UCLouvain

epl

École polytechnique de Louvain

Quantile Regression by Optimal Decision Trees

In collaboration with H&M Group

Author: **Valentin Lemaire**
Supervisor: **Siegfried Nijssen**
Readers: **Pierre Schaus, Marco Trincavelli, Gaël Aglin**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Abstract

Quantile regression is a field that has many interesting properties when compared to standard regression, including robustness with respect to datasets that present a lot of variability and have many outliers. In some contexts, these predictions need to be justifiable and hence, the models making said predictions need to be interpretable. This work will evaluate how optimal decision trees can be used to generate quantile predictions that are highly interpretable and how to analyze the trees obtained to not only explain the decisions for individual quantiles but to provide easily interpretable information on the whole conditional distribution of a dataset.

Keywords Quantile regression, quantile loss, optimal decision trees, conditional density estimations, interpretability.

Acknowledgements

I would like to thank Prof. Siegfried Nijssen for his time, availability and precious counsel in the elaboration and writing of this thesis. As well as Gaël Aglin for his help throughout the year.

I would also like to thank Marco Trincavelli and David Andersson from H&M Group for their valuable input and help in the brainstorming and the conception of the contributions.

A special thank-you goes to my family and friends, especially Alix de Pierpont, Augustin d'Oultremont, Brieuc de Voghel, Gauthier de Moffarts and Gwenaëlle Dembour, who have supported me and provided continuous encouragement throughout this project.

Contents

CHAPTER 1

Introduction

This chapter will introduce the work of this master's thesis. This starts with the context in which the thesis was realized, stating the challenges and the stakes that are raised in said context.

This will be followed by a description of the motivations of the thesis, explaining the research questions that will be analyzed and answered. Finally, we will finish this chapter by explaining the structure of the rest of the thesis.

1.1 Context presentation

Machine learning techniques have been enjoying a huge interest in the previous years, both in industrial and academic settings. Because of their repeatedly shown efficiency and strengths, a lot of research has been conducted to create new models that are always more accurate.

This race to the most efficient models has led to creating more and more complicated models, like deep neural networks, recurrent neural networks and many more sophisticated techniques. While these approaches have shown more than excellent results, they are often treated as black boxes that are completely uninterpretable to a human analyst, both because the values of the weights themselves don't have a precise meaning and because of the very large number of said weights.

This is the reason why, more recently, a lot of new research has been conducted to create simple and interpretable models that still give satisfactory results. This is especially the case when being able to explain the model's decision is as important as having a very accurate result. This is true in several medical settings, business strategy settings, and many more. This search for interpretable models will be a core component of this work.

Interpretable models were of particular interest to H&M Group, with whom this thesis was written in collaboration. They were interested in seeing how optimal decision trees, and more precisely DL8.5 trees, could help them in predicting quantile values, i.e., a value for which a certain percentage of the dataset falls below said value, for a dataset describing the sales of articles. Indeed, they wanted to use the predictions in an optimization framework to ensure the good restocking of shops and warehouses. Standard models predicting the expectation of the target were not enough, as they needed to overestimate how many sales they would make to be sure to always have enough stock to answer the demand.

This is where quantile predictions come in. The special interest in DL8.5, an algorithm able to find optimal decision trees, was due to the fact that it is able to optimize quantile loss with mathematical guarantees and that the final models are interpretable, thus enabling them to explain the decisions made by the model to the business side of the company.

1.2 Motivations

After exploring how we could combine the business requirements of H&M Group and the properties and qualities of DL8.5, we realized that it would be interesting for H&M Group to be able to generate many decision trees, each corresponding to a different quantile, so that they could interpret them together as an ensemble method or choose the "important" quantiles that made the most sense based on the data and the analysis. This became the main focus of the thesis. In this objective, some motivations and research directions became primordial. They are detailed below.

1.2.1 Efficiency

DL8.5 is an algorithm that, due to its optimal nature, is slow and memory consuming. Therefore, when using it and creating variations, it is primordial to have efficiency with respect to these two aspects in mind. In this thesis, we will explore how we can learn many optimal trees for different quantiles at once, limiting both the use of memory and CPU time.

This is especially important as the DL8.5 algorithm is packaged into a python library, and it is the goal of this thesis to bring any new variations to this algorithm to the same package. Therefore, if it is intended to be usable by anyone, it needs to hold an efficient implementation.

1.2.2 Quality

DL8.5 does hold the optimality guarantee, meaning that it will find the tree that fits best the training set, however in practice, to avoid overfitting to a training set, by time constraints and to help interpretability, it is often used with maximum depth and minimum support constraints.

Another goal of this thesis will be to evaluate if DL8.5 is an adequate algorithm to perform regression and, more precisely, to generate quantile predictions on large, real-life datasets whose target feature presents a high variability.

This thesis will explore how the algorithm performs on these points, while comparing it to standard algorithms that hold properties similar to those of DL8.5.

1.2.3 Interpretability

One of the main interests in using decision trees rather than more complicated methods is that they are highly interpretable. This is a core property of such methods, and it is therefore important to keep this property true when creating variations based on this family of algorithms.

In this thesis, we will evaluate how learning many trees whose purpose is to minimize quantile loss impacts the interpretability of the obtained model and how we can actually combine the results of the individual trees to create compound interpretations to better understand and interpret the full distribution of the predicted feature of a dataset.

1.3 Thesis structure

The thesis will follow a clear structure that allows to easily assess what were the contributions of this work and what research question they attest. This structure is explained here.

This introduction is followed by Chapter 2, treating of related work. In that chapter, we will present a few theoretical concepts that form the foundation for the rest of the thesis. This will be followed by a clear explanation of the learning procedure for optimal decision trees using DL8.5, finishing with an overview of similar methods and showing why none of them fully answer the challenges raised at the start of this chapter.

Chapter 3 will present the first contributions of the thesis by exploring how the DL8.5 algorithm can be extended in practice to be usable in a regression context, and more precisely a quantile regression context.

Chapter 4 goes further into the contributions by presenting the full, optimized algorithm that allows to learn many optimal trees, each optimizing the quantile loss for a different quantile. More specifically, this chapter will explain how DL8.5 was extended to *efficiently* learn all these trees at once.

The last chapter treating of contributions of this thesis is Chapter 5, where the experiments to evaluate the new algorithm are detailed. In these experiments, we answer the research questions and bring forward ways to interpret the results and models generated.

Finally, we will end this thesis with Chapter 6 that is composed of a section about future work and a conclusion to this project.

CHAPTER 2

Related work

In this chapter, we present the theoretical concepts on which we will build our quantile estimates. To this effect, we will explore briefly the formal definitions of probability density functions and quantiles, followed by the inner workings of finding an optimal decision tree using DL8.5, and finishing by a review of existing methods that explore the use of decision trees to estimate probability densities.

2.1 Quantiles

As a main building block in the goals of this thesis, we will make heavy use of the notion of quantiles and how they're related to probability density functions. In this section, we will explain and formalize these concepts.

2.1.1 Probability density functions

Let us assume we have a random variable Y. The realizations of Y, denoted y, belong to a certain domain $\mathcal{Y} \subseteq \mathbb{R}$. The random variable Y features a probability density function (pdf) $f : \mathbb{R} \to \mathbb{R} : y \to f(y)$ from which we can determine probabilities of ranges included in the domain. The probability to observe a realization whose value y is comprised between y_1 and y_2 is given by

$$\mathbb{P}[y_1 \leq y \leq y_2] = \int_{y_1}^{y_2} f(\lambda) d\lambda. \tag{2.1}$$

Such functions have interesting properties, such as the fact that they are always positive and that their integral over their entire domain is always equal to 1. It is also such that for any $y \in \mathbb{R} \setminus \mathcal{Y}$, $f(y) = 0$.

From the pdf of a random variable, we can derive its cumulative density function (cdf) $F : \mathbb{R} \to [0, 1] : y \to F(y)$, a concept of which we will also make heavy use in this thesis. Said function is defined using the pdf by

$$F(y) = \mathbb{P}[-\infty \leq y' \leq y] = \int_{-\infty}^{y} f(\lambda) d\lambda. \tag{2.2}$$

As this function is the integral of the same function over a domain increasing with the value y, its value is also monotonically increasing. We can also note that all cdfs share the same asymptotic values as $\mathbb{P}[y \leq -\infty] = 0$ and $\mathbb{P}[y \leq +\infty] = 1$.

2.1.2 Quantiles of a distribution

Using the above defined concepts, we can introduce the quantile notion. Given a probability q, the quantile value is a real $y_q \in \mathcal{Y}$ such that the probability that a realization $y \in \mathcal{Y}$ of the random variable Y is lower than y_q is q. Formally, using the definition of [1], this is defined as

$$y_q = \arg\inf_{y \in \mathcal{Y}} \{\mathbb{P}[y' \leq y] \geq q\} = \arg\inf_{y \in \mathcal{Y}} \{F(y) \geq q\}. \tag{2.3}$$

This definition is of course unique. From this, we can also explore the quantile function of a distribution, $Q : [0, 1] \to \mathcal{Y} : q \to Q(q)$ whose definition is given in equation (2.4).

$$Q(q) = \arg\inf_{y \in \mathcal{Y}} \{F(y) \geq q\} \tag{2.4}$$

From this, we observe that the quantile function is actually the reciprocal of the cumulative density function; $Q(q) = F^{-1}(q)$. Indeed, the cdf gives the probability q that a random realization falls below a certain value y_q and the quantile function gives the value y_q such that the probability of a random realization to fall below y_q is q.

In the remainder of this thesis, when we talk about quantiles we mean the probability $q \in [0, 1]$ and when we talk about quantile values we mean the real value $y_q \in \mathcal{Y}$ corresponding to a quantile q.

2.2 Learning optimal decision trees with DL8.5

Throughout this thesis, the goal is to adapt the DL8.5 algorithm to enable it to optimize the quantile loss in the most efficient way possible. In this section, we will explore how the DL8.5 algorithm exposed in [2] finds optimal decision trees efficiently.

As a prelude, it must be known that DL8.5 strictly operates on boolean data. This will however not be a limiting factor in finding optimal trees, as any categorical feature can be transformed in binary features with a one-hot encoding per category. Continuous features can also be transformed in binary features by creating binary tests with different threshold values.

In the remainder of this thesis, we will assume all features in the data are binary.

While the algorithm is able to do more than classification tasks, the authors of [2] did not expose it. This will be discussed in further sections, until then we will assume that the predicted values are class labels.

2.2.1 Decision trees in the context of itemset Mining

To find optimal decision trees, [2] uses a transactional database $\mathcal{D}$ that can be expressed as

$$\mathcal{D} = \{(t, I, p) \mid t \in \mathcal{T}, I \in \mathcal{I}, p \in \mathcal{P}\} \tag{2.5}$$

Where $\mathcal{T}$ is the set of transaction IDs, $\mathcal{I}$ the set of possible itemsets and $\mathcal{P}$ is the set of possible target values which are class labels.

In this representation, the features are represented as itemsets. Therefore, each row (or transaction) of the original binary database is represented as an itemset where each item is either the positive i or negative $\neg i$ value for each feature i. Tables 2.1a and 2.1b show the binary and transactional representations of an example dataset, respectively.

Using this representation of the data, [2] defines the cover on an itemset I as $cover(I) = \{(t', I', p') \in \mathcal{D} \mid I \subseteq I'\}$, thus corresponding to the transactions t' which have I as a subset of their own itemset I'. Similarly, they define the class support as $Sup(I, p) = |\{(t', I', p') \in cover(I) \mid p' = p\}|$.

A	B	C	p
0	1	1	0
1	0	1	1
0	0	1	1
0	1	0	0
1	0	0	1
0	0	0	0
0	0	1	0
1	1	0	1
0	0	0	1
0	0	1	0
0	0	0	1

(a) Binary dataset

t	I	p
1	$\neg a$, b, c	0
2	a, $\neg b$, c	1
3	$\neg a$, $\neg b$, c	1
4	$\neg a$, b, $\neg c$	0
5	a, $\neg b$, $\neg c$	1
6	$\neg a$, $\neg b$, $\neg c$	0
7	$\neg a$, $\neg b$, c	0
8	a, b, $\neg c$	1
9	$\neg a$, $\neg b$, $\neg c$	1
10	$\neg a$, $\neg b$, c	0
11	$\neg a$, $\neg b$, $\neg c$	1

(b) Transactional dataset

Table 2.1: Binary and transactional version of a small example dataset

Another important notion is the leaf error function. This is a function that takes an itemset I and gives the error over the transactions of $cover(I)$. For this, it must assign a predicted value to the set of transactions. [2] uses the misclassification error, defined as

$$leaf\_error(I) = |cover(I)| - \max_{p \in \mathcal{P}} \{Sup(I, p)\}. \tag{2.6}$$

Again, using this representation, each path in a decision tree can be mapped to a certain itemset. Hence, given a dataset $\mathcal{D}$, the goal is to find a decision tree $\mathcal{DT} \subset \mathcal{I}$ such that

- The itemsets of $\mathcal{DT}$ represent a decision tree,
- $\sum_{I \in \mathcal{DT}} leaf\_error(I)$ is minimal,
- $\forall\ I\ \in \mathcal{DT}$, $|I| \leq maxdepth$, where $maxdepth$ is the maximum depth of the tree,
- $\forall\ I\ \in \mathcal{DT}$, $|cover(I)| \geq minsup$, where $minsup$ is the minimum support threshold

The last two items are additional pruning constraints on the learned decision tree that will help in making the search space smaller and in practice will be used to mitigate overfitting of the learned trees.

2.2.2 Decision tree learning domain

In the domain of decision tree models, two main categories exist. Standard decision tree models that create just one tree that is used to perform the predictions, and ensemble methods that create multiple tree models and combine them to make the predictions. This second category is generally referred to as forests when the underlying individual models are decision trees.

Decision trees Figure 2.1 shows an example of a decision tree for the example dataset shown in Table 2.1a. This is a tree obtained with the standard CART algorithm for decision trees as presented in [3] when constrained to a maximum depth of two. In this figure, the leaf nodes are twice circled, and the values represent the class they predict.

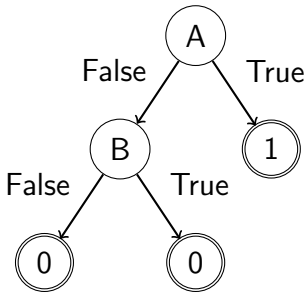


Figure 2.1: Example decision tree

Forests Figure 2.2 shows an example random forest for the same dataset. Here we see that the algorithm generates N possibly different decision trees. In this example, we set $N = 3$. Once these trees have been generated, the algorithm derives importance weights for each of the trees and the predictions will be a weighted majority vote of the predictions of the trees in the case of classification like this example or a weighted sum in the case of regression.

The examples in Figures 2.1 and 2.2 were generated from the `scikit-learn` classes `DecisionTreeClassifier` and `RandomForestClassifier` respectively.

Heuristic versus optimal When learning decision trees, two families of algorithms exist. The first and most common one is the family of algorithms based on heuristics. They grow their trees greedily based on some heuristic, usually creating splits for the features that decrease the most the information of the dataset after the split, to encourage similar samples to end up in the same leafs. On the other side, we have

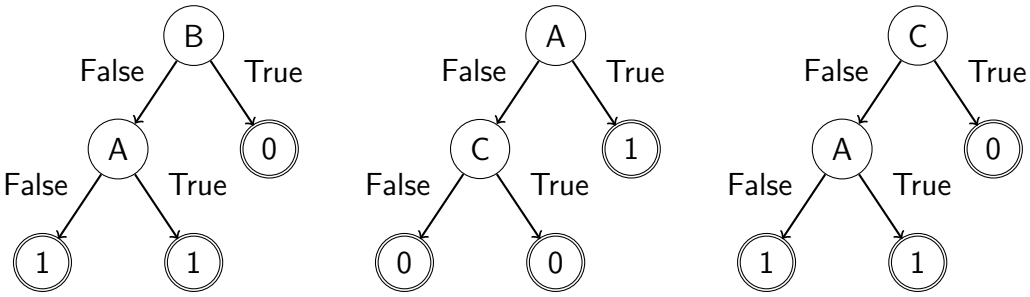


Figure 2.2: Example forest

optimal algorithms. These algorithms are able to find the best tree, under certain constraints, to optimize a certain loss function.

The main challenge of heuristic algorithms is to derive an efficient heuristic that performs well on the loss considered. There has been extensive research in finding efficient heuristics when it comes to minimize categorical cross-entropy for classification and mean squared error for regression, but other losses rarely have custom heuristics to optimize them.

While optimal algorithms do not need any heuristic, one of the core challenges for this family of algorithms is to find the optimum with reasonable time and memory resources, as such algorithms usually rely on exhaustive search.

Because the goal of this thesis is to optimize quantile loss, it makes sense to use the DL8.5 algorithm, which belongs to the optimal family of algorithms. Indeed, defining an efficient heuristic for quantile loss is not at all a straightforward problem.

2.2.3 DL8 learning

To solve the problem stated in Section 2.2, [2] introduced the DL8.5 algorithm, which itself is based on the DL8 algorithm presented in [4]. The DL8 algorithm explores how dynamic programming can be used to find decision trees with a guarantee to be optimal.

This dynamic programming way to explore the search space goes as follows. To compute the best tree for an itemset I, the algorithm computes the best trees for the itemsets I'. The itemsets I' are defined as $I' = I \cup \{i\}$ where i is either f or $\neg f$ for each feature f not already represented in I, i.e., all features f' for which $I \cap \{f', \neg f'\} = \emptyset$. The chosen tree for itemset I is then the one that minimizes the sum of the errors of the positive and negative branches found for each feature.

Algorithm 1 DL8($maxdepth$, $minsup$)

struct $BestTree\{tree : Tree, error : float\}$
$cache \leftarrow HashSet < Itemset, BestTree >$
$(\tau, b) \leftarrow DL8 - Recurse(\emptyset)$
return τ
procedure $DL8 - Recurse(I)$
    $solution \leftarrow cache.get(I)$
    if $solution$ was found **then**
        return $(solution.tree, solution.error)$
    if $leaf\_error(I) = 0$ or $|I| = maxdepth$ **then**
        return $(make\_leaf(I), leaf\_error(I))$
    $(\tau, b) \leftarrow (make\_leaf(I), leaf\_error(I))$
    for all attributes i **do**
        if $cover(I \cup \{i\}) \geq minsup$ and $cover(I \cup \{\neg i\}) \geq minsup$ **then**
            $(\tau_1, e_1) \leftarrow DL8 - Recurse(I \cup \{\neg i\})$
            if $e_1 \leq b$ **then**
                $(\tau_2, e_2) \leftarrow DL8 - Recurse(I \cup \{i\})$
                if $e_1 + e_2 \leq b$ **then**
                    $(\tau, b) \leftarrow (make\_tree(i, \tau_1, \tau_2), e_1 + e_2)$
    $cache.store(I, BestTree(\tau, b))$
    return (τ, b)

This approach might lead to exploring multiple times the same itemset, for example exploring $\neg b$ after a results in the same search as exploring a after $\neg b$, both giving the itemset $\{a, \neg b\}$. To avoid this, DL8 uses a cache of itemsets to reuse the best trees obtained previously for said itemset. These ideas lead to the DL8 algorithm, which is presented in Algorithm 1.

The actual DL8 algorithm as presented in [4] is able to enforce more constraints and is therefore quite more complicated. However, this work is not related to this thesis, and we will therefore not cover it.

2.2.4 DL8.5 learning

While the DL8 algorithm provides an optimal solution and does not explore the same itemset twice if a solution exists for this itemset, the search space can still be substantially reduced by implementing into DL8 a branch-and-bound component. This is what was done in the DL8.5 algorithm presented in [2].

The main idea in this amelioration is the use of upper bounds in the search. This upper bound ensures that the tree that is found has a total error that is strictly lower than this upper bound. When the search starts, that upper bound is set to $+\infty$ but every time a tree is found, its error is used as the upper bound for the next trees to be computed for this itemset and is communicated to children nodes in the search. The DL8.5 algorithm is presented in Algorithm 2.

Algorithm 2 uses the bounds to prune some branches. In line 18 we only explore the positive branch $I \cup \{i\}$ if the negative branch $I \cup \{\neg i\}$ did not already give an error greater than the upper bound.

Another main advantage is the use of the `NO_TREE`. Indeed, if a search over a certain itemset is unable to provide a tree with an error lower than the upper bound, `NO_TREE` is returned and saved in the cache. Meaning that if we encounter once more the same itemset with a lower or equal upper bound, we know that there is no need to conduct the search without having to execute further computations. This is expressed in line 10 of Algorithm 2.

A combination of the two previous points is also used in line 16 of Algorithm 2 as if the negative branch does not give a solution that satisfies the upper bound constraint there is no need to search the positive branch, and it is pruned.

Algorithm 2 DL8.5($maxdepth$, $minsup$)

struct $BestTree\{lb : float, tree : Tree, error : float\}$
$cache \leftarrow HashMap < Itemset, BestTree >$
$best\_solution \leftarrow DL8 - Recurse(\emptyset, +\infty)$
return $best\_solution.tree$
procedure $DL8.5 - Recurse(I,\ ub)$
    if $leaf\_error(I) = 0$ or $|I| = maxdepth$ or time-out is reached **then**
        return $BestTree(ub, make\_leaf(I), leaf\_error(I))$
    $solution \leftarrow cache.get(sort(I))$
    if $solution$ was found **then**
        if solution.tree $\neq$ NO_TREE or $ub \leq solution.lb$ **then**
            return $solution$
    $(\tau, b, left\_ub) \leftarrow (\text{NO\_TREE}, +\infty, ub)$
    for all attributes i in a well-chosen order **do**
        if $cover(I \cup \{i\}) \geq minsup$ and $cover(I \cup \{\neg i\}) \geq minsup$ **then**
            $sol_1 \leftarrow DL8.5 - Recurse(I \cup \{\neg i\}, ub)$
            if $sol_1.tree = \text{NO\_TREE}$ **then**
                continue
            if $sol_1.error < left\_ub$ **then**
                $sol_2 \leftarrow DL8.5 - Recurse(I \cup \{i\}, left\_ub - sol_1.error)$
                if $sol_2.tree = \text{NO\_TREE}$ **then**
                    continue
                $feature\_error \leftarrow sol_1.error + sol_2.error$
                if $feature\_error < left\_ub$ **then**
                    $\tau \leftarrow make\_tree(i, sol_1.tree, sol_2.tree)$
                    $b \leftarrow feature\_error$
                    $left\_ub \leftarrow b$
                if $feature\_error = solution.lb$ **then**
                    break
    $solution \leftarrow BestTree(ub, \tau, b)$
    $cache.store(I, solution)$
    return $solution$

DL8.5 also offers the ability to use heuristics for the order in which we search the different attributes. This is expressed in line 13 and is useful because of the upper bound pruning. Indeed, if an attribute f_1 is visited first and gives an error e_1 and an attribute f_2 is visited next with the error corresponding to its negative branch being $\tilde{e}_2$, and that $\tilde{e}_2 \geq e_1$ then the algorithm will prune the positive branch for f_2 while the other way around the positive branch could not have been pruned.

2.2.5 Better lower bounding

In line 10 of Algorithm 2, we see that the upper bound of a previous search is used as a lower bound for a subsequent search. [5] proposed a better lower bounding technique that was then implemented in the current version of the DL8.5 algorithm.

Indeed, [5] shows that the lower bound can actually be improved by distinguishing two cases:

1. The search finds a solution, in which case the lower bound is the error of the found tree, namely the sum of the errors of its children branches.

2. The search does not find a solution. In which case, the concept of minimum lower bound is introduced, and it is updated for each attribute explored for the current itemset. It is defined as follows,

$$min\_lb = \min_{f_i \in attributes} LB(I, f_i, ub) \tag{2.7}$$

$$LB(I, f_i, ub) = \begin{cases} lsol.lb \text{ **if** } lsol.tree = \texttt{NO\_TREE} \\ lsol.error + rsol.lb \text{ **if** } rsol.tree = \texttt{NO\_TREE} \end{cases} \tag{2.8}$$

   Where $lsol$ is the result of the recursive procedure on the itemset $I \cup \{\neg f_i\}$ with upper bound ub and $rsol$ is the result of the recursive procedure on the itemset $I \cup \{f_i\}$ with the upper bound $ub - rsol.error$.

Finally, the lower bound stored in cache is the maximum between the upper bound used for the search and the minimum lower bound over the features. The inclusion of this new lower bound is shown in Algorithm 3.

2.2.6 Further optimizations

Some further optimizations to the DL8.5 algorithm as presented in Algorithm 3 are possible. [5] showed three new ways to better the algorithm.

Algorithm 3 DL8.5($maxdepth$, $minsup$)

```
1: struct BestTree{lb : float, tree : Tree, error : float}
2: cache ← HashMap < Itemset, BestTree >
3: best_solution ← DL8 − Recurse(∅, +∞, 0)
4: return best_solution.tree
5: procedure DL8.5 − Recurse(I, ub)
6:     if leaf_error(I) = 0 or |I| = maxdepth or time-out is reached then
7:         return BestTree(ub, make_leaf(I), leaf_error(I))
8:     solution ← cache.get(sort(I))
9:     if solution was found then
10:        if solution.tree ≠ NO_TREE or ub ≤ solution.lb then
11:            return solution
12:    (τ, b, left_ub, min_lb) ← (NO_TREE, +∞, ub, +∞)
13:    for all attributes i in a well-chosen order do
14:        if cover(I ∪ {i}) ≥ minsup and cover(I ∪ {¬i}) ≥ minsup then
15:            sol_1 ← DL8.5 − Recurse(I ∪ {¬i}, ub)
16:            if sol_1.tree = NO_TREE then
17:                min_lb ← min(min_lb, sol_1.lb)
18:                continue
19:            if sol_1.error < left_ub then
20:                sol_2 ← DL8.5 − Recurse(I ∪ {i}, left_ub − sol_1.error)
21:                if sol_2.tree = NO_TREE then
22:                    min_lb ← min(min_lb, sol_1.error + sol_2.lb)
23:                    continue
24:                feature_error ← sol_1.error + sol_2.error
25:                min_lb ← min(min_lb, feature_error)
26:                if feature_error < left_ub then
27:                    τ ← make_tree(i, sol_1.tree, sol_2.tree)
28:                    b ← feature_error
29:                    left_ub ← b
30:                if feature_error = solution.lb then
31:                    break
32:    lb ← max(ub, min_lb)
33:    solution ← BestTree(lb, τ, b)
34:    cache.store(I, solution)
35:    return solution
```

Finer lower bounding In [5], they enforce a constraint on the maximum number of nodes a tree can have for a certain itemset. Because the error is monotonically decreasing with the maximum number of nodes increasing, it can be used to refine the lower bounding by taking the minimum not only over the features but also over all configurations of numbers of nodes in the children that respect the maximum node constraint. However, in the implementation we use in this thesis, this constraint is not enforced. We will therefore not use this lower bounding technique.

Depth two tree computer Because the search in DL8.5 is exhaustive, even though there is good pruning, many trees of depth two will be explored. [5] proposed a way to avoid some computation in the trees of depth two. For ease of notation, we limit this explanation to binary classification, but this is easily extended to multiclass classification. Indeed, they define $FQ^+(f_i)$ and $FQ^+(f_i, f_j)$ as the frequency counts of the positive instances for a single feature and for a pair of features and analogously for the negative instances. In the following notations, $\mathcal{D}$ represents the whole dataset and $\mathcal{D}^+$ represents the subset of $\mathcal{D}$ containing only the positive instances. From this, they can compute $FQ^+(\neg f_i)$, $FQ^+(f_i, \neg f_j)$, $FQ^+(\neg f_i, f_j)$ and $FQ^+(\neg f_i, \neg f_j)$. This of course is similar for the negative instances and counts. These definitions are as follows:

$$FQ^+(\neg f_i) = |\mathcal{D}^+| - FQ^+(f_i) \tag{2.9}$$
$$FQ^+(f_i, \neg f_j) = FQ^+(f_i) - FQ^+(f_i, f_j) \tag{2.10}$$
$$FQ^+(\neg f_i, f_j) = FQ^+(f_j) - FQ^+(f_i, f_j) \tag{2.11}$$
$$FQ^+(\neg f_i, \neg f_j) = |\mathcal{D}^+| - FQ^+(f_i) - FQ^+(f_j) + FQ^+(f_i, f_j) \tag{2.12}$$

Using this, the classification score for a classification node with all instances of $\mathcal{D}$ containing both features f_i and f_j, defined as $CS(f_i, f_j)$, is computed by

$$CS(f_i, f_j) = \min\{FQ^+(f_i, f_j), FQ^-(f_i, f_j)\} \tag{2.13}$$

Let us remember we don't want to compute the classification score for a single node but the misclassification score for a decision tree of depth two with feature f_{root} at its root and features f_{left} and f_{right} as its left and right children. The misclassification score, for one of the two branches, $MS(f_{root}, f_{child})$ can be computed in constant time given the frequency counts.

$$MS_{left}(f_{root}, f_{left}) = CS(\neg f_{root}, \neg f_{left}) + CS(\neg f_{root}, f_{left}) \tag{2.14}$$
$$MS_{right}(f_{root}, f_{right}) = CS(f_{root}, \neg f_{right}) + CS(f_{root}, f_{right}) \tag{2.15}$$

The final misclassification score of the depth-two tree is the sum of those two values. They showed in [5] that this optimization provides a significant speedup. While this is easily extended to multiclass classification, it is however much more complicated to do so for regression and not at all applicable to quantile regression, which is the main topic of this thesis. This optimization will therefore not be shown in the algorithms.

Similarity based lower bounding Finally, [5] also proposes a way to use the optimal tree found for a previous dataset. If we already have computed the optimal tree for a certain part of the dataset $\mathcal{D}_{old}$, we can define the similarity-based lower bound for another dataset $\mathcal{D}_{new}$ as

$$LB(\mathcal{D}_{new}, \mathcal{D}_{old}, d) = T(\mathcal{D}_{old}, d) - |\mathcal{D}_{out}| \tag{2.16}$$

Here $\mathcal{D}_{out} = \mathcal{D}_{old} \setminus \mathcal{D}_{new}$ and $T(\mathcal{D}, d)$ is the error of the best tree of maximum depth d on dataset $\mathcal{D}$.

Unfortunately, [5] shows this similarity based lower bounding technique cannot be used or adapted in the context of regression, which is why it won't be shown in the algorithms.

2.3 Existing methods

While the concept of using decision trees, and more precisely optimal decision trees, to predict the quantiles of a distribution and then extracting from it a probability density function is a relatively unexplored field, some work does merit a mention as a comparison point. In this section, we will introduce an overview of these works and briefly mention their methods and results.

2.3.1 AdaBoost to determine class distributions

As a way to estimate conditional probability distributions in the context of binary classification, [6] developed a method based on the AdaBoost classifier to estimate conditional probabilities. They base the use of AdaBoost on the fact that it is a very efficient way of classification and is robust to overfitting.

Their developed method consists of the following steps. They start by setting a quantization level $\delta > 2$ from which they create δ quantiles: $q = 1/\delta, 2/\delta, \ldots, 1 - 1/\delta$

1. Generate δ datasets from the initial dataset, so the q-quantile becomes the median. This is done by randomly sampling from the original dataset, with

replacement, but by over/under-sampling the positive and negative classes so the median of the generated dataset becomes the qth-quantile of the original dataset. This means that in the dataset generation, a negative sample has a probability q to be drawn, and positive samples have a probability $1 - q$ to be drawn.

2. Fit one AdaBoost classifier per dataset and hence quantile.

3. The classifiers then give an estimate for a sample x of $I\{p(x) \geq q\}$ denoted as $\hat{D}_q(x) \in \{0, 1\}$. Here $I\{c\}$ takes values 1 if condition c is true and 0 otherwise.

4. Finally, they estimate the probability $\hat{p}(x)$ of the sample x to belong to the positive class by the following procedure:

   - If $\hat{D}_{0.5}(x) = 1$, then $\hat{p}(x) = \min\{q > 0.5 | \hat{D}_q(x) = 0\} - \frac{1}{2\delta}$. If no such q is found, they take $\hat{p}(x) = 1 - \frac{1}{2\delta}$
   - If $\hat{D}_{0.5}(x) = 0$, then $\hat{p}(x) = \max\{q < 0.5 | \hat{D}_q(x) = 1\} + \frac{1}{2\delta}$. If no such q is found, they take $\hat{p}(x) = \frac{1}{2\delta}$

Multiple limitations arise with this model in the context of the research questions of this thesis. Firstly, it is limited to classification and the goal of this thesis is to use density estimation in a regression context.

Secondly, the authors of [6] do stress that even though over/under-sampling the datasets to change the median value does improve the class probability estimates, it does effectively reduce the dataset size as we under-sample one class and increase the size of another by drawing with replacement.

2.3.2 Quantile regression forests

Regression models usually predict the expectation of a certain variable, however this is usually not very informative of the whole distribution. To palliate this, [7] proposed to create quantile regression forests.

They based themselves on [8] for their growing on the trees, meaning they learn k trees using the CART algorithm as described in [3]. Once that is done, they can compute for any realization of the explanatory variables $\boldsymbol{X} = \boldsymbol{x}$, i.e., the features of the samples, the weight $w_i(\boldsymbol{x}, \theta)$ associated to each sample for every tree.

$$w_i(\boldsymbol{x}, \theta) = \frac{I\{\boldsymbol{X}_i \in R_\ell(\boldsymbol{x}, \theta)\}}{|\{\boldsymbol{X}_j \in R_\ell(\boldsymbol{x}, \theta)\}|} \tag{2.17}$$

In this expression $R_\ell(\boldsymbol{x}, \theta)$ is the set of realizations mapped to the leaf ℓ under the tree θ. $I\{c\}$ takes value 1 if condition c is true and value 0 otherwise.

Using the trees and the weights, they derive an estimation of the conditional cumulative distribution function as follows. Here $w_i(\boldsymbol{x})$ corresponds to the average over all the trees of $w_i(\boldsymbol{x}, \theta_k)$

$$\hat{F}(y|\boldsymbol{X} = \boldsymbol{x}) = \sum_{i=1}^{N} w_i(\boldsymbol{x}) I\{y \geq y_i\} \tag{2.18}$$

Finally, using this estimation of the cumulative distribution function, they can estimate quantiles using the following expression.

$$Q_q(\boldsymbol{x}) = \inf\{y : \hat{F}(y|\boldsymbol{X} = \boldsymbol{x}) \geq q\} \tag{2.19}$$

2.3.3 Combined quantile regression

Going further than just predicting quantiles, [9] proposed a way to predict distributions based on the combined predictions of multiple models and then fitting a quantile regressor on those predictions.

The method used is the following. They created three models to predict the target value using standard MSE as their loss. The models used are a random forest regressor, a gradient boosting decision tree model and finally a support vector regressor.

Then they fitted a quantile regressor as described in [10] on the predictions of those models. By defining $\hat{\boldsymbol{y}}_i = (\hat{y}_i^{(1)}, \hat{y}_i^{(2)}, \hat{y}_i^{(3)})$ as the explanatory variables for the sample i, where $\hat{y}_i^{(1)}$, $\hat{y}_i^{(2)}$ and $\hat{y}_i^{(3)}$ are respectively the predicted values of the three models for this sample. Also defining y_i, the true value for this sample, the method aims to find for a certain quantile q the weights vector $\boldsymbol{\beta}(q)$ that minimizes the following loss.

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{|\boldsymbol{y}|} \max\{(\boldsymbol{\beta}^\top \hat{\boldsymbol{y}}_i - y_i)q, (\boldsymbol{\beta}^\top \hat{\boldsymbol{y}}_i - y_i)(q-1)\} \tag{2.20}$$

After obtaining predicted quantile values for Q quantiles, they use a kernel density estimator (KDE) to generate a probability density function. This is formulated as follows. If the goal is to estimate the probability distribution for a certain sample $\boldsymbol{x}_i$

that outputs the quantile values $\hat{y}_{q_j,i}$ in the quantile regressor, the probability estimation is given by

$$\hat{f}(y) = \frac{1}{Q\omega} \sum_{j=1}^{Q} K\left(\frac{\hat{y}_{q_j,i} - y}{\omega}\right). \tag{2.21}$$

Here ω is the width of the window and $K(\cdot)$ is the kernel function of the KDE. In [9], only Gaussian kernels were used.

The authors of [9] showed their method gave good results for their use case, which was to predict the short term probability density of the electrical load in the US.

While the model gives good results, they are difficult to interpret by a human as it is a combination of three relatively complicated models.

2.3.4 Combination of quantile regression models

The method explored by [9] has proven good results by minimizing a loss function, namely (2.20), that enforces the model to predict values for the quantiles rather than the median or mean value. However, the model obtained is a linear one based on predictions of the mean value by other non-linear models. The authors of [11] propose a way to directly predict quantiles in the initial models.

Indeed, in [11] they explore different widely used predictive models that have been made to optimize the quantile loss function rather than MAE or MSE. This function is shown in equation (2.22).

$$QL_q(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{|\boldsymbol{y}|} \sum_{i=1}^{|\boldsymbol{y}|} \max\{q(\hat{y}_i - y_i), (q-1)(\hat{y}_i - y_i)\} \tag{2.22}$$

The models used are the following. They were all trained on the quantile loss of equation (2.22).

- Linear Regression (LR)
- Gated Recurrent Unit (GRU), a variant of Recurrent Neural Networks, as presented in [12].
- Random Forest (RF), a bagging ensemble method based on CART regression trees.

- Gradient Boosting Regression Tree (GBRT), another ensemble method where each model minimizes the residue of the previous training.
- Light Gradient Boosting Machine (LGBM), a distributed version of GBRT that uses the histogram method to reduce computational cost.

Once these models were trained on multiple quantile values, like in the previous method considered, they applied a KDE on the obtained values for the samples to generate a distribution. The expression of the density estimate is given by equation (2.21).

As an amelioration to this, they also tried to combine the different models linearly to see if they obtained a better prediction. To this effect, they developed an algorithm to find weights that minimize the CRPS loss given in equation (2.23).

$$CRPS(\hat{F}, y) = \int_{-\infty}^{+\infty} (\hat{F}(\lambda) - I(\lambda - y))^2 d\lambda \tag{2.23}$$

Where $\hat{F}$ is the cumulative density function corresponding to a density estimate provided by one of the models, $\hat{F}(y) = \int_{-\infty}^{y} \hat{f}(\lambda) d\lambda$, y is the actual value of a sample and $I(\lambda - y)$ is the Heaviside step function which takes value 1 if $\lambda \geq y$ otherwise takes value 0.

The CRPS measures the integral of the square of the difference between the estimated distribution and the true distribution, which is a step function, as the observed sample has a deterministic y value. It is a metric that evaluates the reliability and sharpness of the density forecasting model.

The authors of [11] show that the combination of the models using their custom algorithms to weigh the different density estimates gives excellent results in predicting electricity load in the US, which was their use-case.

Again, even though the method gives good results, the model is of high complexity and completely uninterpretable by a human as we have many predictive components, some of which are not even tree-based (GRU).

2.3.5 Parametric decision trees for conditional density estimation

The approaches described above go in increasing size of complexity at the cost of interpretability. In [13], they proposed CaDET, a novel way of learning decision trees to obtain *parametric* conditional density estimations (CDE).

Their proposed method of estimating conditional densities is to generate trees in which the leafs correspond to a member of a certain distribution family. The goal is then to find the parameters of the distribution that fit best the examples in the leaf.

To limit memory consumption of their algorithm, they use the concept of sufficient statistics. If $\mathcal{F}$ is a parametric family of pdfs over $\mathcal{Y}$, the target variable space, with parametric space Θ, and by taking $\theta \in \Theta$, the member of $\mathcal{F}$ is identified by θ, denoted as $\rho(\cdot; \mathcal{F}, \theta)$. A sufficient statistic for Θ is a vector-valued function $\boldsymbol{w}(\boldsymbol{Y})$ such that $\boldsymbol{w}$ is as informative as the sample target values $\boldsymbol{Y}$ in the purpose of estimating θ. For the Gaussian family, a sufficient statistic would be:

$$\boldsymbol{w}(\boldsymbol{Y}) = \left(\sum_{y \in \boldsymbol{Y}} y, \sum_{y \in \boldsymbol{Y}} y^2 \right) \tag{2.24}$$

This adaptation allows the CaDET trees to directly produce CDEs in the leafs.

The authors of [13] also bring forth an important amelioration compared to standard decision trees such as CART or C4.5 in the sense that when choosing splits, they don't use the MSE impurity in regression trees, defined as

$$m_{mse}(\boldsymbol{Y}) = \frac{1}{|\boldsymbol{Y}|} \sum_{y \in \boldsymbol{Y}} \left(y - \overline{\boldsymbol{Y}} \right)^2, \text{ where } \overline{\boldsymbol{Y}} = \frac{1}{|\boldsymbol{Y}|} \sum_{y \in \boldsymbol{Y}} y. \tag{2.25}$$

Neither do they use the Entropy impurity in information-gain classification trees, defined as

$$m_H(\boldsymbol{Y}) = -\sum_{i \in \mathcal{Y}} \hat{\mathbb{P}}(i) log \left(\hat{\mathbb{P}}(i) \right), \text{ where } \hat{\mathbb{P}(i)} = \frac{\sum_{y \in \boldsymbol{Y}} \delta(y - i)}{|\boldsymbol{Y}|}. \tag{2.26}$$

In this equation $\delta(y - i)$ is the Kronecker delta which takes value 1 if $y = i$ and 0 otherwise.

Indeed, these impurities are used to optimize the conditional expectation of a random variable Y, i.e., they favor splits that will give a better prediction of Y, not a better estimation of its conditional distribution.

As a solution, [13] proposed their own impurity criterion, which is dependent on the parametric family of pdfs $\mathcal{F}$ that is used. Indeed, CaDET minimizes the Empirical Cross Entropy impurity, defined in (2.27).

$$m_{ece}(\boldsymbol{Y};\mathcal{F}) = -\frac{1}{|\boldsymbol{Y}|}\sum_{y\in\boldsymbol{Y}} log\left(\rho(y;\mathcal{F},\theta(\boldsymbol{w}(\boldsymbol{Y})))\right) \tag{2.27}$$

Because this loss depends on $\mathcal{F}$, the splits will lead to the data being well-fit by $\mathcal{F}$. And because the impurity is dependent on the estimated CDE, this impurity is much more appropriated than MSE or entropy impurities for this context.

[13] also explores the possibility of using random forests composed of CaDET trees to provide a combined prediction of the CDEs. In this approach, they actually combine the sufficient statistics of each of the individual trees in a weighted sum to compute the parameters of the final predicted distribution. This only works if all the corresponding leaves in the different trees map to the same distribution family.

Using these concepts, the authors of [13] show that the use of CaDET produces interpretable and accurate predictions of CDEs and with low overfitting risks.

2.4 Limitations

The methods presented above explore different ways to estimate conditional probability densities. However, they suffer from limitations.

The method presented in [6] is limited to binary classification, does not make full use of the available dataset, and has limited theoretical foundation for its predictions.

The method presented in [7] does not exploit the fact that quantiles will be predicted in the end model when growing the trees. Because the algorithm used to grow the trees is a greedy heuristic designed for MSE regression, it may not be very efficient for quantile estimation. However, no quantiles need to be set *a priori*, meaning the method offers a full estimation of the quantile function.

The methods presented in [9] and [11] offer good results and are usable in a regression context, however they do not offer a good interpretability of their predictions.

Finally, the method proposed in [13] addresses the points raised above with strong mathematical foundations and high interpretability. However, its estimations are parametric, forcing the user to assume the distribution family of the predictions. Moreover, the growing of the trees is based on heuristics, thus maybe not finding the best and/or shallowest trees.

In the following sections we will propose a novel way of estimating distributions that is easy to interpret for a human, with theoretical guarantees and that is non-parametric with respect to the distribution predicted.

CHAPTER 3

Quantile regression

In this chapter we will introduce how the current DL8.5 algorithm can be turned into a regression model, how we can create empirical quantile estimations and how to find the tree that minimizes quantile loss in DL8.5. With each addition to the algorithm, we briefly mention how it was implemented in the current version. When we refer to the current implementation, we mean the one available at [14].

3.1 Regression in DL8.5

As a first step into predicting optimal quantile estimations using decision trees, we must first transform the DL8.5 implementation into one that can predict real values rather than class labels.

Let us remind ourselves that DL8.5 is an exact algorithm that finds the optimal decision tree under given constraints and is able to optimize any additive loss function with no need for any heuristic.

3.1.1 Transforming the algorithm to enable regression

The algorithm as presented in Algorithm 3 needs no adaptation to optimize a regression error as long as it is additive. Indeed, DL8.5 bases itself on the fact that the $leaf\_error$ function is additive because it considers the error of the root of the tree to be the sum of the errors of its children.

This was of course the case for the misclassification error, shown in equation (2.6). This is not the case for MSE, however, as for many other losses that compute a mean over samples. Fortunately, as minimizing MSE gives the same result as optimizing the sum of squared errors (SSE) we can use this adapted loss to still be able to optimize MSE. This is true for most errors computing a mean over samples.

3.1.2 Integration of MSE into DL8.5 implementation

The python DL8.5 package already allowed to optimize arbitrary losses if they were additive. However, this was done by creating a callback to a python function from the C++ implementation. While convenient for the user to only provide a python function, this method is extremely slow.

As a first addition to the implementation of DL8.5, we added in the C++ implementation two new parameters, a vector of `double` values representing the target values of the samples and a code representing the function to optimize if no python error is provided. This can be either `misclassification` or `mse`. Checks were then implemented when the leaf error is computed to call the right function with the right error code.

The vector of target values had to be added, as in the existing implementation the target values (class labels) were stored as class supports in the form of bitsets. While this is very efficient to compute misclassification errors, it is not extendable to regression and target values in the real domain.

3.2 Quantile estimation

As presented in Section 2.1, obtaining the quantile function of a distribution gives all the information about said distribution. Indeed, it is the reciprocal of the cumulative density function, the derivative of which is the probability density function. This shows that if we can perfectly estimate the quantile function, then we can perfectly predict probability densities for the studied random variable.

Deriving the quantile function, as is, is a complicated task. What may be done more easily however is estimating a finite number of quantile values.

3.2.1 Empirical quantile estimation

Given a dataset $\boldsymbol{Y} \in \mathcal{Y}^N$ of i.i.d. realizations of the random variable Y, to compute the empirical quantile value y_q for a certain quantile, we use an estimated cumulative

distribution function $\hat{F}$ and apply equation (2.3).

The easiest way to compute an estimated cdf based on the empirical values is by using $I(y - y')$, the Heaviside step function:

$$\hat{F}(y) = \frac{1}{|\boldsymbol{Y}|} \sum_{y' \in \boldsymbol{Y}} I(y - y'). \tag{3.1}$$

Though, often, a piece-wise linear interpolation of the samples in $\boldsymbol{Y}$ is used. For this, we assume that the samples in $\boldsymbol{Y}$ are in increasing order, such that $y_1 \leq y_2 \leq \cdots \leq y_N$, and we define the component function $\phi_i(\lambda)$ for $2 \leq i \leq N$

$$\phi_i(\lambda) = \begin{cases} 0 \textbf{ if } \lambda < 0 \\ \lambda \textbf{ if } \lambda \in [0, 1] \\ 1 \textbf{ otherwise} \end{cases} \tag{3.2}$$

Using this formulation, we can define the piece-wise linear empirical cdf of the sample $\boldsymbol{Y}$. This is shown in equation (3.3).

$$\hat{F}(y) = \frac{1}{|\boldsymbol{Y}| - 1} \sum_{i=2}^{|\boldsymbol{Y}|} \phi_i \left(\frac{y - y_{i-1}}{y_i - y_{i-1}} \right) \tag{3.3}$$

Once we have the empirical cdf estimation, we can use equation (2.3) to find the estimated quantile value $\hat{y}_q \in \mathcal{Y}$ for a certain quantile $q \in [0, 1]$.

3.3 Quantile loss

The above method gives a way to, given a certain sample set, generate estimates for the quantile values. This gives information about the pdf $f(y)$ from which the sample was drawn. However, we are not interested in finding the distribution of the whole dataset. Indeed, we are interested in the *conditional* probability density of random variable Y, knowing the values of explanatory random variables $\boldsymbol{X}$ whose realizations will be denoted $\boldsymbol{x} \in \mathcal{X}$. We will denote the conditional probability density function of Y knowing $\boldsymbol{X}$ as $f(y|\boldsymbol{x})$.

3.3.1 Quantile loss formulation

In order to predict conditional pdfs, we intend to use a decision tree learned by the DL8.5 algorithm. For this, the most straightforward way of doing would be to

use the MSE loss already implemented and to assign quantile values to the leafs as their predictions rather than the center of mass of the samples classified in said leaf. However, as discussed in Section 2.3.5, [13] showed that MSE is inefficient in enforcing good probability distribution predictions and will favor splits that give a prediction close to the expectation but do not account for over/under predictions weights.

To this effect intervenes the quantile loss. Its formulation for one realization is stated below as

$$QL_q(y, y_p) = \max\left\{q(y_p - y), (q-1)(y_p - y)\right\}. \tag{3.4}$$

In this equation, y_p is the predicted value for a realization of the random variable Y, y is the true value for that realization and q is the quantile we are considering. We can interpret this equation as a loss penalizing over-predictions with weight q and under-predictions with weight $1 - q$. Indeed, in the first case, $y_p - y$ is positive and $QL_q(y, y_p)$ takes the value $q(y_p - y)$. In the second case, $y_p - y$ is negative and $QL_q(y, y_p)$ takes the value $(1 - q)(y - y_p)$.

We usually want to evaluate a loss over a set of samples, and not just one realization. For this, we use the mean over the samples. This is formulated as

$$QL_q(\boldsymbol{Y}, \boldsymbol{Y_p}) = \frac{1}{|\boldsymbol{Y}|}\sum_{i=1}^{|\boldsymbol{Y}|}\max\left\{q(y_{p,i} - y_i), (q-1)(y_{p,i} - y_i)\right\}. \tag{3.5}$$

The interesting property about the function in (3.4) is that it is minimized for the true quantile value of the distribution from which the sample was drawn, as shown in [7].

3.3.2 Quantile prediction to minimize the loss

Because we will use this loss in the context of a decision tree, we will map a set of samples to a certain leaf which will need to take a certain value, i.e., a value that will minimize the mean of the quantile losses for each of the samples mapped to that leaf if they all get the same predicted value. In the following developments, we will assume that the samples of the leaf are sorted: $y_1 \leq y_2 \leq \cdots \leq y_N$

At this point, the literature offers many options to compute quantile values, the most common being explained in [15]. Indeed, empirically any value between $y_{\lfloor h \rfloor}$ and $y_{\lceil h \rceil}$ where $h = q(N-1) + 1$ would give the same value of percentage of samples being under-predicted and that percentage would be the closest to q as can be attained.

However, all predictions between those values do not hold the same properties. We will focus ourselves on two directions. The first one being the optimal way to compute the prediction with respect to the training data, the second being the one that numpy uses as default. These are detailed below.

Optimal way The quantile loss in equation (2.4) can be viewed as a weighted sum that penalizes over and under predictions with the weights q and $1-q$ respectively. Since in a leaf we are only allowed to predict one value for the whole sample set mapped to that leaf, we can distinguish three cases.

1. First case: $q < 0.5$. In this case, the weight $1-q$ is heavier than the weight q, so under-predictions are penalized more than over-predictions. This means that the optimal estimated quantile value is $y_{\lceil h \rceil}$
2. Second case: $q = 0.5$. Here, both under and over-predictions share the same penalization and the optimal estimated quantile value is $\frac{y_{\lfloor h \rfloor}+y_{\lceil h \rceil}}{2}$.
3. Third case: $q > 0.5$. Here, the weight q is heavier than the weight $1-q$, so over-predictions are penalized more than under-predictions. This means that the optimal estimated quantile value is $y_{\lfloor h \rfloor}$

NumPy way Because the goal is to use this algorithm as a python package in a data science context, it is interesting to be compatible with what numpy uses as its default in the `np.quantile` function [16]. The value of the numpy quantile estimation is given by

$$\hat{y}_q = y_{\lfloor h \rfloor} + (h - \lfloor h \rfloor)(y_{\lceil h \rceil} - y_{\lfloor h \rfloor}). \tag{3.6}$$

It is interesting to notice that this way of computing the quantile corresponds to finding the exact quantile value if the distribution is represented using the approximation of the cdf function in equation (3.3).

3.4 Implementation into DL8.5

In this section, we will briefly explain how the quantile error can be computed efficiently and what adjustments had to be made in the current implementation of DL8.5 to enable it to optimize it.

3.4.1 Efficient way of computation

Traversing the data is the most computationally intensive part of DL8.5 learning and it is therefore primordial to use an efficient computation of the quantile values in a production implementation.

A naive way of implementation would be to find the cover associated to a leaf, we will denote the cover's size (support) as N. With $h = q(N-1)+1$, to then find the values associated to the $\lfloor h \rfloor$th and $\lceil h \rceil$th lowest values of $\boldsymbol{Y}$ in the cover, which takes $\mathcal{O}(NlogN)$ time as shown in [17]. Then we would need to traverse the whole data again to compute the quantile loss to each sample given the leaf prediction and compute the mean, adding $\mathcal{O}(N)$ operations, ending with an algorithm in $\mathcal{O}(NlogN)$ time.

If we now consider the array of samples $\boldsymbol{Y}$ to already be sorted, such that $y_1 \leq y_2 \leq \cdots \leq y_N$, we can compute the predicted value in $\mathcal{O}(1)$ time and only traverse the data once to compute the full loss of the leaf, making it $\mathcal{O}(N)$ time.

3.4.2 Integration

To fully integrate the quantile loss in the DL8.5 implementation, a few more changes needed to be made. First, the quantile loss as defined in equation (3.5) is not additive. Fortunately, multiplying by a factor N makes it additive and gives the same result on the final tree.

An additional code was added in the parameters of the C++ implementation to specify which way we want to use to predict quantile values, this code, `quantile_mode`, can take a value corresponding to either the optimal or linear (numpy default) way.

Finally, to be able to use the faster way of computing of the previous section, the algorithm needs to be able to go through the y values in increasing order. For this, the target values array $\boldsymbol{Y}$ was sorted once at the start of the algorithm and the explanatory variables array $\boldsymbol{X}$ was sorted accordingly. Using this and forcing the algorithm to go through the items in increasing order of index, skipping the elements not in cover, we ensure to go through samples in increasing order of y value, making the error computation correct and efficient.

CHAPTER 4

Optimizations and smoothing

In this chapter, we will bring forth the main algorithmic addition to DL8.5 from this thesis. It enables it to optimize for arbitrarily many quantiles at once by sharing information across the search for each of the trees to find an optimal decision tree for each one.

This chapter is structured in the following sections. We will start by an explanation of the challenges of optimizing many quantiles at once and what assumptions can be made to improve efficiency. Following this will be an explanation of the modifications to the main algorithm to enable the optimization. Then, we will explain how to compute quantile estimations for many quantiles over a same set in $\mathcal{O}(N)$ time, followed by a brief explanation of the implementation and integration in the current version of DL8.5.

Finally, we will present how the outputs of the Quantile DL8.5 algorithm can be used to generate smooth pdf estimations.

4.1 Optimizing for many quantiles

In chapter 3, we explored how we can adapt the DL8.5 algorithm to enable it to predict quantile values for the leafs of the decision tree and to optimally minimize the quantile loss.

However, let us remind ourselves that the end goal is to get information about the full probability density function by finding an approximation of the quantile function. Of

course, a single quantile is not very informative about the shape and properties of the whole function. To get more information, we would need to estimate many quantiles at once. This raises multiple challenges, explained below.

4.1.1 Parametric loss function

Because the loss function shown in equation (3.5) is parametric depending on the value q, we have a different loss function for each quantile we wish to find. Therefore, we cannot just learn one tree and use it to compute the different quantile value estimations, as there would be no guarantee that the tree is optimal for the other quantiles.

Because we have many different loss functions, the DL8.5 algorithm, as it is currently, is not able to find a tree that is optimal for all losses.

4.1.2 Quantile dependency on explanatory variables assumption

One way to solve the problem posed in Section 4.1.1 would be to fit a new tree for each quantile for which we wish to generate predictions.

While this solution would work and would give optimal predictions for each quantile, it is a very inefficient method. Indeed, if we assume that the distributions are dependent on the explanatory variables, then quantile predictions are as well. Meaning that we may expect the trees resulting from the search to be very similar. Especially for quantiles close to each other.

However, this assumption is not at all exploited in the method explained above, as we perform independent searches for the different quantiles.

4.2 Quantile DL8.5

In this section, we will introduce a new algorithm whose goal is to find $Q = |\boldsymbol{q}|$ trees, each optimizing quantile loss for a different quantile. This algorithm will perform the search for all these trees at the same time, thus sharing its computations across the trees, while preserving optimum on the individual trees obtained.

This new algorithm transforms the classical DL8.5 algorithm from a decision tree learner to an ensemble method learning many trees.

To that effect, we will present the modified version of DL8.5 that allows it to find one optimal decision tree per quantile while only traversing the search tree once. This is shown in Algorithm 4.

Algorithm 4 shows some differences with Algorithm 3, but the main concept remains the same: explore the search space as long as optimum is not reached. In this case, optimum is only reached when it is reached for each of the individual trees we are learning.

As can be seen in lines 16, 22, 26 and 34, some operations of Algorithm 3 have to be performed once for each quantile.

In lines 19 and 25, we see that we can only stop searching for this feature if *all* searches failed. If even only one search returned a result, we must continue to explore this branch.

In the same vein, line 20 shows that if at least one of the negative branches for one of the quantiles for the feature considered returned a feasible result, then the positive branch has to be explored as well.

When we have performed the full search for one feature, we only allow to break the loop if said feature provides the optimal solution for all trees. This is shown in line 33.

Finally, the main change occurs in line 10, which calls the sub-procedure defined in Algorithm 5. In this procedure, the can_return_flag stays set to true only if, for all quantiles, the cached solution either contains the optimal tree for this itemset, or the problem is infeasible because the upper-bound is lower than the lower bound or the current itemset is optimal in this branch for this quantile. In any other case, the search must continue.

In line 5 of Algorithm 5, we see that if the current itemset is optimal, we save the corresponding error as the final tree error for this branch.

While the pruning in this version of the algorithm seems a lot less restrictive than in Algorithm 3, we may remind ourselves that here we are optimizing for many optimization functions at once, that if we have only 1 quantile the algorithms are equivalent and finally that because we are optimizing conditional predictions, we may expect the resulting trees to be very similar, leading to exploring a similar search space as when performing the search for only one of those quantiles.

Algorithm 4 q-DL8.5($maxdepth$, $minsup$)

struct $BestTree\{\boldsymbol{lbs} : vector < float >, \boldsymbol{trees} : vector < Tree >, \boldsymbol{errors} : vector < float >\}$
$cache \leftarrow HashMap < Itemset, BestTree >$
$best\_solution \leftarrow q - DL8 - Recurse(\emptyset, +\infty^{|\boldsymbol{q}|})$
return $best\_solution.tree$
procedure $q - DL8.5 - Recurse(I, \boldsymbol{ubs})$
    $solution \leftarrow cache.insertOrGet(sort(I))$
    $\boldsymbol{leaf\_errors} \leftarrow quantile\_errors(I, \boldsymbol{q})$
    if $|I| = maxdepth$ or time-out is reached **then**
        return $BestTree(solution.\boldsymbol{lbs}, make\_leafs(I), \boldsymbol{leaf\_errors})$
    if $can\_return(solution, \boldsymbol{ubs}, \boldsymbol{leaf\_errors})$ **then**
        return $solution$
    $\boldsymbol{min\_lbs} \leftarrow +\infty^{|\boldsymbol{q}|}$
    for all attributes F in a well-chosen order **do**
        if $cover(I \cup \{f\}) \geq minsup$ and $cover(I \cup \{\neg f\}) \geq minsup$ **then**
            $sol_1 \leftarrow q - DL8.5 - Recurse(I \cup \{\neg i\}, \boldsymbol{ubs})$
            for $i \in \{1, 2, \ldots, |\boldsymbol{q}|\}$ **do**
                if $sol_1.trees_i =$ NO_TREE **then**
                    $min\_lbs_i \leftarrow min(min\_lbs_i, sol_1.lbs_i)$
            if all $sol_1.\boldsymbol{trees}$ are NO_TREE **then continue**
            if $\exists i \in \{1, 2, \ldots, |\boldsymbol{q}|\} : sol_1.errors_i < solution.errors_i$ **then**
                $sol_2 \leftarrow q - DL8.5 - Recurse(I \cup \{i\}, \boldsymbol{ubs} - \boldsymbol{sol_1.errors})$
                for $i \in \{1, 2, \ldots, |\boldsymbol{q}|\}$ **do**
                    if $sol_2.trees_i =$ NO_TREE **then**
                        $min\_lbs_i \leftarrow min(min\_lbs_i, sol_1.errors_i + sol_2.lbs_i)$
                if all $sol_2.\boldsymbol{trees}$ are NO_TREE **then continue**
                for $i \in \{1, 2, \ldots, |\boldsymbol{q}|\}$ **do**
                    $feature\_errors_i \leftarrow sol_1.errors_i + sol_2.errors_i$
                    $min\_lbs_i \leftarrow min(min\_lbs_i, feature\_errors_i)$
                    if $feature\_errors_i < solution.errors_i$ **then**
                        $solution.trees_i \leftarrow build\_tree(F, sol_1.trees_i, sol_2.trees_i)$
                        $solution.errors_i \leftarrow feature\_errors_i$
                        $ubs_i \leftarrow feature\_errors_i$
                if $\boldsymbol{feature\_errors} = solution.\boldsymbol{lbs}$ **then break**
    for $i \in \{1, 2, \ldots, |\boldsymbol{q}|\}$ **do**
        $solution.lbs_i \leftarrow max(ubs_i, min\_lbs_i)$
    return $solution$

Algorithm 5 $can\_return(solution, \boldsymbol{ubs}, \boldsymbol{leaf\_errors})$

1: $can\_return\_flag \leftarrow true$
2: **for** $i \in \{1, 2, \ldots, |\boldsymbol{q}|\}$ **do**
3: **if** $\neg(solution.trees_i \neq$ NO_TREE or $ubs_i \leq solution.lbs_i$ or $leaf\_errors_i = solution.lbs_i)$ **then**
4: $can\_return\_flag \leftarrow false$
5: **if** $leaf\_errors_i = solution.lbs_i$ **then**
6: $solution.errors_i \leftarrow leaf\_errors_i$
7: **return** can_return_flag

4.3 Linear-time algorithm to find many quantile values

Algorithm 4 makes use of the function, $quantile\_errors(I, \boldsymbol{q})$ which computes a vector of errors corresponding to the quantile errors for each of the quantiles in $\boldsymbol{q}$ for the samples that contain itemset I in their explanatory variables.

In Section 3.4.1, we showed how to compute the quantile error efficiently for one quantile. However, looping over the quantiles to use this function would give a time complexity of $\mathcal{O}(|\boldsymbol{q}|N)$. In this section, we propose an $\mathcal{O}(\max\{|\boldsymbol{q}|, N\})$ algorithm to compute the quantile error for arbitrarily many quantiles. This algorithm is shown in Algorithm 6. The time complexities are under the assumption that the data can be traversed in increasing order without addition of complexity, as explained in Section 3.4.1. For the remainder of this section, we will assume $y_1 \leq y_2 \leq \cdots \leq y_N$.

The idea behind this algorithm is simple. We iterate through the elements of $\boldsymbol{Y}$ in increasing order. While we encounter elements that are below the closest quantile value greater than the current value, we add the y_i values to the s^{under} corresponding counter.

Similarly, we add to the s^{above} counter corresponding to the quantile value closest to but lower than the current y_i value. If the current index corresponds to the h^{low} value of a quantile, we store the current value in the corresponding y^{low} element. Similarly, if the current index corresponds to the h^{up} value of a quantile, then we can compute the corresponding prediction (we will already have encountered h_{low} because we go through elements in increasing order).

Algorithm 6 Quantile Error Computer

for $j \in \{1, 2, \ldots, |\boldsymbol{q}|\}$ **do**
  $h_j \leftarrow (N-1)q_j + 1$
  $(h_j^{up}, h_j^{low}) \leftarrow (\lceil h_j \rceil, \lfloor h_j \rfloor)$
  $(s_j^{under}, s_j^{above}) \leftarrow (0, 0)$
$(k_{low}, k_{up}) \leftarrow (1, 1)$
$(m_{low}, m_{up}) \leftarrow (1, 0)$
for $i \in \{1, 2, \ldots, N\}$ **do**
  if $k_{up} \geq 1$ and $h_{k_{up}}^{up} \neq h_{k_{up}}^{low}$ **then**
    $s_{k_{up}}^{above} \leftarrow s_{k_{up}}^{above} + y_i$
  if $k_{low} \leq |\boldsymbol{q}|$ **then**
    $s_{k_{low}}^{under} \leftarrow s_{k_{low}}^{under} + y_i$
    if $i = h_{low}^{k_{low}}$ **then**
      $k_{low} \leftarrow k_{low} + 1$
      $k_{up} \leftarrow k_{up} + 1$
      if $h_{k_{up}}^{up} \neq h_{k_{up}}^{low}$ **then**
        $s_{k_{up}}^{above} \leftarrow s_{k_{up}}^{above} + y_i$
      if $k_{low} \leq |\boldsymbol{q}|$ **then**
        $s_{k_{low}}^{under} \leftarrow s_{k_{low}-1}^{under}$
  if $m_{low} \leq |\boldsymbol{q}|$ and $i = h_{low}^{m_{low}}$ **then**
    $y_{m_{low}}^{low} \leftarrow y_i$
    $m_{low} \leftarrow m_{low} + 1$
  if $m_{up} \leq |\boldsymbol{q}|$ and $i = h_{up}^{m_{up}}$ **then**
    $y_{m_{up}}^{pred} \leftarrow y_{m_{up}}^{low} + (h_{m_{up}} - h_{m_{up}}^{low}) \cdot (y_i - y_{m_{up}}^{low})$
    $m_{up} \leftarrow m_{up} + 1$
$s \leftarrow 0$
for $j \in \{|\boldsymbol{q}|, |\boldsymbol{q}| - 1, \ldots, 1\}$ **do**
  $s_j^{above} \leftarrow s_j^{above} + s$
  $s \leftarrow s_j^{above}$
  $s_j^{under} \leftarrow (h_j^{low} + 1)y_j^{pred} - s_j^{under}$
  $s_j^{above} \leftarrow (N - (h_j^{low} + 1))y_j^{pred} - s_j^{above}$
  $e_j \leftarrow q_j s_j^{under} + (q_j - 1)s_j^{above}$
return $\boldsymbol{e}$

Using this technique, we store in s_j^{under} the sum of the elements of $\boldsymbol{Y}$ that are such that $y_{j-1}^{pred} < y_i \leq y_j^{pred}$. However, we wish to store the sum of all elements below y_j^{pred}. This is why we do the operation in line 18.

Similarly, without further addition in complexity, we store in s_j^{above} the sum of the elements that are $y_j^{pred} < y_i \leq y_{j+1}^{pred}$. To have the sum of all elements $y_i > y_j^{pred}$, we do the backward loop on quantiles to add to each sum, the sum of all above elements. This is shown in line 27.

Finally, to compute the final errors, we use the fact that the quantile error of equation (3.5), without the $\frac{1}{N}$ factor, can be rewritten as in equation (4.1).

$$QL_q(\boldsymbol{Y}^p, \boldsymbol{Y}) = q \sum_{i:y_i \leq y_i^p} (y_i^p - y_i) + (q-1) \sum_{i:y_i > y_i^p} (y_i^p - y_i) \tag{4.1}$$

Using this formulation and the above sums, lines 29 to 31 show the final error computation. From this formulation of the algorithm, it is straightforward to see that we perform two loops of size $|\boldsymbol{q}|$ only containing constant-time operations, so $\mathcal{O}(|\boldsymbol{q}|)$. We also have the central loop that goes through the samples, which also contains only constant-time operations. This makes it run in $\mathcal{O}(N)$ time. Making the whole algorithm run in $\mathcal{O}(\max\{|\boldsymbol{q}|, N\})$ time. However, it makes little sense to estimate $|\boldsymbol{q}|$ quantiles if there are only $N < |\boldsymbol{q}|$ samples. We therefore recommend to always set the $minsup$ parameter of DL8.5 such that $minsup \geq |\boldsymbol{q}|$, thus ensuring $N \geq |\boldsymbol{q}|$, which makes Algorithm 6 run in $\mathcal{O}(N)$ time.

Algorithm 6 shows how to compute the quantile estimates based on the second method of Section 3.3.2 that is based on linear interpolation of the cdf function. To use the first way which provides the optimum for the quantile loss, the only change that must be made is to distinguish the three cases when computing h^{up} and h^{low}.

1. If $q_j < 0.5$, we set $h_j^{low} = h_j^{up} = \lceil h_j \rceil$
2. If $q_j = 0.5$, we set $h_j^{low} = \lfloor h \rfloor$ and $h_j^{up} = \lceil h_j \rceil$
3. If $q_j > 0.5$, we set $h_j^{low} = h_j^{up} = \lfloor h_j \rfloor$

4.4 Implementation

In the current implementation of DL8.5, for each node of the search tree that was explored, the error was stored, with the best attribute to split this node into if it is not a leaf, and saving the left and right branches of the tree.

Because now we are optimizing many functions at once, for each quantile we do not only store one value but a table of values. Meaning that now the search tree nodes hold arrays for errors, attributes, and left and right branches. The current implementation of the algorithm was also extended to implement the new branching and pruning conditions shown in Algorithm 4 and explained in Section 4.2.

Another main addition to the code base was the `QuantileLossComputer` class, which implements Algorithm 6 in C++ code. Of course, the computations in the algorithm require storing many intermediate results: $\boldsymbol{h}, \boldsymbol{h}^{up}, \boldsymbol{h}^{low}, \boldsymbol{s}^{above}, \boldsymbol{s}^{under}, \boldsymbol{y}^{up}$ and $\boldsymbol{y}^{pred}$. We also know that DL8.5 is very memory intense, so allocating (and freeing) those arrays every time the function is called would be inefficient. Because these arrays have size $|\boldsymbol{q}|$, no matter the itemset they are called on, we can only allocate those vectors once and reuse the same ones for every node explored, this way no new memory needs to be allocated for intermediate results. We do however need to allocate a new $\boldsymbol{e}$ array every time, which is also of size $|\boldsymbol{q}|$.

Finally, in Section 4.3 we discussed two methods of computing the quantile predictions which influence the loss computation and hence might change the final decision tree obtained. Both have been implemented into the C++ code, and the user can set the `quantile_estimation` parameter to a code corresponding to either `optimal` or `linear` estimation.

4.5 Smooth predictions

The quantile values themselves give a lot of information on the distribution of the variable observed, because they are the reciprocal of the cdf. However, they are harder to interpret than the pdf itself.

To mitigate this effect, we propose a way to interpret the quantile values obtained from the DL8.5 algorithm into a smooth, easily interpretable estimation of the pdf. This pdf estimate $\hat{f}$ would need to have the following properties.

$$\lim_{y \to -\infty} \hat{f}(y) = 0 \tag{4.2}$$

$$\lim_{y \to +\infty} \hat{f}(y) = 0 \tag{4.3}$$

$$\int_{-\infty}^{+\infty} \hat{f}(y) dy = 1 \tag{4.4}$$

To this effect, we take from the ideas exposed in [9] and [11]. Indeed, in those papers, they use kernel density estimation to get a smooth pdf estimation from the quantile values. The equation for this estimation is

$$\hat{f}(y) = \frac{1}{|\boldsymbol{q}|h} \sum_{j=1}^{|\boldsymbol{q}|} K\left(\frac{y - y_{q_j}}{h}\right) \tag{4.5}$$

.

Here, K is a kernel function that holds pdf properties (4.2) to (4.4), for example the Gaussian density function of unit variance and zero-mean,

$$K_G(y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}y^2}. \tag{4.6}$$

h in equation (4.5) is a parameter that controls the width of the kernels in the estimation. The higher the h value, the smoother the estimation will be. [18] showed that applying kernel density estimation on quantile values gives a smooth estimation that fits all pdf properties exposed above.

When computing a kernel density, there remains the choice of the h parameter. Multiple approaches exist: it can be done by cross-validation, or rules of thumb can be used. One of which is detailed hereunder.

Scott's rule Presented in [19], Scott's rule defines h as

$$h = \left(\frac{4\hat{\sigma}^5}{3|\boldsymbol{q}|}\right)^{\frac{1}{5}} \tag{4.7}$$

Here, $\hat{\sigma}$ represents the empirical standard deviation of the quantile values predicted.

This value for parameter h was shown in [19] to be optimal for univariate estimations when the kernel used is the Gaussian as in equation (4.6) and the underlying distribution is Gaussian as well. While we cannot make that second assumption in all cases, we will analyze the results this method gives in the following experiments.

Because this is not as much a direct contribution to DL8.5 but a way to treat the output of the models, this was not directly implemented into the DL8.5 package but was rather implemented as a wrapper that takes a `DL85QuantileRegressor` as argument, that fits the kernel density on the predictions of the model and outputs smooth pdf estimations.

For this implementation we used SciPy's `gaussian_kde` class, described in [20], that directly allows fitting a kernel density estimation using Gaussian kernels as in equation (4.6) and that contains the implementation of Scott's rule as defined in (4.7).

CHAPTER 5

Experiments

This chapter will explore the uses of Quantile DL8.5 in practice, and how it performs on different metrics.

Throughout this chapter we will explore how efficient this new algorithm is, the quality of the results it provides and how interpretable the final models are.

To this effect, this chapter is structured by starting to describe the environment of the experiments and the data used in them, followed by precise explanations of the different metrics used and how to interpret them, finishing by a results section that evaluates the metrics and interprets the results with respect to the criteria stated above.

For the experiments, we have created and used two different python packages. The extended pydl8.5 package that holds the code to the learning algorithm is available at `https://github.com/valentinlemaire/pydl8.5`. We have also created a pure python package, that enables easy testing of the algorithm and provides a few helper classes and functions that are relevant to the use of Quantile DL8.5. It is available at `https://github.com/valentinlemaire/dl85-experimental-utils`.

5.1 Experimental setup

In this section, we will describe the testing framework we have used throughout the experiments. Starting with the run environment, followed by a description of the three datasets used and finishing with a short description of the competing algorithms.

5.1.1 Run environment

All experiments were run on Microsoft Azure Databricks, using for each of them the same virtual machines, namely `Standard_DS13_v2` having the following hardware specifications (provided by Microsoft in [21]).

- **vCPU** : 8
- **Memory** : 56 GB
- **Temp storage (SSD)** : 112 GB
- **Max data disks** : 32
- **Max cached and temp storage throughput** : 32000 IOPS / 256 MBps
- **Cache size** : 288 GB
- **Max uncached disk throughput** : 25600 IOPS / 384 MBps
- **Max NICs** : 8
- **Expected network bandwidth** : 6000 Mbps

The VMs also had the following software specifications.

- **Databricks Runtime Version** : 10.4 LTS (including Apache Spark 3.2.1 and Scala 2.12)
- **Python version** : 3.8

Because memory and time are measured in most experiments and to avoid all possible overhead, the notebook making the experiment measurements was the only user-launched process on the VM for each experiment.

5.1.2 H&M Group dataset

As mentioned before, this thesis is done in collaboration with H&M Group and using DL8.5 in a quantile regression context comes from their need to control over and under predictions when estimating their sales for the following weeks. To this effect, they have provided us with a dataset to experiment with. Each row in this dataset corresponds to an article. The columns of the dataset correspond to the following information.

- `SIZE_CODE` - Code that matches to size of variant.

- SELLING_WEEK - How many weeks since they started selling the variant.
- CALENDAR_WEEK - Goes from 1 to 53, weeks in the year.
- SALES_PCS_1W - How many pieces were sold the week before current week. Negative values indicate returned items.
- SALES_PCS_1W_LAG_1 - How many pieces were sold the 2 weeks before current week.
- SALES_PCS_1W_LAG_2 - How many pieces were sold 3 weeks before current week.
- TOTAL_STOCK - Total end-of-week stock last week.
- TOTAL_STOCK_LAG_1 - Total end-of-week stock 2 weeks ago.
- TOTAL_STOCK_LAG_2 - Total end-of-week stock 3 weeks ago.
- BALANCING_WAREHOUSE - Binary feature that indicates the type of the warehouse.
- SALES_TARGET - Sum of sales for this variant at this warehouse for the coming six weeks. This is the target feature.

Binarization While discrete, these features are however not all binary. Since DL8.5 only operates on binary features, the dataset had to be transformed.

We made the arbitrary choice to binarize the dataset with 5 bins per feature if said feature had more than 5 different values, otherwise a one-hot encoding was generated. Of course, the target feature was left as is because we are performing regression.

When choosing thresholds for the bins, we preferred to create equally-sized bins (we selected ranges such that the bins of the features have approximately the same number of samples for which the created feature is positive), rather than equal-range bins (where the range thresholds for the binary features are evenly spread across the feature range, but bins can have very different supports), especially as the features corresponding to the stock and sales have many outliers. Of course, all bins are not exactly equally sized because integer features cannot always be continuously split at any percentile.

Finally, we obtained a dataset with 46 binary features. Because finding how to efficiently binarize a dataset is not the subject of this thesis, this binarized dataset was used as is in all the following experiments, even in the models that can handle non-binary features.

5.1.3 Artificial dataset

While having a real life dataset is very valuable in ensuring that the methods considered work for true applications, it is not always the most efficient way to evaluate performances.

Especially in this context of predicting probability functions, if we generate a dataset based on certain distributions, then we will know the actual true distributions of the data, making performance evaluation more meaningful.

To this effect, we have created a dataset of size `n_samples` composed of `n_features` binary features, and `n_categories` classes within the data. Each of these classes is associated to a random combination of the features (each individual feature being either positive or negative) that is different from all other classes. Each of the classes is also associated to a Gaussian distribution with a certain mean and variance. The target value of the dataset for each sample of the class will be sampled from that Gaussian distribution.

A parameter of the dataset creator is the `sparsity`. The lower this value, the closer the mean and variance of the distributions of classes that are close will be. Here, we define "close" classes as classes that are associated to similar combinations of positive/negative features. Therefore, for a high `sparisity` value, the distributions within the dataset will be very different.

Each class is associated with a certain weight that indicates the probability of a sample to belong to that class. This probability is sampled from a Poisson distribution, so the categories can have very different supports, to simulate class imbalance.

Finally, white Gaussian noise was added to all target samples and the binary features of the dataset were switched from 1 to 0 or from 0 to 1 with a probability controlled by the `feature_noise_percentage` parameter.

The end dataset that was used in the following experiments was generated with the following parameters

- `n_features` : 9
- `n_samples`: 100 000
- `n_categories`: 15

- `sparsity`: 150.0
- `feature_noise_percentage`: 0.02
- `target_white_noise_variance`: 0.6

5.1.4 Air Quality dataset

The two above presented datasets are specific to this thesis. The first showing how this novel algorithm performs on the original use case which was to predict quantiles and full distributions functions for the future sales on the H&M Group dataset. The second showing how Quantile DL8.5 performs when we actually know the true distribution function of the dataset and are able to compare it to the predicted one.

However, when creating new algorithms and comparing them to existing methods, it is common practice to use standard, publicly available datasets, hence allowing the experiments to be reproduced. For this purpose, we have chosen the Air Quality dataset from the UCI Machine Learning repository. It is available at [22].

This dataset is generally used to predict the concentration of different pollutants in the air based on sensor data of 5 different concentrations of components in the air (not including the target concentrations) and the time and date. Because Quantile DL8.5 is limited to univariate predictions, we have chosen to only predict the concentration of C_6H_6 in the air based on the features.

Like the H&M Group dataset, this dataset is not composed of binary features and needs to be binarized. We did this using the same procedure as described in Section 5.1.2.

5.1.5 Competing algorithms

To be able to assess the quality of the generated models, it is good practice to compare said models to others that hold some of the same properties.

To that effect, we have decided to compare our algorithm with two implementations of Random Forests that, for the first one, has to be extended to be able to generate smooth pdf predictions (in the same way as Quantile DL8.5) and one that already predicts smooth functions. Both are ensemble methods that generate multiple decision trees in the learning process.

Quantile Random Forests As discussed in Section 2.3.2, regular Random Forests can be adapted to create conditional quantile predictions, as shown by [7]. Some implementations exist for this adaptation, for our experiments we have chosen the `scikit-garden` implementation (documentation available at [23]).

On top of this model, we can fit a KDE implementation to provide smooth pdf estimations. We have used the same as the one described in Section 4.5.

CaDET Random Forests The other model we shall compare Quantile DL8.5 is CaDET Random Forests, this algorithm is described in Section 2.3.5. We have used the implementation provided by the authors of the paper and that is available at [24]. It is constructed as an extension to scikit-learn.

An important aspect of this method is that it is parametric, meaning that the user has to assume families of distributions to which the estimated densities might belong and will be limited by the ones available in the CaDET implementation.

5.2 Metrics

To ensure that the final algorithm we provide does perform well in terms of sheer efficiency, in terms of quality of the results obtained and in terms of interpretabilty, we need to provide quantitative results that qualify those characteristics. In this section we provide different metrics to estimate the quality of the Quantile DL8.5 model for those properties.

5.2.1 Run metrics

Being an algorithm that performs an exhaustive search that uses caching, DL8.5 is intense both on CPU time and on memory consumed.

Memory Because the Quantile DL8.5 algorithms performs the search for many trees at once, rather than just a constant amount of information (leaf error, pointers to right and left children, the best error, ...) per node explored, it stores each of these once per tree that needs to be learned. So if we train for 100 different quantiles, then at each node, 100 different values will be stored for each of these metrics.

For this reason, Quantile DL8.5 is very memory consuming, when comparing it to other algorithms and evaluating performance, we will measure **mean memory consumption** and **maximal memory consumption**.

In practice, the `memory-profiler` (documentation available at [25]) python library was used and memory was sampled every hundredth of a second.

CPU time Because one of the main contributions of this thesis is an optimization that allows the algorithm to learn many trees at once, we will see how this optimization impacts CPU time of the learning when varying different training parameters and compare it to an unoptimized version that learns all trees one after the other.

5.2.2 Quality metrics

The quality of the end results is a core desirable property of the models generated, we provide in this section 4 different metrics to evaluate the end models, each focusing on a certain aspect of the predictions.

Mean Integrated Squared Error If we use the technique described in Section 4.5, we are able to take the quantile values generated from the algorithm and obtain a smooth pdf prediction for the conditional distributions.

Because this prediction is a smooth function, regular, discrete metrics will not be as descriptive of the whole function obtained. Therefore, we might want to use Mean Integrated Squared Error (MISE), which is defined as

$$MISE(f, \hat{f}) = \int_{-\infty}^{\infty} (f(y) - \hat{f}(y))^2 dy. \tag{5.1}$$

An illustration of the MISE metric is depicted in Figure 5.1. In this figure, MISE is the integral of the square of the colored region.

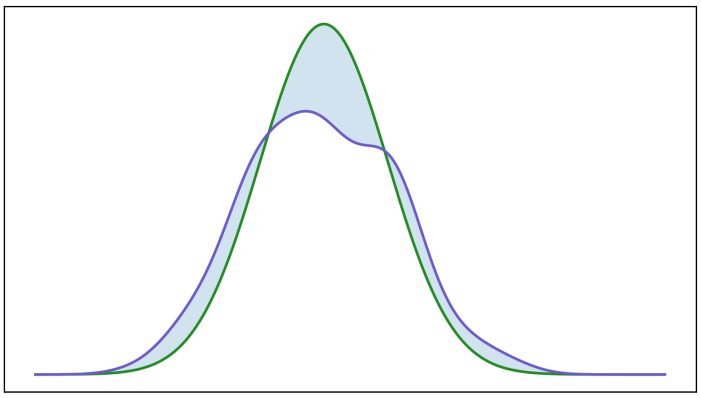


Figure 5.1: MISE illustration

The problem with this metric is that it requires the true pdf f, which, for real datasets, is unknown. Therefore, we will only be able to use this metric for the artificial dataset described in Section 5.1.3 because for this dataset we know from which distribution the samples were generated.

To obtain a MISE value for a whole test dataset, the mean is taken over all the predicted and true distributions for each sample in the test dataset.

Log-likelihood Evaluating the pdf function f at certain points can be interpreted as how likely that point is. Hence, the product of all likelihoods represents how likely the data is, given a certain probability density function. Because the likelihoods are often very small values and their product between all samples would get close to zero, common practice is to take the opposite of the logarithm of this value, giving the log-likelihood (LL) metric, defined as

$$LL(\boldsymbol{Y}, \hat{\boldsymbol{f}}) = -\sum_{i=1}^{|\boldsymbol{Y}|} log\left(\hat{f}_i(y_i)\right). \tag{5.2}$$

We take the negative here for clarity so that the lower this metric is, the better the estimation is, like the others considered.

Log-likelihood is a common measure when trying to estimate the distributions based on sample data, this is for example mentioned in [13].

Mean Quantile Error Because we are optimizing the quantile loss in DL8.5, it is interesting to see how it is performing with respect to that loss compared to other algorithms, even if those algorithms do not give a similar model in the end. To this effect, we can for each quantile compute its loss as described in equation (3.5) associated to the test dataset and then average all the losses across the different quantiles, the Mean Quantile Error (MQE) is defined as

$$MQE(\boldsymbol{Y}, \boldsymbol{Y}^p) = \frac{1}{|\boldsymbol{q}|}\sum_{j=1}^{|\boldsymbol{q}|} QL_{q_j}(\boldsymbol{Y}, \boldsymbol{Y}_j^p) \tag{5.3}$$

Here $\boldsymbol{Y}^p$ corresponds to the quantile value predictions corresponding to the quantiles $\boldsymbol{q}$.

Figure 5.2 shows an illustration of the quantile error for three different quantiles. We can see that as mentioned in Section 3.3, for each individual quantile, the corresponding quantile loss is minimized for the true quantile value of the distribution from which the samples were generated.

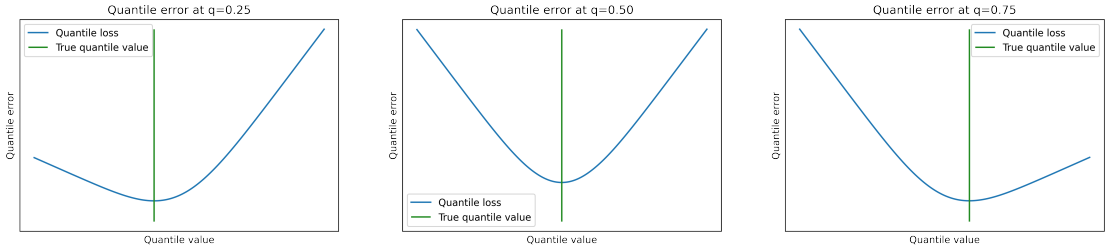


Figure 5.2: Quantile Error illustration

Continuous Ranked Probability Score Finally, another metric that can be used when creating smooth pdf estimations is the Continuous Ranked Probability Score (CRPS). This is a standard metric for this type of applications, used in [11] for example. This metric considers that the true distribution for each sample in the test set is actually a Dirac delta centered at the true value for the sample.

Using this assumption, the cdf for the true pdf is a step function $I(y - y_i)$ for sample i. This is then compared to the estimated cdf $\hat{F}_i$ for this sample, and the integral of the squared difference is computed. This is formulated mathematically as

$$CRPS(\hat{\boldsymbol{F}}, \boldsymbol{y}) = \frac{1}{|\boldsymbol{y}|} \sum_{i=1}^{|\boldsymbol{y}|} \int_{-\infty}^{\infty} (\hat{F}_i(\lambda) - I(\lambda - y_i))^2 d\lambda. \tag{5.4}$$

Here, $\hat{F}(y)$ is the estimated cdf, which can be computed as

$$\hat{F}(y) = \int_{-\infty}^{y} \hat{f}(\lambda) d\lambda \tag{5.5}$$

An illustration of the CRPS metric is depicted in Figure 5.3. In this figure, CRPS is the integral of the square of the colored region.

This metric is interesting because it takes into account both how smooth the prediction is and how well the predictions describe the dataset.

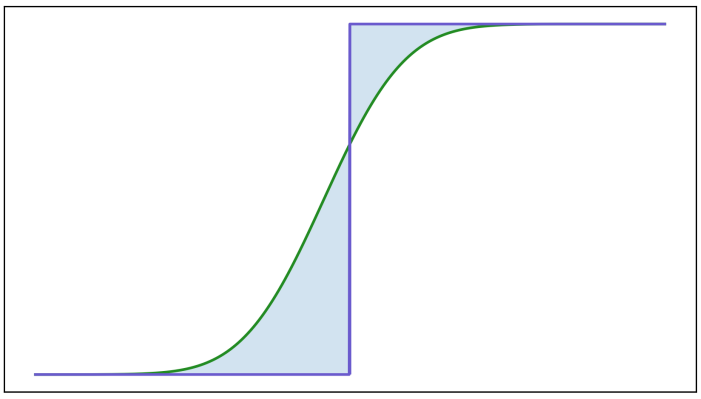


Figure 5.3: CRPS illustration

5.2.3 Interpretability metrics

One very desirable property of the final model is its interpretability. Unfortunately, this is a difficult criterion to quantify. When considering the different trees obtained by DL8.5, each corresponding to a quantile, we might expect trees that correspond to close quantiles to be very similar to each other because they are optimizing functions that are very similar.

Having similar trees would make the interpretation easier, as analyzing trees that evolve little is easier than analyzing trees that are completely different from one another. In addition to this, seeing that the trees evolve smoothly over the quantiles would indicate that the trees are in fact learning well, as each tree is approximating a different region of the monotonic quantile function.

There remains to qualify the similarity of trees, which is not a trivial task. Indeed, trees are a particular space where not only the values in the leafs are important but the structure of the tree itself is part of the information. And while the trees have the same final maximum depth, they might not have the same number of nodes and leafs.

To take these challenges into account, we have derived a comparison metric that considers the tree as a collection of paths, which are sets of features. These paths can be interpreted as partitions of the data. Of course, doing this removes part of the hierarchical information of the trees, but we will perform a level-wise analysis as explained later to still keep this hierarchy element in our analysis.

We now need a metric that can quantify how similar two partitions of the same data are, while taking into account that the partitions might not have the same number of

classes. Taking inspiration from the clustering domain, we have used a metric described in [26]. In this article they consider the two partitions as lists, $\mathcal{C}^1$ and $\mathcal{C}^2$, where $\mathcal{C}_i$ gives the class of sample i and they consider all pairs of samples $(i, j) \in \mathbb{I}^2$ where $\mathbb{I} = \{1, 2, \ldots, |\boldsymbol{y}|\}$ and compute the following values

$$A = \left|\left\{(i,j) \in \mathbb{I}^2 : \mathcal{C}_i^1 = \mathcal{C}_j^1 \text{ and } \mathcal{C}_i^2 = \mathcal{C}_j^2\right\}\right| \tag{5.6}$$

$$B = \left|\left\{(i,j) \in \mathbb{I}^2 : \mathcal{C}_i^1 = \mathcal{C}_j^1 \text{ and } \mathcal{C}_i^2 \neq \mathcal{C}_j^2\right\}\right| \tag{5.7}$$

$$C = \left|\left\{(i,j) \in \mathbb{I}^2 : \mathcal{C}_i^1 \neq \mathcal{C}_j^1 \text{ and } \mathcal{C}_i^2 = \mathcal{C}_j^2\right\}\right| \tag{5.8}$$

Using these pairwise values, the Jaccard index describing the similarity of the partitions can be computed by

$$J = \frac{A}{A + B + C}. \tag{5.9}$$

Tree similarity matrix Because we are faced with not only two trees but a full set of trees, we must compute this similarity for each pair of trees, thus giving a matrix of similarity between the trees.

Finally, trees have a hierarchy. To have interpretability, we would expect the higher levels of the tree to be common to most quantiles and see that the lower levels contain the information about the smaller variations in the data. To this effect, we have computed one similarity matrix by limiting them to every level and for each pair of trees. Hence, if we have a hundred trees of maximum depth 5, we will obtain 5 matrices of shape 100×100.

Because we obtain full matrices and not just one value, an interesting question is to see how we can interpret them. If the trees are very similar to the trees close to them but evolve through the quantile range because the features describing the higher quantiles are very different from the features describing the lower quantiles, we might expect a banded matrix as shown in Figure 5.4. For such a structure to be desirable the band has to be large, to indicate that trees evolve slowly over the range of quantiles, a very narrow band would mean that trees corresponding to close quantiles change a lot, which is not desirable.

If, however, there is less variation over the quantile range and the almost exact same trees describe close quantiles, we might observe more of a block structure in the matrix. Of course very similar trees could describe distant ranges of quantiles if the features

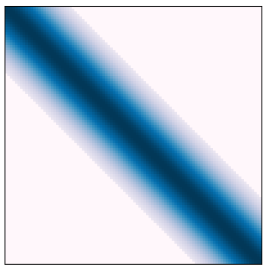


Figure 5.4: Banded matrix

that describe best these two ranges are similar, this would give blocks that are not on the diagonal. An example of such a structure is given in Figure 5.5.

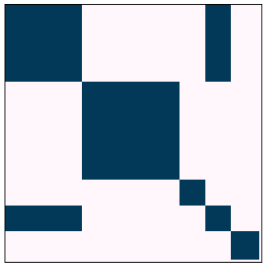


Figure 5.5: Block matrix

Both of the above presented examples are desirable structures of similarity matrices between the trees. Of course, they are ideal cases and in practice we might expect a combination of the two with some form of noise in the matrix. What we don't want is a fully random matrix where each tree is independent of the others. An example of such a matrix is given in Figure 5.6.

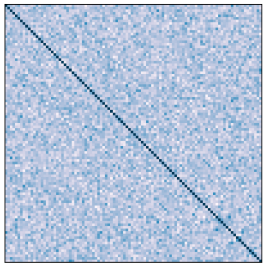


Figure 5.6: Noise matrix

Mean tree similarity We might also want to use this metric to qualify the similarity of the trees obtained by the competing methods, namely the ones described in Section 5.1.5, but in those, the order of the trees does not have meaning, so the matrix representation is not the best way to evaluate. However, we can compute the mean of the matrix for the largest depth to get an idea of how similar the trees are overall.

5.3 Results

For each of the three main points raised in the previous section, we have derived experiments to evaluate the performances of our algorithm, comparing it to the competing methods when relevant to provide a meaningful baseline. In this section, we present the results of those experiments.

5.3.1 Memory consumption

For every experiment here, we used the H&M Group dataset described in Section 5.1.2, but we sampled it randomly to only keep 70 000 rows. For each of the following experiments we varied one parameter of the model and the others were maintained constant with the following values.

- Maximum depth : 4
- Minimum support : 3000
- Quantiles : 25 quantiles distributed evenly between 0.01 and 0.99

To evaluate memory consumption of the Quantile DL8.5 algorithm, we ran it against the unoptimized version of the DL8.5 algorithm that learns one tree per quantile sequentially.

Some graphs have missing data for the unoptimized code, this is when the training time for these parameters surpassed 12 hours.

Influence of maximum depth In Figure 5.7, we can observe the influence of maximum depth on the memory used by the two algorithms.

As expected, running sequentially the searches for each individual tree consumes less memory than launching them all at the same time. We also observe that with maximum depth, memory grows rapidly and that the gap between the two versions of the algorithm gets bigger. This makes sense as we expect deeper trees to differentiate

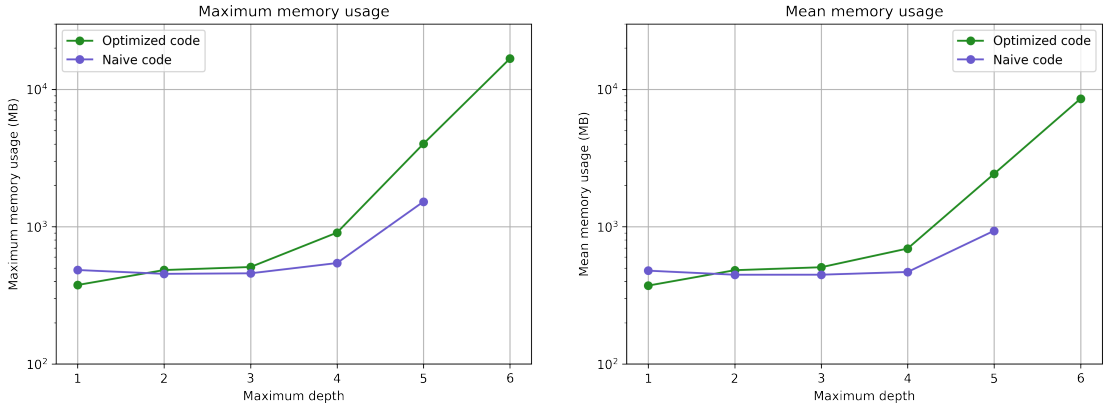


Figure 5.7: Memory consumption evolution with maximum depth

themselves in the lower levels, so doing the whole search at the same time would need to go through many nodes to be able to compute the optimum for all quantiles. The naive version does not suffer from this, as it only needs to optimize for one tree before clearing the cache (and memory).

Influence of minimum support In Figure 5.8, we can observe the influence of minimum support on the memory used by the two algorithms.

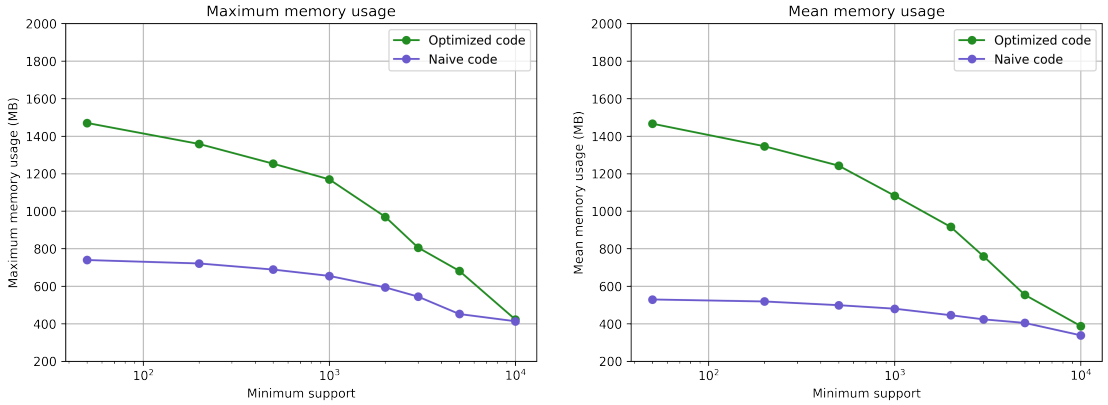


Figure 5.8: Memory consumption evolution with minimum support

Here we again see that the optimized code is much more memory-consuming than the naive one. We also observe that the difference in memory consumed is much reduced when minimum support becomes higher. This is to be expected as with a high minimum support, fewer leafs become feasible and the space of feasible trees is

much reduced, thus most trees would look similar, and the same nodes would need to be explored for all quantiles, thus training them all at once induces less overhead memory-wise.

Influence of number of quantiles In Figure 5.9, we can observe the influence of the number of quantiles for which to learn trees on the memory consumption. For this experience, we chose a linear space of quantiles, with just the number of said quantiles varying.

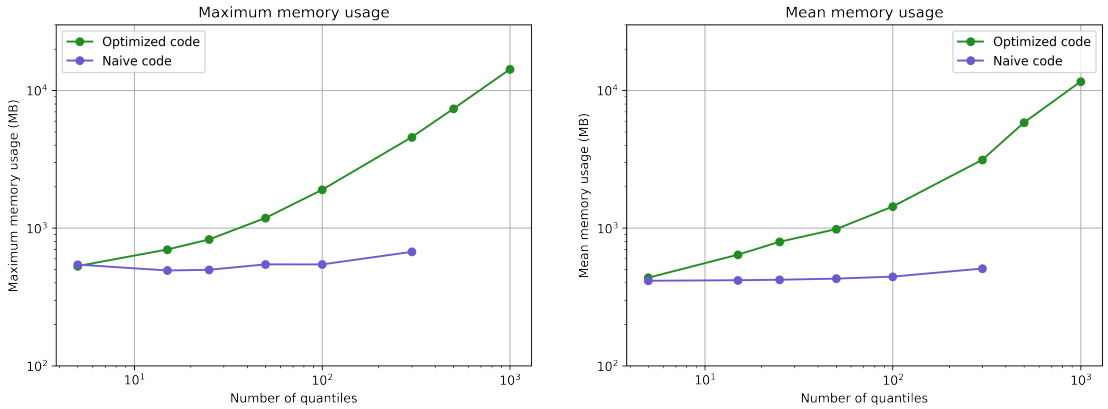


Figure 5.9: Memory consumption evolution with number of quantiles

Here we see that for the naive code, the memory remains almost constant for any number of quantiles, which makes sense as for each tree the explored space might be different, but we might expect that a similar total number of nodes will be explored. For the optimized version, however, we see the memory consumption rapidly growing with the number of quantiles. This is due to two reasons. With a higher number of trees to learn, each node of the cache has to store $|\boldsymbol{q}|$ times each information. The other reason being that with a high number of trees to learn, to find the optimum for all of them, a bigger search tree will be generated and for each node the information about each quantile is saved, even if the optimum for the corresponding quantile has already been found.

5.3.2 Execution time

In this section, we present the influence of the different model parameters on the execution time of the algorithm. For this, we used the exact same experiment setup as described in the beginning of Section 5.3.1.

Influence of maximum depth In Figure 5.10 we can observe how the maximum depth parameter influences execution time on the H&M Group dataset.

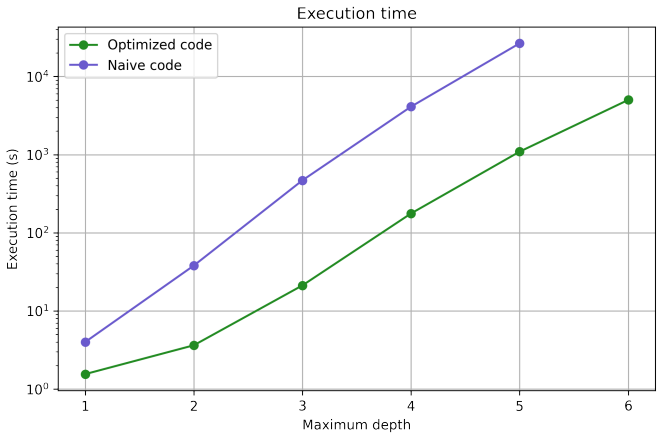


Figure 5.10: Execution time evolution with maximum depth

In this figure, we can see that the optimized code is consistently faster than the naive code. We could however wonder by what factor the algorithm is actually faster. Table 5.1 shows these values, and we can observe that with higher maximum depths the speedup gets close to 25 which is the number of trees we are learning. We do however see that the speedup is much less impressive for shallower trees, which makes sense as the time to learn such trees is much lower, so the speedup difference is less marked.

	Maximum depth				
	1	2	3	4	5
Speedup	2.57	10.48	22.12	23.29	24.36

Table 5.1: Speedup factor with maximum depth

Influence of minimum support In Figure 5.11 we can observe the influence of minimum support on the execution time of the algorithms.

Similarly to maximum depth, the optimized code is consistently faster than the naive code. Table 5.2 shows the values of the speedups with respect to the minimum support. Here again we see that the speedup is quite important, especially for large values of minimum support because in that case the trees are very similar, the search space is very small and only needs to be explored once. Because the errors are computed only

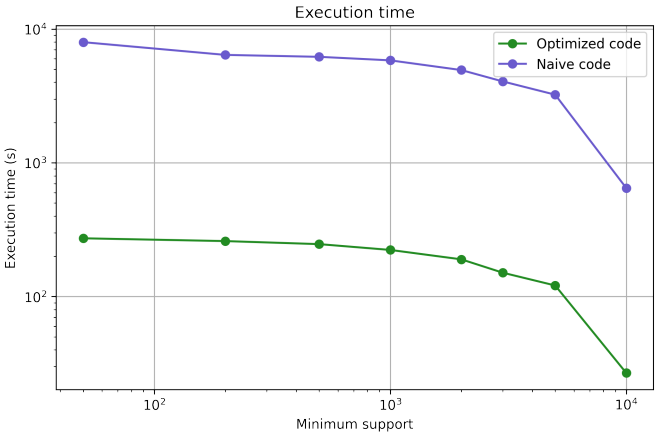


Figure 5.11: Execution time evolution with minimum support

once for all the quantile values, there is much less traversing of the data than when learning all the trees one after the other.

	Minimum support							
	50	200	500	1000	2000	3000	5000	10000
Speedup	2.38	12.43	16.45	22.11	30.73	41.10	52.94	297.07

Table 5.2: Speedup factor with minimum support

Influence of number of quantiles In Figure 5.12 we see the influence of the number of quantiles, and hence, the number of trees to learn, on the execution time of the algorithms.

In this figure, we see the most import aspect of the optimizations brought up in Chapter 4. Indeed, here we see that with the evolution of number of quantiles, we obtain huge speedups with the optimized version learning the trees simultaneously compared to the naive version. Table 5.3 shows the values of those speedups. These results show that for lower number of quantiles to be learned, the speedup is almost equal to the number of trees, i.e., the optimal speedup we can hope for. However, when going to higher number of quantiles the trees start to differ more from each other and the speedups become less impressive compared to the optimal speedup but still allow for much faster learning than the naive version.

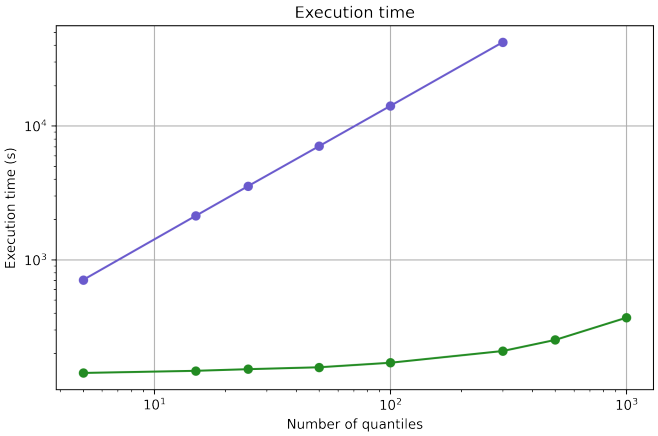


Figure 5.12: Execution time evolution with number of quantiles

	Number of quantiles					
	5	15	25	50	100	300
Speedup	4.96	14.40	23.35	44.94	82.94	203.08

Table 5.3: Speedup factor with minimum support

5.3.3 Quality metrics

In this section, we will explore the performances of the Quantile DL8.5 algorithm in terms of result quality. To do this, we have compared it to the algorithms presented in Section 5.1.5.

For this experiment, we have performed grid search selection with 5-fold cross validation to choose the best model for each of the family of algorithms and for each dataset. For the H&M Group and Artificial datasets, this was done using a training dataset of 50 000 samples. We also sampled a test dataset of 5000 samples independent of the train set. This test set was used to generate the final metrics. For the Air Quality dataset, which is only composed of 8991 samples, we used a 90-10 percent train-test split.

In all the metrics used in this section, the lower the metric is, the better the quality of the result is.

H&M Group dataset After cross validation on the H&M Group dataset, we obtained the best models for each family of algorithms. Some characteristics of these models are indicated in Table 5.4.

	QRF	CaDET RF	QDL8.5
Number of trees	50	100	100
Minimum support	500	1000	500
Maximum depth	8	5	5

Table 5.4: Best model characteristics for the H&M Group dataset

Using best models, we obtained the results shown in Table 5.7. It shows the results of the quality metrics with respect to the H&M Group dataset described in Section 5.1.2. In this table, we see that our algorithm performs poorly on the Log-likelihood metric, where CaDET obtains the best result. This could be expected as CaDET tries to optimize this metric and chooses greedy splits that give the best amelioration for this metric.

Surprisingly, it also performed worse than CaDET on mean quantile error, even when it is the metric Quantile DL8.5 optimizes. This might be explained by the fact that the H&M Group dataset is large and presents high variability on the target variable. On top of that, memory and time limitations prevented us from training models with a maximum depth higher than 5 during cross validation.

Because of this, the trees learned by our algorithm might not have enough degrees of freedom to accurately describe the quantiles of the dataset, which might explain poor performance.

That alone does not explain, however, how CaDET performs better with the same number of trees and the same maximum depth. This can be explained by the fact that the final predictions in the leafs of the CaDET trees are full distribution functions represented by the sufficient statistics for a certain distribution family and the family itself, which also holds information. Whereas Quantile DL8.5 only stores 1 value for every leaf. So for a same depth and number of trees, CaDET actually has a lot more degrees of freedom.

Finally, we do observe that Quantile DL8.5 performs the best on the CRPS metric, especially compared to the CaDET RF, which seems to diverge rather than converge to a good solution on this metric.

Artificial dataset We performed the same model selection process on the artificial dataset and obtained best models for each algorithm considered. Table 5.5 shows some of their characteristics.

	QRF	CaDET RF	QDL8.5
Number of trees	25	100	100
Minimum support	500	1000	500
Maximum depth	7	4	5

Table 5.5: Best model characteristics for the artificial dataset

Using these parameter values, we trained best models for each algorithm and computed the quality metrics on the test set. These results are available in Table 5.8.

In this table, we see that Quantile DL8.5 performs much better than on the first dataset. Indeed, while not the best one on log-likelihood, it now outperforms Quantile Random Forest and is the best algorithm for both MQE and CRPS. This change in performance could be explained by the same intuition that justified poor performance for the H&M Group dataset.

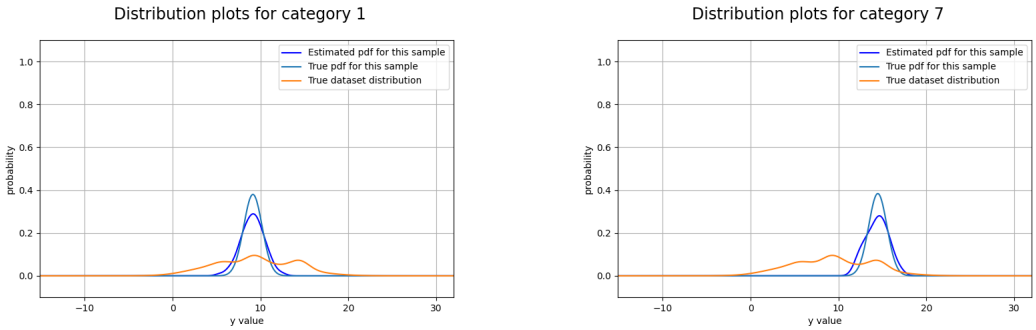
Indeed, because this dataset has only 9 features, only 15 latent categories and a lot of samples for each of those, a maximum depth of 5 is able to capture a lot more of the information about the distribution of those latent categories and therefore performs generally better on all metrics.

For this dataset, because it was generated based on known distributions, we can compute the MISE. And here we observe that rather surprisingly, Quantile Random Forest performs best when it did not perform as well on the rest of the metrics. However, it is very closely followed by Quantile DL8.5. CaDET is however not performing as well as the two others on this metric.

Something interesting to notice is that for MISE, MQE and CRPS, Quantile Random Forest and Quantile DL8.5 get similar results but CaDET is outperformed by both by a large margin. This could mean that for these metrics, using quantile based methods is better when the dataset allows for the learning of deep enough trees for these methods.

To get a better idea of what kind of results a KDE estimation on top of a DL8.5 quantile model might give, we show in Figures 5.13a and 5.13b an example of estimated pdf for two of the categories of the artificial dataset.

Air Quality dataset Again, we performed model selection for the Air Quality dataset described in 5.1.4 and obtained best models for each family of algorithms, some of their characteristics are shown in Table 5.6.



(a) Category 1 (b) Category 7

Figure 5.13: Example estimated pdfs

	QRF	CaDET RF	QDL8.5
Number of trees	200	100	100
Minimum support	100	100	185
Maximum depth	6	5	5

Table 5.6: Best model characteristics for the Air Quality dataset

Using these best models, we were able to generate the results of Table 5.9 on the corresponding test set.

In this dataset, we once again observe that CaDET Random Forests performs the best on the Log-likelihood metric. And this does indeed confirm the findings of [13] as they had pitted their algorithm against Quantile Random Forests on the Air Quality dataset and concluded that CaDET did in fact perform better.

On the other metrics considered, i.e., Mean Quantile Error and Continuous Ranked Probability Score, we see that the two quantile-based methods give the best results. Indeed, while we observe a slight difference favoring Quantile Random Forests, the two methods give almost identical results on all metrics. However, we can see that Quantile Random Forests uses twice as many and deeper trees than Quantile DL8.5 thus having a much higher model complexity.

Obtaining these results on a standard dataset confirms that Quantile DL8.5 is competitive with existing methods in terms of quality of results and depending on the metric that one wants to optimize, it actually gives the best results with the large advantage of high interpretability as will be further shown in the next section.

	Quantile RF	CaDET RF	Quantile DL8.5
Log-likelihood	25415.329	**8664.939**	152754.906
Mean Quantile Error	22.348	**19.756**	21.525
Continuous Ranked Prob. Score	338.006	173554253.658	**75.407**

Table 5.7: Quality metrics on H&M Group dataset

	Quantile RF	CaDET RF	Quantile DL8.5
Mean Integrated Squared Error	**0.067**	1.839	0.077
Log-likelihood	12501.895	**9122.300**	12266.351
Mean Quantile Error	2.440	5.663	**2.372**
Continuous Ranked Prob. Score	1.682	996.737	**1.633**

Table 5.8: Quality metrics on artificial dataset

	Quantile RF	CaDET RF	Quantile DL8.5
Log-likelihood	1787.671	**1480.154**	1787.406
Mean Quantile Error	**1.947**	4.961	1.955
Continuous Ranked Prob. Score	**1.368**	3971.031	1.397

Table 5.9: Quality metrics on Air Quality dataset

5.3.4 Tree similarity

In this section, we will explore the results obtained by the tree similarity metric from an interpretability point of view.

Similarity matrices As described in Section 5.2.3, we have computed similarity matrices for the different trees obtained by the Quantile DL8.5 algorithm. Because we want to see the evolution of the similarities with the depth of the trees, we have computed one for each depth limit. This was done on all three datasets. The results are available in Table 5.10.

These matrices correspond to what we might have expected. Indeed, in the first levels, we obtain very similar trees, especially for the H&M Group and Air Quality datasets, which indicates that one feature is very important in quantile estimation.

By going further down in the levels, we observe that for some quantiles, the trees are different from the rest of them, indicating that these are important values in the quantile range.

In the final levels, we observe that the trees differentiate themselves more and more, so more noise appears in the matrices, but we still observe a very distinct structure.

This leads us to another important observation, on all levels, including the ones corresponding to higher depths, we observe a strong block structure as the one depicted in Figure 5.5. As stated above, this is a very desirable structure as it means that close quantiles get mapped to very similar trees, which indicates that the model is learning adequately the distribution information.

Because of the block nature of the matrices, we can infer that the distribution being estimated by these models is described by quantile ranges, or intervals of quantiles and the trees within these intervals represent similar partitions of the data.

Comparison with other models As has been done multiple times in this chapter, we will also compare the results to what can be obtained in the competing methods. For each method and dataset, we have computed the mean of the similarity matrix for the final level. These results are shown in Table 5.11.

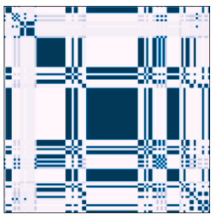
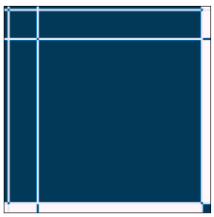
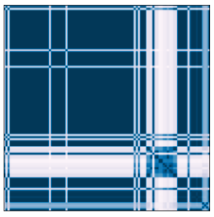
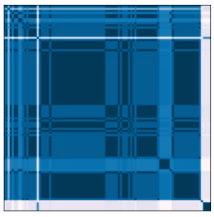
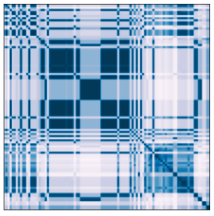
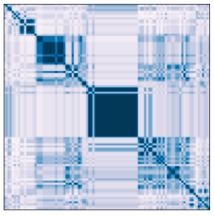
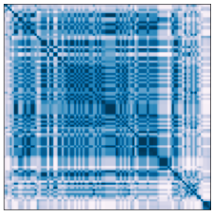
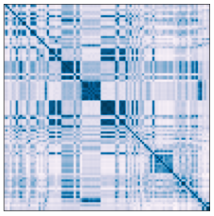
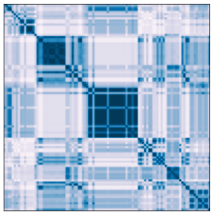
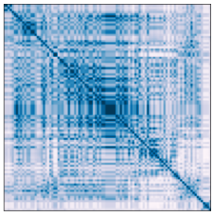
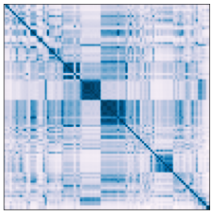
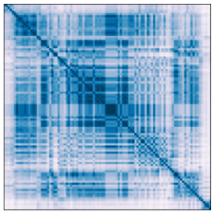
	H&M Group	Artificial	Air Quality
Depth 1			
Depth 2			
Depth 3			
Depth 4			
Depth 5			

Table 5.10: Similarity matrices for both datasets

	Models		
	QRF	CaDET RF	QDL8.5
H&M Group dataset	0.442	0.358	0.414
Artificial dataset	0.953	0.805	0.731
Air Quality dataset	0.469	0.616	0.567

Table 5.11: Mean tree similarities

This table shows that CaDET Random Forests and Quantile DL8.5 get comparable similarity measures, but Quantile Random Forests outperforms them both on the first two datasets. This can be explained by the fact that the best models for Quantile Random Forests for the first two datasets are composed of fewer but deeper trees, making it more probable that two samples get mapped to the same partition in different trees as there are more degrees of freedom in the individual trees. For the last dataset, however, Quantile Random Forests chose to use twice as many trees as the other two algorithms, explaining the decreased interpretability score.

Concerning the comparison between CaDET Random Forests and Quantile DL8.5, while they get similar overall scores on the mean similarity, the main difference is that in Quantile DL8.5, the order of the trees has meaning and that because they are each mapped to a certain quantile, this information can be exploited as well in the interpretation, when in CaDET Random Forests, tree order is meaningless and does not correspond to a specific metric or value.

It is also worth noting that CaDET is a parametric method that requires to pass a list of distribution families that can then be instantiated in the leafs, but only one distribution per leaf per tree is possible. Meaning that if the samples getting mapped to a leaf are not well described by the distributions available, we might get very skewed estimations. In this sense, Quantile DL8.5 has the advantage of being a non-parametric method.

5.3.5 Inferring clusters

It is possible to go even further in the interpretation of the similarity matrices and hence the trees in the case of Quantile DL8.5. Indeed, due to the blocky nature of those matrices, we can see that certain trees and hence, quantiles, are pivot points in those matrices, representing a significant change in the features needed to estimate the quantile values.

One way to identify those pivot points is to compute a new matrix, being the absolute difference between the similarity matrix and itself, shifted by 1 column. Then we can sum each column of this matrix and obtain a vector $\boldsymbol{d}$ of immediate difference in the tree similarities. If $\boldsymbol{M}$ is the similarity matrix of size $N \times N$, this is computed as

$$d_j = \sum_{j=1}^{`N-1} |M_{i,j} - M_{i+1,j}|. \tag{5.10}$$

Using this procedure, we were able to create the difference vector on the H&M Group dataset. This is shown in Figure 5.14.

Figure 5.14: Tree similarity differences for H&M Group dataset

Now, using this vector, we can generate clusters in the quantile range by taking the indices of the maximal elements of this vector. Doing this, we get pivot points in the matrix that define intervals, and then by selecting the intervals that have the largest sum of similarities in the original matrix $\boldsymbol{M}$, we obtain quantile ranges that correspond to very similar trees. With this technique, we were able to generate the clusters shown in Figure 5.15.

(a) Similarity matrix for depth 5

(b) Detected clusters

Figure 5.15: Cluster detection for H&M Group dataset

This analysis of the tree similarity actually allows us to select important clusters that contain very similar trees and would allow a human to easily analyze one of the trees in this cluster and interpret the decisions made there.

To illustrate this, we have taken the quantiles at the middle of the ranges detected here-above and plotted the corresponding trees. They are available in Appendix A in Figures A.1 to A.4. In these trees, the left branch corresponds to the feature being positive and inversely for the right branch.

Because of the size of these trees we did not show the full names of the binary features, but acronyms, the correspondence is shown in Table A.1, each non-binary column has 5 binary features that each correspond to an approximately equally sized bin of the feature defined by a range. We have also hidden the values of the ranges for the binarization of the features, as that information is sensitive business information for the H&M Group. For the same reason, the predicted values, corresponding to the quantiles of future sales, have been scaled by dividing all values by the median over the whole dataset. Hence, giving information about how they compare but not on absolute values.

In these trees we can make a few observations. First, even though these four trees represent four distinct parts of the distribution and can be considered as representative of dissimilar clusters, we do still observe that they share a lot of common points.

Indeed, we see that there is significant overlap of the features used in the splits in all 4 trees. A few numbers might help illustrate this. There are 46 binary features in the full dataset, only 23 are used in the combination of the four trees and each tree uses a proportion between 0.65 and 0.74 of those 23 features.

Moreover, even if the exact features are not the same, they are often closely related. If we look at the trees in figures A.1 and A.2, we see that the two first levels are the exact same except for the right-most split of the second level, which actually splits on the same underlying continuous feature but on a binary feature that represents a different range within the original feature. This indicates how even if the similarity metric shows that the trees represent very different parts of the quantile range, we still observe common characteristics that can be explained by the fact that the trees are estimating parts of the same conditional distribution.

The analysis of the trees further confirms our observation based on the similarity matrices of Section 5.3.4 as we see that the trees share a lot of similarity in the higher levels and differentiate more in the lower levels.

Generally, to the human eye and to the analyst that wants to interpret the resulting trees, it is fairly simple to take these trees and note the differences in them as they are

closely related, and those differences are minimal. This gives significant insight on how the models change the rules in the trees to make final predictions with the different quantiles and allows an analyst to easily interpret the data.

This kind of easy interpretation of a few trees to get an idea of the decision process over the whole distribution is something that is not possible with the other methods because the trees don't correspond to an interpretable value, especially for Quantile Random Forests where only the combination of all trees is meaningful for the estimation of quantiles. This makes their similarity matrices uninterpretable, thus further indicating Quantile DL8.5 is a highly interpretable method.

CHAPTER 6

Discussion

As research is a never ending path, the work in this thesis can be taken further by many directions. In this chapter, we will start by exploring different research paths that might improve the work in this thesis. After this, we will conclude this thesis by synthesizing the main aspects of the work.

6.1 Future work

In this section, we will provide a non-exhaustive list of possible amelioration directions that could be explored in future work concerning this domain.

6.1.1 Further optimization of the Quantile DL8.5 algorithm

In Quantile DL8.5, we store at each node of the search tree, the information for each of the quantiles, meaning that if we are creating N trees for N quantiles, each node of the search tree saves N different values about the error, children nodes, etc.

Because if we have discrete data like in the H&M Group dataset, and we use the optimal quantile estimation, we might often predict the same value for different quantiles for a same itemset. Because of this, having a run-length encoding of the node information, might limit the use of memory of Quantile DL8.5 which is one of its main caveats.

6.1.2 Optimizing function metrics directly

In the current implementation, we optimize quantile loss to find the optimal trees corresponding to said quantiles and then once the optimal trees for these quantiles have been found, we fit a KDE to get smooth pdf estimations.

If the end goal is to perform smooth pdf estimations, it might be relevant to find the trees based on this metric directly. So, rather than learning one tree per quantile, we would straight away fit a KDE on the samples being mapped to a leaf. This would yield only one tree, hence improving interpretability, and would allow the algorithm to directly optimize Log-likelihood or CRPS.

However, the danger of training KDEs on samples directly and not on quantiles is that it makes it more susceptible to outliers and might encourage overfit. Another option would be to compute the quantiles of the samples in the leaf and fit a KDE on those values, but here, the danger might be that partitions describe better some quantiles than others, hence the advantage of having one tree per quantile.

This being said, having just one tree that is optimal for predicting a smooth pdf would be highly interpretable and the optimality guarantee, with good pruning parameters (maximum depth and minimum support) might yield interesting results that are worth investigating.

6.1.3 Further encouraging similar trees

In the current version of Quantile DL8.5, the paths are explored in a fixed order, that can either be the alphabetical order or in increasing or decreasing order of information gain.

For each quantile, we only change the best tree found so far if the loss is strictly better and the size of the tree is lower. Hence, if we were to consider paths in an order that creates the optimal trees for most quantiles, we would greatly help the branch-and-bound search, and we would encourage the final trees to be more similar. Both of which are desirable features.

Another way to encourage similar trees would be to allow for a certain error bound for each tree. Indeed, if we have a tree A that is optimal for quantile q_a, we might allow assigning to quantile q_b the same tree A even if it's not optimal, if we ensure that it is within the error bound of the optimum. This would improve interpretability while keeping good quality results.

6.1.4 Better detection of clusters

The current way of detecting clusters in the similarity matrices, using a vector of immediate differences, is rather naive. While it does already provide satisfactory results, it would benefit from more advanced methods.

One possible direction would be using graph theory. Indeed, the similarity matrix could be seen as the adjacency matrix of a graph, and we could use clustering and community detection techniques from graph theory to detect clusters.

6.1.5 Heuristic methods for quantile loss

While DL8.5 provides the optimal result for the training set when optimizing a loss function and has the distinct advantage of being independent of any heuristic, it is quickly limited by time and memory constraints for bigger datasets.

An interesting research direction would be to see if an efficient heuristic could be derived to grow the trees greedily while still obtaining good results on the quantile loss.

While we would expect the results for same depth and minimum support to be slightly worse, having a heuristic would allow going for deeper maximum depths and might give quality results.

6.1.6 Parametric DL8.5

We saw in Section 5.3.3 that for a complicated dataset like the H&M Group dataset described in Section 5.1.2, CaDET provides good results, even when limited to the same depth and number of trees than Quantile DL8.5.

It might be interesting to reuse the concepts of sufficient statistics and parametric estimations in the context of optimal decision trees to see how that influences the quality and efficiency of the final models.

6.2 Conclusion

In this thesis, we proposed a novel way to use the DL8.5 learning algorithm in the context of quantile regression.

In doing this, we were brought to first enable DL8.5 to be used in regression and to optimize quantile loss. Going further, we created a new version of the DL8.5 algorithm that enabled it to learn many decision trees at once, each optimizing for a different quantile.

Using the different trees created by this new algorithm, we provided a mathematically guided way to estimate full, smooth, conditional density functions for the samples of a test set. Using these theoretical contributions, we put them in code and implemented them straight in the C++ source code of the `pydl8.5` package. With this implementation in place, we set out to show that the created algorithm really did match the evaluation criteria we set at the beginning of this thesis.

Concerning the efficiency, we have shown that the parallel learning of the trees provided a major speedup in the training time for many trees, almost reaching in some cases the optimal speedup we could have expected.

Concerning the quality of the results, we have shown that Quantile DL8.5 was competitive with the other methods existing, even being the best one for some of the metrics and datasets considered.

Finally, concerning the interpretability of the Quantile DL8.5 model, we have shown by similarity metrics between the trees, that the algorithm indeed learned as we would have expected, the trees being similar when the corresponding quantiles were close.

Further in that direction, we provided two different ways to interpret the results of the trees, aggregating them differently. The first one, namely density estimation, aggregates for a sample, the predictions of all the trees in a smooth pdf estimation that is easy for a human analyst to interpret and that allows him/her to see how for this sample, the target variable will behave. The second one, namely quantile cluster detection, bases itself on the tree similarity matrices and allows detecting ranges of quantiles in which the trees are similar, aggregating this time on the samples and not the trees, and allowing the analyst to identify what regions of the quantile range are the most important and enabling him/her to analyze only a few trees to get an idea about the whole distribution.

Some points, however, remain to be improved. On the purely computer-science aspect, some optimizations may be made in the code base to further improve performance. If the end goal is only to predict smooth distributions, the corresponding metrics could be directly included in the algorithm. Further research could go into generating trees that are more similar, whether they be optimal or within a fixed bound of optimum. The cluster detection could be bettered using more advanced techniques, possibly from the graph theory domain. And taking inspiration from the competing methods, heuristic and parametric versions of Quantile DL8.5 might provide interesting results.

APPENDIX A

Additional results

Original feature	Corresponding binary features
SIZE_CODE	SC_1 to SC_5
SELLING_WEEK	SW_1 to SW_5
CALENDAR_WEEK	CW_1 to CW_5
SALES_PCS_1W	SP1W_1 to SP1W_5
SALES_PCS_1W_LAG_1	SP1WL1_1 to SP1WL1_5
SALES_PCS_1W_LAG_2	SP1WL2_1 to SP1WL2_5
TOTAL_STOCK	TS_1 to TS_5
TOTAL_STOCK_LAG_1	TSL1_1 to TSL1_5
TOTAL_STOCK_LAG_2	TSL2_1 to TSL2_5
BALANCING_WAREHOUSE	BW

Table A.1: H&M Group dataset feature correspondence

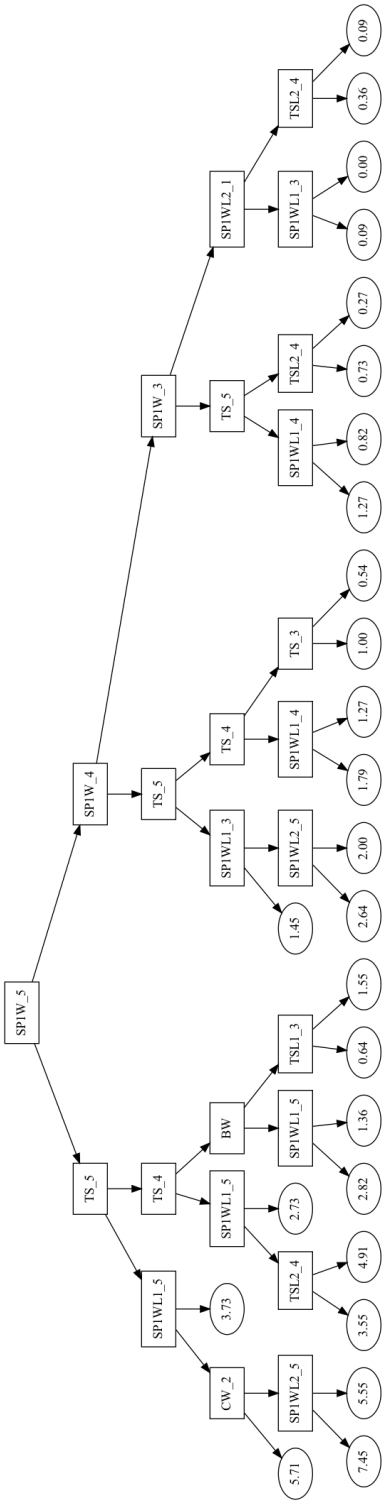


Figure A.1: Decision tree for quantile $q = 0.19$

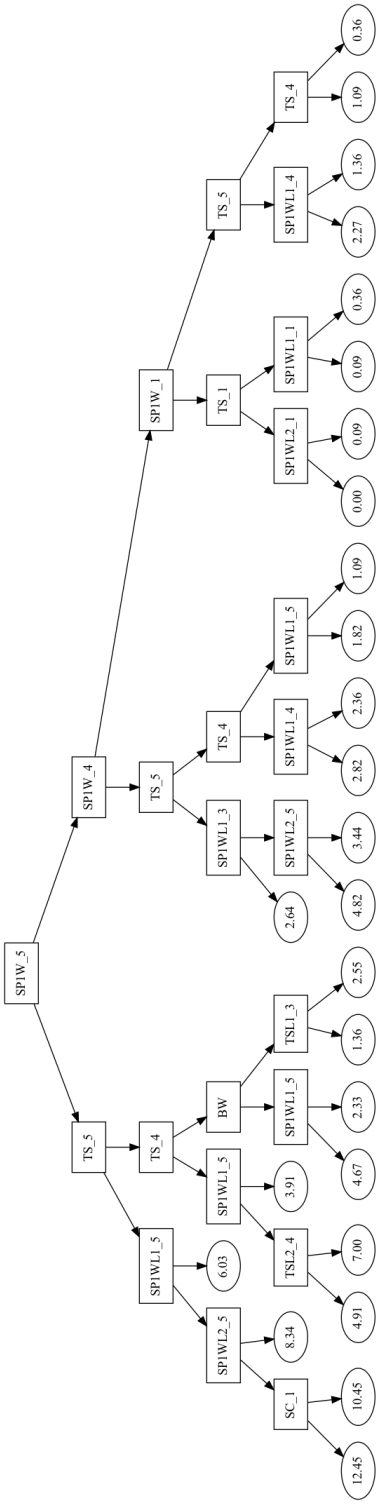


Figure A.2: Decision tree for quantile $q = 0.42$

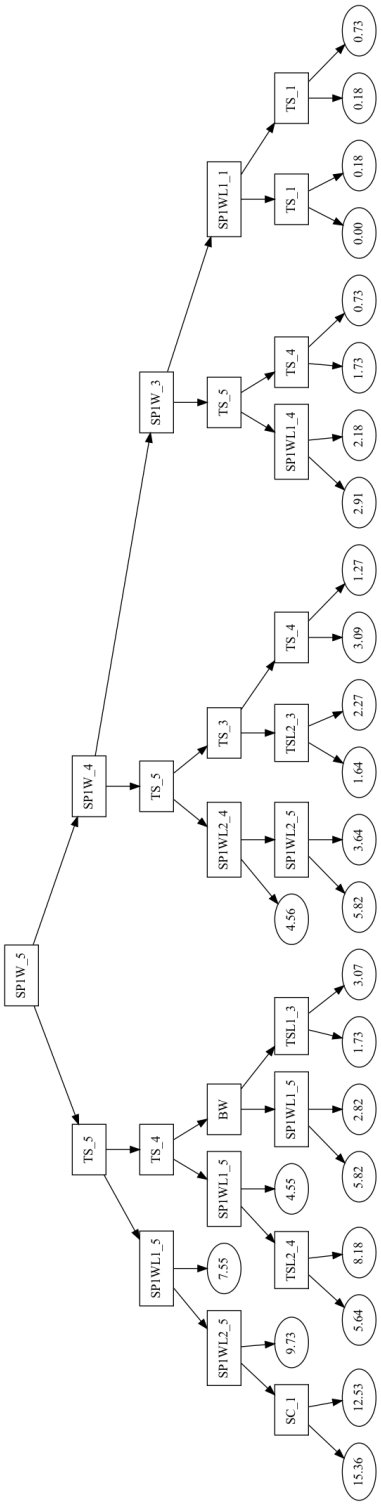


Figure A.3: Decision tree for quantile $q = 0.52$

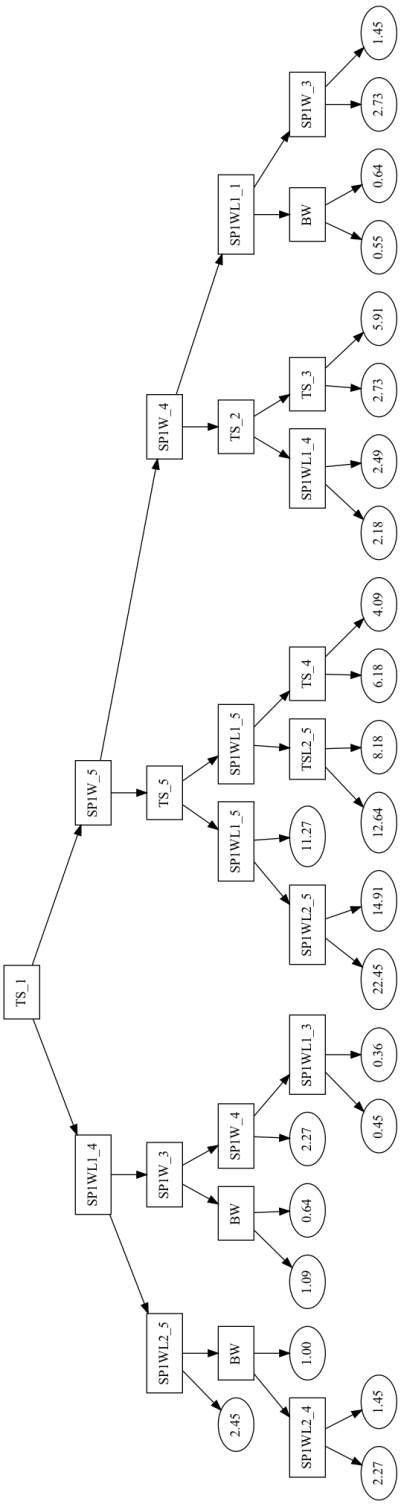


Figure A.4: Decision tree for quantile $q = 0.76$

Bibliography

[1] Lingxin Hao, Daniel Q Naiman, and Daniel Q Naiman. *Quantile regression*. Number 149 in Quantitative Applications in the Social Sciences. Sage, 2007.

[2] 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*, volume 34, pages 3146–3153, 2020.

[3] Kamil A Grajski, Leo Breiman, Gonzalo Viana Di Prisco, and Walter J Freeman. Classification of eeg spatial patterns with a tree-structured methodology: Cart. *IEEE transactions on biomedical engineering*, 12:1076–1086, 1986.

[4] Siegfried Nijssen and Elisa Fromont. Mining optimal decision trees from itemset lattices. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 530–539, 2007.

[5] Emir Demirović, Anna Lukina, Emmanuel Hebrard, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J Stuckey. Murtree: Optimal decision trees via dynamic programming and search. *Journal of Machine Learning Research*, 23(26):1–47, 2022.

[6] David Mease, Abraham J Wyner, and Andreas Buja. Boosted classification trees and class probability/quantile estimation. *Journal of Machine Learning Research*, 8(3), 2007.

[7] Nicolai Meinshausen and Greg Ridgeway. Quantile regression forests. *Journal of Machine Learning Research*, 7(6), 2006.

[8] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[9] Shaomin Wang, Shouxiang Wang, and Dan Wang. Combined probability density model for medium term load forecasting based on quantile regression and kernel density estimation. *Energy Procedia*, 158:6446–6451, 2019.

[10] Roger Koenker and Gilbert Bassett Jr. Regression quantiles. *Econometrica: journal of the Econometric Society*, pages 33–50, 1978.

[11] Shu Zhang, Yi Wang, Yutian Zhang, Dan Wang, and Ning Zhang. Load probability density forecasting by transforming and combining quantile forecasts. *Applied energy*, 277:115600, 2020.

[12] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[13] Cyrus Cousins and Matteo Riondato. Cadet: interpretable parametric conditional density estimation with decision trees and forests. *Machine Learning*, 108(8):1613–1634, 2019.

[14] (aia-uclouvain/pydl8.5) An algorithm for learning optimal decision trees, with Python interface. `https://github.com/aia-uclouvain/pydl8.5`. Accessed: 2022-06-03.

[15] Rob J Hyndman and Yanan Fan. Sample quantiles in statistical packages. *The American Statistician*, 50(4):361–365, 1996.

[16] (NumPy API Reference) numpy.quantile. `https://numpy.org/doc/stable/reference/generated/numpy.quantile.html`. Accessed: 2022-04-11.

[17] Travis Gagie, Simon J Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *International Symposium on String Processing and Information Retrieval*, pages 1–6. Springer, 2009.

[18] James P McDermott, G Jogesh Babu, John C Liechty, and Dennis KJ Lin. Data skeletons: simultaneous estimation of multiple quantiles for massive streaming datasets with applications to density estimation. *Statistics and Computing*, 17(4):311–321, 2007.

[19] George R. Terrell and David W. Scott. Variable kernel density estimation. *The Annals of Statistics*, 20(3):1236–1265, 1992.

[20] (SciPy v1.4.1 Reference Guide) scipy.stats.gaussian_kde. https://docs.scipy.org/doc//scipy-1.4.1/reference/generated/scipy.stats.gaussian_kde.html. Accessed: 2022-05-19.

[21] (Azure Virtual Machines | Microsoft Docs) Memory optimized Dv2 and Dsv2-series VMs. https://docs.microsoft.com/en-us/azure/virtual-machines/dv2-dsv2-series-memory. Accessed: 2022-05-21.

[22] (UCI Machine Learning Repository) Air Quality Data Set. https://archive.ics.uci.edu/ml/datasets/air+quality#. Accessed: 2022-06-01.

[23] (API Reference - Scikit-Garden) skgarden.quantile.RandomForestQuantileRegressor. https://scikit-garden.github.io/api. Accessed: 2022-05-21.

[24] (CADET) Interpretable Parametric Conditional Density Estimation with Decision Trees and Forests. http://cs.brown.edu/~ccousins/projects/cadet/home.html. Accessed: 2022-05-21.

[25] (PyPI) memory-profiler. https://pypi.org/project/memory-profiler. Accessed: 2022-05-21.

[26] Gilbert Saporta and Genane Youness. Comparing two partitions: Some proposals and experiments. In *Compstat*, pages 243–248. Springer, 2002.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**