], [], []]
458     MSE_multiples = []

```

```

459     training_MSE_multiples = []
460
461     SPO_extra_trav_norms = []
462     SPO_MSEs = [[], [], [], []]
463     SPO_trainingMSEs = [[], [], [], []]
464     SPO_extra_travs = []
465     SPO_extra_travs5 = [[], [], [], []]
466     SPO_training_extra_travs5 = [[], [], [], []]
467     SPO_MSE_multiples = []
468     SPO_training_MSE_multiples = []
469
470     # Do all depths for each dataset
471     for l in range(4):
472         print('Max depth = %s' % (max_depth_list[l]))
473
474         # BASELINE MODEL CART CLASSIFIER-----
475         baseline = tree.DecisionTreeClassifier(max_depth=max_depth_list[l])
476         paths = get_training_paths(C)
477         # get baseline from paths to int categories
478         categories = []
479         mapping = {}
480         demapping = {}
481         counter = 0
482         for path in paths:
483             if str(path) in mapping.keys():
484                 categories.append(mapping[str(path)])
485             else:
486                 mapping[str(path)] = counter + 1
487                 demapping[counter + 1] = path
488                 counter += 1
489                 categories.append(mapping[str(path)])
490         baseline = baseline.fit(X, categories)
491
492         baseline_predictions = baseline.predict(X_test)
493         predictions = []
494         for pred in baseline_predictions:
495             predictions.append(demapping[pred])
496         baseline_predictions = predictions
497
498         baseline_trainingpredictions = baseline.predict(X)
499         predictions = []
500         for pred in baseline_trainingpredictions:
501             predictions.append(demapping[pred])
502         baseline_trainingpredictions = predictions
503         # -----
504

```

```

505  # CART REGRESSION TREE-----
506  sk_clf = tree.DecisionTreeRegressor(max_depth=max_depth_list[1])
507  sk_clf = sk_clf.fit(X, C)
508  sk_predictions = sk_clf.predict(X_test)
509  sk_trainingpredictions = sk_clf.predict(X)
510  # -----
511
512  # CALCULATE MSE UPPER BOUND FROM FITTED CART REGRESSION TREE-----
513  samples = collections.defaultdict(list)
514  dec_paths = sk_clf.decision_path(X)
515
516  for d, dec in enumerate(dec_paths):
517      for node in range(sk_clf.tree_.node_count):
518          if dec.toarray()[0][node] == 1:
519              samples[node].append(d)
520
521  CARTError = 0
522  for sample in range(len(samples)):
523      CARTError += error(samples[sample])
524  # -----
525
526  # CALCULATE SPO UPPER BOUND FROM FITTED CART REGRESSION TREE-----
527  CARTSPOLoss = 0
528  for sample in range(len(samples)):
529      CARTSPOLoss += get_SPO_loss(samples[sample])
530  # -----
531
532  # DL8.5 WITH MSE DECISION TREE-----
533  clf = DL85Predictor(max_depth=max_depth_list[1],
534                      error_function=error,
535                      leaf_value_function=leaf_value,
536                      max_error=CARTError, time_limit=14400)
537  clf = clf.fit(X)
538  predictions = clf.predict(X_test)
539  trainingpredictions = clf.predict(X)
540
541  # -----
542
543  # DL8.5 WITH SPO DECISION TREE-----
544  SPO_clf = DL85Predictor(max_depth=max_depth_list[1],
545                          error_function=get_SPO_loss,
546                          leaf_value_function=leaf_value,
547                          max_error=CARTSPOLoss, time_limit=14400)
548  SPO_clf = SPO_clf.fit(X)
549  SPO_predictions = SPO_clf.predict(X_test)
550  SPO_trainingpredictions = SPO_clf.predict(X)

```

```

551
552 # -----
553
554 # Get all the data necessary for the figures
555 baseline_extra_trav_norm, baseline_extra_trav, baseline_paths = \
556     get_baseline_extra_travel_time(C_test, baseline_predictions)
557 baseline_training_extra_trav_norm, baseline_training_extra_trav, \
558     baseline_training_paths = \
559     get_baseline_extra_travel_time(C, baseline_trainingpredictions)
560 sk_extra_trav_norm, sk_MSE, sk_trainingMSE, sk_extra_trav, \
561     sk_MSE_multiple, sk_training_MSE_multiple, sk_paths, \
562     sk_travel_costs, sk_training_extra_trav = \
563     get_list_extra_travel_time(deg[i], max_depth_list[l], X, X_test,
564                               C, C_test, sk_predictions,
565                               sk_trainingpredictions, 'skl')
566 extra_trav_norm, MSE, trainingMSE, extra_trav, MSE_multiple, \
567     training_MSE_multiple, paths, travel_costs, training_extra_trav = \
568     get_list_extra_travel_time(deg[i], max_depth_list[l], X, X_test,
569                               C, C_test, predictions,
570                               trainingpredictions, 'dl8')
571 SPO_extra_trav_norm, SPO_MSE, SPO_trainingMSE, SPO_extra_trav, \
572     SPO_MSE_multiple, SPO_training_MSE_multiple, SPO_paths, \
573     SPO_travel_costs, SPO_training_extra_trav = \
574     get_list_extra_travel_time(deg[i], max_depth_list[l], X, X_test,
575                               C, C_test, SPO_predictions,
576                               SPO_trainingpredictions, 'SPO')
577
578 # Add all the information to the stocking variables to use later
579
580
581 baseline_extra_travs5[l].append(baseline_extra_trav)
582 baseline_training_extra_travs5[l].append(baseline_training_extra_trav)
583 sk_extra_travs5[l].append(sk_extra_trav)
584 sk_training_extra_travs5[l].append(sk_training_extra_trav)
585 extra_travs5[l].append(extra_trav)
586 training_extra_travs5[l].append(training_extra_trav)
587 SPO_extra_travs5[l].append(SPO_extra_trav)
588 SPO_training_extra_travs5[l].append(SPO_training_extra_trav)
589
590 baseline_count_correct_paths.append(baseline_extra_trav.count(0))
591 sk_count_correct_paths.append(sk_extra_trav.count(0))
592 count_correct_paths.append(extra_trav.count(0))
593 SPO_count_correct_paths.append(SPO_extra_trav.count(0))
594
595 path_distrib[l] = update_path_distribution(path_distrib[l],
596                                           baseline_paths[0], "predbaseline")

```

```

597     path_distrib[l] = update_path_distribution(path_distrib[l],
598                                                baseline_paths[1], "realbaseline")
599
600     path_distrib[l] = update_path_distribution(path_distrib[l],
601                                                get_training_paths(C), "training")
602
603     sk_extra_trav_norms.append(sk_extra_trav_norm)
604     sk_MSEs[l].append(sk_MSE)
605     sk_trainingMSEs[l].append(sk_trainingMSE)
606     sk_extra_travs.extend(sk_extra_trav)
607     sk_MSE_multiples.extend(sk_MSE_multiple)
608     sk_training_MSE_multiples.extend(sk_training_MSE_multiple)
609     path_distrib[l] = update_path_distribution(path_distrib[l],
610                                                sk_paths[0], "predskl")
611     path_distrib[l] = update_path_distribution(path_distrib[l],
612                                                sk_paths[1], "real skl")
613
614     extra_trav_norms.append(extra_trav_norm)
615     MSEs[l].append(MSE)
616     trainingMSEs[l].append(trainingMSE)
617     extra_travs.extend(extra_trav)
618     MSE_multiples.extend(MSE_multiple)
619     training_MSE_multiples.extend(training_MSE_multiple)
620     path_distrib[l] = update_path_distribution(path_distrib[l],
621                                                paths[0], "pred dl8")
622     path_distrib[l] = update_path_distribution(path_distrib[l],
623                                                paths[1], "real dl8")
624 \
625     SPO_extra_trav_norms.append(SPO_extra_trav_norm)
626     SPO_MSEs[l].append(SPO_MSE)
627     SPO_trainingMSEs[l].append(SPO_trainingMSE)
628     SPO_extra_travs.extend(SPO_extra_trav)
629     SPO_MSE_multiples.extend(SPO_MSE_multiple)
630     SPO_training_MSE_multiples.extend(SPO_training_MSE_multiple)
631     path_distrib[l] = update_path_distribution(path_distrib[l],
632                                                SPO_paths[0], "pred SPO")
633     path_distrib[l] = update_path_distribution(path_distrib[l],
634                                                SPO_paths[1], "real SPO")
635

```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl