Estimation of conditional CDFs using machine learning

Estimation of conditional CDFs using machine learning

By

J. Rang

A thesis submitted in partial fulfilment
of the requirements for the degree
of Master of Science

Student number:	4603567
Project duration:	October 20, 2022 – October 24, 2023
Thesis committee:	Prof. dr. ir. G. Jongbloed, TU Delft (Responsible supervisor)
	Dr. A. F. F. Derumigny, TU Delft (Daily supervisor)
	Dr. ir. G. F. Nane, TU Delft

This thesis is to be defended publicly on Tuesday October 24, 2023 at 10:00 AM

Contents

Summary

This paper presents a novel approach for the estimation of conditional multivariate cumulative distribution functions (CDFs) within a nonparametric framework. To achieve this, we introduce a binary random variable that indirectly represents conditional CDFs and construct a dataset by pairing input vectors with the binary variables. We developed a general approach compatible with various machine learning methods.

We have also developed an R package that facilitates the application of machine learning methods. This package leverages a range of machine learning models, including decision trees, neural networks, random forests, and bagging neural networks. Through systematic learning of the intricate relationships between the covariates and the binary variables, we effectively estimate conditional CDFs.

To enhance the accuracy and reliability of the estimated CDFs, we incorporate a rearrangement technique which transforms the estimated functions into monotonic representations, aligning them more closely with the target CDFs and mitigating potential inconsistencies [6].

Through simulations, we evaluate the performance of the estimation approach under various scenarios and assess the impact of sample size and correlation on estimation accuracy, using Mean Integrated Squared Error as a key performance metric. The results demonstrate the effectiveness and robustness of the methodology in estimating conditional CDFs, providing a valuable tool for capturing complex dependencies in multivariate data, with potential applications in risk assessment, finance, and environmental modeling.

Acknowledgements

This thesis concludes my time as a master's student in Applied Mathematics at Delft University of Technology. This moment also marks the end of my university education, which has been a tremendously enjoyable chapter in my life. My adventure in Delft began in 2016 when I first pursued my bachelor's degree in Applied Mathematics. During this period, my love for mathematics further deepened and I discovered my fondness for its applications in finance. I found completing the courses in the Financial Engineering specialization highly enjoyable and gained valuable experience from the data analyst internships I completed. When Alexis presented the opportunity to work on this project in the fall of 2023, I was immediately excited to start.

I would like to take this opportunity to extend my heartfelt gratitude to Alexis for his unwavering guidance and support throughout the entire project. Your dedication and the substantial time you invested in working with me are genuinely appreciated. Our combined effort in unravelling the problem was both enlightening and enjoyable, and I have thoroughly enjoyed our collaboration. I would also like to express my sincere appreciation to the other two esteemed members of the thesis committee, Prof. Dr. Ir. G. Jongbloed and Dr. Ir. G. F. Nane, for their valuable time spent reviewing my report and attending the presentation.

Furthermore, I want to acknowledge the role played by DelftBlue, the supercomputer, in running the simulations for my research. The computational power provided by DelftBlue was instrumental in achieving the results presented in this thesis.

Finally, I want to thank my family and friends for their support and love of throughout this academic journey. Your encouragement and belief in me have provided me with the motivation and strength to overcome challenges and reach this milestone.

Jugo Rang
The Hague, October 2023

List of symbols

Random variables and related quantities

$\mathbf{Y} = (Y_1, Y_2, \ldots, Y_p) \in \mathbb{R}^p$, a vector of response variables.

$\mathbf{X} = (X_1, X_2, \ldots, X_m) \in \mathbb{R}^m$, a vector of corresponding observations of covariates.

$\mathbf{Y}_i = (Y_{i,1}, \ldots, Y_{i,p})$, the i-th observation of $\mathbf{Y} = (Y_1, Y_2, \ldots, Y_p)$.

$\mathbf{X}_i = (X_{i,1}, \ldots, X_{i,m})$, the i-th observation of $\mathbf{X} = (X_1, X_2, \ldots, X_m)$.

$\mathcal{D} = \big((\mathbf{X}_1, \mathbf{Y}_1), \ldots, (\mathbf{X}_n, \mathbf{Y}_n)\big) \in \mathbb{R}^{n(m+p)}$, a dataset of independent and identically distributed observations $\big((\mathbf{X}_1, \mathbf{Y}_1), \ldots, (\mathbf{X}_n, \mathbf{Y}_n)\big)$ of $(\mathbf{X}, \mathbf{Y})$.

$F_{\mathbf{Y}|\mathbf{X}}$, the conditional multivariate cumulative distribution function (CDF) of the response vector $\mathbf{Y}$ given the covariate vector $\mathbf{X}$.

$\widehat{F}_{\mathbf{Y}|\mathbf{X}}$, an estimator of the conditional multivariate CDF $F_{\mathbf{Y}|\mathbf{X}}$ of $\mathbf{Y}$ given $\mathbf{X}$.

$\mathbf{y} = (y_1, y_2, \ldots, y_p) \in \mathbb{R}^p$, a vector of threshold values.

$\mathbf{x} = (x_1, x_2, \ldots, x_m) \in \mathbb{R}^m$, a vector of values of interest of the covariates.

$W_{\mathbf{y}} = \mathbf{1}\{\mathbf{Y} \leq \mathbf{y}\}$, a binary random variable based on the indicator function of $\mathbf{Y} \leq \mathbf{y}$.

$W_{\mathbf{y}}^i = \mathbf{1}\{\mathbf{Y}_i \leq \mathbf{y}\}$, the i-th binary random variable based on the i-th observation $\mathbf{Y}_i$ and the vector of threshold values $\mathbf{y}$.

$\mathcal{D}_{W,\mathbf{y}} = \big((\mathbf{X}_1, W_{\mathbf{y}}^1), \ldots, (\mathbf{X}_n, W_{\mathbf{y}}^n)\big) \in \mathbb{R}^m \times \mathbb{R}^p$, a dataset of independent and identically distributed observations $\big((\mathbf{X}_1, W_{\mathbf{y}}^1), \ldots, (\mathbf{X}_n, W_{\mathbf{y}}^n)\big)$ of $(\mathbf{X}, W_{\mathbf{y}})$.

Models

"Tree", the model where a decision tree is fitted.
"NN", the model where a neural network is fitted.
"RF", the model where a random forest is fitted.
"NNForest", the model where a bagging ensemble of neural networks is fitted.

Hyperparameters

`mindev`, the minimum improvement in deviance required for a node to split in a decision tree.
`minsize` the minimum number of samples required in a leaf node in a decision tree.
`maxiter`, the maximum number of training iterations or epochs in a neural network.
`n_neurons`, the number of neurons in one layer of a neural network.
`n_bootstraps`, the number of bootstrap samples for a random forest or bagging neural network.
`pctObsBootstrap`, the percentage of data sampled from the original dataset when creating bootstrap samples for a random forest or bagging neural network.

1

Introduction

Conditional CDFs play a fundamental role in statistical modelling, providing a probabilistic description of the relationship between a response variable and a set of covariates allowing for predictive modelling, risk assessment, and decision-making under uncertainty. In particular, the estimation of conditional CDFs has garnered significant attention in various fields such as finance, economics, environmental sciences, and engineering. For instance, in financial markets, accurate estimation of conditional CDFs is vital for portfolio optimization and risk management [5]. In environmental sciences, estimating the conditional CDFs of weather variables assists in predicting extreme events and designing resilient infrastructure [14]. Furthermore, in engineering applications, such as structural reliability analysis, the estimation of conditional CDFs enables the assessment of failure probabilities under different conditions [23]. In this paper, we aim to develop a new method for estimating conditional multivariate cumulative distribution functions (CDFs) in a nonparametric framework.

Much of the earliest work on this topic focuses on the estimation of univariate conditional CDFs. In the univariate case, the modeling of conditional CDF is closely related to the modeling of the (conditional) quantile, often called *quantile regression*. This concept has been known since the late 1970 [16] and provides a valuable alternative to traditional least squares regression that only estimates the conditional mean. By estimating the quantiles at various levels, the full conditional distribution can be characterized.

Building upon this, [10] proposed a kernel-based method called *Local Quantile Regression* for estimating conditional CDFs. Their paper introduced the concept of local quantile regression, which estimates the quantiles of the conditional distribution function. By estimating multiple quantiles, the full conditional CDF can be obtained.

However, estimating conditional multivariate CDFs poses several new challenges. First, as the dimensionality of data increases, the curse of dimensionality arises [9].

1

Higher data dimensionality leads to increased sparsity, posing challenges in precisely estimating and capturing the underlying distribution patterns, thereby hindering reliable estimation. Additionally, the computational complexity and data requirements grow exponentially with dimensionality, making the estimation process computationally demanding. Besides the challenges posed by the curse of dimensionality, estimating conditional multivariate CDFs also encounters difficulties in capturing nonlinear dependencies as the relationships between variables cannot be adequately described by linear models. Accounting for nonlinear dependencies often requires employing advanced modelling techniques, such as kernel methods, nonparametric regression, or machine learning algorithms, to capture and represent complex relationships accurately.

In recent years, multiple approaches have been researched for estimating conditional multivariate CDFs. Building upon the foundation of Rosenblatt [21], Genest and Favre proposed a methodology for estimating conditional CDFs based on copula functions to model the dependence structure between variables [12]. By estimating the conditional copula function, one can obtain the conditional multivariate CDF.

Another approach to estimating conditional CDFs presented by [4] uses Distributional Random Forests (DRF). This method models the conditional distribution of the response variables as a function of the predictors. DRF goes beyond standard point estimates of the random forest algorithm [3] and aims to estimate the entire distribution of the response variables. Instead of predicting a single value, each decision tree in the DRF ensemble generates predictions that represent samples from the conditional distribution. By combining the predictions from all the decision trees, DRF obtains an estimate of the conditional distribution.

In the context of estimating conditional multivariate distributions, we aim to estimate the conditional CDFs $F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ of a response vector $\mathbf{Y} = (Y_1, Y_2, \ldots, Y_p) \in \mathbb{R}^p$ given corresponding observation of covariates $\mathbf{X} = (X_1, X_2, \ldots, X_m) \in \mathbb{R}^m$ without imposing specific parametric assumptions on the underlying distribution. For this purpose, we assume that we have a dataset

$$\mathcal{D} = \begin{bmatrix} X_{1,1} & \cdots & X_{1,m} & Y_{1,1} & \cdots & Y_{1,p} \\ X_{2,1} & \cdots & X_{2,m} & Y_{2,1} & \cdots & Y_{2,p} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ X_{n,1} & \cdots & X_{n,m} & Y_{n,1} & \cdots & Y_{n,p} \end{bmatrix} \tag{1.1}$$

of independent and identically distributed observations $\big((\mathbf{X}_1, \mathbf{Y}_1), \ldots, (\mathbf{X}_n, \mathbf{Y}_n)\big)$ of $(\mathbf{X}, \mathbf{Y})$. For example, $\mathbf{X}_1 = (X_{1,1}, \ldots, X_{1,m})$ is the first observation of $\mathbf{X} = (X_1, X_2, \ldots, X_m)$. In this thesis, we focus on the case $m = p = 2$ even though the methods we propose are valid for arbitrary positive integers m and p. In this bivariate case, we define the conditional multivariate CDF

$$F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) := \mathbb{P}\big(Y_1 \leq y_1, Y_2 \leq y_2 | X_1 = x_1, X_2 = x_2\big). \tag{1.2}$$

for any $x_1, x_2, y_1, y_2 \in \mathbb{R}$, where $\mathbf{x} = (x_1, x_2)$ is the value of interest of the covariates and $\mathbf{y} = (y_1, y_2)$ are called the threshold values. Equivalently, we can remark that

$F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ is the expected value of the binary variable $\mathbf{1}\{Y_1 \leq y_1, Y_2 \leq y_2\}$ given $X_1 = x_1$ and $X_2 = x_2$, meaning that

$$F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) = E\big[\mathbf{1}\{Y_1 \leq y_1, Y_2 \leq y_2\}|X_1 = x_1, X_2 = x_2\big] \tag{1.3}$$

In order to estimate Equation (1.3) we propose a nonparametric approach based on machine learning techniques. First, for a fixed value $\mathbf{y} \in \mathbb{R}^2$, we introduce a new binary random variable $W_{\mathbf{y}} := \mathbf{1}\{\mathbf{Y} \leq \mathbf{y}\}$ and create a new dataset, $\mathcal{D}_{W,\mathbf{y}} = \big((\mathbf{X}_1, W_{\mathbf{y}}^1), \ldots, (\mathbf{X}_n, W_{\mathbf{y}}^n)\big)$, by combining the covariates $\mathbf{X}_1, \ldots, \mathbf{X}_n$ with the vector of binary random variables $(W_{\mathbf{y}}^1, \ldots, W_{\mathbf{y}}^n)$, where $W_{\mathbf{y}}^i := \mathbf{1}\{\mathbf{Y}_i \leq \mathbf{y}\}$. By training a machine learning classifier on this new dataset, we can learn the relationships between the covariates $\mathbf{X}$ and the binary random variables $W_{\mathbf{y}}$ for a fixed $\mathbf{y}$, which indirectly represent the conditional CDFs.

While the conditional CDF is inherently an increasing function in $\mathbf{y}$, it is important to note that the estimator of the conditional CDF that we obtained in this way might not be increasing in $\mathbf{y}$ itself. To address this, we therefore use the rearrangement method (see for example the textbook [13]) to construct a monotonic function. This method systematically rearranges the values of a function in a way that preserves the monotonicity and mass of the function by transforming a function to its quantile function. The resulting rearranged monotonic function exhibits a closer alignment with the target function [6].

The contributions of this thesis are twofold. First, we propose a nonparametric approach for estimating conditional multivariate CDFs using machine learning techniques. By avoiding specific parametric assumptions, we provide a method with greater flexibility and adaptability to various data scenarios. Second, we introduce a meta-algorithm that outlines the estimation process for conditional CDFs. The meta-algorithm allows for the use of different machine learning models to train classifiers. Therefore other machine learning techniques not mentioned in this paper can be applied as well. Additionally, to facilitate practical implementation, we have developed an R package named `estimCondCDF` enabling statisticians to readily apply the meta-algorithm in their work.

The structure of this thesis is organized as follows: in Chapter 2, we provide an overview of the fundamental theory underlying the various machine learning methods, along with the rearrangement technique. We do this by first explaining the concept of classification tasks. We then introduce the four machine learning techniques we will use, namely, decision trees, neural networks, random forests and bootstrap aggregation using a neural network as a classifier. Lastly, we elaborate on the concept of rearrangement using examples for enhanced understanding. In Chapter 3, we present the new approach to estimating conditional multivariate CDFs. We start by introducing the general framework and the meta-algorithm model, which uses the machine learning techniques discussed in Chapter 2 as classifiers for the estimation process. Subsequently, we delve into the foundational computer science concepts underpinning this approach. Following that, we examine the inner workings of the constructed R package, addressing any computational challenges that

arose and explaining the design choices made. We provide a detailed examination of the modeling and prediction aspects of the R package and conclude by providing instructions on the practical application of the methodology using the R package. Chapter 4 presents the results of the simulation study, focusing on two main objectives. First, we determine the optimal hyperparameters under the predefined reference setting. Then, we demonstrate the effectiveness of the introduced estimation method in accurately predicting the conditional CDF for various types of datasets. These datasets cover different degrees of dependency between predictor variables $\mathbf{X}$, varying distributions of response variables $\mathbf{Y}$, and datasets of various sizes. To assess the performance of the model, we utilize the Mean Integrated Squared Error (MISE) as the primary evaluation metric and consider the average computation time. Finally, in Chapter 5, we summarize the results and draw conclusions based on the outcomes of the research.

2

Preliminaries

In this chapter, we establish the groundwork for introducing the new estimating method for conditional CDFs. We begin by introducing the concept of supervised machine learning, with a particular focus on binary classification tasks. We then give a detailed explanation of how the various machine learning techniques we apply as a classifier work. Lastly, we elaborate on the concept of rearrangement using examples for enhanced understanding. This will provide the necessary background knowledge needed to understand the estimation method introduced in Chapter 3.

2.1. Classification tasks

Supervised machine learning is a form of machine learning where a program is taught to predict or classify new examples using labelled data. The algorithm is given a dataset comprised of labelled input-output pairs in order to find a function that maps the input values to their respective labelled outputs. This function can then be used to make predictions or classifications for new, unseen examples. An example of this progress is illustrated in Figure 2.1.

One common task in supervised machine learning is binary classification, where the goal is to find a function $f : \mathcal{X} \to \{0, 1\}$ which maps the inputs $\mathbf{X} \in \mathcal{X}$ to a corresponding binary class label $Y \in \{0, 1\}$. The program is given a training dataset

$$\mathcal{D}_Y = \big((\mathbf{X}_1, Y_1), (\mathbf{X}_2, Y_2), \ldots, (\mathbf{X}_n, Y_n)\big),$$

consisting of input vectors $\mathbf{X}_i \in \mathcal{X}$ and output values $Y_i \in \{0, 1\}$ where,

$$\mathbf{X}_i = \begin{bmatrix} X_{i,1} \\ X_{i,2} \\ \vdots \\ X_{i,m} \end{bmatrix}$$

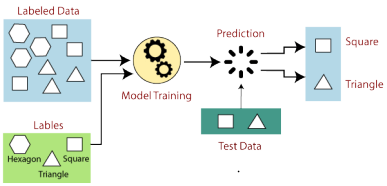

Figure 2.1: Supervised machine learning.

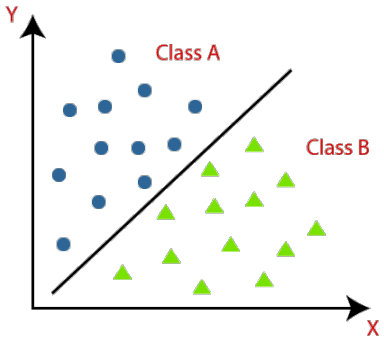

Figure 2.2: Binary classification.

is the i-th input feature vector of dimension m. We adopt these notations commonly used in classification techniques to illustrate this specific case within the general framework where $p = 1$.

A classification task that aims to assign each input observation to more than two possible classes is called a multi-class classification problem. In this case, the output consists of more than two possible values while the input remains the same. However, the function $f(\mathbf{X})$ needs to be modified in order to accommodate multiple output values. For the problem at hand, we are only interested in binary classification problems.

The concept of classification tasks was first introduced by [11]. It proposed a method for classifying objects into different classes based on multiple measurements or features. Statistical methods are used to analyze the relationships between the measurements and the classes to find a linear combination of the measurements that maximally separates these classes. An example of this is shown in Figure 2.2.

The binary classification problem can be solved using various methods. In this paper, we focus on decision trees, neural networks, and two bootstrap aggregating methods, namely, random forests and bagging neural networks. The following sections will discuss these approaches in detail.

2.2. Decision trees

Decision trees are a non-parametric method for supervised machine learning, particularly for binary classification problems. They are based on a hierarchical structure that recursively partitions the feature space into subsets that are increasingly homogeneous with respect to the response variable.

The visual structure of decision trees makes them effective for understanding complex problems as it allows for easy interpretation of the decision-making process. Furthermore, the non-parametric nature of the algorithm allows it to handle complex, non-linear relationships in the data. Lastly, the robustness to outliers and scalability make them a dependable and efficient tool for large and complex datasets.

We aim to generate a decision tree $T = \{\mathcal{N}, \mathcal{L}\}$ that maps any input to a binary

output. Using the formulation of the binary classification problem, each decision node $\texttt{node} \in \mathcal{N}$ corresponds to a feature $j_{\texttt{node}} \in \{1, \ldots, m\}$ and a threshold value $t_{\texttt{node}}$. Furthermore, each leaf node $l \in \mathcal{L}$ corresponds to a binary output value $y_l \in \{0, 1\}$, which represents the classification decision.

In order to create a decision tree we let $S \subset \mathcal{D}_Y$ be the subset of data points that are assigned to a specific node `node` in the decision tree. The goal is to find the best split for the current subset S. At each decision node, we partition the subset S of data points as $S = S_{\text{left}} \cup S_{\text{right}}$, where $S_{\text{left}} := \{(\mathbf{X}_i, Y_i) \in S : X_{i,j_{\texttt{node}}} \leq t_{\texttt{node}}\}$ and $S_{\text{right}} := \{(\mathbf{X}_i, Y_i) \in S : X_{i,j_{\texttt{node}}} > t_{\texttt{node}}\}$.

Here $t_{\texttt{node}}$ is chosen such that the impurity of the resulting partitions is minimised. This impurity is a measure of the non-homogeneity of the partition S_{left}, S_{right} of S. Before introducing the impurity formula, we define the Gini impurity of the set S as:

$$\text{Gini}(S) := 1 - (p_1^2 + p_0^2). \tag{2.1}$$

Here p_1 represents the proportion of points labelled as 1 within the set S and p_0 the proportion labelled as 0 within the same set S. The impurity of the partition $\{S_{\text{left}}, S_{\text{right}}\}$ is then calculated by

$$\text{Impurity} = \frac{\text{Card}(S_{left})}{\text{Card}(S)} \cdot \text{Gini}(S_{left}) + \frac{\text{Card}(S_{right})}{\text{Card}(S)} \cdot \text{Gini}(S_{right}), \tag{2.2}$$

where $\text{Card}(S)$ denotes the total number of points in the set S. This process is recursively repeated for each subset S until one of the stopping criteria is met.

To construct a decision tree model, these stopping criteria are essential input parameters for the function responsible for generating the decision tree. These criteria are commonly referred to as the hyperparameters of the model. The two hyperparameters dictating the stopping criteria in the decision tree model are the minimum deviance of the tree, denoted as `mindev`, and the minimum number of samples required in a leaf node, referred to as `minsize`.

The `mindev` parameter specifies the minimum improvement in the deviance required for a node to split in a decision tree. The deviance is a measure of the goodness-of-fit of the model, and nodes with deviance improvements below the `mindev` threshold will not be split.

The `minsize` parameter sets the minimum number of observations required in a leaf node of the decision tree. Leaves with fewer observations than the `minsize` threshold will not be split further and will become terminal nodes. Together these two hyperparameters play an essential role in shaping the structure and complexity of the decision tree. In R we generate a decision tree model using the function `tree` from the package `tree`.

The decision tree model obtained from this process is used for classification by traversing the decision tree from the root node to a leaf node. This leaf node corresponds to a particular value of the binary label y_l.

There are however drawbacks to classification using decision trees. One of the main drawbacks of decision trees is their tendency to overfit the data if they are

Algorithm 1: Decision tree algorithm.

Input : Training set $\mathcal{D}_Y$, `mindev`, `minsize`
Output: Decision tree $T = \{\mathcal{N}, \mathcal{L}\}$

Function `CreateTree`($\mathcal{D}_Y$, *mindev*, *minsize*)**:**
  if *MINIMUM DEVIANCE* $>$ *mindev* ***OR***
    NUMBER OF SAMPLES IN LEAF NODE $>$ *minsize*
  then
    Compute $y_l = \mathbf{1}\{\text{Card}(\{i \in S : Y_i = 0\}) < \text{Card}(\{i \in S : Y_i = 1\})\}$
    return *leaf node* $\{y_l\}$
  end
  for *all features* $j = 1, ..., m$ **do**
    for *all* $t \in \mathbb{R}$ **do**
      Determine the partition $\{S_{\text{left}}, S_{\text{right}}\}$
      Calculate Impurity using (2.2)
    end
  end
  Choose split $j_{\texttt{node}}, t_{\texttt{node}}$ with minimum impurity ;
  Partition S into subsets S_{left} and S_{right} ;
  Create subtrees T_1 := `CreateTree`(S_{left}) and T_2 := `CreateTree`(S_{right});
  return $(j_{node}, t_{node}, T_1, T_2)$

grown too deep and complex. To address this problem we can prune the decision tree by deleting any branches that do not improve the accuracy of the model. This however can be time-consuming and can lead to underfitting, if not done correctly. To mediate this we will generate a multitude of decision tree models with different node sizes, `minsize`, and within-node deviances, `mindev`.

Another limitation of decision trees is their inability to handle missing data. Decision trees require complete data for each observation in order to make a prediction. Therefore if a feature has missing data, the algorithm may either exclude the observation or impute the missing value using mean imputation or regression imputation which can lead to bias and reduce the accuracy of the model.

Lastly, decision trees are sensitive to minor changes in the data. A small change in the input data can produce an entirely different decision tree, making the model less stable and more difficult to interpret. This issue can be mitigated by implementing ensemble methods, such as bagging, which combines multiple decision trees to reduce model variability. This will be discussed in Section 2.4.

2.3. Neural networks

Neural networks are computational models inspired by the structure and function of the human brain. The first computational model was introduced in 1943 [17] when Warren McCulloch and Walter Pitts published their mathematical model of a neuron. Later Frank Rosenblatt [20] worked on developing the perceptron, an

early type of neural network capable of learning simple linear patterns. The first multilayer neural network was published in 1965 by [15]. A significant advancement occurred in 1986, when [22] introduced the backpropagation algorithm, allowing for efficient training of multilayer neural networks.

A neural network consists of a large number of interconnected processing units called neurons, which are organized into layers. The neurons in each layer receive inputs from the neurons in the previous layer, perform a computation on these inputs and then pass the result to the neurons in the next layer. A feedforward neural network is a specific type of neural network in which the neurons are organized into layers that are connected such that the information flows from the input layer to the output layer without any loops or feedback connections. The aforementioned backpropagation algorithm is an iterative optimization algorithm used in training neural networks, which calculates and adjusts the parameter gradients of the network in a backward pass to minimize the error between predicted and actual output, enabling effective learning and adaptation. The advantages of a feedforward neural network are that it is simple to train using backpropagation and is computationally efficient making it suitable for large-scale computations. We will therefore use a feedforward neural network.

We will first examine a neural network with an input vector $\mathbf{X} = (X_1, \ldots, X_m)$ of m features, a single neuron, and a single output value. To get a prediction from a neural network we use two steps. First, we compute

$$z = \sum_{i=1}^{m} X_i \cdot w_i + b = \mathbf{X} \cdot \mathbf{w} + b \tag{2.3}$$

where $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ is the weight vector needed in order to obtain the weighted sum of inputs, and b is an offset. The weights determine the strength of the connection between neurons and dictate the magnitude of the influence of each input on the output of the neuron, with larger weights resulting in greater influence. We then find the final prediction of this single neuron by applying the sigmoid activation function to z. This introduces non-linearity into the output of the neurons. The sigmoid activation function is defined as:

$$sigmoid(z) = \frac{1}{1 + e^{-z}}. \tag{2.4}$$

This function is useful for binary classification problems as it maps any real-valued number to a value between 0 and 1. Another benefit of the sigmoid activation function is that the function is differentiable, meaning backpropagation algorithms can be used to train the neural network.

A neural network can thus be represented as a function f_σ that maps any input vector $\mathbf{X}$ to a corresponding output $\widehat{Y}$, where σ represents the set of learnable parameters in the network. An example of a neural network consisting of one layer with 4 neurons is visualised in Figure 2.3.

In Figure 2.3 each neuron $f_\sigma^i(\mathbf{X})$ is trained as described in the single neuron single output case, using the sigmoid activation function described in Equation 2.4.

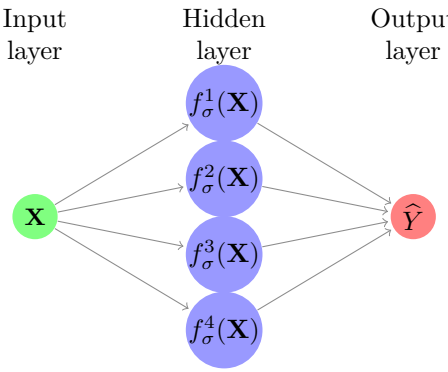

Figure 2.3: Single layer neural network with 4 neurons.

This means each neuron $f_\sigma^i(\mathbf{X})$ in the layer provides one prediction. To form a single prediction $\widehat{Y}$, the outputs of each $f_\sigma^i(\mathbf{X})$ are combined using a weighted sum method of the neuron outputs, in this case:

$$\widehat{Y} = \sum_{i=1}^{4} w_i \cdot f_\sigma^i(\mathbf{X}). \tag{2.5}$$

When training a neural network model, the network is presented with pairs of data instances from a dataset denoted as $\mathcal{D}_Y$. It then calculates the sigmoid activation function 2.4 for each neuron $f_\sigma^i(\mathbf{X})$ in the hidden layer. Then a weighted sum method is used in Equation 2.5 to calculate the prediction $\widehat{Y}$. The weights of the neural networks are continuously updated using the backpropagation algorithm. The key component of this algorithm is to minimize the binary cross-entropy loss function, which measures the difference between the predicted probabilities, $\widehat{Y}$, of the two classes and the true class labels, Y:

$$Loss(Y, \widehat{Y}) = -Y \log(\widehat{Y}) - (1 - Y) \log(1 - \widehat{Y}). \tag{2.6}$$

Then the gradient of the loss function with respect to the weights in the neural network is calculated, enabling the optimization of the weights through the use of an optimization algorithm. The backpropagation algorithm performs the following steps:

1. Forward pass: Calculate $f_\sigma(\mathbf{X})$ using Equations (2.3) and (2.4)
2. Calculation of error: Calculate $Loss(Y, \widehat{Y})$ using Equation (2.6)
3. Backward pass: Backpropagate the error through the network from the output layer to the input layer, and calculate the gradient of the loss with respect to each weight.

4. Weight update: Update the weights using the BFGS optimization algorithm [18]. The optimization algorithm adjusts the weights in the direction of the negative gradient of the loss, with the learning rate determining the size of the weight update.

We establish two stopping criteria for the model, firstly the hyperparameter, `maxiter`, which represents the maximum number of epochs that the model will be trained for. Secondly, we specify the hyperparameter `n_neurons` to indicate the number of neurons in the one-layered neural network. In R we generate a neural network using the function `nnet` from the package `nnet`.

The algorithm of the neural network is displayed in Algorithm 2.

Algorithm 2: Neural network algorithm.

Input: Training set $\mathcal{D}_Y$, `maxiter`, `n_neurons`
Output: $\widehat{Y}$
Function `CreateNN(` $\mathcal{D}_Y$, *`maxiter`*, *`n_neurons`*`):`
  Initialize weights σ ;
  for $k \leftarrow 1$ *to* *`maxiter`* **do**
    for $i \leftarrow 1$ *to* *`n_neurons`* **do**
      Calculate $f_\sigma^i(\mathbf{X})$ using Equation (2.3) and (2.4) ;
    end
    Calculate $\widehat{Y} = \sum_{j=1}^{\texttt{n\_neurons}} w_j \cdot f_\sigma^j(\mathbf{X})$;
    Calculate the error $Loss(Y, \widehat{Y})$ using Equation(2.6) ;
    Calculate the gradient of $Loss(Y, \widehat{Y})$ using backpropagation ;
    Update σ using a BFGS optimization algorithm ;
  end
  return $\widehat{Y}$.

2.4. Bootstrap aggregation

Bootstrap aggregation (bagging) is an ensemble learning method that combines the predictions of multiple models trained on different subsets of the training data to improve the accuracy and stability of the overall prediction. In binary classification problems, bagging can be used to improve classification accuracy by reducing the variance of the prediction given by the model. The concept of bagging was initially introduced by Breiman in 1996 [2], using it within the context of decision trees.

The basic idea of bagging is to create a set of training data subsets by randomly sampling the original training data with replacement. Each subset is then used to train a base classifier. This is typically a decision tree but other classifiers such as neural networks can also be used for this. The predictions of all the individual classifiers trained on each subset are combined to obtain the final prediction. The key advantage of bagging is that it can reduce the prediction variance by averaging over the predictions of multiple classifiers trained on different subsets of the data, which is particularly effective for unstable models.

The bagging algorithm for binary classification problems can be described as follows. We randomly sample (with replacement) the set $\mathcal{D}_Y$ to create B bootstrap samples $\mathcal{D}_Y^1, \mathcal{D}_Y^2, \ldots, \mathcal{D}_Y^B$. For each bootstrap sample $\mathcal{D}_Y^i$, we train a binary classifier $f_i(\mathbf{X})$ which is used to make predictions. Lastly, we use a majority vote to gain the final prediction,

$$f(\mathbf{X}) := \mathbf{1}\left\{\frac{1}{B}\sum_{i=1}^{B} f_i(\mathbf{X}) > \frac{1}{2}\right\}. \tag{2.7}$$

2.4.1. Random forests

A particular extension of the bagging algorithm for decision trees is called random forest. Instead of using a single decision tree as the classifier, we use an ensemble of decision trees. We first create B bootstrap samples from the dataset $\mathcal{D}_Y$. Then a new decision tree is generated for each bootstrap sample, creating an ensemble of decision trees.

After a random forest model is constructed, consisting of an ensemble of decision trees, a prediction for a new input vector $\mathbf{x} = (x_1, \ldots, x_m)$ can be made. By traversing each decision tree in the ensemble similar to the decision tree model, each decision tree votes for the predicted label of $\mathbf{x}$. The final prediction of a random forest model is made by applying the majority vote scheme and can be written as:

$$f(\mathbf{x}) = \mathbf{1}\left\{\frac{1}{B}\sum_{i=1}^{B} T_i(\mathbf{x}) > \frac{1}{2}\right\}, \tag{2.8}$$

where $T_i(\mathbf{x})$ is the prediction of the i-th decision tree in the forest for the new input $\mathbf{x}$, and `n_bootstraps` is the total number of trees in the forest.

The random forest algorithm takes in four hyperparameters. Because we need to generate decision trees we need to specify the hyperparameters for the decision tree algorithm, `mindev` and `minsize`. Besides these, we also need to specify the hyperparameters, `pctObsBootstrap`, indicating the percentage of data sampled from the original dataset $\mathcal{D}_Y$ and `n_bootstraps` (called B before), indicating how many bootstrap samples should be taken. The number of bootstraps, `n_bootstraps`, thus directly influences the total count of decision trees in the ensemble as each bootstrap sample serves as the basis for constructing an individual decision tree.

Since each decision tree in the ensemble is constructed from a different random sample of the original dataset, the number of bootstraps, `n_bootstraps`, also determines how many decision trees will be generated. The algorithm of the random forest is displayed in Algorithm 3.

Algorithm 3: Random forest algorithm.

Input : Training set $\mathcal{D}_Y$, `mindev`, `minsize`, `n_bootstraps`, `pctObsBootstrap`
Output: Random forest $F = \{T_1, T_2, \ldots, T_{\texttt{n\_bootstraps}}\}$

Function `RandomForest`($\mathcal{D}_Y$, *mindev, minsize, n_bootstraps, pctObsBootstrap*)**:**
  for $b \leftarrow 1$ **to** *n_bootstraps* **do**
    Generate a bootstrap sample $\mathcal{D}_Y^b$ from $\mathcal{D}_Y$, containing a specified percentage `pctObsBootstrap` of the total data in $\mathcal{D}_Y$.;
    Train decision tree T_b = `CreateTree`($\mathcal{D}_Y^b$, *mindev, minsize*) using Algorithm 1;
  end
  return $\{T_1, T_2, \ldots, T_{n\_bootstraps}\}$

2.4.2. Bagging neural network

There are multiple types of bagging algorithms besides random forests. As we have already discussed the workings of a neural network we can construct the bagging algorithm using a neural network as a classifier. This will work in a similar fashion to the random forest ensemble technique but instead of using a decision tree as a classifier, we use a neural network. We will call this method where a bagging ensemble of neural networks is used the bagging neural network method.

Each neural network is trained on a bootstrap sample, consisting of a percentage of the dataset $\mathcal{D}$. The final prediction is again determined by a majority vote scheme. We need to specify four hyperparameters. These are the hyperparameters of the neural network, `maxiter` and `n_neurons`, as we need to generate a neural network for each bootstrap. We also need to specify, just as in the random forest algorithm, Algorithm 3, the hyperparameter `pctObsBootstrap`, indicating the percentage of data sampled from the original dataset, and `n_bootstraps`, the number of neural networks we want to generate. The algorithm of the bagging neural network is displayed in Algorithm 4.

Algorithm 4: Bagging neural network algorithm.

Input : Training set $\mathcal{D}_Y$, `maxiter`, `n_neurons`,`n_bootstraps`, `pctObsBootstrap`

Output: Bagging neural network $\mathbf{NN}_{Bagging} = \{\mathbf{NN}_1, \mathbf{NN}_2, \ldots, \mathbf{NN}_{\texttt{n\_bootstraps}}\}$

Function `BaggingNN`($\mathcal{D}_Y$, *maxiter*, *n_neurons*, *n_bootstraps*, *pctObsBootstrap*)**:**

  for $b \leftarrow 1$ **to** *n_bootstraps* **do**

    Generate a bootstrap sample $\mathcal{D}_Y^b$ from $\mathcal{D}_Y$, containing a specified percentage `pctObsBootstrap` of the total data in $\mathcal{D}_Y$. ;

    Train neural network $\mathbf{NN}_b$ = `CreateNN`($\mathcal{D}_Y^b$, *maxiter*, *n_neurons*) using Algorithm 2.;

  end

  return $\{NN_1, NN_2, \ldots, NN_{n\_bootstraps}\}$

2.5. Rearrangement

We aim to estimate conditional CDFs which are inherently increasing functions. However, the estimator of the conditional CDF that we obtain this way might not be increasing itself. To address this, we therefore use the rearrangement method (see for example the textbook [13]) to construct a monotonic function that is closer aligned with the target conditional CDF function [6]. The naive idea of rearranging a function to make it monotonic is to simply swap the points of a function such that the function appears to be monotonic. This, however, is not a valid approach because it only changes the visual appearance of the function, but does not preserve its mathematical properties. This approach can result in a function that violates basic mathematical properties such as continuity and differentiability. We will therefore use a well-studied method of rearrangement which enjoys nice mathematical properties.

In the general case of this rearrangement method, we assume a target function $f_0 : \mathbb{R}^d \to \mathbb{R}$ to be monotonic, i.e. weakly increasing, and the existence of an initial, not necessarily monotonic, estimate $\hat{f}$. It was shown in [13] that rearrangement methods could be used to convert $\hat{f}$ into a monotonic estimate $\hat{f}^*$. As [6] demonstrated, this rearranged estimate $\hat{f}^*$ is invariably closer to the target function f_0 in standard metrics. In order to more accurately estimate conditional CDFs we thus want to use rearrangement. The following sections elaborate on rearrangement in the univariate and multivariate cases

2.5.1. Univariate rearrangement

First, we examine univariate rearrangement. We consider a univariate function $f : \mathrm{D} \mapsto \mathrm{R}$ where D is a compact interval on $[0, 1]$ and R is a bounded subset of $\mathbb{R}$. The rearrangement of a continuous function f can be defined as a sorting operation. Specifically, we can obtain the rearranged function f^* by sorting the

values of f evaluated at a fine enough net of equidistant points in increasing order. This sorting process results in a rearranged function f^* that represents the quantile function of the random variable $f(X)$, where $X \sim \mathcal{U}(0,1)$. We can define this quantile function as:

$$f^*(X) = \inf\left[k \in \mathbb{R} : \int_{\mathrm{D}} \mathbf{1}\{f(u) \leq k\} du \geq X\right] \tag{2.9}$$

To illustrate the concept of rearrangement in the context of a univariate function f, we now present an example.

Example 2.1. *Assume the univariate function*

$$f(x) = \begin{cases} 4x, & \text{if } x \in [0, \frac{1}{2}], \\ 3 - 2x, & \text{if } x \in [\frac{1}{2}, 1], \end{cases} \tag{2.10}$$

such that $x \in [0,1]$. This function is shown in Figure 2.4. We can obtain the rearranged function f^* by calculating the quantile function in Equation (2.9). First, we will evaluate the indicator function $\mathbf{1}\{f(x) \leq k\}$ for both cases.

$$\mathbf{1}\{f(x) \leq k\} = \begin{cases} \mathbf{1}\{4x \leq k\} \text{ for } x \in [0, \frac{1}{2}] \\ \mathbf{1}\{3 - 2x \leq k\} \text{ for } x \in [\frac{1}{2}, 1] \end{cases} = \begin{cases} \mathbf{1}\{x \leq \frac{k}{4}\} \text{ for } x \in [0, \frac{1}{2}] \\ \mathbf{1}\{x \geq \frac{3-k}{2}\} \text{ for } x \in [\frac{1}{2}, 1] \end{cases}. \tag{2.11}$$

We can thus rewrite the integral in Equation 2.9 as:

$$\int_{\mathrm{D}} \mathbf{1}\{f(u) \leq k\} du = \int_0^{\frac{1}{2}} \mathbf{1}\{u \leq \frac{k}{4}\} du + \int_{\frac{1}{2}}^1 \mathbf{1}\{u \geq \frac{3-k}{2}\} du. \tag{2.12}$$

We now assess all cases for the integral $\int_0^{\frac{1}{2}} \mathbf{1}\{u \leq \frac{k}{4}\} du$.
Case 1: $k \leq 0$
In this case, $\mathbf{1}\{u \leq \frac{k}{4}\} = 0$ since $u \in [0, \frac{1}{2}]$ and thus we have

$$\int_0^{\frac{1}{2}} \mathbf{1}\{u \leq \frac{k}{4}\} du = \int_0^{\frac{1}{2}} 0 du = 0. \tag{2.13}$$

Case 2: $0 < k < 2$
In this case, $\mathbf{1}\{u \leq \frac{k}{4}\} = 1$ for $u \in [0, \frac{k}{4}]$ and $\mathbf{1}\{u \leq \frac{k}{4}\} = 0$ for $u \in [\frac{k}{4}, \frac{1}{2}]$ thus we have

$$\int_0^{\frac{1}{2}} \mathbf{1}\{u \leq \frac{k}{4}\} du = \int_0^{\frac{k}{4}} 1 du + \int_{\frac{k}{4}}^{\frac{1}{2}} 0 du = \frac{k}{4}. \tag{2.14}$$

Case 3: $k \geq 2$
In this case, $\mathbf{1}\{u \leq \frac{k}{4}\} = 1$ since $u \in [0, \frac{1}{2}]$ and thus we have

$$\int_0^{\frac{1}{2}} \mathbf{1}\{u \leq \frac{k}{4}\} du = \int_0^{\frac{1}{2}} 1 du = \frac{1}{2}. \tag{2.15}$$

We can summarise these 3 cases by

$$\int_0^{\frac{1}{2}} \mathbf{1}\{u \leq \frac{k}{4}\}du = \begin{cases} 0 \text{ If } k \leq 0 \\ \frac{k}{4} \text{ If } 0 < k < 2 \\ \frac{1}{2} \text{ If } k \geq 2 \end{cases} . \tag{2.16}$$

We also asses all cases for the integral $\int_{\frac{1}{2}}^1 \mathbf{1}\{u \geq \frac{3-k}{2}\}du$.
Case 1: $k \leq 1$
In this case, $\frac{3-k}{2} \geq 1$ *thus* $\mathbf{1}\{u \leq \frac{3-k}{2}\} = 0$ *since* $u \in [\frac{1}{2}, 1]$. *We get that*

$$\int_{\frac{1}{2}}^1 \mathbf{1}\{u \geq \frac{3-k}{2}\}du = \int_{\frac{1}{2}}^1 0du = 0. \tag{2.17}$$

Case 2: $1 < k < 2$
In this case, $\frac{1}{2} < \frac{3-k}{2} < 1$ *thus* $\mathbf{1}\{u \leq \frac{3-k}{2}\} = 0$ *for* $u \in [\frac{1}{2}, \frac{3-k}{2}]$ *and* $\mathbf{1}\{u \leq \frac{3-k}{2}\} = 1$ *for* $u \in [\frac{3-k}{2}, 1]$. *We get*

$$\int_{\frac{1}{2}}^1 \mathbf{1}\{u \geq \frac{3-k}{2}\}du = \int_{\frac{1}{2}}^{\frac{3-k}{2}} 0du + \int_{\frac{3-k}{2}}^1 1du = 1 - \frac{3-k}{2} = \frac{k-1}{2}. \tag{2.18}$$

Case 3: $k \geq 2$
In this case, $\frac{3-k}{2} \leq \frac{1}{2}$ *thus* $\mathbf{1}\{u \leq \frac{3-k}{2}\} = 1$ *since* $u \in [\frac{1}{2}, 1]$. *We thus get*

$$\int_{\frac{1}{2}}^1 \mathbf{1}\{u \geq \frac{3-k}{2}\}du = \int_{\frac{1}{2}}^1 1du = \frac{1}{2} \tag{2.19}$$

We can summarise these 3 cases by

$$\int_{\frac{1}{2}}^1 \mathbf{1}\{u \geq \frac{3-k}{2}\}du = \begin{cases} 0 \text{ If } k \leq 1 \\ \frac{k-1}{2} \text{ If } 1 < k < 2 \\ \frac{1}{2} \text{ If } k \geq 2 \end{cases} . \tag{2.20}$$

Combining Equations (2.16) and (2.20) into Equation (2.12) we get

$$\int_{\mathrm{D}} \mathbf{1}\{f(u) \leq k\}du = \begin{cases} 0 \text{ If } k \leq 0 \\ \frac{k}{4} \text{ If } 0 < k \leq 1 \\ \frac{k}{4} + \frac{k-1}{2} \text{ If } 1 < k < 2 \\ \frac{1}{2} + \frac{1}{2} \text{ If } k \geq 2 \end{cases} \tag{2.21}$$

As a consequence, solving $\frac{k}{4} = x$ *gives* $k = 4x$.

Solving $\frac{k}{4} + \frac{k-1}{2} = x$ *gives* $\frac{3k}{4} - \frac{1}{2} = x$, *and therefore* $k = \frac{4}{3}\left(x + \frac{1}{2}\right) = \frac{4}{3}x + \frac{2}{3}$.
Therefore

$$f^*(x) = \begin{cases} 4x \text{ if } 0 \leq x \leq \frac{1}{4} \\ \frac{4}{3}x + \frac{2}{3} \text{ if } \frac{1}{4} < x \leq 1. \end{cases} \tag{2.22}$$

We confirm this by running:

```
Rearrangement::rearrangement(x = list(c(0, 1/2, 1)),
                             y = c(0, 2, 1))
```

in R, which indeed gives $f^*(0) = 0$, $f^*(1/2) = 4/3 \approx 1.332$ *and* $f^*(1) = 2$.

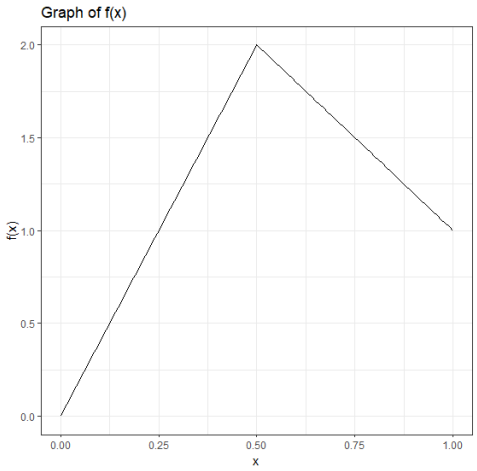

Figure 2.4: Visual representation of f

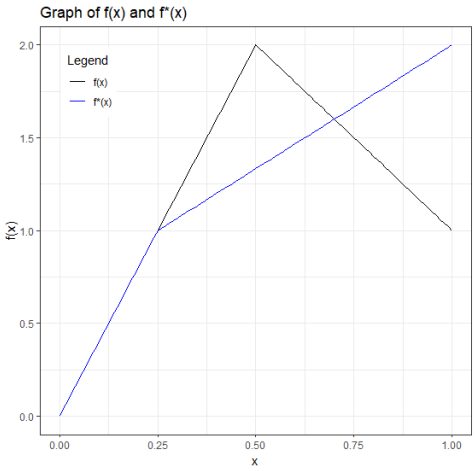

Figure 2.5: Visual representation of f and f^*.

As illustrated in Figure 2.5, the proposed method of rearranging a function results in a new function f^* with the same area or mass under its graph as the original function f. Intuitively we can see this as shifting the mass of the function f such that it becomes the monotonic function f^*. Therefore the integral in Equation (2.9) is important as it preserves the mass. We show that the areas underneath the graph are equal and thus the mass is preserved:

$$\int_0^1 f(x)dx = \int_0^{\frac{1}{2}} 4x\ dx + \int_{\frac{1}{2}}^1 3 - 2x\ dx = \frac{1}{2} + \frac{3}{4} = \frac{5}{4} \tag{2.23}$$

and

$$\int_0^1 f^*(x)dx = \int_0^{\frac{1}{4}} 4x\ dx + \int_{\frac{1}{4}}^1 \frac{4}{3}x + \frac{2}{3}\ dx = \frac{1}{8} + \frac{9}{8} = \frac{5}{4}. \tag{2.24}$$

2.5.2. Multivariate rearrangement

For multivariate rearrangement, we consider the multivariate function $f : \mathrm{D}^d \mapsto R$, where $\mathrm{D}^d = [0,1]^d$ and R is a bounded subset of $\mathbb{R}$. The monotonicity that we require from the function f is as follows: f is weakly increasing in the vector $\mathbf{V} \in \mathrm{D}$ if $f(\mathbf{V}') \leq f(\mathbf{V})$ whenever $\mathbf{V}' \geq \mathbf{V}$ pointwise. We will define $f(\mathrm{v}_j, \mathbf{V}_{-j})$ as the dependence of f on one of its arguments, v_j, while holding all other arguments, $\mathbf{V}_{-j}$, fixed. The definition of monotonicity given for f is equivalent to requiring that $\forall j \in \{1, \ldots, d\}$ the mapping $\mathrm{v}_j \mapsto f(\mathrm{v}_j, \mathbf{V}_{-j})$ is weakly increasing in $\mathbf{V}_{-j}$ for

all $\mathbf{V}_{-j} \in \mathrm{D}^{d-1}$. We now define the function f^* to be the rearranged function w.r.t. v_j as

$$f_j^*(\mathbf{V}) = R_j f(\mathbf{V}) = \inf\Big[k : \int_{\mathrm{D}} \mathbf{1}\{f(\mathrm{v}_j', \mathbf{V}_{-j}) \leq k\} d\mathrm{v}_j' \geq \mathrm{v}_j\Big], \tag{2.25}$$

where we define R_j to be the rearrangement operator. Here we have applied a one-dimensional increasing rearrangement to the one-dimensional function $\mathrm{v}_j \mapsto f(\mathrm{v}_j, \mathbf{V}_{-j})$ while holding all other arguments $\mathbf{V}_{-j}$ fixed. R_j is then recursively applied to every value $\mathrm{v} \in \mathbf{V}_{-j}$.

Assume an ordering $\sigma = (\sigma_1, \ldots, \sigma_d)$ of integers $1, \ldots, d$ and define R_σ to be the σ-rearrangement operator. Furthermore, define $f_\sigma^* = R_\sigma f = R_{\sigma_1} \cdots R_{\sigma_d} f$ as the σ-rearranged function. It is, however, possible for any two non-identical orderings σ and τ on the same set of integers $1, \ldots, d$ to have different rearranged functions f_σ and f_τ. In order to mitigate this problem we want to take the average over all orderings given by

$$f^* = \frac{1}{|\mathcal{S}|} \sum_{\sigma \in \mathcal{S}} f_\sigma^*, \tag{2.26}$$

where we assume $\mathcal{S}$ as any finite collection of orderings σ and $|\mathcal{S}|$ the number of elements in the ordering set $\mathcal{S}$

In practice, we can apply the rearrangement of a d-dimensional multivariate function by first only focusing on rearranging a function over only one axis, holding all other $d-1$ axes. We then apply this process for each axis to get one of the possible orderings. Since we do not want to rely on just one way of rearranging, the goal is to average over all possible orderings. We thus repeat the process but change the order in which we rearrange each axis for all d possible rearrangements. We then average these rearrangements to find f^*. If the domain of the multivariate function is outside $[0, 1]$ we can always normalize it such that it will be inside this interval. To further illustrate the concept of rearrangement in the context of a multivariate function f, we now present an example.

Example 2.2. *Assume the multivariate function $f : [0,1]^2 \mapsto \mathbb{R}$ defined by*

$$f(\mathbf{V}) = f(\begin{bmatrix} \mathrm{v}_1 \\ \mathrm{v}_2 \end{bmatrix}) = (v_1 - \frac{1}{2})(v_2 - \frac{1}{2}) \tag{2.27}$$

We first choose to rearrange w.r.t v_1 and hold v_2. We evaluate the indicator function in Equation (2.25)

$$\begin{aligned}
\mathbf{1}\{f(\mathrm{v}_1, \mathbf{V}_{-1}) \leq k\} &= \mathbf{1}\Big\{(\mathrm{v}_1 - \frac{1}{2})(\mathrm{v}_2 - \frac{1}{2}) \leq k\Big\} \\
&= \begin{cases} \mathbf{1}\Big\{\mathrm{v}_1 \geq \frac{k}{\mathrm{v}_2 - \frac{1}{2}} + \frac{1}{2}\Big\} & \text{if } 0 < \mathrm{v}_2 < \frac{1}{2} \\ \mathbf{1}\Big\{\mathrm{v}_1 \leq \frac{k}{\mathrm{v}_2 - \frac{1}{2}} + \frac{1}{2}\Big\} & \text{if } \frac{1}{2} < \mathrm{v}_2 < 1 \end{cases} \\
&= \begin{cases} \mathbf{1}\Big\{\mathrm{v}_1 \geq \frac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1}\Big\} & \text{if } 0 < \mathrm{v}_2 < \frac{1}{2} \\ \mathbf{1}\Big\{\mathrm{v}_1 \leq \frac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1}\Big\} & \text{if } \frac{1}{2} < \mathrm{v}_2 < 1 \end{cases}
\end{aligned}$$

We can thus rewrite the integral as follows:

$$\begin{aligned}&\int_{\mathrm{D}} \mathbf{1}\{f(\mathrm{v}_1', \mathbf{V}_{-1}) \leq k\} d\mathrm{v}_1' \\ &= \begin{cases} \int_0^1 \mathbf{1}\Big\{\mathrm{v}_1' \geq \dfrac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1}\Big\} d\mathrm{v}_1' & \text{if } 0 < \mathrm{v}_2 < \dfrac{1}{2} \\ \int_0^1 \mathbf{1}\Big\{\mathrm{v}_1' \leq \dfrac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1}\Big\} d\mathrm{v}_1' & \text{if } \dfrac{1}{2} < \mathrm{v}_2 < 1 \end{cases}\end{aligned} \tag{2.28}$$

We look at both integrals separately, if $0 < \mathrm{v}_2 < \frac{1}{2}$

$$\begin{aligned}&\int_0^1 \mathbf{1}\Big\{\mathrm{v}_1' \geq \frac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1}\Big\} d\mathrm{v}_1' \\ &= \begin{cases} 1 & \text{if } \dfrac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} \leq 0 \\ \dfrac{-2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} & \text{if } 0 < \dfrac{2k + v_2 - \frac{1}{2}}{2v_2 - 1} < 1 \\ 0 & \text{if } \dfrac{2k + v_2 - \frac{1}{2}}{2v_2 - 1} > 1 \end{cases}\end{aligned} \tag{2.29}$$

Expressing $\frac{2k+\mathrm{v}_2-\frac{1}{2}}{2\mathrm{v}_2-1}$ *in terms of* k *using* $0 < \mathrm{v}_2 < \frac{1}{2}$ *we get*

$$\begin{aligned}&\int_0^1 \mathbf{1}\Big\{\mathrm{v}_1' \geq \frac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1}\Big\} d\mathrm{v}_1' \\ &= \begin{cases} 1 & \text{if } 0 < \mathrm{v}_2 < \dfrac{1}{2},\ k \geq \dfrac{1}{4} - \dfrac{\mathrm{v}_2}{2} \\ \dfrac{-2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} & \text{if } 0 < \mathrm{v}_2 < \dfrac{1}{2},\ \dfrac{\mathrm{v}_2}{2} - \dfrac{1}{4} < k < \dfrac{1}{4} - \dfrac{\mathrm{v}_2}{2} \\ 0 & \text{if } 0 < \mathrm{v}_2 < \dfrac{1}{2},\ k \leq \dfrac{\mathrm{v}_2}{2} - \dfrac{1}{4} \end{cases}\end{aligned} \tag{2.30}$$

Doing the same for the case where $\frac{1}{2} < \mathrm{v}_2 < 1$ *we get*

$$\begin{aligned}&\int_0^1 \mathbf{1}\Big\{\mathrm{v}_1' \leq \frac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1}\Big\} d\mathrm{v}_1' \\ &= \begin{cases} 0 & \text{if } \dfrac{1}{2} < \mathrm{v}_2 < 1,\ \dfrac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} \leq 0 \\ \dfrac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} & \text{if } \dfrac{1}{2} < \mathrm{v}_2 < 1,\ 0 < \dfrac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} < 1 \\ 1 & \text{if } \dfrac{1}{2} < \mathrm{v}_2 < 1,\ \dfrac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} \geq 1 \end{cases} \\ &= \begin{cases} 0 & \text{if } \dfrac{1}{2} < \mathrm{v}_2 < 1,\ k \leq \dfrac{1}{4} - \dfrac{\mathrm{v}_2}{2} \\ \dfrac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} & \text{if } \dfrac{1}{2} < \mathrm{v}_2 < 1,\ \dfrac{1}{4} - \dfrac{\mathrm{v}_2}{2} < k < \dfrac{\mathrm{v}_2}{2} - \dfrac{1}{4} \\ 1 & \text{if } \dfrac{1}{2} < \mathrm{v}_2 < 1,\ k \geq \dfrac{\mathrm{v}_2}{2} - \dfrac{1}{4} \end{cases}\end{aligned} \tag{2.31}$$

Combining Equation (2.30) and (2.31) in Equation (2.28) we get

$$\mathbf{I}(k, \mathrm{v}_1) = \int_{\mathrm{D}} \mathbf{1}\{f(\mathrm{v}_1', \mathbf{V}_{-1}) \leq k\} d\mathrm{v}_1'$$
$$= \begin{cases} 1 & \text{if } 0 < \mathrm{v}_2 < \frac{1}{2},\ k \geq \frac{1}{4} - \frac{\mathrm{v}_2}{2} \\ \frac{-2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} & \text{if } 0 < \mathrm{v}_2 < \frac{1}{2},\ \frac{\mathrm{v}_2}{2} - \frac{1}{4} < k < \frac{1}{4} - \frac{\mathrm{v}_2}{2} \\ 0 & \text{if } 0 < \mathrm{v}_2 < \frac{1}{2},\ k \leq \frac{\mathrm{v}_2}{2} - \frac{1}{4} \\ 0 & \text{if } \frac{1}{2} < \mathrm{v}_2 < 1,\ k \leq \frac{1}{4} - \frac{\mathrm{v}_2}{2} \\ \frac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} & \text{if } \frac{1}{2} < \mathrm{v}_2 < 1,\ \frac{1}{4} - \frac{\mathrm{v}_2}{2} < k < \frac{\mathrm{v}_2}{2} - \frac{1}{4} \\ 1 & \text{if } \frac{1}{2} < \mathrm{v}_2 < 1,\ k \geq \frac{\mathrm{v}_2}{2} - \frac{1}{4} \end{cases} \tag{2.32}$$

Using Equation (2.25) we thus get

$$f_1^*(\mathbf{V}) = R_1 f(\mathbf{V}) = \inf\left[k : \mathbf{I}(k, \mathrm{v}_1) \geq \mathrm{v}_1\right]$$
$$= \begin{cases} \inf\left[\frac{-2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} \geq \mathrm{v}_1\right] & \text{if } 0 < \mathrm{v}_2 < \frac{1}{2} \\ \inf\left[\frac{2k + \mathrm{v}_2 - \frac{1}{2}}{2\mathrm{v}_2 - 1} \geq \mathrm{v}_1\right] & \text{if } \frac{1}{2} < \mathrm{v}_2 < 1 \end{cases}$$
$$= \begin{cases} \inf\left[k : k \geq -\mathrm{v}_1\mathrm{v}_2 + \frac{1}{2}\mathrm{v}_1 + \frac{1}{2}\mathrm{v}_2 + \frac{1}{4}\right] & \text{if } 0 < \mathrm{v}_2 < \frac{1}{2} \\ \inf\left[k : k \geq \mathrm{v}_1\mathrm{v}_2 - \frac{1}{2}\mathrm{v}_1 - \frac{1}{2}\mathrm{v}_2 + \frac{1}{4}\right] & \text{if } \frac{1}{2} < \mathrm{v}_2 < 1 \end{cases} \tag{2.33}$$

Finally, we obtain the explicit expression

$$f_1^*(\mathbf{V}) = \begin{cases} (\frac{1}{2} - \mathrm{v}_1)(\mathrm{v}_2 - \frac{1}{2}) & \text{if } 0 < \mathrm{v}_2 < \frac{1}{2} \\ (\mathrm{v}_1 - \frac{1}{2})(\mathrm{v}_2 - \frac{1}{2}) & \text{if } \frac{1}{2} < \mathrm{v}_2 < 1 \end{cases} \tag{2.34}$$

We now rearrange this newly found function w.r.t v_2*. Again we evaluate the*

indicator function in Equation (2.25)

$$\mathbf{1}\{f_1^*(\mathrm{v}_2, \mathbf{V}_{-2}) \le k\} = \begin{cases} \mathbf{1}\Big\{(\frac{1}{2} - \mathrm{v}_1)(\mathrm{v}_2 - \frac{1}{2}) \le k\Big\} & \text{if } 0 < \mathrm{v}_2 < \frac{1}{2} \\ \mathbf{1}\Big\{(\mathrm{v}_1 - \frac{1}{2})(\mathrm{v}_2 - \frac{1}{2}) \le k\Big\} & \text{if } \frac{1}{2} < \mathrm{v}_2 < 1 \end{cases}$$

$$= \begin{cases} \mathbf{1}\Big\{\mathrm{v}_2 \le \dfrac{k}{\frac{1}{2} - \mathrm{v}_1} + \dfrac{1}{2}\Big\} & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ 0 < \mathrm{v}_2 < \frac{1}{2} \\ \mathbf{1}\Big\{\mathrm{v}_2 \ge \dfrac{k}{\frac{1}{2} - \mathrm{v}_1} + \dfrac{1}{2}\Big\} & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ 0 < \mathrm{v}_2 < \frac{1}{2} \\ \mathbf{1}\Big\{\mathrm{v}_2 \ge \dfrac{k}{\mathrm{v}_1 - \frac{1}{2}} + \dfrac{1}{2}\Big\} & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ \frac{1}{2} < \mathrm{v}_2 < 1 \\ \mathbf{1}\Big\{\mathrm{v}_2 \le \dfrac{k}{\mathrm{v}_1 - \frac{1}{2}} + \dfrac{1}{2}\Big\} & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ 0 < \mathrm{v}_2 < \frac{1}{2} \end{cases} \tag{2.35}$$

We can furthermore rewrite the integral in Equation (2.25) *as*

$$\begin{aligned} &\int_{\mathrm{D}} \mathbf{1}\{f_1^*(\mathrm{v}_2', \mathbf{V}_{-2}) \le k\} d\mathrm{v}_2' \\ &= \int_{\mathrm{v}_2=0}^{\frac{1}{2}} \mathbf{1}\{f_1^*(\mathrm{v}_2', \mathbf{V}_{-2}) \le k\} d\mathrm{v}_2' + \int_{\mathrm{v}_2=\frac{1}{2}}^{1} \mathbf{1}\{f_1^*(\mathrm{v}_2', \mathbf{V}_{-2}) \le k\} d\mathrm{v}_2' \end{aligned} \tag{2.36}$$

We evaluate the integral $\int_{\mathrm{D}} \mathbf{1}\{f_1^*(\mathrm{v}_2', \mathbf{V}_{-2}) \le k\} d\mathrm{v}_2'$ *for each case in Equation* (2.35).

First if $0 < \mathrm{v}_1 < \frac{1}{2}$ *and* $0 < \mathrm{v}_2 < \frac{1}{2}$ *we get*

$$\int_{\mathrm{D}} \mathbf{1}\{f_1^*(\mathrm{v}_2', \mathbf{V}_{-2}) \le k\} d\mathrm{v}_2' = \int_{\mathrm{v}_2=0}^{\frac{1}{2}} \mathbf{1}\Big\{\mathrm{v}_2 \le \frac{k}{\frac{1}{2} - \mathrm{v}_1} + \frac{1}{2}\Big\} d\mathrm{v}_2'$$

$$= \begin{cases} 0, & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ k \le \frac{\mathrm{v}_1}{2} - \frac{1}{4}, \\ \frac{k}{\frac{1}{2} - \mathrm{v}_1} + \frac{1}{2}, & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ \frac{1}{4} - \frac{\mathrm{v}_1}{2} < k < 0, \\ \frac{1}{2}, & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ k \ge 0. \end{cases} \tag{2.37}$$

If $\frac{1}{2} < \mathrm{v}_1 < 1$ *and* $0 < \mathrm{v}_2 < \frac{1}{2}$ *we get*

$$\int_{\mathrm{D}} \mathbf{1}\{f_1^*(\mathrm{v}_2', \mathbf{V}_{-2}) \le k\} d\mathrm{v}_2' = \int_{\mathrm{v}_2=0}^{\frac{1}{2}} \mathbf{1}\Big\{\mathrm{v}_2 \ge \frac{k}{\frac{1}{2} - \mathrm{v}_1} + \frac{1}{2}\Big\} d\mathrm{v}_2'$$

$$= \begin{cases} 0 & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ k \le 0 \\ \frac{-k}{\frac{1}{2} - \mathrm{v}_1} & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ 0 < k < \frac{\mathrm{v}_1}{2} - \frac{1}{4} \\ \frac{1}{2} & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ k \ge \frac{\mathrm{v}_1}{2} - \frac{1}{4} \end{cases} \tag{2.38}$$

If $0 < \mathrm{v}_1 < \frac{1}{2}$ *and* $\frac{1}{2} < \mathrm{v}_2 < 1$

$$\int_{\mathrm{D}} \mathbf{1}\{f_1^*(\mathrm{v}_2', \mathbf{V}_{-2}) \leq k\} d\mathrm{v}_2' = \int_{\mathrm{v}_2=\frac{1}{2}}^{1} \mathbf{1}\Big\{\mathrm{v}_2 \geq \frac{k}{\mathrm{v}_1 - \frac{1}{2}} + \frac{1}{2}\Big\} d\mathrm{v}_2'$$
$$= \begin{cases} 0 & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ k \leq \frac{\mathrm{v}_1}{2} - \frac{1}{4} \\ \frac{1}{2} - \frac{k}{\mathrm{v}_1 - \frac{1}{2}} & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ \frac{\mathrm{v}_1}{2} - \frac{1}{4} < k < 0 \\ \frac{1}{2} & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ k \geq 0 \end{cases} \tag{2.39}$$

And lastly if $\frac{1}{2} < \mathrm{v}_1 < 1$ *and* $\frac{1}{2} < \mathrm{v}_2 < 1$

$$\int_{\mathrm{D}} \mathbf{1}\{f_1^*(\mathrm{v}_2', \mathbf{V}_{-2}) \leq k\} d\mathrm{v}_2' = \int_{\mathrm{v}_2=\frac{1}{2}}^{1} \mathbf{1}\Big\{\mathrm{v}_2 \leq \frac{k}{\frac{1}{2} - \mathrm{v}_1} + \frac{1}{2}\Big\} d\mathrm{v}_2'$$
$$= \begin{cases} 0 & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ k \leq 0 \\ \frac{k}{\mathrm{v}_1 - \frac{1}{2}} & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ 0 < k < \frac{\mathrm{v}_1}{2} - \frac{1}{4} \\ \frac{1}{2} & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ k \geq \frac{\mathrm{v}_1}{2} - \frac{1}{4} \end{cases} \tag{2.40}$$

We can now use the integral in Equation (2.36) and combine Equation (2.37) and (2.39) for the case $0 < \mathrm{v}_1 < \frac{1}{2}$ *and Equation (2.38) and (2.40) for the case* $\frac{1}{2} < \mathrm{v}_1 < 1$*. We then get*

$$\mathbf{I}(k, \mathrm{v}_2) = \int_{\mathrm{D}} \mathbf{1}\{f_1^*(\mathrm{v}_2', \mathbf{V}_{-2}) \leq k\} d\mathrm{v}_2'$$
$$= \begin{cases} 0 & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ k \leq \frac{\mathrm{v}_1}{2} - \frac{1}{4} \\ 1 + \frac{2k}{\frac{1}{2} - \mathrm{v}_1} & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ \frac{\mathrm{v}_1}{2} - \frac{1}{4} < k < 0 \\ 1 & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ k \geq 0 \\ 0 & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ k \leq 0 \\ \frac{2k}{\mathrm{v}_1 - \frac{1}{2}} & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ 0 < k < \frac{\mathrm{v}_1}{2} - \frac{1}{4} \\ 1 & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ k \geq \frac{\mathrm{v}_1}{2} - \frac{1}{4} \end{cases} \tag{2.41}$$

Using Equation (2.25) we thus get

$$f_{1,2}^*(\mathbf{V}) = R_2 f_1^*(\mathbf{V}) = \inf\Big[k : \mathbf{I}(k, \mathrm{v}_2) \geq \mathrm{v}_1\Big]$$
$$= \begin{cases} \inf\Big[1 + \frac{2k}{\frac{1}{2} - \mathrm{v}_1} \geq \mathrm{v}_2\Big] & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2} \\ \inf\Big[\frac{2k}{\mathrm{v}_1 - \frac{1}{2}} \geq \mathrm{v}_2\Big] & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1 \end{cases} \tag{2.42}$$
$$= \begin{cases} \inf\Big[k \geq \frac{1}{2}(\mathrm{v}_2 - 1)(\frac{1}{2} - \mathrm{v}_1)\Big] & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2} \\ \inf\Big[k \geq \frac{\mathrm{v}_2}{2}(\mathrm{v}_1 - \frac{1}{2})\Big] & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1 \end{cases}$$

Therefore, we obtain

$$f_{1,2}^*(\mathbf{V}) = \begin{cases} \frac{1}{2}(\mathrm{v}_2 - 1)(\frac{1}{2} - \mathrm{v}_1) & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2} \\ \frac{\mathrm{v}_2}{2}(\mathrm{v}_1 - \frac{1}{2}) & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1 \end{cases} \tag{2.43}$$

However, this is the result if we first rearrange w.r.t v_1 *and then rearrange w.r.t* v_2*. We will get a different result if we first rearrange w.r.t* v_2 *and then w.r.t* v_1*. We, therefore, determine the function for this rearrangement as well and take the average over the two orderings as described in Equation (2.26). Due to the symmetry of the function in Equation (2.27), we know that rearranging w.r.t* v_2 *and then* v_1 *will give us the function*

$$f^*_{2,1}(\mathbf{V}) = \begin{cases} \frac{1}{2}(\mathrm{v}_1 - 1)(\frac{1}{2} - \mathrm{v}_2) & \text{if } 0 < \mathrm{v}_2 < \frac{1}{2} \\ \frac{\mathrm{v}_1}{2}(\mathrm{v}_2 - \frac{1}{2}) & \text{if } \frac{1}{2} < \mathrm{v}_2 < 1 \end{cases} \tag{2.44}$$

Combining Equation (2.43) and (2.44) in Equation (2.26) we thus get

$$\begin{aligned} f^* &= \frac{1}{|\mathcal{S}|}\sum_{\sigma\in\mathcal{S}} f^*_\sigma = \frac{1}{2}\Big(f^*_{1,2}(\mathbf{V}) + f^*_{2,1}(\mathbf{V})\Big) \\ &= \begin{cases} \frac{1}{8}(-4\mathrm{v}_1\mathrm{v}_2 + 3\mathrm{v}_1 + 3\mathrm{v}_2 - 2) & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ 0 < \mathrm{v}_2 < \frac{1}{2} \\ \frac{1}{8}(\mathrm{v}_1 + \mathrm{v}_2 - 1) & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ 0 < \mathrm{v}_2 < \frac{1}{2} \\ \frac{1}{8}(\mathrm{v}_1 + \mathrm{v}_2 - 1) & \text{if } 0 < \mathrm{v}_1 < \frac{1}{2},\ \frac{1}{2} < \mathrm{v}_2 < 1 \\ \frac{1}{8}(4\mathrm{v}_1\mathrm{v}_2 - \mathrm{v}_1 - \mathrm{v}_2) & \text{if } \frac{1}{2} < \mathrm{v}_1 < 1,\ \frac{1}{2} < \mathrm{v}_2 < 1 \end{cases} \end{aligned} \tag{2.45}$$

As illustrated in Figures 2.6 to 2.9, we first rearrange the function $f(\mathbf{V})$ *of Equation (2.27) to the function* $f^*_{1,2}(\mathbf{V})$ *of Equation (2.43) conserving the mass under the graph. By symmetry, we obtain the function* $f^*_{2,1}(\mathbf{V})$ *of Equation (2.43). Then by averaging over both rearrangements* $f^*_{1,2}(\mathbf{V})$ *and* $f^*_{2,1}(\mathbf{V})$*, we find the function* $f^*(\mathbf{V})$ *of Equation (2.45), with the same mass under its graph as the original function* $f(\mathbf{V})$*.*

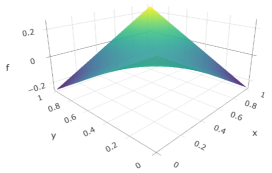

Figure 2.6: The original function $f(\mathbf{V})$ as given by Equation (2.27).

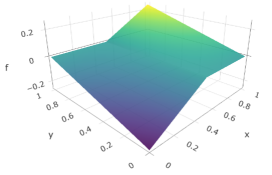

Figure 2.7: The function $f^*_{1,2}(\mathbf{V})$ obtained by rearranging $f(\mathbf{V})$.

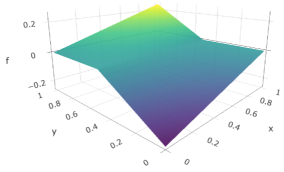

Figure 2.8: The function $f^*_{2,1}(\mathbf{V})$ obtained by rearranging $f(\mathbf{V})$.

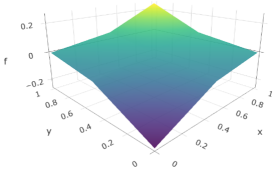

Figure 2.9: The function $f^*(\mathbf{V})$ obtained by averaging over both rearrangements $f^*_{1,2}(\mathbf{V})$ and $f^*_{2,1}(\mathbf{V})$.

3

ML methods for conditional CDFs

3.1. General framework

In this section, we introduce a method of estimating conditional multivariate CDFs using different forms of machine learning and present the different models used. In this setting, we are interested in two random vectors $(\mathbf{X}, \mathbf{Y}) \in \mathbb{R}^m \times \mathbb{R}^p$. We will denote by $\mathbf{X} = (X_1, \ldots, X_m)$ and $\mathbf{Y} = (Y_1, \ldots, Y_p)$. We observe a dataset $\mathcal{D} = \big((\mathbf{X}_1, \mathbf{Y}_1), \ldots, (\mathbf{X}_n, \mathbf{Y}_n)\big)$ of n i.i.d. replications of $(\mathbf{X}, \mathbf{Y})$. We define the conditional multivariate CDF of $\mathbf{Y}$ given $\mathbf{X}$ by

$$F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) := \mathbb{P}(\mathbf{Y} \leq \mathbf{y}|\mathbf{X} = \mathbf{x}), \tag{3.1}$$

for a given vector of threshold values $\mathbf{y} \in \mathbb{R}^p$ and a vector of values of interest $\mathbf{x} \in \mathbb{R}^m$. Observe that the conditional CDF can be expressed as

$$F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) = E\big[\mathbf{1}\{\mathbf{Y} \leq \mathbf{y}\}|\mathbf{X} = \mathbf{x}\big]. \tag{3.2}$$

To estimate this expectation, we introduce the new random variable $W_{\mathbf{y}} = \mathbf{1}\{\mathbf{Y} \leq \mathbf{y}\}$, yielding the equation:

$$F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) = E\big[W_{\mathbf{y}}|\mathbf{X} = \mathbf{x}\big]. \tag{3.3}$$

In order to estimate $F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ we now create a new dataset by combining the vectors $\mathbf{X}$ with corresponding $W_{\mathbf{y}}$ and use this dataset to train the classifiers described in Chapter 2. In this dataset, $\mathbf{X}$ represents the input feature vector and $W_{\mathbf{y}}$ the corresponding binary random variables that serve as the label. This transforms the problem into a binary classification problem. It is important to note that as each new threshold value $\mathbf{y}$ is introduced, a new binary random variable $W_{\mathbf{y}}$ is created. As a result, it is necessary to train a new classifier for each new $\mathbf{y}$. To further illustrate this, we present an example

Example 3.1. *As an example, we can consider the case where* $n = 8$ *such that* $\mathcal{D} = \big((\mathbf{X}_1, \mathbf{Y}_1), \ldots, (\mathbf{X}_8, \mathbf{Y}_8)\big)$ *and* $m = p = 2$ *such that* $\mathbf{X}_i = \begin{bmatrix} X_{i,1} \\ X_{i,2} \end{bmatrix}$ *and* $\mathbf{Y}_i = \begin{bmatrix} Y_{i,1} \\ Y_{i,2} \end{bmatrix}$. *The dataset is illustrated as a matrix in Equation* (3.4)

$$\begin{bmatrix} X_{1,1} & X_{1,2} & Y_{1,1} & Y_{1,2} \\ X_{2,1} & X_{2,2} & Y_{2,1} & Y_{2,2} \\ \vdots & \vdots & \vdots & \vdots \\ X_{8,1} & X_{8,2} & Y_{8,1} & Y_{8,2} \end{bmatrix} \tag{3.4}$$

Since we want to estimate the conditional CDF

$$\begin{aligned} F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) &= \mathbb{P}(\mathbf{Y} \leq \mathbf{y}|\mathbf{X} = \mathbf{x}) \\ &= E\big[\mathbf{1}\{\mathbf{Y} \leq \mathbf{y}\}|\mathbf{X} = \mathbf{x}\big] \\ &= E\big[W_{\mathbf{y}}|\mathbf{X} = \mathbf{x}\big] \end{aligned} \tag{3.5}$$

we need to determine $W_{\mathbf{y}} = \mathbf{1}\{\mathbf{Y} \leq \mathbf{y}\}$ *for a given* $\mathbf{y}$*:*

$$W_{\mathbf{y}}^i = \mathbf{1}\{\mathbf{Y}_i \leq \mathbf{y}\} = \mathbf{1}\left\{ \begin{bmatrix} Y_{i,1} \\ Y_{i,2} \end{bmatrix} \leq \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \right\}. \tag{3.6}$$

This provides the new dataset $\mathcal{D}_{W,\mathbf{y}}$*,*

$$\mathcal{D}_{W,\mathbf{y}} = \begin{bmatrix} X_{1,1} & X_{1,2} & W_{\mathbf{y}}^1 \\ X_{2,1} & X_{2,2} & W_{\mathbf{y}}^2 \\ \vdots & \vdots & \vdots \\ X_{8,1} & X_{8,2} & W_{\mathbf{y}}^8 \end{bmatrix}. \tag{3.7}$$

We then use this dataset to train a classifier to estimate the probability that $W_{\mathbf{y}}$ *is* 0 *or* 1 *given* $\mathbf{X}$*.*

To estimate the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$, we train a classifier using the dataset containing pairs of $\mathbf{X}$ and corresponding $W_{\mathbf{y}}$ values. This classifier aims to capture the relationship between the input vector $\mathbf{x}$ and the binary random variable $W_{\mathbf{y}}$ for a vector of threshold values $\mathbf{y}$.

One important aspect to consider is that each new vector of threshold values $\mathbf{y}$ corresponds to a new binary random variable $W_{\mathbf{y}}$, which necessitates training a separate classifier for each unique value of $\mathbf{y}$. Theoretically, to estimate $F_{\mathbf{Y}|\mathbf{X}}$ accurately, we would need to train an infinite number of classifiers, one for every possible value of $\mathbf{y} \in \mathbb{R}^d$. In practice, it is impossible to train an infinite number of classifiers. However, we can train classifiers for a carefully chosen set of threshold values $\mathbf{y}$. Doing so we obtain predictions that relate $\mathbf{x}$ to $W_{\mathbf{y}}$ for a predefined set of $\mathbf{y}$ values. These predictions cover a grid-like structure, as we have a prediction for every combination of $\mathbf{x} = (x_1, \ldots, x_r)$ and the selected $\mathbf{y} = (y_1, \ldots, y_s)$ values. Note that these predictions are thus available only for the selected values of $\mathbf{y}$ and not

for every possible $\mathbf{y}$. The reason behind the asymmetry between $\mathbf{x}$ and $\mathbf{y}$ lies in the nature of the problem: the relationship between $\mathbf{x}$ and $W_{\mathbf{y}}$ is explicitly learned and modelled, while the relationship between $\mathbf{x}$ and the full range of $\mathbf{Y}$ is not directly captured.

We use interpolation in order to find predictions for $\mathbf{y}$ values that are not on the specified grid. By leveraging the trained classifiers and the available predictions on the grid, we can estimate the conditional CDF $\widehat{F}_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ for $\mathbf{y}$ values that were not part of the selected set.

The estimator is denoted as $\widehat{F}_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ and is computed for all $\mathbf{y}_i$ within the grid $\mathbf{y}_1, \ldots, \mathbf{y}_s$, and all $\mathbf{x}$ within the grid $\mathbf{x}_1, \ldots, \mathbf{x}_r$. The true function $F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ is increasing in $\mathbf{y}$ for every value of $\mathbf{y} \in \mathbb{R}^p$ and $\mathbf{x} \in \mathbb{R}^m$. However, due to the limitations of the training data and the model, this estimator might exhibit an inconsistency as it is possible that for certain $\mathbf{x}$ values, $\widehat{F}_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}_i|\mathbf{x}) < \widehat{F}_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}_j|\mathbf{x})$ even when $\mathbf{y}_i > \mathbf{y}_j$. This discrepancy between the true behaviour of $F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ and the estimator $\widehat{F}_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ is problematic as the inconsistency in the estimated probabilities for different $\mathbf{y}$ values also extends to the interpolated regions. As a result, the estimator fails to accurately capture the increasing nature of the true conditional CDF $F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$, resulting in an invalid representation of the underlying distribution.

To address the issue of inconsistencies in the estimated conditional CDF, we apply rearrangement, as introduced in Section 2.5, to the estimated function. Rearrangement involves transforming the estimated function into a monotonic function, which provides a more accurate and reliable estimate of the underlying conditional CDF.

This approach is motivated by the fact that the inconsistency in the estimated probabilities for different $\mathbf{y}$ values is a common problem in the context of increasing monotone functions. It has been proven, as shown in Chernozuhkov et al. [6], that rearranging the function reduces the estimation error as the rearranged estimate is invariably closer to the target function than the estimate before rearrangement is. However, it is important to note that performing the rearrangement for every value of $\mathbf{x}$ is not feasible in practice due to computational limitations. Therefore, a grid for $\mathbf{x}$ needs to be defined, similar to the grid for $\mathbf{y}$ values. Once the grids for both $\mathbf{x}$ and $\mathbf{y}$ are established, the rearrangement can be applied to the interpolated function. By rearranging the function, we mitigate the inconsistencies and improve the accuracy of the estimated conditional CDF

We present a "meta-algorithm" in Alorithm 5, in which we elaborate on the process described. Since many classifiers can be used for estimating Equation (3.3) this meta-algorithm will not specify a classifier. In practice, we will apply the ML algorithms discussed in Chapter 2 in the meta-algorithm:

- "Tree": this is the model where a decision tree is fitted;
- "NN": this is the model where a neural network is fitted
- "RF": this is the model where a random forest is fitted

- "NNForest": this is the model where a bagging ensemble of neural networks is fitted

Algorithm 5: Meta-algorithm for estimating conditional CDF using ML techniques.

Input: Training dataset $\mathcal{D}$
Input: A grid of threshold values $\mathbf{y}_1, \ldots, \mathbf{y}_s$
Input: A grid of $\mathbf{x}$ values $\mathbf{x}_1, \ldots, \mathbf{x}_r$
Input: ML specific parameters
Output: Estimate $\widehat{F}^*_{\mathbf{Y}|\mathbf{X}}$
for $i \leftarrow 1$ *to* s **do**
  Calculate $W^j_{\mathbf{y}_i} = \mathbf{1}\{\mathbf{Y} \leq \mathbf{y}_i\}$ for every $j = 1, \ldots, n$;
  Create new dataset $\mathcal{D}_{W,\mathbf{y}_i} = \left((\mathbf{X}_1, W^1_{\mathbf{y}_i}), \ldots, (\mathbf{X}_n, W^n_{\mathbf{y}_i})\right)$;
  Train ML algorithm using Algorithm from Chapter 2 using $\mathcal{D}_{W,\mathbf{y}_i}$ and ML specific parameters ;
  for $\mathbf{x}_i$ *on the grid* $\mathbf{x}_1, \ldots, \mathbf{x}_r$ **do**
    Calculate the estimated conditional CDF $\widehat{F}_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}_i|\mathbf{x}_j)$ using ML model ;
  end
end
Rearrange $\widehat{F}_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}_i|\mathbf{x})$ on the grid of $\mathbf{x}_1, \ldots, \mathbf{x}_r$ and $\mathbf{y}_1, \ldots, \mathbf{y}_s$ values as detailed in Chapter 2.5 ;
return $\widehat{F}^*_{\mathbf{Y}|\mathbf{X}}$

3.2. Background on computer science concepts

In this section, we delve into three key programming approaches that underpin the research: meta-programming, functional programming, and object-oriented programming (OOP). Each of these approaches plays a crucial role in the problem-solving methodology, ensuring code functionality, eliminating redundancy, and promoting readability.

3.2.1. Meta-programming

Meta-programming can be defined as the art of programming that revolves around the intricate world of programs themselves. In conventional programming, code execution follows a linear path, adhering to a predefined set of instructions. In contrast, metaprogramming represents a higher-level approach, focusing on the manipulation and generation of code within the programming environment itself. It revolves around the concept of programming with the aim of creating, modifying, or generating code dynamically and then executing this generated code. The key distinction between conventional programming and metaprogramming lies in the source of the executed code: while traditional programming involves the explicit writing of code by the programmer, metaprogramming entails code generation and customization in response to specific requirements and contextual factors. This distinction empowers developers to craft more adaptable and context-aware systems by allowing code to be dynamically generated and executed based on varying conditions and needs.

Meta-programming, furthermore, provides dynamic control over code execution during runtime. This capability offers a wide range of possibilities, enabling the development of functions and scripts for code generation, enhancement, and analysis, particularly beneficial for tasks like automation, code abstraction, and dynamic code manipulation. Notably, meta-programming excels in abstracting recurring patterns into reusable functions, promoting code abstraction and streamlining data manipulation across diverse datasets. Furthermore, it automates code generation based on specific criteria or data, reducing manual coding efforts and enhancing efficiency while minimizing errors. Moreover, the dynamic code manipulation feature of meta-programming empowers real-time modification of code and objects, enhancing adaptability and flexibility, making it a valuable tool for code quality improvement.

We primarily employ meta-programming for code optimization, dynamically generating more efficient and cleaner code while considering data characteristics. This approach enhances code performance and readability, emphasizing the primary focus on improving code efficiency and quality.

Meta-programming relies on three core components: expression manipulation, quoting, and evaluation. Expressions, representing code as data in R, form the foundation of this technique, enabling dynamic alterations to their structure, selective evaluation, or even the creation of entirely new expressions during runtime, facilitating code adaptation to evolving requirements. Quoting temporarily suspends expression evaluation, preserving them for later execution, while unquoting selectively and dynamically evaluates specific components within an expression. This interplay grants developers precise control over the timing and manner of code ex-

ecution, fostering adaptable and responsive program behaviour. Lastly, evaluation, facilitated by functions like `eval` and `substitute`, executes expressions within specified environments, enabling the runtime execution of generated or altered code.

To illustrate this further, we provide a snippet of the algorithm in Figure 3.1, which dynamically constructs R code for decision tree generation.


```
totalXnames = paste0("X", 1:dim_x, collapse = "+")
tree_controls = paste0("minsize=", ML_param$minsize,
                       ", mindev=", ML_param$mindev)
my_code = paste0("theTree = tree::tree(X", ncol(dataframe),
                 "~", totalXnames,
                 ", data = dataframe, " ,tree_controls,")")
eval(parse(text = my_code))
```

Figure 3.1: An example of meta-programming as used in the development of the R package `estimCondCDF`.

In this snippet, the string `my_code` is constructed using expression manipulation. Here various components, including the data source, control parameters, and formula, are programmatically assembled, allowing for code adaptation to diverse datasets. Quoting subtly suspends the immediate evaluation of the `my_code` expression, holding it for deferred execution. Finally, the evaluation aspect comes into play with `eval(parse(text = my_code))`, where the dynamically generated code is executed within the R environment, translating the meta-coded string into executable instructions.

3.2.2. Functional programming

Functional programming is a programming paradigm centred on the evaluation of functions, prioritising the avoidance of mutable data and state changes. Key concepts include first-class functions, which treat functions as first-class citizens, allowing them to be assigned to variables, passed as arguments, and returned as values from other functions, as well as higher-order functions, which either take functions as arguments or return functions. This approach revolves around the creation and application of functions to data, emphasizing immutability, referential transparency, and functional constructs.

Unlike conventional imperative programming reliant on sequential statements for state alteration, functional programming underscores expressing the desired behaviour of the program through interactions through functions. This foundational contrast results in concise, predictable, and less error-prone code, favouring declarative and functional constructs over detailed step-by-step directives.

One of its primary advantages is immutability, which means that once a data structure or variable is created, it cannot be changed. This immutability reduces the likelihood of bugs, simplifies debugging, and makes code more predictable. Additionally, functional programming emphasizes purity, where functions produce the same output for the same input without any side effects. This predictability makes it easier to reason about code behaviour and test individual functions in isolation,

leading to more robust and maintainable code. Moreover, functional programming encourages a declarative style of programming, where developers specify what should be done rather than how to do it, making code more self-explanatory and easier to understand.

Since we propose a general procedure for estimating a conditional CDF we need functional programming. This allows for a more readable code as well as the ability to easily add more machine learning methods later on.

For illustrative purposes, we show a simplified version of the code in Figure 3.2 showcasing how the `estimate_condCDF_general` function employs the concepts of higher-order functions and first-class functions to create a list of ML models.


```
estimate_condCDF_general = function(FitMLmodel, ML_param, (other arguments) )
{
  # ... (Previous code)

    fittedModel = FitMLmodel(dataframe = dataframe, ML_param = ML_param)

    listOfMLmodels[[i]] = list(y = y, model = fittedModel)
  }
  result = list(listOfMLmodels = listOfMLmodels,
                dim = c(dimx = ncol(datax), dimy = ncol(datay)))
}

estimate_condCDF = function(MLmodelName, (other arguments))
{
  switch(

    MLmodelname,

    "Tree" = {

      FitMLmodel = Fit_treeModel

      if (is.null(ML_param)){
        ML_param = list(minsize = 10, mindev = 0.01)
        }
      }
    # ... (Other ML methods)
    )

  result = estimate_condCDF_general(
    FitMLmodel = FitMLmodel, ML_param = ML_param,  (other arguments) )
  return (result)
}
```

Figure 3.2: An example of functional programming as used in the development of the R package `estimCondCDF`.

In this snippet, we have the known first-class function `fit_treeModel`, the variable `FitMLmodel` and the higher-order function `estimate_condCDF_general`. First the function `fit_treeModel` is assigned to the variable `FitMLmodel`. `FitMLmodel` is then passed on to the higher-order function `estimate_condCDF_general` as an argument. The function `fit_treeModel` is then executed in the line `fittedModel = FitMLmodel(dataframe = dataframe, ML_param = ML_param)`. This example demonstrates the workings of functional programming, offering code that is not only more self-explanatory but also aligns with the principles of purity.

3.2.3. Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a paradigm that structures code around the concept of objects, which represent real-world entities or abstract concepts. OOP promotes code organization and reusability by encapsulating data and behaviour within objects. One of its primary advantages is encapsulation, which means that the internal state of an object is hidden from external access, promoting data integrity and reducing the risk of unintended modifications. OOP also emphasizes inheritance, allowing new classes (objects) to inherit properties and methods from existing ones, facilitating code reuse and promoting the creation of hierarchical relationships. Moreover, OOP encourages polymorphism, where different objects can respond to the same method call in a way that is appropriate for their specific types, enhancing flexibility and extensibility.

OOP is guided by several key principles. Encapsulation ensures that the internal state of an object is accessed and modified through well-defined interfaces, promoting data security and reducing code coupling. Inheritance enables the creation of a hierarchy of classes, where child classes inherit attributes and behaviours from parent classes, fostering code reuse and maintenance.

Lastly, polymorphism allows different objects, often belonging to various subclasses, to respond to the same method or message in a way that is contextually relevant to the specific implementation of each object. This means that, even though objects may have different underlying structures and behaviours, they can be treated uniformly when they share a common interface or superclass. Polymorphism promotes code flexibility, extensibility, and reusability by allowing objects of various types to seamlessly interact with shared methods.

Together, these principles in OOP provide a powerful framework for organizing, modeling, and managing complex systems, making code more modular, comprehensible, and maintainable.

To illustrate how OOP is used in the `estimCondCDF` package, a snippet of the prediction method is presented in Figure 3.3. This code provides a unified and consistent interface (predict) for making predictions across different types of machine learning models.

```
predict.estimCondCDFTree = function(object, newdata,
                                    Rearrange = TRUE, ...)
{
  return(predictCommon(object, newdata, predictTreeModel, Rearrange))
}
```

Figure 3.3: An example of OOP as used in the development of the R package `estimCondCDF`.

In the snippet, polymorphism is used to interact with the predict function through the `predict.estimCondCDFTree` method. During the modeling phase, each specific model instance is assigned a class. For instance, when generating a "Tree" model using the `estimCondCDF` package, the resulting model is given the class `estimCondCDFTree`. The `predict.estimCondCDFTree` method is defined with

a method signature that adheres to the conventions of the standard `predict` method in R. This uniformity in method signature enables it to be seamlessly interchangeable with the typical `predict` function in R.

Inside `predict.estimCondCDFTree`, it calls `predict_common` and passes the `object`, `newdata`, and other arguments. `predict_common` is responsible for handling the actual prediction process. The key here is that `predict_common` is a generic function that can work with different types of machine learning models. It uses dynamic dispatch to determine the appropriate behaviour based on the type of object.

This makes it easier to work with various models interchangeably and ensures that the `predict` function behaves appropriately based on the specific model type, enhancing code flexibility and maintainability.

3.3. Internals of the package

When applying the estimation methodology the statistician will first create a model based on one of the four ML methods. This model is then used to make predictions for estimating the conditional CDF. Lastly, these predictions are rearranged to more accurately estimate the conditional CDF. The created `estimCondCDF` package is thus split up into two parts, the modeling and the prediction part. In this section, we will discuss the challenges encountered during the package development and explain the design choices made. We will then provide insights into the inner workings of the modeling and prediction components of the package. Lastly, we will offer practical guidance on using the package effectively.

3.3.1. Challenges and design decisions

During the development of the estimation method and the construction of the R package, we encountered numerous challenges necessitating design decisions. In this subsection, we will elaborate on the problems that arose and the solutions we came up with.

The first decision we had to make was the way to save the build models. As seen in Algorithm 5, a new model, based on the specified machine learning method, is trained for each threshold value $\mathbf{y}_i$. This results in one model consisting of multiple models. These multiple models are based on the specified machine learning method and the number of models on the number of threshold values. We thus need to save each model in combination with the threshold values it was trained on. Besides this, the dimensions of the vector of threshold values $\mathbf{y}$ and the vector containing the values of interest $\mathbf{x}$ should be saved in order to be easily accessed during the prediction phase. In the case of the "Tree" and "NN" models, the model, corresponding threshold values and dimensions are saved as a list of lists as visualised in Figure 3.4.

To create a model for the bagging methods, "RF" and "NNorest" we again need to think about the way we store the models. There are multiple packages in R that generate a random forest model. However, there is no package for a "NNForest" model and we already have the existing code for the decision tree and

neural network methods. We thus take a number of B bootstrap samples from the data and build a decision tree or neural network for each bootstrap sample. To save these B models, one extra list is added containing the B corresponding trained models. This is illustrated by the nested list in Figure 3.5. For the prediction of the bagging methods, we take the mean over the individual predictions of these trees or neural networks.

Now that we have established a method for saving the model, we need to dynamically fit each model based on the dimension of the predictor values `x`. As we do not want to just estimate the conditional CDF for x_1 and x_2 but want to be able to do this for multiple dimensions, we use meta-programming as described in subsection 3.2.1 to dynamically fit the machine learning models based on the input of the statistician.

Furthermore, we want the code to interact with the general `predict` function in R. To achieve this we assign a specific class to the model returned, determined by the employed machine learning method. We then use OOP principles to enable this interaction. For instance, in the case of the "Tree" model with the class `estimCondCDFTree`, we create a dedicated function `predict.estimCondCDFTree`. This allows the statistician to input a model with class `estimCondCDFTree` in the general `predict` function in R. Because the class of this model is `estimCondCDFTree` the specified function `predict.estimCondCDFTree` is automatically invoked.

We use a range of different R packages for multiple different functions within the `estimCondCDF` package. Specifically, we use the `stats` package to employ the `quantile` function, which enables the calculation of the 10% and 90% quantiles of the input variables `y`. Additionally, we rely on the `predict` function from the same package to make predictions for each decision tree or neural network in the model. To construct these decision trees and neural networks, we turn to the `tree` function from the `tree` package and the `nnet` function from the `nnet` package, respectively. For column selection based on specific character prefixes, we employ the `select` and `starts_with` functions from the `dplyr` package. Lastly, to rearrange the predictions effectively, we make use of the `rearrangement` function from the `Rearrangement` package. Prior to this, we prepare the data using a combination of functions from the `tidyr` package, including `pivot_wider`, `pivot_longer`, and `all_of`.

- The main list
  - The list of models and predictor values:
    - ◇ The first trained model:
      - · The threshold values $\mathbf{y}_1$.
      - · The corresponding trained decision tree or NN model.
    - ◇ The second trained model:
      - · The threshold values $\mathbf{y}_2$.
      - · The corresponding trained decision tree or NN model.

    ⋮
    - ◇ The s-th trained model:
      - · The threshold values $\mathbf{y}_s$.
      - · The corresponding trained decision tree or NN model.
  - The dimensions of $\mathbf{x}$ and $\mathbf{y}$

Figure 3.4: Visual representation of how the decision tree or neural network models are saved in the main "Tree" or "NN" model.

- The main list
  - The list of models and predictor values:
    - ◇ The first trained bagging model:
      - · The threshold values $\mathbf{y}_1$.
      - · a list of B corresponding trained decision tree or NN models
    - ◇ The second trained model:
      - · The threshold values $\mathbf{y}_2$.
      - · a list of B corresponding trained decision tree or NN models

    ⋮
    - ◇ The s-th trained model:
      - · The threshold values $\mathbf{y}_s$.
      - · a list of B corresponding trained decision tree or NN models
  - The dimensions of $\mathbf{x}$ and $\mathbf{y}$

Figure 3.5: Visual representation of how the decision tree or neural network models are saved in the main "RF" or "NNForest" model.

3.3.2. Modeling mechanics

We aim to construct a model for estimating the conditional CDF, denoted as $P(\mathbf{Y} \leq \mathbf{y}|\mathbf{X} = \mathbf{x})$, where the vector $\mathbf{X}$ corresponds to the vector of covariates, while $\mathbf{Y}$ represents the dependent response vector with associated values. Furthermore, $\mathbf{y}$ represents a vector of threshold values and $\mathbf{x}$ is the vector of values of interests of the covariates. As seen in Algorithm 5 in Section 3.1, we train a new model for each new $\mathbf{y}$ value. In order to build the model using one of the machine learning techniques, the primary function, `estimate_condCDF()`, determines the suitable machine learning model based on the `MLmodelName` given by the statistician. We detail the internals of the function `estimate_condCDF()`.


```
estimate_condCDF = function(datax, datay, MLmodelName,
                            y_values = NULL, ML_param = NULL)
{
  switch(

    MLmodelName,

    "Tree" = {

      FitMLmodel = Fit_treeModel

      if (is.null(ML_param)){
        ML_param = list(minsize = 10, mindev = 0.01)
      }
    },

    ...(The other ML techniques)
  )
  result = estimate_condCDF_general(
    datax = datax, datay = datay,
    FitMLmodel = FitMLmodel,
    y_values = y_values,
    ML_param = ML_param,
    MLmodelName = MLmodelName)
  return (result)
}
```

Figure 3.6: The `estimate_condCDF()` function called by the statistician.

The function takes in a matrix or dataframe of $\mathbf{X}$ values and a matrix or dataframe of corresponding $\mathbf{Y}$ values, denoted by `datax` and `datay` respectively, as well as the specified machine learning technique to be used, denoted by `MLmodelName`. Optionally a vector or list of threshold values $\mathbf{y}$ can be specified, denoted by `y_values`. If no $\mathbf{y}$ vector is specified the 10th and 90th percentile of the dataset are used as input vectors. Furthermore, a list of hyperparameters specific to the chosen machine learning technique can be specified. If no such list is supplied the standard hyperparameters of the machine learning technique are used. The `fitMLmodel` function is the function where the model is actually constructed.

The `estimate_condCDF` function, as shown in Figure 3.6, passes the chosen `fitMLmodel` function to the `estimate_condCDF_general()` function, illustrated in Figure 3.7. This function then cleans and prepares the data as input for the `fitMLmodel` function.

As seen in Algorithm 5 we first calculate the Indicator function $\mathbf{1}\{\mathbf{Y} \leq \mathbf{y}_i\}$.

```
estimate_condCDF_general = function(datax, datay, FitMLmodel,
                                    y_values = NULL, ML_param,
                                    MLmodelName)
{
  ...(Data prepping)

  listOfMLmodels = list()
  for (i in 1:nrow(y_values)) {

    y = y_values[i, , drop = FALSE]
    colnames(y) = c(paste0("y", 1:(dim_y)))
    data_frame = as.data.frame(cbind(datax, Indicator(datay, y) ) )
    colnames(dataframe) = c(paste0("X", 1:(dim_x+1)))

    fittedModel = FitMLmodel(dataframe = dataframe, ML_param = ML_param)

    listOfMLmodels[[i]] = list(y = y, model = fittedModel)
  }
  result = list(listOfMLmodels = listOfMLmodels,
                dim = c(dimx = ncol(datax), dimy = ncol(datay)))
  class(result) = paste0("estimCondCDF", MLmodelName)
  return(result)
}
```

Figure 3.7: The `estimate_condCDF_general()` function, cleaning and preparing the data into a dataframe, to pass into the `fitMLmodel` function.

For this, the function `Indicator` calculates $\mathbf{1}\{\mathbf{Y} \leq \mathbf{y}_i\}$ for the dataset `datay` and returns a binary value $W_{\mathbf{y}_i}$ per row. These values are used to create the new dataframe `data_frame`. This dataframe is passed into the `fitMLmodel` function where the specified machine learning technique is used to create a model. We use a `for` loop in order to make a model for each provided vector of threshold values.

As an example, we show how this works for the "Tree" model where the chosen ML model is the decision tree. In this case, the variable `fitMLmodel` is assigned the function `Fit_treeModel`, as shown in Figure 3.8. Inside this function, we use the `tree` function from the R package `tree` to generate a decision tree.

```
Fit_treeModel = function(dataframe, ML_param = ML_param)
{
  theTree = tree::tree(X3 ~ X1+X2, data = dataframe,
                       minsize = minsize, mindev = mindev)
  return (theTree)
}
```

Figure 3.8: The `Fit_treeModel` function.

In this example code a two-column matrix of $\mathbf{X}$ values is provided. The new dataset $\mathcal{D}_{W,\mathbf{y}}$ is referred to by `dataframe` and consists of these two columns of $\mathbf{X}$ values and the binary column $W_{\mathbf{y}_i}$, here referred to as `X3`, created using the indicator function. The function tree grows a decision tree using the variables `X1` and `X2` as predictors to predict the variable `X3`. This will return a trained decision tree model based on these provided $\mathbf{y}$ values.

The code used for the development of the R package `estimCondCDf` however,

needs to be able to dynamically determine the number of columns of the dataframe `datax` predictor **X** and adjust the `tree` function accordingly and automatically. We therefore use meta programming as described in the code block of Figure 3.1

When switching from the "Tree" model to the "NN" model, the function assigned to `fitMLmodel` becomes `Fit_NNModel`. Within this function, we construct a neural network using the `nnet` function from the `nnet` package.

For the remaining two models, "RF" and "NNForest", the corresponding functions are `Fit_RFModel` and `Fit_NNForestModel`, respectively. The procedures for both of these models closely resemble those of the 'Tree" model and the "NN" model, respectively, with the inclusion of an additional bagging step. The bagging step is executed by introducing an extra `for` loop within the corresponding `Fit` functions, iterating for a predetermined number of decision trees or neural networks to construct. Within this loop, data is initially sampled, upon which the decision trees or neural networks are generated.

In the final step, each model is assigned a distinct class, determined by the applied machine learning method used, indicated by `MLmodelName`. For instance, in the case of the "Tree" model, the class will be named `estimConCDFTree`. Assigning different classes to each model is essential for seamless integration with the standard `predict` function in R.

3.3.3. Prediction process

With the model in place, we can start the prediction process where we use OOP for compatibility with the general `predict` function in R. When a model, trained using the `estim_condCDF` function, is input into the `predict` function, it triggers the execution of a designated `predict.condCDF` function. For instance, if the input is a model of class `estimcondCDFTree` the function `predict` actually calls the `predict.condCDFTree` function, illustrated in Figure 3.9

```
predict.estimCondCDFTree = function(object, newdata,
                                    Rearrange = TRUE, ...)
{
  return(predictCommon(object, newdata, predictTreeModel, Rearrange))
}
```

Figure 3.9: The `predict.estimCondCDFTree` function called by entering a model with class `estimCondCDFTree` into `predict`.

This function takes the trained model, of class `estimcondCDFTree`, denoted as `object`, and a dataframe of new `x` values for which to make predictions, denoted by `newdata`. It furthermore takes an optional boolean input variable `Rearrange` to control the rearrangement of the resulting predictions. This design is consistent with other machine learning methods to ensure that objects of the specified classes can interact seamlessly with the `predict` function in R.

This data, in combination with the predict function specified by the object class, is then passed to the `predictCommon` function illustrated in Figure 3.10. In the case of the "Tree" model, this is the function `predictTreeModel`. The other models have

their corresponding prediction functions passed to the `predictCommon` function in a similar fashion as in the estimation step. In these respective prediction functions the actual predictions for each model is executed using `predict`.

```
predictCommon <- function(object, newdata, predictFunction,
                          Rearrange = TRUE) {
  result = predictCondCDF_general(
    trainedModel = object,
    newx = newdata,
    predictMLmodelName = predictFunction)

  if (Rearrange) {
    result = Rearranged_dim2(result)
  }

  return(result)
}
```

Figure 3.10: The `predictCommon` function.

The `predictCommon` function acts as a common wrapper for making predictions. It abstracts away details and provides a consistent interface for making predictions using the different machine learning models. This modular design makes it easier to extend the code to support future additional machine learning techniques.

The inputs are passed into the `predictCondCDF_general` function. Here the position of the **y** values for which the model was trained on are determined in the list of lists and general data cleaning and preparation is done to pass everything into the specified `predictFunction`

In the `predictCondCDF_general` function a dataframe, denoted by `dataframe`, is created to store the results of the predictions. This dataframe consists of the columns from `newx`, indicating the vector of values of interests **x**, the columns from `newy` indicating the threshold values **y** the model is trained on and a column for predictions initialized with missing values (`NA`). The code then enters a `for` loop that iterates through each row of the dataframe. For each row, a subset of `newx` is extracted and then used in the `predictMLmodel` function to make predictions. The results are stored in the `prediction` column of the dataframe.

Depending on the `predictMLmodel` which was determined by the class of the object passed into the predict function the predictions are made. In case that object was of class `estimCondCDFTree` the code would be

```
predict_treeModel = function(trainedModel, thePosition,
                             newx)
{
  output = stats::predict(trainedModel[[thePosition]]$model,
                          newdata = newx,
                          type = "vector")
  return(output)
}
```

Here the `predict` function from the R package `stats` is used to find a prediction. Since we unpacked a decision tree, an object of class tree, from the list of lists the actual function that R calls is the `predict.tree` function integrated into the predict

```
predictCondCDF_general = function(trainedModel,
                                  newx,
                                  predictMLmodel)
{
  ....(Data prepping)

  positions = rep(positions, times = nrep_positions)
  new_y = as.data.frame(newy)

  dataframe <- data.frame(
    dplyr::select(newx, dplyr::starts_with("X")),
    dplyr::select(newy, dplyr::starts_with("y")),
    prediction = NAreal,
    titles = NAcharacter
  )
  dataframe$titles = apply(newx, 1, function(row) {
    paste0(paste(names(newx), " = ", row, collapse = ", "))
  })

  for (i in seq(nrow(dataframe))) {
    xnew = as.data.frame(dataframe[i,c(paste0("X", 1:dimx))])
    colnames(xnew) = c(paste0("X", 1:dimx))
    dataframe[i,"prediction"] =
      predictMLmodel(trainedModel = listOfMLmodels,
                     thePosition = positions[i],
                     newx = xnew)

  }
  return(dataframe)
}
```

Figure 3.11: The `predictCondCDF_general` function.

function just as the `predict.estimateCondCDF` function is.

In this code, the `predict` function from the R package `stats` is employed to carry out predictions. Specifically, as we have previously unpacked an object of class `tree` from the list of models, the actual function that R invokes is the `predict.tree` function. This is seamlessly integrated into the `predict` function, much like the custom `predict.estimateCondCDF` function. Similarly, for neural networks, the prediction function would be `predict.nnet`.

At the end of the `for` loop in the `predictCondCDF_general` function, a dataframe containing predictions is returned. If it was specified to use rearrangement on these predictions the `predictCommon` function now passes this dataframe into the `Rearranged_dim2` function. Here the dataframe is again prepared to now work with `rearrangemnt` function from the R package `rearrangment` and a dataframe containing the `x` values, `y` values, predictions and rearranged predictions is returned.

3.3.4. Practical application

To use the `estimateCondCDF` package the first step is to install it. Next, the statistician trains a model from a dataset. This model is used in the final step to make predictions given new values of explanatory variables. An illustrative example of this process using the package in R is provided in Figure 3.12:

Initially, the statistician creates a dataframe comprising the predictor variables, denoted as `X1` and `X2`, along with the response variables `Y1` and `Y2`. In Algorithm

```
X1 = runif(n = 100, min = 0, max = 1)
X2 = runif(n = 100, min = 0, max = 1)
Y1 = rnorm(n = 100, X1, 1)
Y2 = rnorm(n = 100, X2, 1)

dataframe = data.frame(X1, X2, Y1, Y2)

mymodel = estimCondCDF::estimate_condCDF(datax = dataframe[,1:2],
                                         datay = dataframe[,3:4],
                                         MLmodelName = "tree")


newx = data.frame("x1" = c(1,2),"x2" = c(3,4))
predict(mymodel, newdata = newx)
```

Figure 3.12: Practical example of fitting a model and making predictions using the `estimCondCDF` package.

5, this dataframe corresponds to the training dataset $\mathcal{D}$. In this simple example, the data is generated using predefined distributions, whereas real-life applications necessitate loading and cleaning a dataset of observations to adhere to this format.

Subsequently, the model is trained using the `estimate_condCDF` function from the `estimCondCDF` package. This function requires five inputs: `datax`, `datay`, `MLmodelName`, `y_values`, and `ML_param`. Specifically, `datax` and `datay` should be dataframes, while `MLmodelName` should be a string indicating the chosen machine learning method for classification. Optionally, `y_values` and `ML_param` can be provided, although they possess default values. `y_values` should consist of a list of threshold values, represented by $\mathbf{y}_i$ in Algorithm 5, and `ML_param` should be a list containing the hyperparameters specific to the chosen machine learning method.

Then, a new dataframe containing the x values is constructed, referred to as the grid of $\mathbf{x}$ values in Algorithm 5.

Finally, the process of making predictions begins by invoking the `predict` function. This function expects several inputs: an `object`, in this case, `mymodel` carrying the class `estimCondCDFTree`, and a `newdata` dataframe on which predictions are to be generated, here denoted as `newx`. Additionally, to obtain an increasing function, that improves estimations closer to the true conditional CDF, we can apply a rearrangement process to the predicted values, as explained in Section 2.5. To indicate this, a boolean input variable, `Rearrange`, which defaults to `TRUE`, signifies whether the predicted values should be rearranged.

The `predict` function then returns a dataframe encompassing the following components: the new data $\mathbf{x}$ for which predictions were computed, the corresponding $\mathbf{y}$ values on which predictions were made, the estimations of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ evaluated at the new observations $\mathbf{x}$ and $\mathbf{y}$ and the rearrangement of these estimations. This resulting dataframe is illustrated in Figure 3.13.

3

Console Terminal

R 4.2.2 · C:/Users/jrang/Desktop/Thesis/GitHub/MScProjectJugo/

```
> newx = matrix(c(1,3,2,4), nrow = 2, byrow = TRUE)
> predict(object = Mymodel, newdata = newx)
   x1 x2    y1    y2 prediction   rearranged
1   1  3 -0.82 -1.00 0.00000000 0.000000e+00
2   1  3 -0.22 -1.00 0.00000000 0.000000e+00
3   1  3  0.39 -1.00 0.00000000 0.000000e+00
4   1  3  0.99 -1.00 0.02564103 1.282047e-02
5   1  3  1.60 -1.00 0.02564103 2.564103e-02
6   1  3 -0.82 -0.36 0.00000000 0.000000e+00
7   1  3 -0.22 -0.36 0.00000000 2.443602e-05
8   1  3  0.39 -0.36 0.22222222 1.053072e-01
9   1  3  0.99 -0.36 0.28571429 1.469430e-01
10  1  3  1.60 -0.36 0.28571429 2.025899e-01
11  1  3 -0.82  0.29 0.00000000 0.000000e+00
12  1  3 -0.22  0.29 0.00000000 3.882589e-05
13  1  3  0.39  0.29 0.33333333 1.534015e-01
14  1  3  0.99  0.29 0.00000000 2.198244e-01
15  1  3  1.60  0.29 0.71428571 3.189464e-01
16  1  3 -0.82  0.93 0.00000000 9.140047e-10
17  1  3 -0.22  0.93 0.00000000 5.879972e-05
18  1  3  0.39  0.93 0.11111111 2.225715e-01
19  1  3  0.99  0.93 0.12500000 2.743152e-01
20  1  3  1.60  0.93 0.14285714 4.863598e-01
21  1  3 -0.82  1.58 0.01492537 3.731343e-06
22  1  3 -0.22  1.58 0.00000000 1.429734e-02
23  1  3  0.39  1.58 0.30769231 3.205128e-01
24  1  3  0.99  1.58 0.57142857 5.081571e-01
25  1  3  1.60  1.58 0.33333333 7.142857e-01
26  2  4 -0.82 -1.00 0.00000000 0.000000e+00
27  2  4 -0.22 -1.00 0.00000000 0.000000e+00
28  2  4  0.39 -1.00 0.00000000 0.000000e+00
29  2  4  0.99 -1.00 0.02564103 1.282047e-02
30  2  4  1.60 -1.00 0.02564103 2.564103e-02
31  2  4 -0.82 -0.36 0.00000000 0.000000e+00
32  2  4 -0.22 -0.36 0.00000000 2.443602e-05
33  2  4  0.39 -0.36 0.22222222 1.053072e-01
34  2  4  0.99 -0.36 0.28571429 1.469430e-01
35  2  4  1.60 -0.36 0.28571429 2.025899e-01
```

Figure 3.13: Dataframe returned by `predict`.

4

Simulations

In this chapter, we aim to assess and enhance the precision of estimations of the conditional CDF by investigating their relative performance through simulations, as inspired by [8]. We use the four models specified in Section 3.1 to estimate the conditional CDF

$$F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) = \mathbb{P}(Y_1 \leq y_1, Y_2 \leq y_2 | X_1 = x_1, X_1 = x_2) \tag{4.1}$$

Specifically, each model consists of multiple distinct blocks that can be chosen independently:

(i) The hyperparameters of the ML method.

(ii) The bivariate distribution of $\mathbf{X} = (X_1, X_2)$.

(iii) The conditional distribution of $\mathbf{Y} = (Y_1, Y_2)$ given $\mathbf{X}$.

(iv) The sample size n.

For these blocks, we chose the reference setting to be defined as:

(i) Specified hyperparameters chosen after the hyperparameter simulations.

(ii) $\text{Cor}(X_1, X_2) = 0$ where $(X_1, X_2) \sim \mathcal{N}((0,0), I_2)$.

(iii) $Y_1 \sim \mathcal{N}(X_1, 1)$ and $Y_2 \sim \mathcal{N}(X_2, 1)$.

(iv) Sample size $n = 5000$.

For each variation, we conduct 200 distinct replications where we estimate the model based on generated data following the distributions defined in the reference setting. We run the simulations using computational resources of the DelftBlue supercomputer, provided by Delft High Performance Computing Centre [7].

In order to define the reference setting for the hyperparameters of each machine learning technique we will first find the optimal hyperparameters (i) for each ML technique under this reference setting. We will assess the performance of each machine learning technique using the Mean Integrated Squared Error (MISE) as an evaluation metric. The MISE quantifies the average squared difference between the estimated conditional CDF, $\widehat{F}_{\mathbf{Y}|\mathbf{X}}$, and the true conditional CDF, $F_{\mathbf{Y}|\mathbf{X}}$, integrated over $\mathbf{x}$ and $\mathbf{y}$. We define the MISE as

$$\begin{aligned}\text{MISE} &= \mathbb{E}\left[\int \left(\widehat{F}_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}) - F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})\right)^2 d\mathbf{x}d\mathbf{y}\right] \\ &= \mathbb{E}\left[\int \Big(\widehat{\mathbb{P}}\big(Y_1 \le y_1, Y_2 \le y_2 | X_1 = x_1, X_1 = x_2\big)\right. \\ &\left. -\mathbb{P}\big(Y_1 \le y_1, Y_2 \le y_2 | X_1 = x_1, X_1 = x_2\big)\Big)^2 dx_1\, dx_2\, dy_1\, dy_2\right]\end{aligned} \tag{4.2}$$

In addition to the MISE, we also assess the methods based on their average computation time measured in seconds. Since the number of simulations we run is finite, we, however, approximate the MISE by taking points on the grid $\mathbf{x}$, $\mathbf{y}$ and replacing the integral and the expectation operators with finite averages.

We begin the simulation study by determining the optimal hyperparameters for each machine learning technique. Subsequently, we will investigate the robustness of the predictive models by altering key factors. First, we will explore the impact of varying the correlation (ii) between the predictor variables X_1 and X_2. Next, we will assess the adaptability of the model to different distributions (iii) of the response variables, Y_1 and Y_2. Lastly, we will evaluate the performance of the model under differing sample sizes n (iv). This comprehensive analysis will provide valuable insights into the versatility and reliability of the prediction method demonstrated through different machine learning techniques across various scenarios.

4.1. Optimizing hyperparameters for ML models

To determine the optimal hyperparameters for each machine learning technique, we maintain the reference setting while systematically varying the hyperparameters (i). For each combination of these hyperparameters, we compute the MISE to gauge performance. The set of hyperparameters yielding the most favourable MISE results, while also considering computation time, then becomes the updated reference setting for each respective machine learning technique. This method allows us to identify and fine-tune the hyperparameters that yield the best overall model performance.

4.1.1. Decision tree

In the context of decision trees, the analysis initiates with the assessment of the influence of the two hyperparameters `minsize` and `mindev`. As previously discussed in Section 2.2, within the "Tree" model, `minsize` denotes the minimum number of observations required in a terminal node, and `mindev` represents the minimum

deviation in response values for a split to occur. For `minsize`, five distinct values are considered: 1, 25, 50, 75 and 100. In the case of `mindev`, a range of twelve distinct values is examined: $0.0001, 0.0002, 0.0005, \ldots, 0.1, 0.2, 0.5$.

The simulation study reveals that for smaller values of the `mindev` parameter, estimations of the conditional CDF exhibit improvement, indicated by a lower MISE. This observation aligns with the expectation that a reduced `mindev` allows for capturing finer details within the training data since splits occur more frequently. Figure 4.1 visualises the improvement in estimation for lower values of `mindev`. Notably, we observe that similar `mindev` values maintain a consistent relative order when associated with different `minsize` values. This consistent relative relationship between `minsize` and `mindev` further underscores the robustness and interdependence of these hyperparameters in optimizing model performance.

We furthermore find that for higher values of the `minsize` parameter, we have a lower average computation time. This is in line with intubation since a larger `minsize` value means that each leaf node of the decision tree must contain more data points before it can be considered a valid, final prediction node, resulting in a simpler tree structure with fewer nodes and less depth. The effect of the `minsize` hyperparameter on the average computation time and MISE of the "Tree" model is visualised in Figure 4.2.

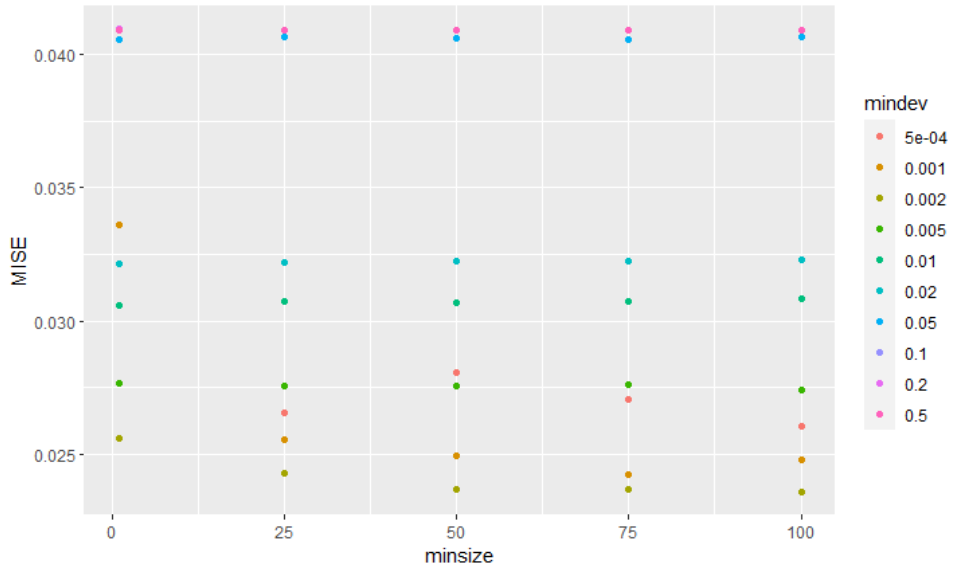

Figure 4.1: MISE of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ using the ML model "Tree", for various combinations of the tuning parameters `minsize` and `mindev`.

Using both Figures 4.1 and 4.2, we identify the hyperparameters yielding the most favourable MISE, approximately 0.0236, to be `minsize` = 100 and `mindev` = 0.002. This combination of hyperparameters not only results in a favourable

MISE but also exhibits a reasonable average computation time for estimating the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$, as shown in Figure 4.2, averaging at approximately 3.2 seconds. While this computation time is not the fastest, it is still efficient. Considering the substantial improvement in MISE achieved with these hyperparameters, the slight trade-off in computation time is justifiable and yields a worthwhile payoff. Therefore, the reference setting for the hyperparameters of the "Tree" model will be defined as `minsize` = 100 and `mindev` = 0.002.

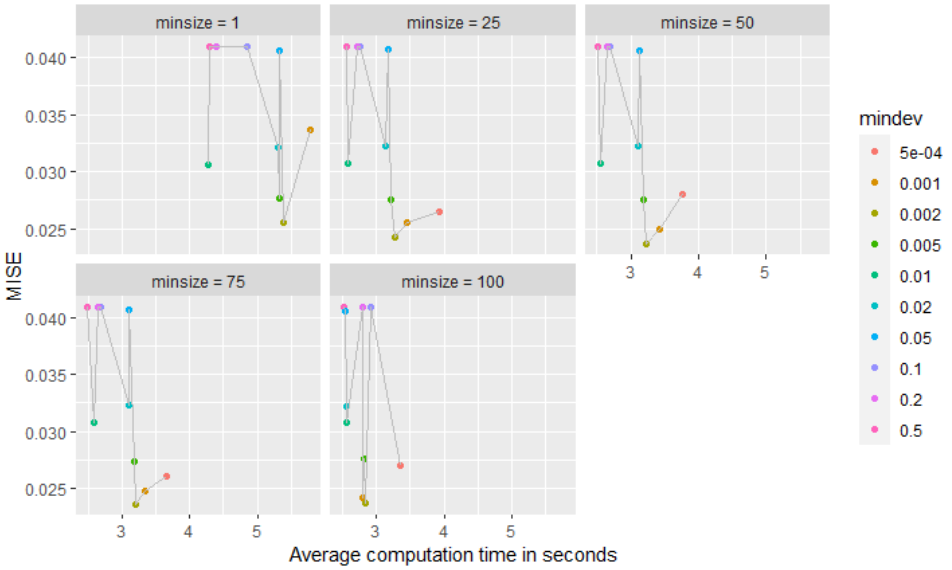

Figure 4.2: MISE and average computation time of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ using the ML model "Tree", for various combinations of the tuning parameters `minsize` and `mindev`.

It is noteworthy that both figures, Figure 4.1 and Figure 4.2, do not encompass certain hyperparameter combinations, particularly those involving `mindev` values of 0.0001 and 0.0002. The absence of these combinations in the visual representations is due to the occurrence of specific errors, namely the `Tree is too big` or `Maximum depth reached` errors, when such combinations are employed. These errors are summarized in Table 4.1.

The emergence of both the `Maximum depth reached` and `Tree is too big` errors with small `mindev` values can be attributed to the propensity of small `mindev` values to promote finer and more intricate splits within the decision tree. This inclination towards finer partitioning has the potential to yield a tree structure with numerous levels and nodes. The `Maximum depth reached` error occurs when the ongoing node partitioning process extends the depth of a decision tree beyond a predetermined threshold, serving as a protective measure against excessive complexity and overfitting. Conversely, the `Tree is too big` error arises when the decision tree becomes excessively large, demanding higher memory and computational resources. A decision tree characterized by a profusion of nodes and extensive

	Maximum depth reached				Tree is too big		
minsize \mindev	**1e-04**	**2e-04**	**5e-04**	**0.001**	**1e-04**	**2e-04**	**5e-04**
1	100	100	100	0.5	0	0	0
25	100	100	51	0	0	0	0
50	50.5	62	44	0	44.5	38	0
75	12.5	18.5	23.5	0	87.5	80.5	2
100	4	10.5	17.5	0	96	89.5	51.5

Table 4.1: Estimated probability (%) of obtaining an error when using Algorithm 5 with the ML method "Tree" for extreme choices of tuning parameters `minsize` and `mindev`.

Example: when using the tuning parameters `minsize = 100` and `mindev = 1e-04`, the probability of obtaining the error `Maximum depth reached` is 4%.

depth necessitates substantial resources for efficient operation. Thus, small `mindev` values, by fostering finer partitions, can contribute to this scenario by facilitating the creation of a larger and more intricate tree structure.

4.1.2. Neural network

To determine the optimal hyperparameters for the neural network method, a range of distinct values are considered for the hyperparameters `n_neurons` and `maxiter` which are detailed in Table 4.2. As discussed in Section 2.3, the parameter `n_neurons` signifies the number of neurons within each layer of the neural network. In this approach, we employ a one-layer neural network. Additionally, `maxiter` corresponds to the maximum allowable number of iterations for the training process, determining when the training algorithm should converge or terminate. It is worth noting that selecting values exceeding 250 for `n_neurons` would result in the `too many (1001) weights` error due to an excessive number of weights in a single layer.

maxiter	1	2	3	5	7	10
n_neurons	2	5	10	20	50	100

Table 4.2: Chosen hyperparameter combinations for neural network.

The simulation study shows that, in line with intuition, increasing the number of neurons, `n_neurons`, leads to a smaller MISE, as illustrated in Figure 4.3. Additionally, as shown in Figure 4.4, increasing the number of neurons corresponds to an increase in computation time. The trade-off observed in Figure 4.4 between average computation time and MISE is intuitive, as one would expect that a longer average computation time should result in more accurate estimations.

However, as demonstrated in Figures 4.3 and 4.4 increasing the maximum number of iterations `maxiter` leads to an increase in average computation time but deterioration in the performance of the estimator, as indicated by the increase in MISE. Therefore, having a reduced amount of iterations can actually improve the performance of the estimator. This pattern is a well-established concept in machine learning known as "early stopping" [19]. Early stopping involves monitoring the

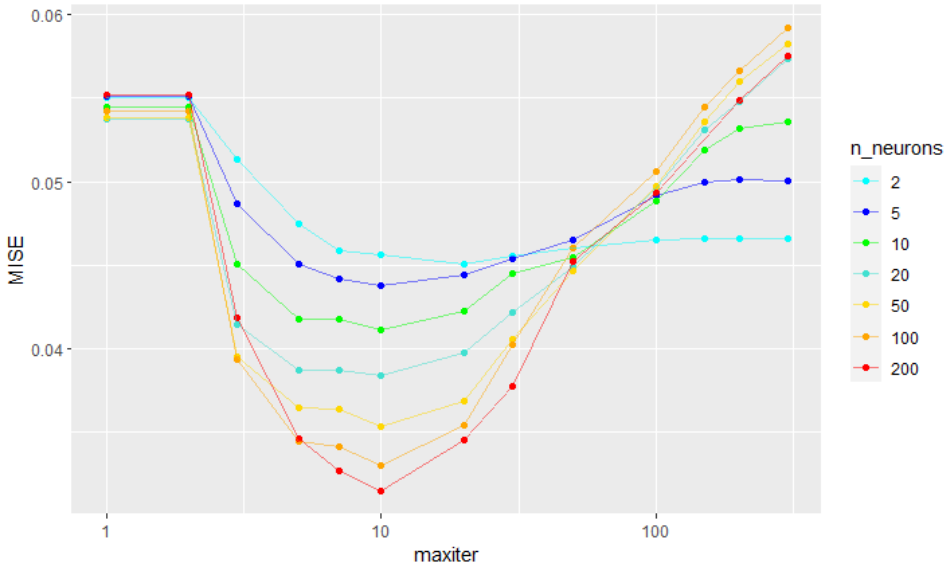

Figure 4.3: MISE of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ using the ML model "NN", for various combinations of the tuning parameters `maxiter` and `n_neurons`.

performance of a model during training and halting the training process once a certain criterion is met. The aim of early stopping is to prevent overfitting, where the model becomes too specialized to the training data and fails to generalize well to unseen data. Early stopping allows to terminate training when the performance of the model starts to degrade on the validation set, rather than continuing until it overfits the training data. This approach helps strike a balance between model complexity and generalization, resulting in models that perform better on new, unseen data.

We, therefore, choose the hyperparameters for the "NN" model to be `maxiter` = 10 and `n_neurons` = 20. As depicted in Figure 4.3, the hyperparameters leading to the most favourable MISE of approximately 0.0315, are `maxiter` = 10 and `n_neurons` = 200. However, it is important to consider the substantial computational time of 24 seconds required for this outcome. As a more balanced alternative, `maxiter` = 10 and `n_neurons` = 20 produces a MISE of approximately 0.384 and entails an average computation time of 5.98 seconds. The reference setting for the hyperparameters of the "NN" model will thus be defined as `maxiter` = 10 and `n_neurons` = 20.

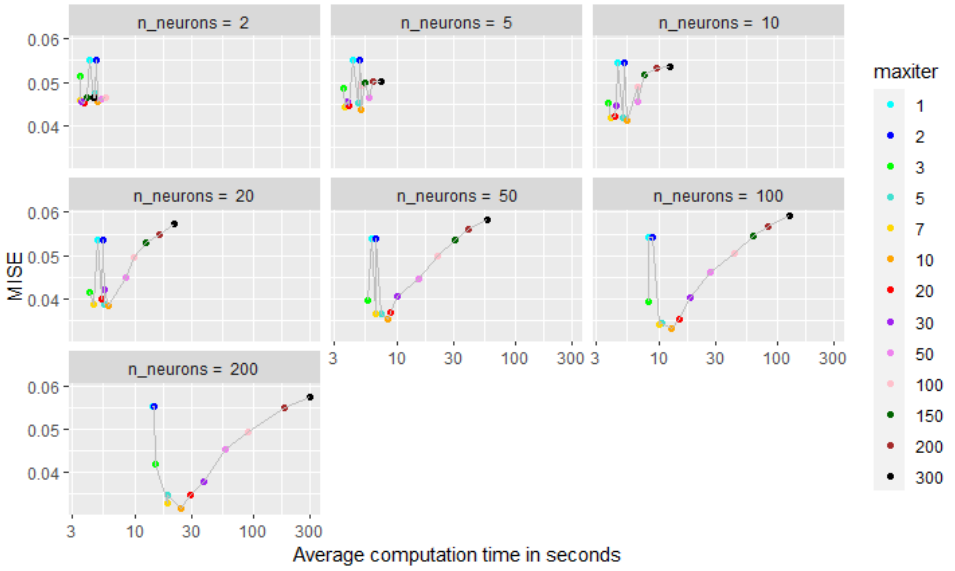

Figure 4.4: MISE and average computation time of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ using the ML model "NN", for various combinations of the tuning parameters `maxiter` and `n_neurons`.

4.1.3. Random forest

To determine the best settings for the random forest method this analysis uses the optimal hyperparameter values found in the decision tree experiments, namely `minsize` = 100 and `mindev` = 0.002. In the evaluation, we focus on the two remaining hyperparameters: `n_bootstraps`, which signifies how many decision trees will be generated, evaluated at the values 5, 10, 20, 50, and 100, and the `pctObsBootstrap` hyperparameter, denoting the percentage of observations (from the dataset) used in each decision tree. The latter was assessed across a spectrum of percentages: 75%, 80%, 85%, 90%, and 95%.

The simulation study reveals that the MISE is approximately constant: the MISEs for all combinations of hyperparameters lie very close together. Indeed, we find that the improvement from the worst-performing combination to the best-performing combination is only 5%. This can be seen in Figure 4.5, which displays the MISE of the estimator using the ML model "RF" for various combinations of hyperparameters.

Figure 4.5 does however show a trend indicating that the performance of the "RF" model is better for smaller percentages of observations per bootstrap, as the MISE slightly increases as the percentage of observations per bootstraps increases. Additionally, Figure 4.6 confirms the intuition that a higher number of bootstraps increases the mean computation time.

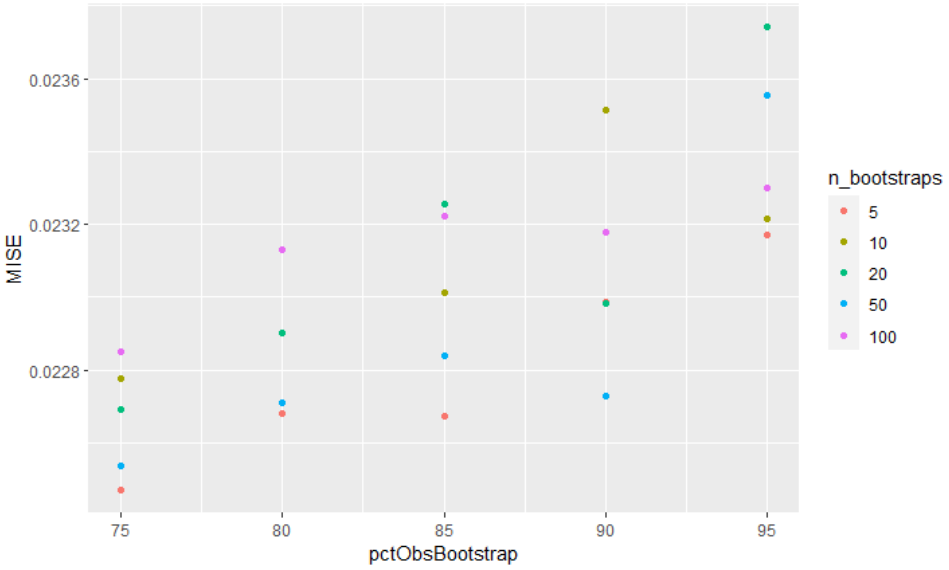

Figure 4.5: MISE of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ using the ML model "RF", for various combinations of the hyperparameters `n_bootstraps` and `pctObsBootstrap`.

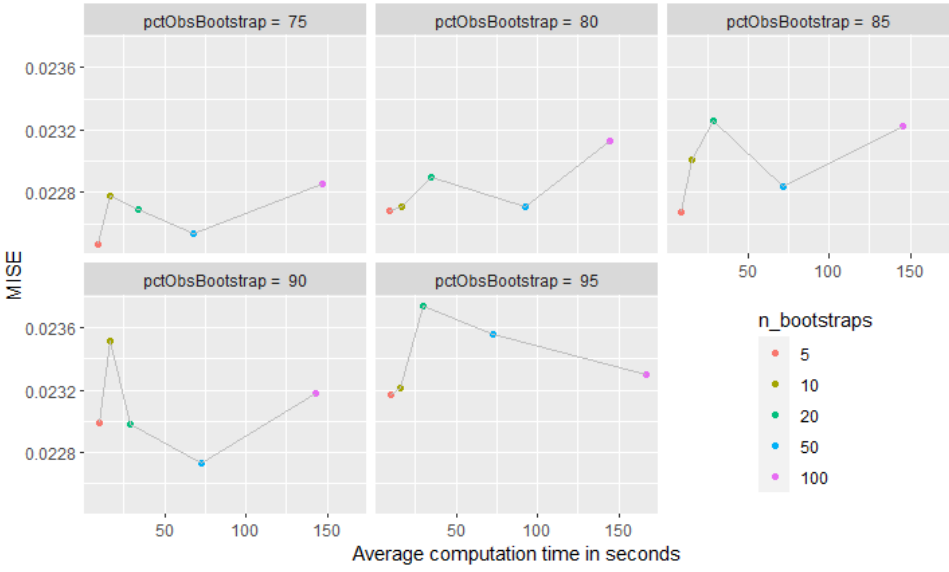

Figure 4.6: MISE and average computation time of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ using the ML model "RF", for various combinations of the hyperparameters `n_bootstraps` and `pctObsBootstrap`.

Given the similarity in MISE values across the hyperparameter combinations, the choice we make is not critical. Therefore, we opt for a combination with a short average computation time and low observation percentage per bootstrap. Specifically, we select `n_bootstraps` = 5 and `pctObsBootstrap` = 75%. These settings yield a minimal MISE of 0.022, as shown in Figure 4.5, and boast one of the shortest computation times, completing in only 9.2 seconds, as depicted in Figure 4.6. The reference setting for the hyperparameters of the "RF" model will thus be defined as `n_bootstraps` = 5 and `pctObsBootstrap` = 75%

4.1.4. Bagging neural network

To obtain the most effective hyperparameter configurations for the bagging neural network method, we will draw upon the hyperparameter values derived from the Neural Network model simulation, namely `maxiter` = 10 and `n_neurons` = 200. As before we choose the same setting for the bagging neural network method as for the random forest method, `n_bootstraps` evaluated at the values 5, 10, 20, 50, and 100, and `pctObsBootstrap` evaluated at 75%, 80%, 85%, 90%, and 95%. This approach enhances the comparability and robustness of the simulation results.

We observe similar results as in Section 4.1.3 as there are indeed not many differences between the MISEs with different combinations of the hyperparameters: the improvement from the worst-performing combination to the best-performing combination is only 2%. The MISE of the estimator using the ML model "NNForest" for various combinations of hyperparameters is illustrated in Figure 4.7.

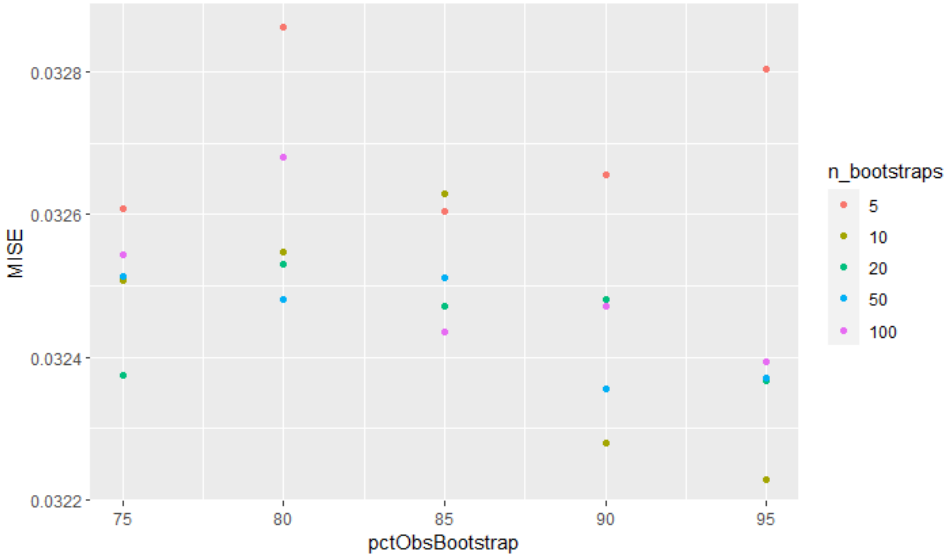

Figure 4.7: MISE and average computation time of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ using the ML model "NNForest", for various combinations of the hyperparameters `n_bootstraps` and `pctObsBootstrap`.

Figure 4.7 further illustrates a minor trend indicating that the performance of the bagging neural network method improves when a higher percentage of the original dataset is used for bootstrapping. Additionally, Figure 4.8 validates that computation time increases as the number of bootstraps rises. However, the computation time does not appear to vary significantly for an increasing percentage of observations which might have to do with the size of the original observed data.

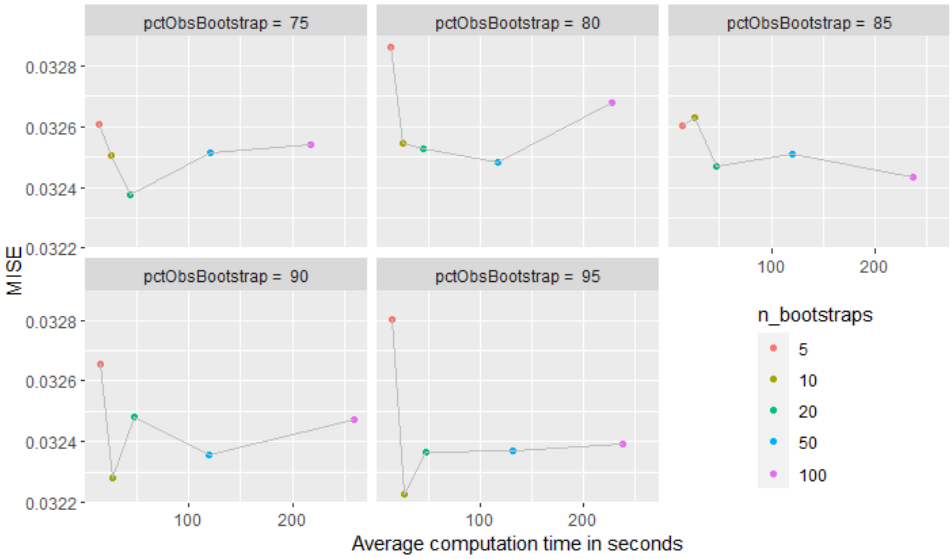

Figure 4.8: MISE and average computation time of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ using the ML model "NNForest", for various combinations of the hyperparameters `n_bootstraps` and `pctObsBootstrap`.

Given the similarity in MISE values across the hyperparameter combinations, the choice we make is not critical. Therefore, we opt for a combination with a short average computation time and a high observation percentage per bootstrap. Specifically, we select `n_bootstraps` = 10 and `pctObsBootstrap` = 95%. These settings yield a minimal MISE of 0.032, as shown in Figure 4.7, and a relative short average computation time, taking 26.6 seconds to complete, as depicted in Figure 4.8. The reference setting for the hyperparameters of the "NNForest" model will thus be defined as `n_bootstraps` = 10 and `pctObsBootstrap` = 95%

4.2. Effect of the dependence between the predictor variables X_1 and X_2

In this section, we explore the impact of the correlations between the predictor variables. Understanding this connection is important for several reasons. Firstly, it unveils how machine learning models react to varying degrees of predictor interdependencies, aiding in model selection for datasets with distinct correlation patterns

and enhancing overall performance. Additionally, this investigation offers insights into model adaptability concerning intricate data relationships, such as those found in financial forecasting. Understanding how models respond to these complex patterns can lead to more precise predictions in such scenarios.

We, therefore, analyse the behaviour of the algorithm for each machine learning technique under five different dependencies for the predictor variables, X_1 and X_2. These dependencies are $\text{Cor}(X_1, X_2) = -0.9, -0.7, -0.5, -0.25, 0, 0.25, ..., 0.9$

We will furthermore use the updated reference setting using the hyperparameters found in Section 4.1, only varying the bivariate distribution of X_1 and X_2 (ii).

The simulation results indicate that using random forests as an estimator provides the most accurate estimation of the conditional CDF for varying correlations using the aforementioned reference setting and MISE as an evaluation metric, as illustrated in Figure 4.9. Following closely is the decision tree estimator. In contrast, the Neural Network estimator, while less accurate, still provides reliable estimates, with MISE ranging from 0.024 to 0.026, underscoring its good statistical performance.

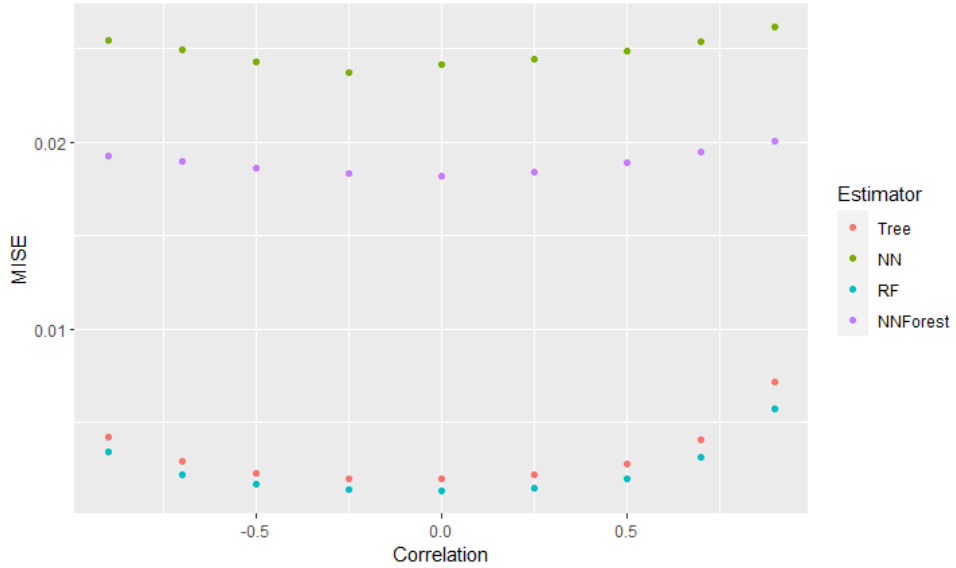

Figure 4.9: MISE of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ for each ML model, for various correlations of predictor variables X_1 and X_2.

It is worth noting that employing a bagging method for estimation enhances model performance compared to using the individual base estimator. The bagging neural network estimator demonstrates a significant improvement in estimation accuracy compared to the standalone Neural Network estimator. The random forest estimator also shows an improvement compared to decision trees, albeit to a lesser extent. This enhanced accuracy, however, comes at the expense of increased compu-

tational time, as depicted in Figure 4.10. Most notably, the bagging neural network estimator incurs a substantial increase in computation time compared to the standalone Neural Network estimator.

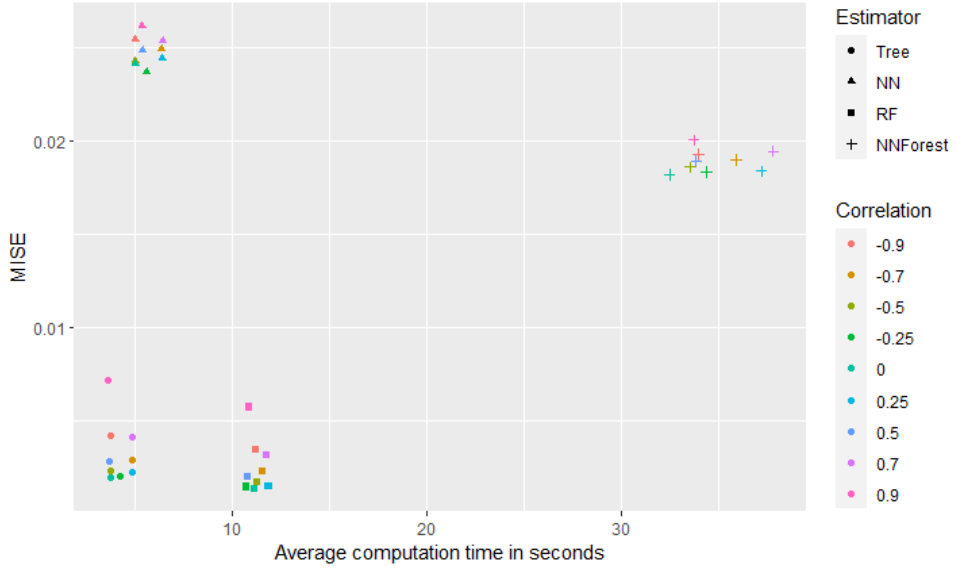

Figure 4.10: MISE and average computation time of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ for each ML model, for various correlations of predictor variables X_1 and X_2.

Finally, given that the estimators were trained on a dataset with $\text{Cor}(X_1, X_2) = 0$, they are expected to exhibit superior performance when applied to other datasets featuring low correlations among predictor variables. This observation becomes evident when examining Figure 4.9, where the curve exhibited by each estimator illustrates that for reduced levels of correlation, the estimation of the conditional CDF improves.

4.3. Effect of the conditional distribution of the response variables Y_1 and Y_2

Conducting a simulation study with different distributions for the response variables $\mathbf{Y} = (Y_1, Y_2)$ allows for assessing the robustness and generalizability of the estimator of the conditional CDF. By testing the model under various distributions of $\mathbf{Y}$, we gain insights into how well it performs across different data scenarios, revealing its sensitivity to deviations from the initially assumed normal distributed $\mathbf{Y}$ variables. This exploration can help identify potential limitations or biases in your model, provide a better understanding of its reliability in practical applications with diverse data types, and guide adjustments or enhancements to ensure more accurate and robust CDF estimations in real-world scenarios. It showcases how well we can estimate a dataset if it resembles one of these distributions.

We therefore keep the updated reference setting using the hyperparameters found in section 4.1, only varying the conditional distribution of Y_1 and Y_2 (iii). We consider the following distributions because of their use in [1]

1. **Log-normal distribution**

   $$F(y) = \Phi\left(\frac{\ln(y) - \mu}{\sigma}\right)$$

   using $\mu = X$ and $\sigma = 1$.

2. **Exponential distribution**

   $$F(y;\lambda) = \begin{cases} 1 - e^{-\lambda y} & y \geq 0, \\ 0 & y < 0. \end{cases}$$

   using $\lambda = X$.

3. **Poison distribution**

   $$F(y;k;\lambda) = \frac{\Gamma(\lfloor k+1 \rfloor, \lambda)}{\lfloor k \rfloor !},$$

   again using $\lambda = X$.

4. **Uniform distribution**

   $$F(y) = \begin{cases} 0 & \text{for } y < a, \\ \frac{y-a}{b-a} & \text{for } a \leq y \leq b, \\ 1 & \text{for } y > b. \end{cases}$$

   where $a := \min(X_1, X_2)$ and $b := \max(X_1, X_2)$. Note that Y_1 and Y_2 are generated independently from this distribution, conditionally on $\mathbf{X}$.

5. **Gamma distribution**

   $$F(y;k;\theta) = \frac{1}{\Gamma(k)}\gamma\left(k, \frac{y}{\theta}\right)$$

   Here we will again sample two X values for each Y value

From the simulations, it is apparent that all estimators perform best when the $\mathbf{Y}$ variables are log-normally distributed, as illustrated in Figure 4.11. This figure furthermore shows that estimating the uniform distribution and gamma distribution is harder for the constructed model. This might be due to the fact that each random variable in $\mathbf{Y}$ is now dependent on two random variables $\mathbf{X}$ instead of one. Optimising the hyperparameters of the estimators specifically for this type of dataset might increase estimation accuracy.

An additional observation drawn from Figure 4.11 is that, with the exception of the log-normal case, both the "RF" and "Tree" models exhibit superior performance

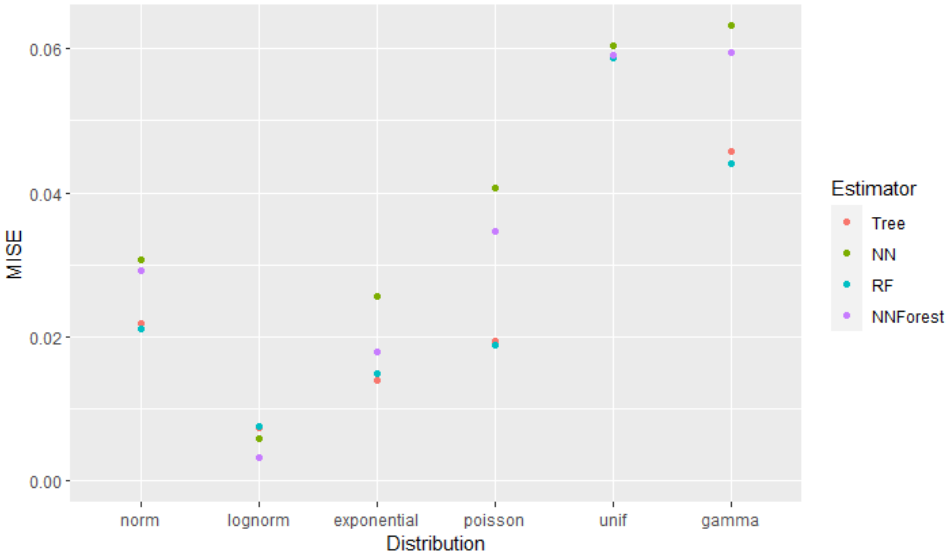

Figure 4.11: Effect of distribution on estimation of conditional CDF per machine learning estimator.

compared to the "NN" and "NNForest" models. To gain a comprehensive understanding of the Estimator that performs the best across all scenarios, we present Table 4.3, where we calculate the mean MISE and average computation time for each estimator by aggregating the values across all distributions.

Estimator	Average MISE	Average Computation Time (s)
"Tree"	0.0279	4.46
"NN"	0.0377	5.83
"RF"	0.0275	12.08
"NNForest"	0.0339	34.60

Table 4.3: The average MISE and average computation time compared over all distributions.

In Table 4.3, it is evident that both the "Tree" model and the "RF" model yield the most accurate estimations of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ as they exhibit nearly identical MISE. However, it is worth noting that the "RF" model requires nearly three times the average computation time compared to the "Tree" model, suggesting that, for various distributions of the response variables $\mathbf{Y}$, the "Tree" model is the more optimal choice.

4.4. Influence of the sample size n

Estimating the model based on varying sample sizes is important for several reasons. Firstly, it allows us to evaluate the robustness of the performance of the model, de-

termining whether it consistently performs well across different sample sizes or if its performance varies significantly. Secondly, assessing the performance of the model with different sample sizes helps us understand its generalizability to unseen data, indicating its ability to make accurate predictions on new and unseen observations.

Furthermore, examining the stability of the estimated model across different sample sizes provides insights into the reliability of the estimated parameters and relationships within the model. Lastly, understanding the influence of sample size on model performance helps in resource allocation by determining a suitable sample size that balances accurate estimation with the runtime of the algorithm, thus ensuring efficient use of resources. To thus explore the influence of the sample size n on the model estimation, a range of nine distinct values is examined: 100, 200, 500, ..., 10000, 20000, 50000.

Additionally, we will use the updated reference setting using the hyperparameters found in section 4.1 only varying the sample size n (iv).

The simulation results indicate that, for most sample sizes random forests emerge as the estimator that provides the most accurate estimation of the conditional CDF, using the MISE as an evaluation metric. This observation is depicted in Figure 4.12.

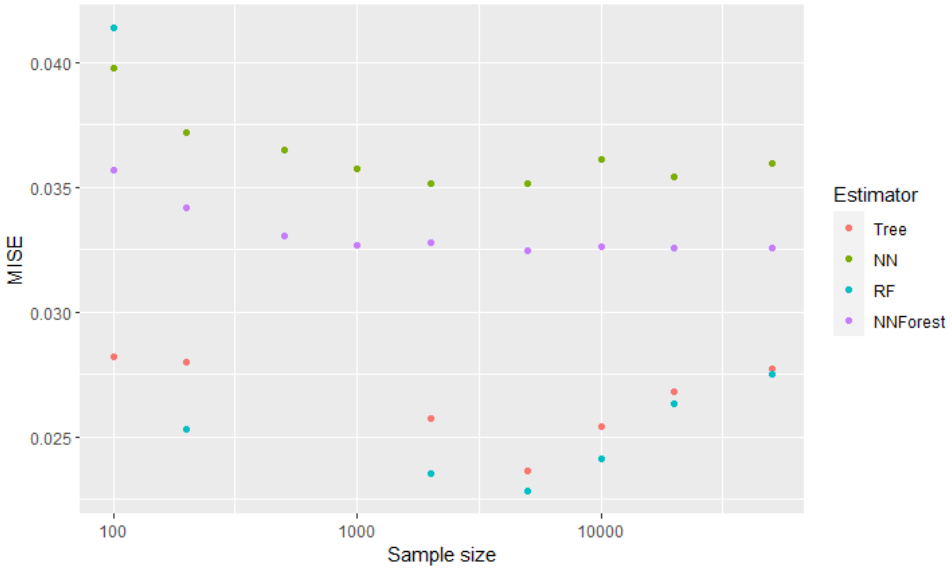

Figure 4.12: MISE of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ for each ML model, for various sample sizes n. Note that for the "Tree" and the "RF" estimators, MISE were computed using only replications that did not produce an error (see Table 4.4).

However, for sample sizes of 2000 and below, random forest appears to be a computationally unreliable estimator under the current reference setting. This ineffectiveness stems from the fact that smaller sample sizes result in the previously mentioned `Tree is too big` error. The same limitation applies when a decision tree is used as an estimator. Due to this error, estimations could not be generated

for certain sample sizes, and for others, the error occurred frequently, resulting in a lack of data points, as can be clearly seen in Figure 4.12. For a more detailed breakdown of this error concerning each estimator and sample size, we refer to Table 4.4.

Error: `Tree is too big`

Sample Size:	200	500	1000	2000
"RF"	97.5	100	100	87
"Tree"	79	100	100	0

Table 4.4: Estimated probability (%) of obtaining the error `Tree is too big` for different choices of sample sizes.

It is worth noting that the hyperparameters used for these estimators were optimized based on a dataset with a sample size of 5000. The performance curve in Figure 4.12 further highlights the superior performance for datasets approaching the size of 5000. Therefore, a potential solution to address the observed error is to select different hyperparameters tailored to smaller datasets. Implementing such adjustments would require conducting additional simulations.

Furthermore, it can be argued that the Neural Network estimator, although less accurate, consistently provides computationally reliable estimates, with MISE ranging from 0.035 to 0.040, highlighting its good statistical performance. Importantly, both the Neural Network and the superior-performing bagging neural network do not encounter errors when applied to smaller datasets.

Similar to the correlation study, the use of a bagging method for estimation proves to enhance model performance over employing the individual base estimators. However, as depicted in Figure 4.13, this improved performance comes at the cost of increased computational time. Particularly when employing a bagging neural network as the estimator for large datasets the mean computation time extends beyond one minute; for a sample size of $50,000$ its average computation time even exceeds three minutes. To enhance the clarity of Figure 4.13, we limited the range of the computation time to $[0, 50$ seconds$]$ so these points do not appear.

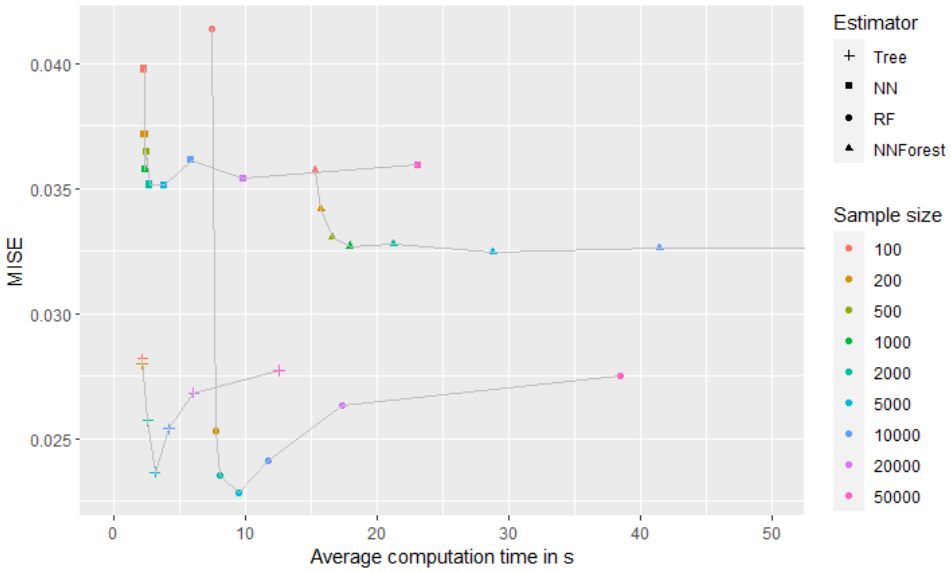

Figure 4.13: MISE and average computation time of the estimator of the conditional CDF $F_{\mathbf{Y}|\mathbf{X}}$ for each ML model, for various sample sizes n.

5

Conclusion

In this thesis, we have provided an alternative approach to estimating conditional multivariate cumulative distribution functions using machine learning and rearrangement techniques. This method, outlined in the "meta-algorithm", Algorithm 5, allows for flexible and adaptive estimation of conditional CDFs without imposing specific parametric assumptions.

By introducing a new binary random variable, denoted as $W_{\mathbf{y}} = \mathbf{1}\{\mathbf{Y} \leq \mathbf{y}\}$, the estimation problem of estimating $F_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ has been transformed into the task of estimating $P\big(W_{\mathbf{y}} = 1|\mathbf{X} = \mathbf{x}\big)$. This transformation allowed the classifier to capture the relation between the input feature vector $\mathbf{x}$ and the random variable $W_{\mathbf{y}}$, constructing an estimator of the conditional CDF. To capture the increasing nature of the true conditional CDF we used rearrangement on the estimator function.

We have developed a dedicated R package `estimCondCDF` that encapsulates the estimation method, making it accessible and user-friendly for statisticians and data analysts. This package serves as a practical resource, enabling the usage of the estimation method for different data analysis processes.

To assess the performance of the approach we conducted a simulation study. We began by optimising the hyperparameters for the involved machine learning methods used as classifiers. In these simulations, the estimation method exhibited good statistical performance for a wide range of tuning parameters.

Subsequently, we evaluated how correlations between the predictor variables X_1 and X_2 affect estimation. We found that the estimation method effectively estimates the conditional CDF across a range of predictor interdependencies, with the "RF" and "Tree" models producing the most accurate estimations. These results underscore the efficacy of the method in handling datasets with diverse correlation patterns in predictor variables.

We furthermore explored how different distributions of the response variables Y_1 and Y_2 influenced the estimation accuracy of the method. The results indicate that even when response variables do not follow a normal distribution, the models

generally performed well, giving confidence in their applicability to real-world situations with diverse data. However, in situations where two predictor variables jointly influenced one response variable, estimation accuracy decreased.

Lastly, sample size plays a big role in practical consideration. We found that smaller sample sizes could challenge the "Tree" and "RF" models due to computational constraints, emphasizing the need for customized parameters for different data sizes. In contrast, the "NN" and "NNForest" models showed resilience, providing dependable estimates across various sample sizes indicating the effectiveness of the estimation method when choosing the right machine learning technique as a classifier with the right hyperparameters.

The estimation of conditional multivariate CDFs holds significant relevance across a diverse range of data analyses. However, the computational intensity of nonparametric methods and model selection are both challenges underscoring the complexity of estimating.

Nevertheless, the methodology we have introduced not only addresses these challenges effectively but also invites the integration of new machine learning techniques as classifiers. This flexible framework offers the potential for enhancing estimation accuracy further by incorporating these new machine learning methods as well as fine-tuning hyperparameters to suit specific dataset characteristics. This adaptability enables the generation of even more precise estimations customized to the distinct requirements of each analysis. Lastly, the demonstrated adaptability and performance of the introduced estimation method suggest promising opportunities for extending its applicability to higher dimensional estimation, an exciting avenue for future research.

Bibliography

[1] R. J. Beckman and M. D. McKay. Monte carlo estimation under different distributions using the same simulation. *Technometrics*, 29(2):153–160, 1987.

[2] L. Breiman. Bagging predictors. *Machine learning*, 24:123–140, 1996.

[3] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[4] D. Ćevid, L. Michel, J. Näf, N. Meinshausen, and P. Bühlmann. Distributional random forests: Heterogeneity adjustment and multivariate distributional regression. *arXiv preprint arXiv:2005.14458*, 2020.

[5] X. Chen and Y. Fan. Estimation of copula-based semiparametric time series models. *Journal of Econometrics*, 130(2):307–335, 2006.

[6] V. Chernozhukov, I. Fernandez-Val, and A. Galichon. Improving point and interval estimators of monotone functions by rearrangement. *Biometrika*, 96(3):559–575, 2009.

[7] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 1). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

[8] A. Derumigny and J.-D. Fermanian. A classification point-of-view about conditional kendall's tau. *Computational statistics & data analysis*, 135:70–94, 2019.

[9] D. L. Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS math challenges lecture*, 1(2000):32, 2000.

[10] J. Fan and Q. Yao. *Nonlinear time series: nonparametric and parametric methods*, volume 20. Springer, 2003.

[11] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[12] C. Genest and A.-C. Favre. Everything you always wanted to know about copula modeling but were afraid to ask. *Journal of hydrologic engineering*, 12(4):347–368, 2007.

[13] G. H. Hardy, J. E. Littlewood, G. Pólya, G. Pólya, et al. *Inequalities*. Cambridge university press, 1952.

[14] W. K. Huang, A. H. Monahan, and F. W. Zwiers. Estimating concurrent climate extremes: A conditional approach. *Weather and Climate Extremes*, 33:100332, 2021.

[15] A. G. Ivakhnenko and V. G. Lapa. *Cybernetic predicting devices*. CCM Information Corporation, 1965.

[16] R. Koenker and G. Bassett Jr. Regression quantiles. *Econometrica: journal of the Econometric Society*, pages 33–50, 1978.

[17] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[18] J. Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of computation*, 35(151):773–782, 1980.

[19] L. Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 2002.

[20] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[21] M. Rosenblatt. Remarks on a multivariate transformation. *The annals of mathematical statistics*, 23(3):470–472, 1952.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[23] Y.-G. Zhao, P.-P. Li, and Z.-H. Lu. Efficient evaluation of structural reliability under imperfect knowledge about probability distributions. *Reliability Engineering & System Safety*, 175:160–170, 2018.