

BACHELOR'S THESIS COMPUTING SCIENCE



RADBOD UNIVERSITY NIJMEGEN

Demystifying Black-Box Survival Models

Adapting and comparing feature summary methods for survival analysis

Author:
Quintine Sol
s1052045

First supervisor/assessor:
Dr. Ir. Tom Claassen

Daily supervisor:
Wieske de Swart MSc

Second assessor:
Prof. Dr. Marco Loog

June 27, 2023

Abstract

In this thesis, we adapt three feature summary methods for survival analysis. Namely, the Partial Dependence Plot (PDP), Individual Conditional Expectation (ICE) and Accumulated Local Effects (ALE). These methods belong to the field of "Explainable Artificial Intelligence", which aims to demystify black-box machine learning models by explaining their inner workings. The methods are adapted, in order to apply them to the survival setting, by incorporating the survival probability and notion of time.

Apart from this, we conduct a comparative analysis of the adapted methods in terms of accuracy and efficiency. The results indicate that the accuracy of PDP and ICE is unaffected by the degree of feature correlation in the data. In contrast, the accuracy of ALE decreases when there is moderate to high feature correlation. However, in terms of execution time, ALE demonstrates better scalability than PDP and ICE.

Keywords Explainable AI, feature summary methods, survival analysis, accuracy, efficiency

Contents

1	Introduction	3
2	Preliminaries	5
2.1	Survival Analysis	5
2.1.1	Censoring	5
2.1.2	Survival and Hazard Function	6
2.1.3	Survival Models	6
2.2	Interpretability Methods	7
2.3	Partial Dependence Plot	8
2.3.1	1D-PDP	8
2.3.2	2D-PDP	9
2.4	Individual Conditional Expectation	10
2.5	Accumulated Local Effects	11
2.5.1	1D-ALE	12
2.5.2	2D-ALE	13
2.6	Evaluation of interpretability	15
3	Related Work	16
4	Adapting PDP, ICE and ALE	18
4.1	Required Changes	18
4.2	Partial Dependence Plot	18
4.2.1	1D-PDP	18
4.2.2	2D-PDP	19
4.3	Individual Conditional Expectation	19
4.4	Accumulated Local Effects	19
4.4.1	1D-ALE	19
4.4.2	2D-ALE	19
5	Methodology	21
5.1	Accuracy	21
5.1.1	Brier Score	21
5.1.2	Feature Importance Weights	22
5.1.3	Evaluation of Interpretability Methods	22
5.2	Efficiency	24
6	Experimental Setup	25
6.1	Accuracy: Experiment 1 and 2	25
6.1.1	Simulating Data	26
6.1.2	Survival Models	27
6.2	Efficiency: Experiment 3	27

6.2.1	Simulating Data	28
6.2.2	Survival Model	28
7	Results	29
7.1	Experiment 1	29
7.1.1	Ranking Plot	29
7.1.2	Faithfulness	32
7.1.3	Monotonicity	33
7.2	Experiment 2	35
7.2.1	Ranking Plot	35
7.2.2	Faithfulness	38
7.2.3	Monotonicity	39
7.3	Experiment 3	40
7.3.1	1D-PDP	40
7.3.2	ICE	40
7.3.3	1D-ALE	41
7.3.4	2D-PDP	42
7.3.5	2D-ALE	42
8	Conclusions	44
8.1	Findings	44
8.2	Limitations	45
8.3	Future Work	45
A	Interpretability Methods for Survival Analysis	49
A.1	Partial Dependence Plot	49
A.2	Individual Conditional Expectation	52
A.3	Accumulated Local Effects	53
B	Simulated Data	60
C	Evaluation Metrics for Interpretability Methods	64
D	Experiments	67
D.1	Experiment 1	68
D.2	Experiment 2	73
D.3	Experiment 3	79
E	Hardware Specifications	83

Chapter 1

Introduction

Machine learning plays an increasingly important role in society. For example, the International Data Corporation (IDC) expects global spending on artificial intelligence to reach \$154 billion in 2023, an increase of 26.9% compared to 2022 [2]. In 2026, this number is expected to surpass \$300 billion [2]. The primary factor behind this trend is the increasing ability of machine learning systems to demonstrate superhuman performance [8]. For example, an AI model trained on X-ray images was able to outperform human radiologists in detecting breast cancer [28].

However, this rise in performance is often achieved through increased model complexity, turning these systems into so-called "black boxes" [27]. As a result, their inner workings are invisible to users and even experts cannot fully understand the rationale behind their predictions [5]. This poses a key issue in the adoption of AI. Particularly in sensitive but critical domains, such as healthcare, where they could be of great value [27]; in those cases, understanding the reasoning behind a prediction is crucial, otherwise, it is hard to trust [3]. Furthermore, the General Data Protection Regulation (GDPR), imposed by the European Parliament in May 2018, forces algorithmic decision-making to be interpretable: "a data subject has the right to meaningful information about the logic involved" [1].

A research field that aims to solve this problem is Explainable Artificial Intelligence (XAI). It focuses on producing more transparent AI while maintaining a high level of performance [3]. Over the last few years, scientific interest in this field has increased. However, this mainly resulted in the development of numerous interpretability methods [3]. As pointed out by several literature reviews, there have been few works on evaluating these methods [3, 27, 41]. Therefore, the need for comparing and thus evaluating these methods arises.

A field where interpretable AI is particularly important is survival analysis. It is concerned with predicting the time until an event of interest occurs [39]. This is often used in healthcare. For example, to predict the time until a diagnosis of dementia, relapse into drug addiction or experiencing a heart attack. However, clinicians are reluctant to use high-performing survival models as they are often hard to understand and trust [9]. Therefore, more transparent but worse-performing models are used in practice [37].

One category of interpretability methods that are particularly useful for survival analysis is 'feature summary methods'. These show the importance of each feature in the prediction of the model [5]. This allows clinicians to understand and trust complex models. Furthermore, knowing the rationale behind the predictions enables the identification of risk factors. Clinicians could use this to make screening decisions or to prescribe treatment, while patients could use the information to adjust their lifestyles [25].

In this thesis, we aim to bridge the gap between survival analysis and its practical implications. Therefore, we are going to adapt and compare different feature summary methods for survival analysis. Our research question is: *What are the advantages and disadvantages of different feature summary methods for survival analysis?*

The structure of the thesis is as follows: Chapter 2 presents the necessary theoretical background to understand this thesis. In Chapter 3, related work is discussed. Chapter 4 provides the definitions of the adapted feature summary methods for survival analysis. This is followed by Chapters 5, 6 and 7, which are dedicated to the methodology, setup and results of the experiments. Chapter 8 concludes the thesis by summarising our findings, discussing the limitations of this study and providing potential directions for future research.

Chapter 2

Preliminaries

This chapter introduces the theoretical background of this research. More concretely, we discuss the key concepts of survival analysis, the selected interpretability methods and the challenges of evaluating interpretability methods.

2.1 Survival Analysis

Survival analysis is concerned with predicting the time until an event of interest occurs [39]. For example, the time until death after the diagnosis of cancer. It uses time-to-event data; each individual i is represented by a triplet (X_i, δ_i, y_i) describing (1) the observed covariates, (2) a label indicating whether the event of interest occurred or not (i.e., $\delta_i = 1$ or $\delta_i = 0$), and (3) the observed time until the event [22, 39].

2.1.1 Censoring

The main challenge in survival analysis is the fact that the event of interest may not be observed for all individuals during the study period. As a result, the survival time may not be known for each individual. This phenomenon is called *censoring* [6]. The most common scenario is right-censoring, which means that the true survival time is larger than the observed survival time [19]. Figure 2.1 illustrates an example of this.

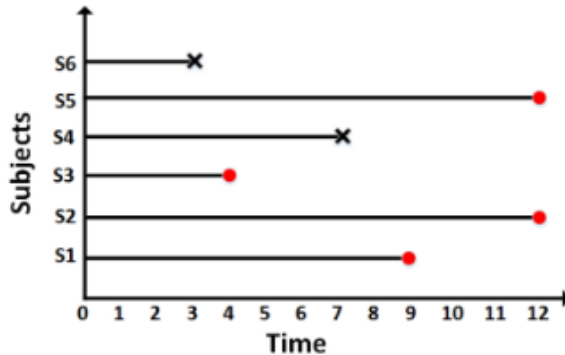


Figure 2.1: Example of survival data where a black cross indicates the occurrence of the event and a red dot indicates right-censoring [39]

Right-censoring happens when an individual does not experience the event during the study period (e.g. subjects S2 and S5 in Figure 2.1), is lost to follow-up or drops out of the study

(e.g. subjects S1 and S3) [33]. Therefore, only for individuals who experienced the event during the study period, the observed time y_i indicates the time until the event of interest occurred (also known as time T) [39]. For censored individuals, y_i represents the time until censoring (also known as time C) [39]. It is important to note that in survival analysis, censoring is often assumed to be non-informative (i.e. due to reasons unrelated to the study) [35, 39]. In this research, we also use this assumption.

2.1.2 Survival and Hazard Function

In survival analysis, there are two important quantities: the survival and the hazard function [11]. The survival function, denoted by $S(t)$, is the probability that the event of interest does not occur before a specific time t [19]. In other words, [22]

$$S(t) = \mathbb{P}(T > t), 0 < t < \infty \quad (2.1)$$

where T is a random variable representing the time until the event of interest. In the survival function, $t = 0$ corresponds to a survival probability of 1, since the event of interest has not occurred for any individual yet [19]. Over time, when t increases, the survival function monotonically decreases [39].

The hazard function, denoted by $h(t)$, is the risk of experiencing the event of interest at time t given that no event has occurred before that time [11]. Mathematically, the hazard function can be defined as [39]

$$h(t) = \lim_{\Delta t \rightarrow 0} \frac{\mathbb{P}(t \leq T < t + \Delta t | T \geq t)}{\Delta t}. \quad (2.2)$$

While the survival function decreases over time, the hazard function can have a variety of shapes [39], as depicted in Figure 2.2. However, the survival and hazard function are closely related. The hazard function corresponds to the decrease rate of the survival function; the faster the survival function decreases, the higher the hazard [11]. This is also illustrated in Figure 2.2. The survival function in Equation (2.1) can be rewritten as

$$S(t) = \exp(-H(t)) \quad (2.3)$$

where $H(t)$ is the cumulative hazard function defined as $H(t) = \int_0^t h(u) du$ [39].

2.1.3 Survival Models

A widely used survival model is the Random Survival Forest (RSF) [20]. It is constructed by fitting multiple decision trees and aggregating their predictions. Each decision tree is built on a bootstrap sample of the data and is grown by recursively partitioning the data based on specific features and splitting rules. At each split, a random subset of predictors is considered and the best predictor and threshold are chosen to maximise the separation between survival times or events. This randomisation reduces overfitting and decorrelates the trees in the ensemble.

Another popular survival model is the Cox Proportional Hazards (CPH) [7]. It is constructed using the Cox partial likelihood function, which combines information from censored and non-censored individuals. The model relies on the proportional hazards assumption. This states that the hazard ratio between two individuals, which represents the hazard rate of the first individual divided by the hazard rate of the second individual, remains constant over time. It implies that the effect of covariates on the hazard function does not change over time. This is an important difference with the Random Survival Forest since RSF does not rely on this assumption [20]. As a result, RSF is able to model time-dependent feature effects, whereas CPH is not.

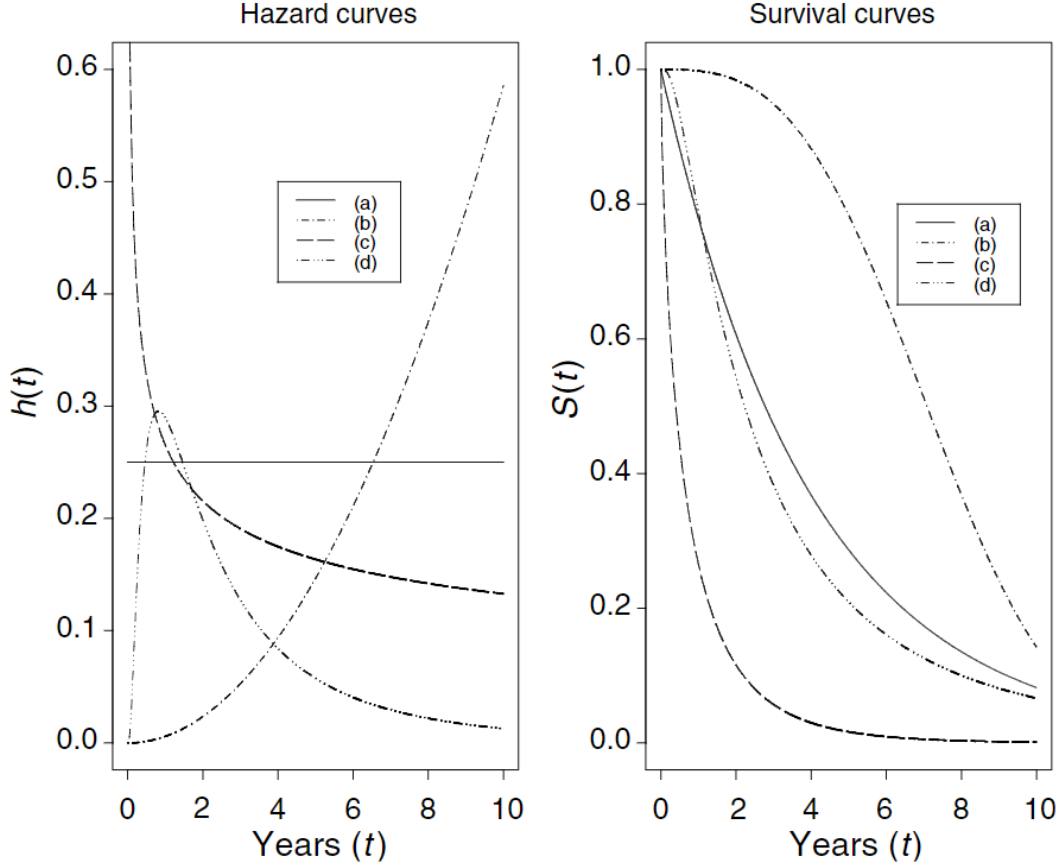


Figure 2.2: Relationship between the hazard (left figure) and survival curve (right figure) for four different cases: a, b, c and d [6]. The figures show that the hazard function corresponds to the decrease rate of the (logarithm of the) survival function.

2.2 Interpretability Methods

The goal of interpretability methods is to describe the inner workings of a black-box machine learning model in a way that is understandable to humans [12]. The category of interpretability methods we focus on in this research is 'feature summary methods'. These methods produce, as the name suggests, summary statistics for each feature [5]. This can be a single number per feature, indicating the importance of the feature on the prediction of the model, or a plot per feature, visualising how the feature influences the prediction of the model [5]. We focus on the latter category because visualisation is the easiest way to understand the inner workings of a machine learning model [3], which is the main goal of interpretability. Furthermore, in survival analysis, it is more informative to know how, rather than only knowing how much, a feature affects the prediction of the model.

This subset of interpretability methods can be further divided using the criterion local vs. global [5]. Local interpretability aims to explain the prediction of a single instance whereas global interpretability aims to explain the overall behaviour of the machine learning model [5]. We believe that global interpretability contributes most to developing trust and understanding in survival models since it enables the identification of population-level rather than instance-level risk factors and exposes possible discrimination towards certain groups of people. Therefore, in this research, we focus on global feature summary methods.

Again, we can make a further division using the criterion intrinsic vs. post hoc [5]. It determines whether interpretability is achieved through constraints on the complexity of the machine learning model (intrinsic) or by applying methods that analyse the model after training (post hoc) [5]. As explained in Chapter 1, putting constraints on the complexity of the model is likely to decrease its performance.

Therefore, in this research, we selected three popular global post-hoc feature summary visualisation methods: the Partial Dependence Plot, Individual Conditional Expectation and Accumulated Local Effects. Each method will be explained in the following sections. In Chapter 4, we explain how we apply these methods to the survival setting.

2.3 Partial Dependence Plot

The Partial Dependence Plot (PDP) was introduced by Friedman [10] in 2001. It shows the average marginal effect of one or two features on the prediction of the model, as illustrated in Figure 2.3 and 2.4. To create a PDP, the value of the feature(s) of interest is changed over its range and the predicted outcome is observed while keeping all other features constant. The predicted outcome is a class probability in classification problems and a determined value in regression [23]. In the following sections, we first explain the computation of a PDP for a single feature of interest (1D-PDP) and then for two features (2D-PDP).

2.3.1 1D-PDP

To compute the partial dependence plot of a single discrete or continuous feature the set of input features X is divided into two subsets: the feature of interest (X_j) and the remaining features ($X_{\setminus j}$), i.e. $X_j \cap X_{\setminus j} = \emptyset$ and $X_j \cup X_{\setminus j} = X$ [10]. Suppose X_j has unique values $\{x_1, x_2, \dots, x_a\}$ in the data. Then, the Partial Dependence Plot of X_j is computed as follows [15]:

1. Copy the data.
2. For $x_j \in \{x_1, x_2, \dots, x_a\}$:
 - (a) Replace the original value of X_j with constant x_j for all data instances.
 - (b) Compute the vector of predicted values from the modified copy of the data.
 - (c) Compute the average prediction to obtain $\hat{f}_{j,PDP}(x_j)$.
3. Plot the pairs $\{x_j, \hat{f}_{j,PDP}(x_j)\}$ for $x_j \in \{x_1, x_2, \dots, x_a\}$.

Equation (2.4) parametrizes the overall approach used in 1D-PDP.

$$\hat{f}_{j,PDP}(x_j) = \frac{1}{n} \sum_{i=1}^n \hat{F}(x_j, x_{i,\setminus j}) \quad (2.4)$$

where $\hat{f}_{j,PDP}(x_j)$ estimates the average marginal effect of X_j at a given value x_j , n is the number of observations in the data, $x_{i,\setminus j}$ are the original values of the features in $X_{\setminus j}$ for the i th observation, and $\hat{F}(x_j, x_{i,\setminus j})$ is the model output for the i th observation where the original value of X_j is replaced with x_j [23].

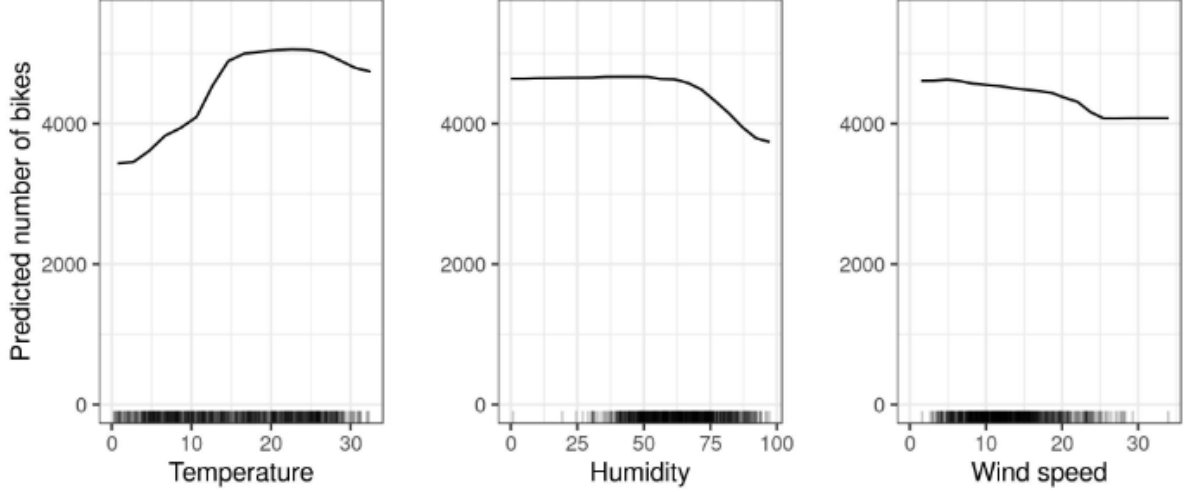


Figure 2.3: Example of 1D-PDPs depicting the relationship between the predicted number of rented bikes and the feature 'Temperature', 'Humidity' and 'Wind speed' respectively [29]. The figures show that higher temperatures (except above 25 degrees Celsius), lower humidity and lower wind speeds increase the number of rented bikes.

2.3.2 2D-PDP

A Partial Dependence Plot for two features shows the combined effect of these features on the prediction of the model [29]. The computation follows a similar procedure to the Partial Dependence Plot of a single feature. We divide the input features X into three subsets: the first feature of interest (X_j), the second feature of interest (X_l) and the remaining features ($X_{\setminus\{j,l\}}$) [10]. Suppose that X_j has unique values $\{x_1, x_2, \dots, x_a\}$ and X_l has unique values $\{x_1, x_2, \dots, x_b\}$ in the data. Then, the Partial Dependence Plot of X_j and X_l is computed as follows [15]:

1. Copy the data.
2. For $(x_j, x_l) \in \{(x_{ja}, x_{lb}) \mid x_{ja} \in \{x_1, x_2, \dots, x_a\} \text{ and } x_{lb} \in \{x_1, x_2, \dots, x_b\}\}$:
 - (a) Replace the original value of X_j with constant x_j and the original value of X_l with constant x_l for all data instances.
 - (b) Compute the vector of predicted values from the modified copy of the data.
 - (c) Compute the average prediction to obtain $\hat{f}_{\{j,l\},PDP}(x_j, x_l)$.
3. Plot $\{(x_j, x_l), \hat{f}_{\{j,l\},PDP}(x_j, x_l)\}$ for $(x_j, x_l) \in \{(x_{ja}, x_{lb}) \mid x_{ja} \in \{x_1, x_2, \dots, x_a\} \text{ and } x_{lb} \in \{x_1, x_2, \dots, x_b\}\}$.

Equation (2.5) parametrises the overall approach used in 2D-PDP.

$$\hat{f}_{\{j,l\},PDP}(x_j, x_l) = \frac{1}{n} \sum_{i=1}^n \hat{F}(x_j, x_l, x_{i,\setminus\{j,l\}}) \quad (2.5)$$

where $\hat{f}_{\{j,l\},PDP}(x_j, x_l)$ estimates the average marginal effect of X_j and X_l at given values x_j and x_l respectively, n is the number of observations in the data, $x_{i,\setminus\{j,l\}}$ are the original values of the features in $X_{\setminus\{j,l\}}$ for the i th observation, and $\hat{F}(x_j, x_l, x_{i,\setminus\{j,l\}})$ is the model output for the i th observation where the original value of X_j is replaced with x_j and the original value of X_l with x_l [23].

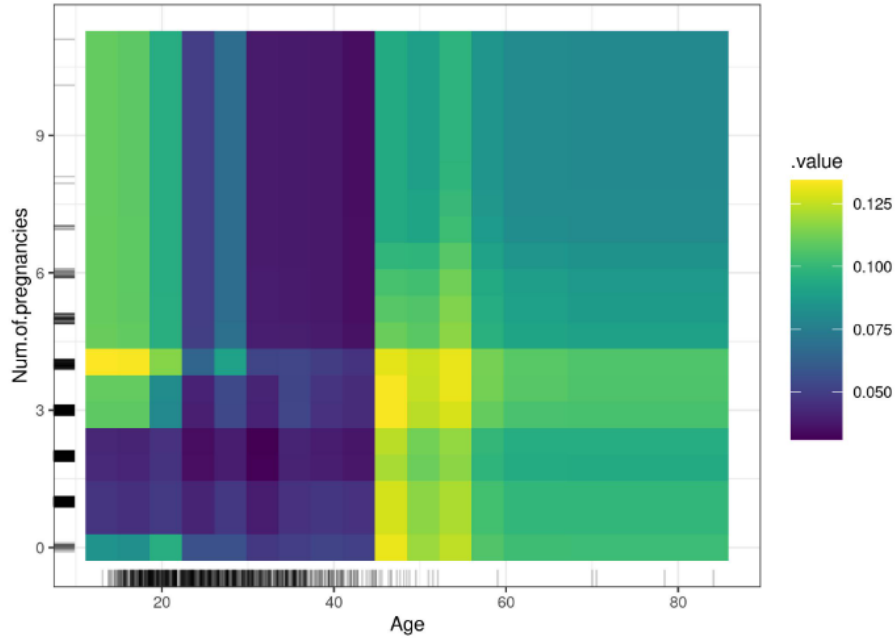


Figure 2.4: Example of 2D-PDP which depicts the relationship between the features 'Age' and 'Number of pregnancies' and the predicted cancer probability [29]. The figure shows an increase in the predicted cancer risk at age 45 and older. Furthermore, women below 25 who had 1 or 2 pregnancies have a lower cancer probability than women who had 0 or more than 2 pregnancies.

For large data sets, often a subset of the unique values of the feature(s) of interest, also referred to as grid values, is used in 1D-PDP and 2D-PDP [16]. This is also the case for Individual Conditional Expectation, which will be discussed now.

2.4 Individual Conditional Expectation

Since PDP shows the *average* marginal effect of one or two features on the prediction of the model, the complexity of the modelled relationship may be hidden, as illustrated in Figure 2.5 [27].

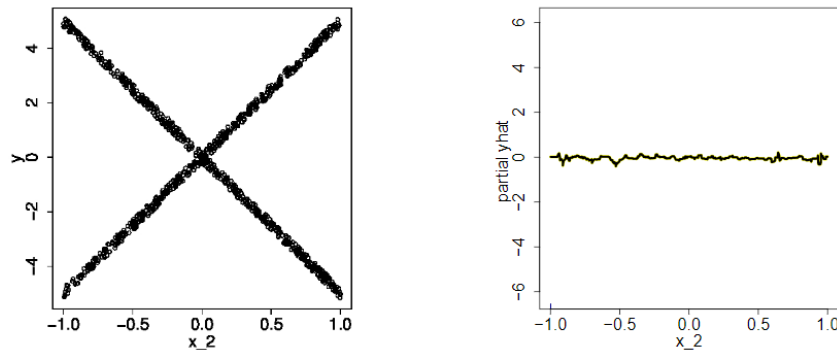


Figure 2.5: Scatterplot (left figure) and 1D-PDP (right figure) of feature X_2 and predicted value Y [13]. The figure shows that PDP incorrectly indicates that variable X_2 does not have an effect on the predicted value of Y .

To solve this issue, Goldstein et. al [13] introduced Individual Conditional Expectation (ICE). Rather than displaying the average relationship, ICE graphs the relationship between a feature and the predicted outcome for each individual instance, as illustrated in Figure 2.6. As a result, heterogeneity across observations can be identified [23]. However, due to the individual lines, ICE can only display the relationship for a single feature [29].

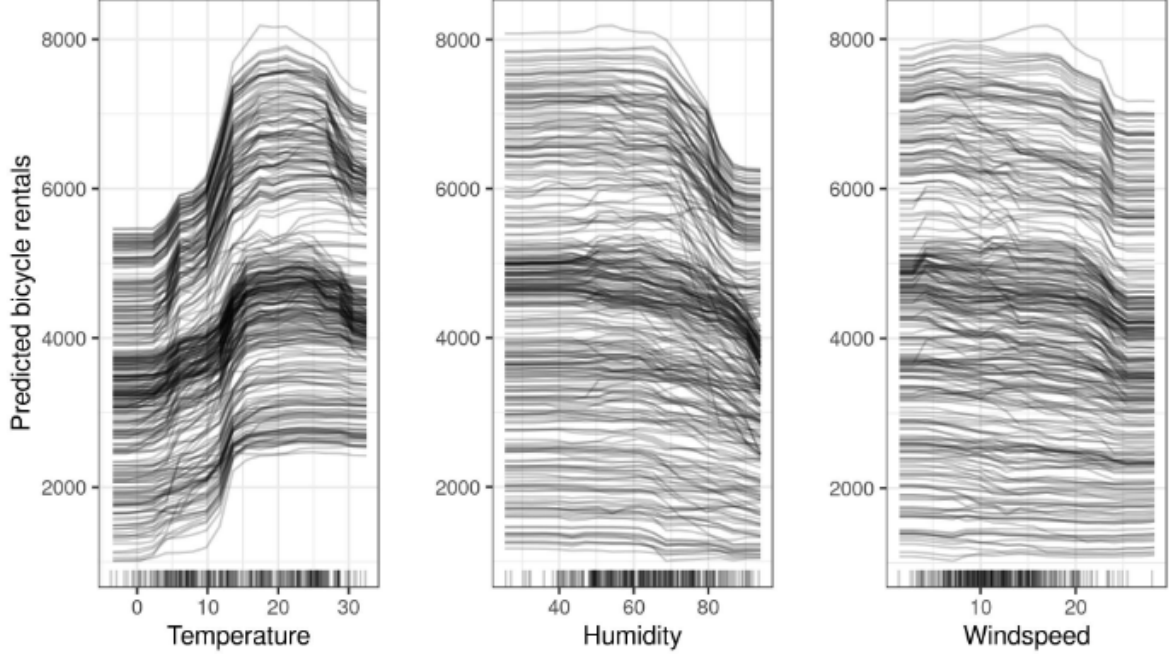


Figure 2.6: Example of ICE plots depicting the relationship between the predicted number of rented bikes and the feature 'Temperature', 'Humidity' and 'Wind speed' respectively [29]

Generating ICE plots is similar to constructing PDP plots for a single feature. The only difference is that ICE does not take the average of the predicted values (step 2c). Instead, it uses the predictions of each observation separately to draw the corresponding curve in the ICE plot. Consequently, 1D-PDP is the average of the lines in an ICE plot, as illustrated by Figure 2.3 and Figure 2.6 [29].

The overall approach of ICE can be parametrized as follows [13]:

$$\hat{f}_{i,j,ICE}(x_j) = \hat{F}(x_j, x_{i,\setminus j}) \quad (2.6)$$

where $\hat{f}_{i,j,ICE}(x_j)$ estimates the marginal effect of feature X_j at a given value x_j for observation i , $x_{i,\setminus j}$ are the original values of the features in $X_{\setminus j}$ for the i th observation, and $\hat{F}(x_j, x_{i,\setminus j})$ is the model output for the i th observation where the original value of X_j is replaced with x_j .

2.5 Accumulated Local Effects

The Accumulated Local Effects (ALE) was introduced by Apley and Zhu [4] to address another shortcoming of PDP. Namely that in the case of strongly correlated features, PDP generates synthetic data instances that are unlikely in reality. For example, as explained by Molnar et al. [29], imagine a machine learning model that predicts the price of a house based on the number of rooms and the total size of the living area. Suppose we want to draw the Partial Dependence Plot depicting the effect of the living area on the price. Then following the calculation of 1D-PDP, described in section 2.3.1, we would replace the total living area size for all instances by

- say 20 m^2 - even for houses with 10 rooms. However, this is unrealistic as the average room size would become 2 m^2 . However, these unrealistic instances are still used in the computation of the average marginal effect.

To solve this issue, ALE divides the data into intervals. Rather than averaging the predictions over the marginal distribution, it averages the predictions over the conditional distribution [29]. In other words, it takes the average predicted outcome within an interval (1D-ALE) or cell (2D-ALE) instead of over the whole range of the feature. This accounts for potential correlation between features and prevents the generation of unrealistic synthetic data [27]. In the following sections, we first describe the computation of an ALE plot for a single feature (1D-ALE), as illustrated in Figure 2.7, and then for two features (2D-ALE), as illustrated in Figure 2.8.

2.5.1 1D-ALE

Similar to 1D-PDP and ICE, the set of input features X is divided into two subsets: the feature of interest (X_j) and the remaining features ($X_{\setminus j}$) [4]. Then, the computation of the ALE plot for X_j consists of several steps. First, the sample range of X_j is partitioned into K intervals. Then, the uncentered effect of X_j is computed for each observed value x of feature X_j as follows [4]:

$$\hat{g}_{j,ALE}(x) = \sum_{k=1}^{k_j(x)} \frac{1}{n_j(k)} \sum_{\{ix_{i,j} \in N_j(k)\}} [\hat{F}(z_{k,j}, x_{i,\setminus j}) - \hat{F}(z_{k-1,j}, x_{i,\setminus j})]. \quad (2.7)$$

In this formula, $k_j(x)$ is the interval of X_j within which x lies ($k_j(x) \in \{1, 2, \dots, K\}$), $n_j(k)$ is the number of observations in interval k , $N_j(k)$ is the subset of observations that occur in k , $z_{k,j}$ and $z_{k-1,j}$ are the upper and lower bound of X_j in k respectively, $x_{i,\setminus j}$ is the value of the features in $X_{\setminus j}$ for the i th observation in k and $\hat{F}(\cdot)$ is the trained model.

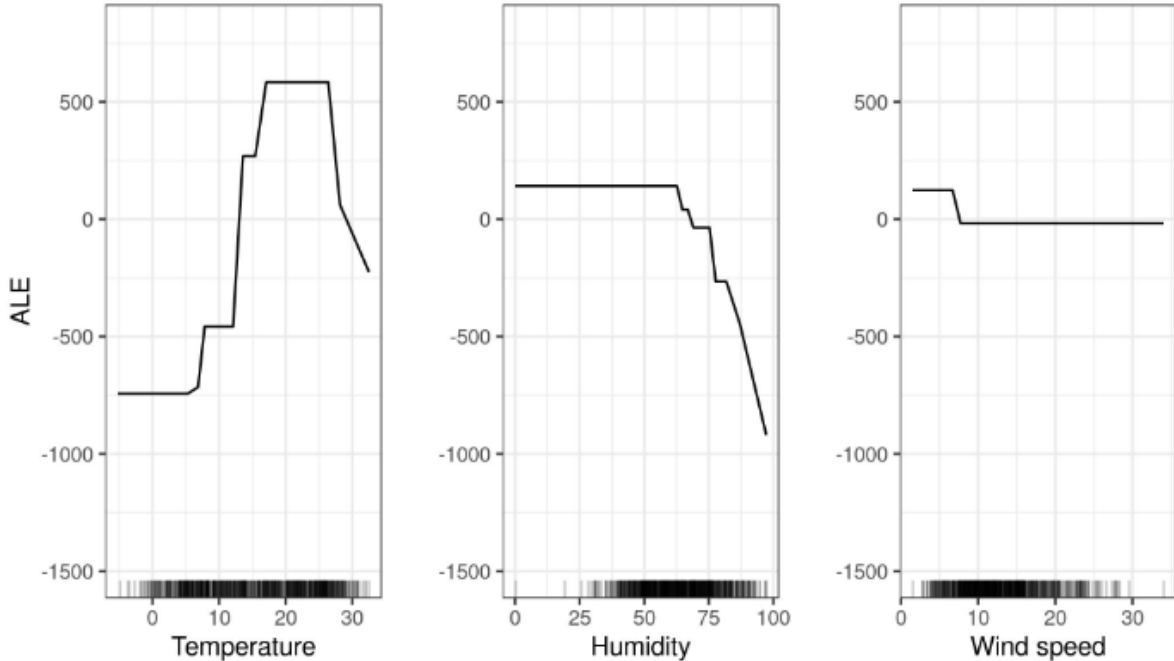


Figure 2.7: Example of 1D-ALE plots depicting the relationship between the predicted number of rented bikes and the feature 'Temperature', 'Humidity' and 'Wind speed' respectively [29]. The figures show that the temperature has the largest effect on the number of rented bikes, followed by humidity and then wind speed.

In other words, starting from the right side of the formula, the difference in prediction is calculated by first replacing the value of X_j with the upper bound and then the lower bound of X_j in the interval. This is the effect feature X_j has for an individual observation in a certain interval [29]. The sum on the right adds the effects of all instances within an interval. This is then divided by the number of observations in the interval to obtain the average difference. The left sum accumulates the average differences across all intervals. For example, the uncentered effect of a feature value that lies in the third interval is the sum of the effects of the first, second and third intervals.

Lastly, the uncentered effect $\hat{g}_{j,ALE}(x_j)$ is centered to obtain a mean effect of zero:

$$\begin{aligned}\hat{f}_{j,ALE}(x) &= \hat{g}_{j,ALE}(x) - \frac{1}{n} \sum_{i=1}^n \hat{g}_{j,ALE}(x_{i,j}) \\ &= \hat{g}_{j,ALE}(x) - \frac{1}{n} \sum_{k=1}^K n_j(k) \hat{g}_{j,ALE}(z_{k,j}).\end{aligned}\tag{2.8}$$

2.5.2 2D-ALE

Similarly to 2D-PDP, the set of input features X is partitioned into three subsets: the first feature of interest (X_j), the second feature of interest (X_l) and the remaining features ($X_{\setminus\{j,l\}}$) [4]. Then, the computation of the ALE plot for the pair of features (X_j, X_l) consists of several steps. First, both the sample range of X_j and the sample range of X_l is partitioned into K intervals, creating K^2 rectangular cells. Then, the uncentered effect of (X_j, X_l) is computed for each observed value combination (x_j, x_l) of (X_j, X_l) as follows [4]:

$$\hat{h}_{\{j,l\},ALE}(x_j, x_l) = \sum_{k=1}^{k_j(x_j)} \sum_{m=1}^{k_l(x_l)} \frac{1}{n_{\{j,l\}}(k,m)} \sum_{\{i: x_{i,\{j,l\}} \in N_{\{j,l\}}(k,m)\}} \Delta_{\hat{F}}^{\{j,l\}}(K, k, m; x_{i,\setminus\{j,l\}}) \tag{2.9}$$

In this formula, $k_j(x_j)$ is the interval of X_j within which x_j lies, $k_l(x_l)$ is the interval of X_l within which x_l lies, (k, m) denotes the index of a rectangular cell with k corresponding to the index of $k_j(x_j)$ and m corresponding to the index of $k_l(x_l)$, $n_{\{j,l\}}(k, m)$ is the number of observations in (k, m) , $N_{\{j,l\}}(k, m)$ is the subset of observations that occur in (k, m) and $\Delta_{\hat{F}}^{\{j,l\}}(K, k, m; x_{i,\setminus\{j,l\}})$ is defined as follows:

$$\begin{aligned}\Delta_{\hat{F}}^{\{j,l\}}(K, k, m; x_{i,\setminus\{j,l\}}) &= [\hat{F}(z_{k,j}, z_{m,l}, x_{i,\setminus\{j,l\}}) - \hat{F}(z_{k-1,j}, z_{m,l}, x_{i,\setminus\{j,l\}})] \\ &\quad - [\hat{F}(z_{k,j}, z_{m-1,l}, x_{i,\setminus\{j,l\}}) - \hat{F}(z_{k-1,j}, z_{m-1,l}, x_{i,\setminus\{j,l\}})]\end{aligned}\tag{2.10}$$

where $z_{k,j}$ and $z_{k-1,j}$ are the upper and lower bound of X_j in (k, m) respectively, $z_{m,l}$ and $z_{m-1,l}$ are the upper and lower bound of X_l in (k, m) respectively, $x_{i,\setminus\{j,l\}}$ is the value of the features in X_j for the i th observation in (k, m) and $\hat{F}(\cdot)$ is the fitted model.

In other words, in Equation (2.10) the difference in prediction is calculated by replacing the original values of X_j and X_l with their lower and upper bounds in the cell. This is the effect feature X_j and X_l have for an individual observation in a certain cell. Then, in Equation (2.9), starting from the right side of the formula, the sum adds the effects of all instances within a cell. This is divided by the number of instances in the cell to obtain the average difference. The two left sums accumulate the average differences across all cells. For example, the uncentered effect of feature values that lie in cell (3, 2) is the sum of the effects of cells (1, 1), (1, 2), (2, 1), (2, 2), (3, 1) and (3, 2).

An Accumulated Local Effects plot for two features only shows the interaction effect [29]. It does not include the individual effect of each feature. Therefore, the individual effects of X_j and X_l are subtracted from the uncentered effect $\hat{h}_{\{j,l\},ALE}(x_j, x_l)$ to obtain $\hat{g}_{\{j,l\},ALE}(x_j, x_l)$:

$$\begin{aligned}
& \hat{g}_{\{j,l\},ALE}(x_j, x_l) \\
&= \hat{h}_{\{j,l\},ALE}(x_j, x_l) - \sum_{k=1}^{k_j(x_j)} \frac{1}{n_j(k)} \sum_{\{i: x_{i,j} \in N_j(k)\}} [\hat{h}_{\{j,l\},ALE}(z_{k,j}, x_{i,l}) - \hat{h}_{\{j,l\},ALE}(z_{k-1,j}, x_{i,l})] \\
&- \sum_{m=1}^{k_l(x_l)} \frac{1}{n_l(m)} \sum_{\{i: x_{i,l} \in N_l(m)\}} [\hat{h}_{\{j,l\},ALE}(x_{i,j}, z_{m,l}) - \hat{h}_{\{j,l\},ALE}(x_{i,j}, z_{m-1,l})] \\
&= \hat{h}_{\{j,l\},ALE}(x_j, x_l) - \sum_{k=1}^{k_j(x_j)} \frac{1}{n_j(k)} \sum_{m=1}^K n_{\{j,l\}}(k, m) [\hat{h}_{\{j,l\},ALE}(z_{k,j}, z_{m,l}) - \hat{h}_{\{j,l\},ALE}(z_{k-1,j}, z_{m,l})] \\
&- \sum_{m=1}^{k_l(x_l)} \frac{1}{n_l(m)} \sum_{k=1}^K n_{\{j,l\}}(k, m) [\hat{h}_{\{j,l\},ALE}(z_{k,j}, z_{m,l}) - \hat{h}_{\{j,l\},ALE}(z_{k,j}, z_{m-1,l})].
\end{aligned} \tag{2.11}$$

In this formula, the individual effect of each feature can be computed in two ways. The first approach computes the difference between the uncentered effect when the value of the feature is replaced with its upper bound and lower bound in the interval. This is the individual effect of the feature for a single observation in a certain interval. Then, the effects of all instances within an interval are added together. This is divided by the number of observations in the interval to obtain the average difference. Finally, the average difference across all intervals is accumulated. For example, the individual effect of a feature value that lies in the third interval is the sum of the individual effects of the feature in the first, second and third intervals.

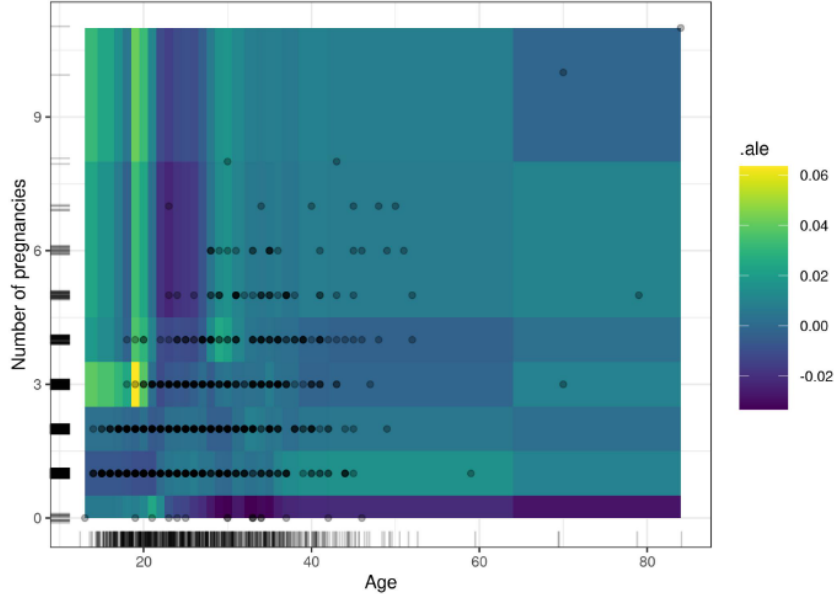


Figure 2.8: Example of a 2D-ALE plot which depicts the relationship between the features 'Age' and 'Number of pregnancies' and the predicted cancer probability [29]. The figure shows a low risk of cancer for women above 22 who had 0 pregnancies and a high risk for women below 22 who had 3 or more pregnancies.

Alternatively, the second approach computes the individual effect in each cell separately, weighted by the number of observations in the cell. Then, the individual effect within an interval is obtained by adding the individual effects of all the cells within the interval together. Similarly to the first approach, this is divided by the number of observations in the interval to obtain the average difference. Then, the average difference across all intervals is accumulated.

Lastly, the centered effect $\hat{f}_{\{j,l\},ALE}(x_j, x_l)$ is computed by subtracting the mean of $\hat{g}_{\{j,l\},ALE}(x_j, x_l)$:

$$\begin{aligned}\hat{f}_{\{j,l\},ALE}(x_j, x_l) &= \hat{g}_{\{j,l\},ALE}(x_j, x_l) - \frac{1}{n} \sum_{i=1}^n \hat{g}_{\{j,l\},ALE}(x_{i,j}, x_{i,l}) \\ &= \hat{g}_{\{j,l\},ALE}(x_j, x_l) - \frac{1}{n} \sum_{k=1}^K \sum_{m=1}^K n_{\{j,l\}}(k, m) \hat{g}_{\{j,l\},ALE}(z_{k,j}, z_{m,l}).\end{aligned}\tag{2.12}$$

2.6 Evaluation of interpretability

Currently, there is no clear consensus on how to evaluate post-hoc interpretability methods [30]. The main challenge is that there is no ground truth, as the real explanation of the model is not known [41]. Furthermore, the perceived quality of an explanation is contextual and subjective as it depends on the type of information users are interested in [41]. The type of explanations generated by different interpretability methods are also quite divergent [41]. As a result, general evaluation metrics are hard to formalise [41]. Therefore, determining the right evaluation metrics for each case is challenging and remains an open problem [27].

However, the community agrees that there are three clear goals interpretability methods should try to achieve, namely accuracy, efficiency and understandability [5]. In this research, we focus on two of the three goals, namely accuracy and efficiency. The reason for this choice will be discussed in more detail in Chapter 3.

Chapter 3

Related Work

In this chapter, we discuss the body of existing research and literature relating to this study. We also highlight the knowledge gap we aim to fill and situate our study within the broader academic context.

A recent study by Krzyżiński et al. [24] focused, just like our research, on adapting and comparing interpretability methods for survival analysis. More concretely, they introduced SurvSHAP(t), the first time-dependent feature summary method for survival models, and compared it with SurvLIME, an approach based on LIME, adapted for survival models but without a time dimension. They concluded that SurvSHAP(t) provides a more accurate ranking of variables than SurvLIME, especially for variables with a time-dependent effect. Since we are also going to adapt PDP, ICE and ALE for survival analysis, we want to evaluate the methods based on similar cases. Therefore, our experimental setup is based on this study. However, SurvSHAP(t) and SurvLIME are local explanation methods; they explain the model’s prediction for a particular observation. In contrast, the interpretability methods in our research are global; they provide explanations about the overall behaviour of the model. Therefore, we need to use different evaluation metrics that are more appropriate for global instead of local interpretability methods.

Another study by Bakhshi et al. [23] compared PDP, ICE and ALE. However, this was in the field of Real-Time Risk Assessment (RTRA). They concluded that in the absence of feature correlation, PDP accompanied by ICE is preferred, whereas in the case of highly-correlated variables, ALE is a better choice. Unfortunately, these findings were specific to RTRA models and are not generalisable to survival models. Furthermore, they evaluated the interpretability methods in terms of understandability for the end-user. However, in this research, we intend to compare the interpretability methods based on their accuracy and efficiency.

In general, we found that most studies, comparing interpretability methods, focused on local interpretability methods. For example, comparing local interpretability methods for time series forecasting models [32], on real-world healthcare data [9], in the field of attack detection [36], or on binary categorical data [38]. Furthermore, most studies use human-centred evaluation metrics. For example, satisfaction with, trust in, or ease of interpretation of provided explanations [41]. These studies solely focus on evaluating the interaction between interpretability methods and humans. Even though this is important, they fail to evaluate the interpretability methods themselves. For example, the accuracy and efficiency of the produced explanations. Moreover, as previous research suggests, we should be wary of comparing interpretability methods using merely human evaluations; it implies a strong bias towards simpler explanations which threatens transparency [18].

To the best of our knowledge, there exists no comparison of PDP, ICE and ALE for survival analysis. More importantly, there do not exist implementations of these interpretability methods that are adapted for survival models. Therefore, in this study, we aim to fill the knowledge gap by adapting and comparing PDP, ICE and ALE for survival analysis. Furthermore, unlike most previous research, we are going to compare the interpretability methods based on their accuracy and efficiency using non-human-centred evaluation metrics.

Chapter 4

Adapting PDP, ICE and ALE

As discussed in Chapter 2, the selected feature summary methods cannot be applied to survival models directly. Therefore, this chapter discusses the required changes for the survival setting and provides the definitions of the adapted methods. We only highlight the changes made in comparison to the definitions provided in Chapter 2. The complete implementations of the adapted methods can be found in Appendix A.

4.1 Required Changes

The main issue with existing implementations, such as the Partial Dependence and Individual Conditional Expectation plots from `scikit-learn`¹, is that they only allow the predicted outcome to be a class probability (for classification) through the `predict_proba()` method or a determined value (for regression) through the `predict()` method. However, in survival analysis, the predicted outcome is survival probability $S(t)$, which is obtained using the `predict_survival_function()` method. Therefore, we need to incorporate this into the definition of the feature summary methods.

Additionally, the distinction between the survival probability and the 'normal' class probability is rooted in the concept of time. Unlike the fixed class probability, an individual's survival probability can vary over time. Consequently, the influence of features may also differ over time. Hence, we need to incorporate the notion of time into the definition of the feature summary methods.

4.2 Partial Dependence Plot

4.2.1 1D-PDP

The Partial Dependence Plot for a single feature is adapted as follows:

$$\hat{f}_{j,PDP}(t, x_j) = \frac{1}{n} \sum_{i=1}^n \hat{S}(t|x_j, x_{i,\setminus j}) \quad (4.1)$$

where $\hat{f}_{j,PDP}(t, x_j)$ estimates the average marginal effect of X_j at a given value x_j and now also a given time t . Furthermore, $\hat{S}(t|x_j, x_{i,\setminus j})$ is the survival probability at time t for the i th observation where the original value of X_j is replaced with x_j .

¹https://scikit-learn.org/stable/modules/partial_dependence.html

4.2.2 2D-PDP

Similarly, the Partial Dependence Plot for two features is adapted:

$$\hat{f}_{\{j,l\},PDP}(t, x_j, x_l) = \frac{1}{n} \sum_{i=1}^n \hat{S}(t|x_j, x_l, x_{i,\setminus\{j,l\}}) \quad (4.2)$$

such that $\hat{f}_{\{j,l\},PDP}(t, x_j, x_l)$ estimates the average marginal effect of X_j and X_l at given values x_j and x_l respectively and given time t . Furthermore, $\hat{S}(t|x_j, x_l, x_{i,\setminus\{j,l\}})$ is the survival probability at time t for the i th observation where the original value of X_j is replaced with x_j and the original value of X_l is replaced with x_l .

4.3 Individual Conditional Expectation

The definition of the Individual Conditional Expectation becomes:

$$\hat{f}_{i,j,ICE}(t, x_j) = \hat{S}(t|x_j, x_{i,\setminus j}) \quad (4.3)$$

where $\hat{f}_{i,j,ICE}(t, x_j)$ estimates the marginal effect of feature X_j at a given value x_j and time t for observation i .

4.4 Accumulated Local Effects

4.4.1 1D-ALE

In the computation of an ALE plot for a single feature, the uncentered effect is now computed at a specific time t as follows:

$$\hat{g}_{j,ALE}(t, x_j) = \sum_{k=1}^{k_j(x_j)} \frac{1}{n_j(k)} \sum_{\{i: x_{i,j} \in N_j(k)\}} [\hat{S}(t|z_{k,j}, x_{i,\setminus j}) - \hat{S}(t|z_{k-1,j}, x_{i,\setminus j})] \quad (4.4)$$

where $\hat{S}(t|z_{k,j}, x_{i,\setminus j})$ and $\hat{S}(t|z_{k-1,j}, x_{i,\setminus j})$ are the survival probability for the i th observation at time t where the original value of X_j is replaced with the upper bound $z_{k,j}$ and lower bound $z_{k-1,j}$ of X_j in k respectively.

Furthermore, the computation of the centred effect becomes:

$$\begin{aligned} \hat{f}_{j,ALE}(t, x_j) &= \hat{g}_{j,ALE}(t, x_j) - \frac{1}{n} \sum_{i=1}^n \hat{g}_{j,ALE}(t, x_{i,j}) \\ &= \hat{g}_{j,ALE}(t, x_j) - \frac{1}{n} \sum_{k=1}^K n_j(k) \hat{g}_{j,ALE}(t, z_{k,j}). \end{aligned} \quad (4.5)$$

such that $\hat{f}_{j,ALE}(t, x_j)$ and $\hat{g}_{j,ALE}(t, \cdot)$ now also include the notion of time.

4.4.2 2D-ALE

Similarly, the computation of the uncentered effect of a pair of features is computed at a specific time t :

$$\hat{h}_{\{j,l\},ALE}(t, x_j, x_l) = \sum_{k=1}^{k_j(x_j)} \sum_{m=1}^{k_l(x_l)} \frac{1}{n_{\{j,l\}}(k,m)} \sum_{\{i: x_{i,\{j,l\}} \in N_{\{j,l\}}(k,m)\}} \Delta_{\hat{S}}^{\{j,l\}}(t, K, k, m; x_{i,\setminus\{j,l\}}) \quad (4.6)$$

such that $\Delta_{\hat{S}}^{\{j,l\}}(t, K, k, m; x_{i,\setminus\{j,l\}})$ is defined as follows:

$$\Delta_{\hat{S}}^{\{j,l\}}(t, K, k, m; x_{i,\setminus\{j,l\}}) = [\hat{S}(t|z_{k,j}, z_{m,l}, x_{i,\setminus\{j,l\}}) - \hat{S}(t|z_{k-1,j}, z_{m,l}, x_{i,\setminus\{j,l\}})] - [\hat{S}(t|z_{k,j}, z_{m-1,l}, x_{i,\setminus\{j,l\}}) - \hat{S}(t|z_{k-1,j}, z_{m-1,l}, x_{i,\setminus\{j,l\}})] \quad (4.7)$$

where $\hat{S}(t|z_{k,j}, z_{m,l}, x_{i,\setminus\{j,l\}})$, $\hat{S}(t|z_{k-1,j}, z_{m,l}, x_{i,\setminus\{j,l\}})$, $\hat{S}(t|z_{k,j}, z_{m-1,l}, x_{i,\setminus\{j,l\}})$ and $\hat{S}(t|z_{k-1,j}, z_{m-1,l}, x_{i,\setminus\{j,l\}})$ are the survival probability for the i th observation at time t where the original value of X_j is replaced with its lower bound $z_{k-1,j}$ or upper bound $z_{k,j}$ in interval k and the original value of X_l is replaced with its lower bound $z_{m-1,l}$ or upper bound $z_{m,l}$ in interval m .

Furthermore, the subtraction of the individual effect of each feature changes:

$$\begin{aligned} & \hat{g}_{\{j,l\},ALE}(t, x_j, x_l) \\ &= \hat{h}_{\{j,l\},ALE}(t, x_j, x_l) - \sum_{k=1}^{k_j(x_j)} \frac{1}{n_j(k)} \sum_{\{i:x_{i,j} \in N_j(k)\}} [\hat{h}_{\{j,l\},ALE}(t, z_{k,j}, x_{i,l}) - \hat{h}_{\{j,l\},ALE}(t, z_{k-1,j}, x_{i,l})] \\ &- \sum_{m=1}^{k_l(x_l)} \frac{1}{n_l(m)} \sum_{\{i:x_{i,l} \in N_l(m)\}} [\hat{h}_{\{j,l\},ALE}(t, x_{i,j}, z_{m,l}) - \hat{h}_{\{j,l\},ALE}(t, x_{i,j}, z_{m-1,l})] \\ &= \hat{h}_{\{j,l\},ALE}(t, x_j, x_l) - \sum_{k=1}^{k_j(x_j)} \frac{1}{n_j(k)} \sum_{m=1}^K n_{\{j,l\}}(k, m) [\hat{h}_{\{j,l\},ALE}(t, z_{k,j}, z_{m,l}) - \hat{h}_{\{j,l\},ALE}(t, z_{k-1,j}, z_{m,l})] \\ &- \sum_{m=1}^{k_l(x_l)} \frac{1}{n_l(m)} \sum_{k=1}^K n_{\{j,l\}}(k, m) [\hat{h}_{\{j,l\},ALE}(t, z_{k,j}, z_{m,l}) - \hat{h}_{\{j,l\},ALE}(t, z_{k,j}, z_{m-1,l})]. \end{aligned} \quad (4.8)$$

such that $\hat{g}_{\{j,l\},ALE}(t, x_j, x_l)$ and $\hat{h}_{\{j,l\},ALE}(t, \cdot, \cdot)$ are computed at a specific time t .

Lastly, the computation of the centred effect becomes:

$$\begin{aligned} \hat{f}_{\{j,l\},ALE}(t, x_j, x_l) &= \hat{g}_{\{j,l\},ALE}(t, x_j, x_l) - \frac{1}{n} \sum_{i=1}^n \hat{g}_{\{j,l\},ALE}(t, x_{i,j}, x_{i,l}) \\ &= \hat{g}_{\{j,l\},ALE}(t, x_j, x_l) - \frac{1}{n} \sum_{k=1}^K \sum_{m=1}^K n_{\{j,l\}}(k, m) \hat{g}_{\{j,l\},ALE}(t, z_{k,j}, z_{m,l}). \end{aligned} \quad (4.9)$$

such that $\hat{f}_{\{j,l\},ALE}(t, x_j, x_l)$ and $\hat{g}_{\{j,l\},ALE}(t, \cdot, \cdot)$ now also include the notion of time.

Chapter 5

Methodology

In this chapter, we discuss the methodology for comparing the adapted interpretability methods. We focus on accuracy and efficiency since these are two main goals of interpretability, as discussed in Chapter 2. First, we outline the methodology for comparing the accuracy of the methods. Then, we continue with the methodology for comparing their efficiency.

5.1 Accuracy

As explained in Chapter 2, there is no consensus on how to evaluate the accuracy of interpretability methods. However, in Chapter 3 we discussed that it is important to use non-human-centred evaluation metrics to prevent a strong bias towards simpler explanations as this threatens transparency. Therefore, we selected three evaluation metrics [24, 31] that allow us to evaluate the global interpretability methods without the involvement of humans, namely the ranking plot, faithfulness and monotonicity metric. Prior to discussing these metrics, it is necessary to explain two other concepts, namely the Brier score and feature importance weights, as they are used in the evaluation.

5.1.1 Brier Score

When evaluating the accuracy of the interpretability methods, it is important to take the accuracy of the underlying survival model into account. A well-known approach to evaluate survival models is the weighted Brier score for right-censored data. It estimates the accuracy of a survival model at a specific time t by computing the mean squared error [14]:

$$BS^c(t) = \begin{cases} \frac{1}{n} \sum_{i=1}^n \frac{(0 - \hat{S}(t, x_i))^2}{\hat{G}(y_i)} & \text{if } y_i \leq t \wedge \delta_i = 1 \\ \frac{1}{n} \sum_{i=1}^n \frac{(1 - \hat{S}(t, x_i))^2}{\hat{G}(t)} & \text{if } y_i > t \end{cases} \quad (5.1)$$

where $\hat{S}(t, x_i)$ is the predicted survival probability of instance i with covariates x_i at time t , y_i is the observed time of instance i until the event of interest or censoring, δ_i indicates whether instance i experienced the event of interest or not (i.e. $\delta_i = 1$ or $\delta_i = 0$) and $\hat{G}(t)$ is the Kaplan-Meier estimate [21] of the censoring distribution.

The overall accuracy of a survival model over a specific time interval $t_{min} \leq t \leq t_{max}$ can be calculated with the integrated Brier score, which is defined as [14]:

$$IBS = \int_{t_{min}}^{t_{max}} BS^c(t) dw(t) \quad (5.2)$$

where $w(t)$ is the weighting function defined as $w(t) = \frac{t}{t_{max}}$.

The Brier score and integrated Brier score are implemented in `scikit-survival` [34], an open-source Python package for time-to-event analysis fully compatible with `scikit-learn`. Therefore, we will use these implementations^{1,2} in this research.

5.1.2 Feature Importance Weights

The interpretability methods, adapted in Chapter 4, generate a plot which shows how each feature affects the prediction of the model. However, they do not provide concrete values - also called weights - indicating the importance of each feature. Nevertheless, this is required for the three evaluation metrics we intend to use. Therefore, we apply the feature importance measure introduced by Greenwell et al. [16]. It is based on the "flatness" of the plot generated by the interpretability method. Any variable for which the plot is "flat", is likely to be less important than those predictors whose plot varies across a wider range of the response. Therefore, the importance of any variable X_j can be computed as follows:

$$I(X_j) = \sqrt{\frac{1}{K-1} \sum_{k=1}^K [\hat{f}_j(x_j^{(k)}) - \frac{1}{K} \sum_{k=1}^K \hat{f}_j(x_j^{(k)})]^2} \quad (5.3)$$

where \hat{f}_j is the plot function and $x_j^{(k)}$ are the K unique values of X_j . Originally, this approach was only used for PDP plots. However, since ICE and ALE plots are similar, we also apply it to those. The implementation of the feature importance measure can be found in Appendix C.

It is important to note that the feature importance measure can only assign positive weights to features; it can indicate how much influence a feature has but not whether this influence is positive or negative. Apart from this, the average of an ICE plot gives the corresponding PDP plot. Therefore, the feature importance measure yields the same result for PDP and ICE. Consequently, in Chapter 7, where the results of the experiments are discussed, we group PDP and ICE together and simply refer to this as 'PDP/ICE'.

5.1.3 Evaluation of Interpretability Methods

The three evaluation metrics we are going to use to compare the accuracy of the interpretability methods will be discussed now. The implementations of the evaluation metrics can be found in Appendix C.

Ranking Plot

First, we are going to compare the coefficients of the variables in the underlying data with the assigned weights by the interpretability method as done in the study by Krzyżiński et al. [24]. This is achieved by plotting the assigned weights by the interpretability methods over time, which we call the ranking plot. Of course, the correctness of the weights does not solely depend on the accuracy of the interpretability method, it also depends on the accuracy of the survival model. For example, if the survival model does not learn the underlying relationships in the data properly, the interpretability method is unlikely to give accurate results [30]. Therefore, when assessing the accuracy of the weights, we also take the Brier scores of the survival models into account.

¹https://scikit-survival.readthedocs.io/en/stable/api/generated/sksurv.metrics.brier_score.html

²https://scikit-survival.readthedocs.io/en/stable/api/generated/sksurv.metrics.as_integrated_brier_score_scorer.html

While the ranking plot can visualise the difference between the assigned and actual weights of the features, its dependency on the accuracy of the survival model is a problem. Therefore, we also use the faithfulness and monotonicity metrics that do not suffer from this dependency.

Faithfulness

Secondly, we are going to use the faithfulness metric introduced by Oblizanov et al. [31]. It measures the correspondence between the assigned weights by the interpretability methods and the actual influence of each feature on the prediction of the model. It is calculated as the Pearson sample correlation between the weights and the change in model prediction when the feature is set to a fixed value. Since the variables are sampled from different distributions, we decided to fix each variable to its mean value.

Figure 5.1 illustrates a schematic representation of the calculation. The metric can take values from -1, indicating a complete mismatch between the weights and contributions of the features, to 1, indicating the correctness of the distribution of weights. A value of 0 indicates no correlation.

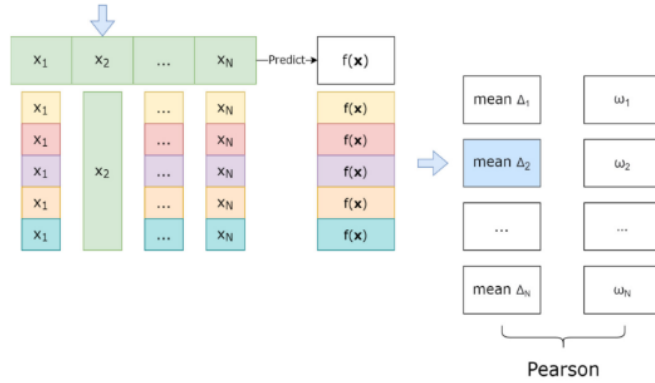


Figure 5.1: Schematic representation of the faithfulness metric [31]

Monotonicity

Thirdly, we are going to use the monotonicity metric which was also introduced by Oblizanov et al. [31]. It measures the extent to which the interpretability methods have a distortion of feature priorities (i.e. when less influential features receive a higher weight than more influential features). It is calculated by iteratively fixing features, ordered by increasing weights, obtained from the interpretability method. Again, we decided to fix each feature to its mean value. The main difference with the faithfulness metric is that monotonicity determines the cumulative effect of features, which is more reliable when features are highly correlated [31].

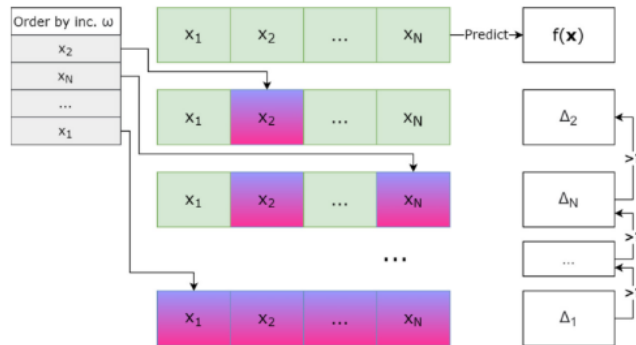


Figure 5.2: Schematic representation of the monotonicity metric [31]

A schematic representation of the calculation can be found in Figure 5.2. The metric can take values from 0, indicating a complete distortion of feature priorities, to 1, indicating more important features correctly received more weight than less important features.

5.2 Efficiency

The efficiency of the interpretability methods will be compared by measuring their execution times. It is important to note that we only include the computation time of the plot values in the time measurements; the actual plotting itself is not included. The main reason for this decision is that I/O operations are generally slower than computational operations. Therefore, as previous research suggests, excluding I/O operations from execution time measurements provides a more reliable metric for evaluating the efficiency of an algorithm [17].

Chapter 6

Experimental Setup

In this chapter, we discuss the setup of the experiments. First, we outline the setup of the accuracy experiments. Then, we continue with the efficiency experiment.

6.1 Accuracy: Experiment 1 and 2

The accuracy of the interpretability methods will be compared by conducting two experiments. In the first experiment, we assess whether the interpretability methods correctly describe variables with an insignificant, constant or time-dependent effect, provided that the underlying survival model can make use of such dependencies. In the second experiment, we assess the effect of feature correlation on the accuracy of the interpretability methods.

In both experiments, we also vary the parameters passed to the interpretability methods. In the case of PDP and ICE, this is the number of grid values (i.e. the subset of unique values) it should use in the computation. For ALE, this is the number of intervals the data instances should be divided into. In the experiments, we want to cover a wide range of possible parameter values while keeping the amount of results manageable. Therefore, we use 10, 40, 100 and 200 grid values and intervals.

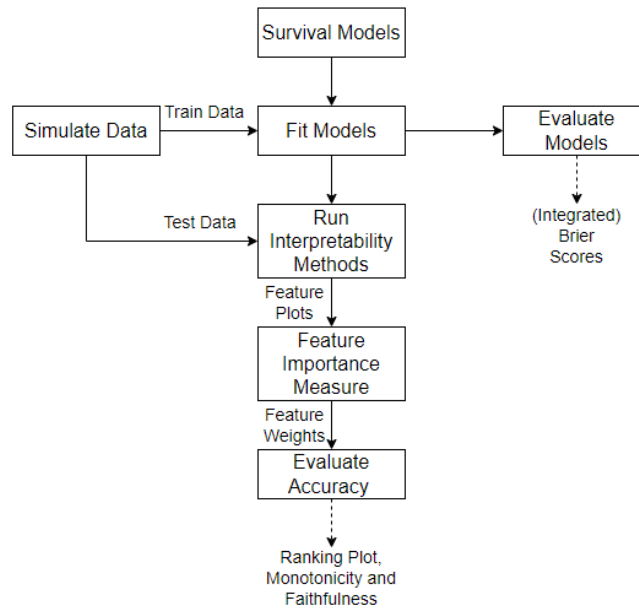


Figure 6.1: Schematic representation of Experiment 1 and 2

Figure 6.1 gives an overview of Experiments 1 and 2. Chapter 5 discussed the 'Feature Importance Weights', 'Evaluate Accuracy' and 'Evaluate Models' parts. In the following sections, we discuss the other aspects.

6.1.1 Simulating Data

For the experiments, we simulate data based on the simulation studies in the papers by Lin et al. [26] and Krzyżiński et al. [24]. The script for the data simulation was provided by Wieske de Swart and can be found in Appendix B.

Experiment 1

For the first experiment, we generate four datasets. Each dataset consists of $N = 2000$ observations. To ensure that the interpretability methods do not rely on the underlying distribution of the baseline hazard function, we use three different ones. Two are common survival distributions, namely an exponential (6.1) and Weibull distribution (6.2). The third one (6.3) is more complex, its logarithm is a fractional polynomial function. The baseline hazard functions are defined as follows:

$$h_0^I(t) = 0.08, \quad (6.1)$$

$$h_0^{II}(t) = 1.2 \cdot 0.1 \cdot t^{0.2}, \quad (6.2)$$

$$h_0^{III}(t) = \exp(-17.8 + 6.5t - 11\sqrt{t} \cdot \ln t + 9.5\sqrt{t}). \quad (6.3)$$

We generate three datasets `EXP1_exponential`, `EXP1_weibull` and `EXP1_complex` based on the following hazard function

$$h(t) = h_0(t) \cdot \exp[(-0.9 + 0.1t + 0.9 \ln t)x^{(1)} + 0.5x^{(2)} - 0.2x^{(3)} + 0.1x^{(4)} + 10^{-6}x^{(5)}] \quad (6.4)$$

where the baseline hazard function $h_0(t)$ is (6.1), (6.2) and (6.3) respectively. The coefficients in the hazard function give feature $x^{(1)}$ a time-dependent effect, $x^{(2)}$, $x^{(3)}$ and $x^{(4)}$ a constant effect, and $x^{(5)}$ an insignificant effect, representing noise. Furthermore, variables $x^{(1)}$ and $x^{(2)}$ are sampled from a binomial distribution, such that

$$\mathbb{P}(x^{(1)} = 0) = \mathbb{P}(x^{(1)} = 1) = \mathbb{P}(x^{(2)} = 0) = \mathbb{P}(x^{(2)} = 1) = 0.5.$$

The other variables are sampled from normal distributions, namely $x^{(3)} \sim \mathcal{N}(10, 2)$, $x^{(4)} \sim \mathcal{N}(20, 4)$, and $x^{(5)} \sim \mathcal{N}(0, 1)$.

Additionally, a fourth dataset `EXP1_non_td` is generated which does not contain time-dependent feature effects. It uses the baseline hazard function in (6.3). However, the time-dependent coefficient of $x^{(1)}$ in the hazard function in (6.4) is replaced with 1.

The survival function is determined based on the cumulative hazard function, namely $S(t) = \exp(-H(t))$, as discussed in section 2.1.2.

Experiment 2

For the second experiment, we generate six datasets. Each consists of $N = 2000$ observations. Since we are mainly interested in the feature correlation, we keep the hazard function as simple as possible. Therefore, for each dataset, we use the baseline hazard function in (6.1). Furthermore, in the hazard function in (6.4) we replace the time-dependent effect of $x^{(1)}$ with 1.

The sampling of variables $x^{(1)}$, $x^{(2)}$ and $x^{(5)}$ is the same as in Experiment 1. However, we create feature correlation between $x^{(3)}$ and $x^{(4)}$ by sampling from a multivariate normal distribution $\mathcal{N}(\begin{bmatrix} 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 & c \\ c & 2 \end{bmatrix})$. The correlation coefficient c is replaced with 0, 0.2, 0.4, 0.6, 0.8 and 1 to generate the six datasets `EXP2_0`, `EXP2_02`, `EXP2_04`, `EXP2_06`, `EXP2_08` and `EXP2_1` respectively.

Train and Test Data

In both experiments, the simulated data is split into train (70%) and test data (30%). The training data is used to fit the survival models and the test data is used as input for the interpretability methods. The 70-30 split is chosen to ensure a sufficient amount of data is used to train the models, so they can properly learn the underlying relationships in the data, while also enough data is set apart for the interpretability methods, so they can properly explain the models. Since the values of the features are randomly generated in the simulation script, more advanced ways of splitting the data are not required to prevent selection bias.

6.1.2 Survival Models

For the accuracy experiments, we use two popular survival models, namely Random Survival Forest [20] and Cox Proportional Hazards [7]. Both models are implemented in `scikit-survival`. Therefore, we will use these implementations^{1,2} in the experiments. It is important to note that we train separate models on each of the generated datasets. Hence, in Experiment 1 we have four CPH and RSF models and in Experiment 2 we have six CPH and RSF models.

6.2 Efficiency: Experiment 3

In the third experiment, we are going to compare the execution time of the interpretability methods. Based on the definitions of the interpretability methods in Chapter 4, the execution time seems to depend on (1) the number of instances in the data and (2) the number of grid values, in the case of PDP and ICE, and the number of intervals, in the case of ALE. Therefore, we are going to compute the execution times with varying numbers of observations and numbers of grid values/intervals used in the computation. To improve the reliability of the experiment, we repeat each measurement four times and take the average.

Figure 6.2 gives an overview of Experiment 3. Chapter 5 discussed the 'Evaluate Efficiency' part. In the following sections, we discuss the other aspects.

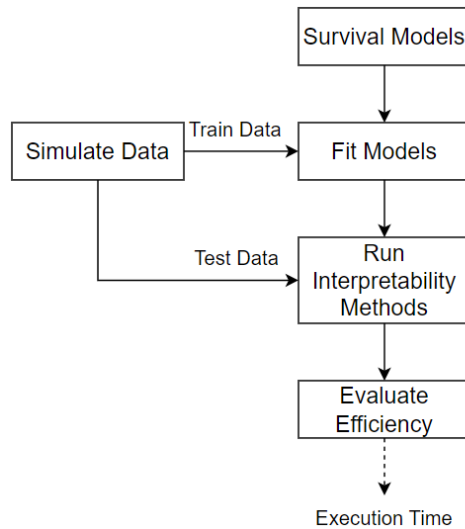


Figure 6.2: Schematic representation of Experiment 3

¹https://scikit-survival.readthedocs.io/en/stable/api/generated/sksurv.linear_model.CoxPHSurvivalAnalysis.html

²<https://scikit-survival.readthedocs.io/en/stable/api/generated/sksurv.ensemble.RandomSurvivalForest.html>

6.2.1 Simulating Data

We assume the relations between a given number of variables to be unimportant; based on the definitions in Chapter 4, the specific functional form between variables does not seem to influence the execution time of the interpretability methods. Therefore, we generate one dataset with $N = 10000$ observations using the exponential baseline hazard function in (6.1) and the hazard function in (6.4). The variables are sampled from the same distributions as in Experiment 1. We also use the same split of training and test data. The number of observations are varied by only using a subset of the test data as input to the interpretability methods.

6.2.2 Survival Model

In this experiment, we also assume the underlying survival model to be unimportant. The primary rationale behind this assumption is that the functional difference between the Cox Proportional Hazards and Random Survival Forest is not relevant for comparing execution times since the accuracy of the survival model does not influence the execution time of the interpretability method. Furthermore, while the computation time for generating survival probabilities by the survival model influences the execution time of the interpretability methods, it should not impact the relative efficiency comparison among the methods. Therefore, we only use a Random Survival Forest.

Chapter 7

Results

In this chapter, we discuss the results of the experiments. Due to the large volume of results, it is not feasible to show all of them. Therefore, we restrict ourselves to the most important ones. The code used to generate the results can be found in Appendix D.

7.1 Experiment 1

In the first experiment, we intend to determine whether the interpretability methods correctly explain variables with a time-dependent, constant or insignificant effect. Thus, we fit a separate Cox Proportional Hazards (CPH) and Random Survival Forest (RSF) model to the training part (70%) of each of the four datasets generated for Experiment 1. Hence, we have four CPH and four RSF models. The performance of the models, expressed in the Brier score, is illustrated in Figure 7.1. The corresponding integrated Brier scores are presented in Table 7.1. We can see that the integrated Brier score of CPH is smaller than RSF for all four datasets. Therefore, we know that CPH outperforms RSF in all cases. However, the difference is relatively small.

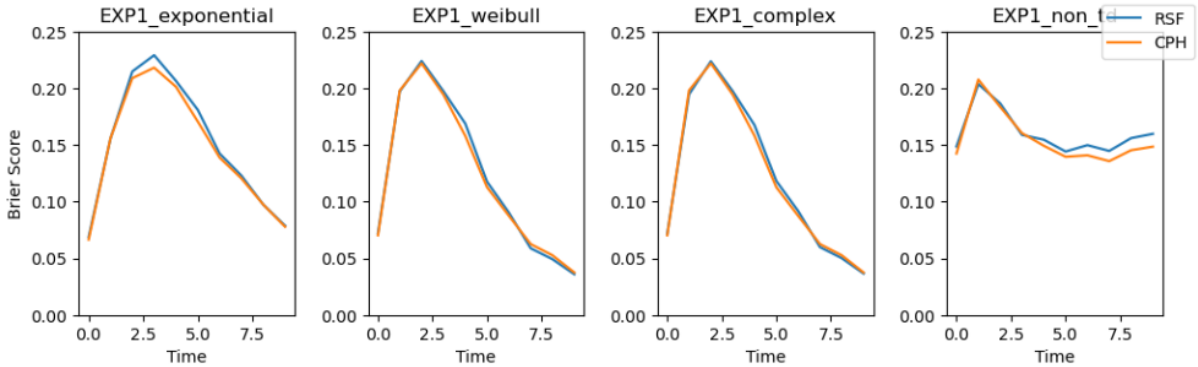


Figure 7.1: Time-dependent performance of the RSF and CPH models trained on the four datasets of Experiment 1 measured by Brier score (lower is better; 0.25 indicates random guessing [24])

7.1.1 Ranking Plot

Figure 7.2 shows the feature weights assigned by PDP, ICE and ALE, computed with the test part (30%) of the four generated datasets, at different time points. In this case, PDP and ICE used ten grid values and ALE used ten intervals in their computation. In the first row, the weights are shown for the RSF model, whereas in the second row, they are shown for the CPH

Dataset	RSF	CPH
EXP1_exponential	0.158233	0.153815
EXP1_weibull	0.128656	0.126914
EXP1_complex	0.128637	0.126914
EXP1_non_td	0.161479	0.156494

Table 7.1: Integrated Brier score of the RSF and CPH models trained on the four datasets of Experiment 1

model. Both models are trained on the `EXP1_exponential` dataset. As explained in Chapter 5, the feature importance measure yields the same result for PDP and ICE. Therefore, they are grouped together and referred to as 'PDP/ICE'.

Since the data in this experiment is synthetically generated, we know that variable $x^{(1)}$ has a time-dependent effect. The effect of $x^{(1)}$ is positive in the beginning and negative later. However, the feature importance measure we use, explained in Chapter 5, only assigns positive weights to features; it indicates how much influence a feature has but not whether the influence is positive or negative. Therefore, the time-dependent effect of $x^{(1)}$ should first decrease, and instead of becoming negative, start increasing again between time points 2 and 3. In Figure 7.2, we can see that this time-dependent effect of $x^{(1)}$ is correctly modelled by all interpretability methods for RSF but not for CPH. This can be explained by the fact that CPH assumes variable effects to be constant over time [7], whereas RSF does not [20], as discussed in Chapter 2. This shows that PDP, ICE and ALE explain variables with a time-dependent effect correctly, provided that the underlying survival model can make use of such dependencies.

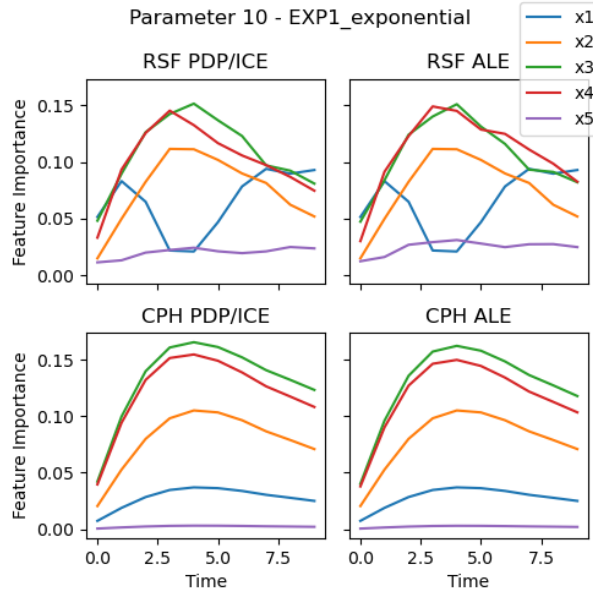


Figure 7.2: Feature weights of RSF and CPH trained on `EXP1_exponential` assigned by PDP, ICE or ALE using ten grid values or intervals

From the simulated data, we also know that variable $x^{(5)}$ represents noise. Therefore, it should get no weight assigned. In Figure 7.2, we can see that the interpretability methods correctly assign zero weight to variable $x^{(5)}$ in the CPH model. However, this is not the case for the RSF

model. Since CPH outperforms RSF, it might be the case that RSF is overfitting rather than the interpretability methods being inaccurate. Especially since learning noise is one of the main causes of overfitting [40]. Interestingly, we noticed for ALE that a rise in the number of intervals increases the weight assigned to the noise variable in the RSF model. For example, Figure 7.3 displays the feature weights assigned by the interpretability methods when two hundred grid values or intervals are used. By comparing the top-left plot in Figure 7.2 and 7.3, we can see that the weight assigned by PDP and ICE to variable $x^{(5)}$ is similar in both cases. However, for ALE we can see that variable $x^{(5)}$ gets more weight assigned when two hundred instead of ten grid intervals are used. It is unclear whether this is due to the RSF model being inaccurate, for example by overfitting, or ALE becoming less accurate as the number of intervals increases. However, based on the faithfulness of ALE, which will be discussed later, we suspect the latter.

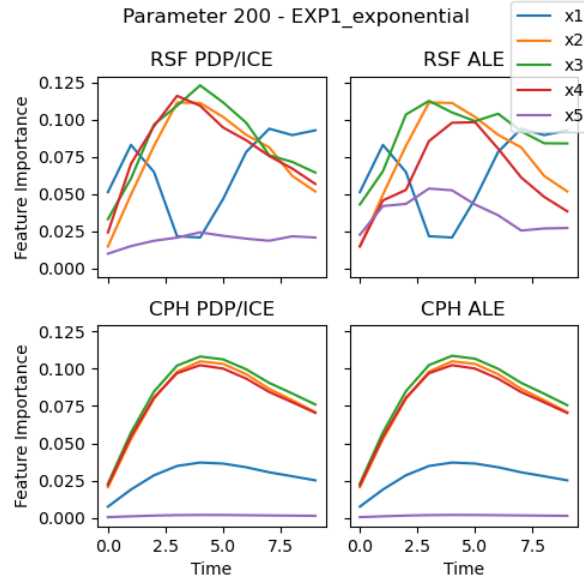


Figure 7.3: Feature weights of RSF and CPH trained on `EXP1_exponential` assigned by PDP, ICE or ALE using two hundred grid values or intervals

The models trained on `EXP1_weibull` and `EXP1_complex` showed similar results compared to `EXP1_exponential`. However, there is an interesting difference between the results of the models trained on `EXP1_complex` and `EXP1_non_td`. As explained in Chapter 6, these datasets are nearly identical, with the exception of variable $x^{(1)}$. In `EXP1_complex`, variable $x^{(1)}$ has a time-dependent effect, whereas in `EXP1_non_td`, it has a constant effect. In both datasets, all other variables have a constant effect. However, we can see in Figure 7.4, displaying the assigned weights in the models trained on `EXP1_complex`, the weights of $x^{(2)}$, $x^{(3)}$ and $x^{(4)}$ change over time. In contrast, in Figure 7.5, displaying the assigned weights in the models trained on `EXP1_non_td`, they stay relatively constant. This could indicate that the time-dependent effect of $x^{(1)}$ in `EXP1_non_td` disrupts the weights assigned by the interpretability methods to variables with a constant effect. However, it could also be the case that it is a result of changing feature priorities in the underlying survival model. Since the Brier scores of the models trained on `EXP1_non_td` and `EXP1_complex` are different, we suspect the latter.

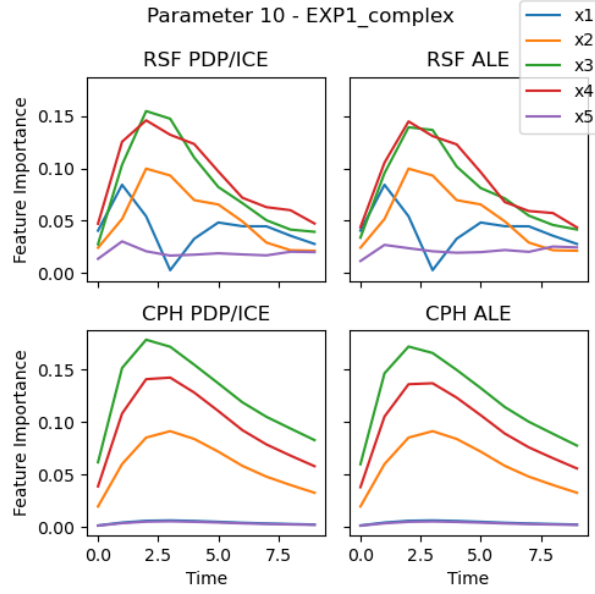


Figure 7.4: Feature weights of RSF and CPH trained on EXP1_complex assigned by PDP, ICE or ALE using ten grid values or intervals

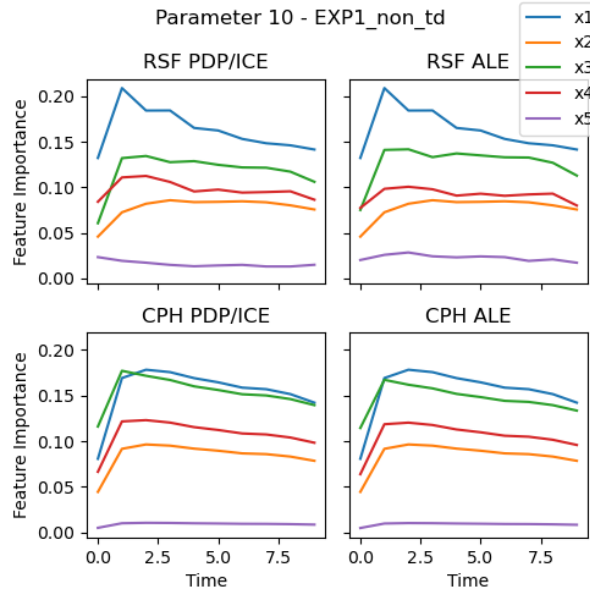


Figure 7.5: Feature weights of RSF and CPH trained on EXP1_non_td assigned by PDP, ICE or ALE using ten grid values or intervals

7.1.2 Faithfulness

In Figure 7.6, the average faithfulness of the interpretability methods over time is displayed when ten grid values or intervals are used. We can see that the faithfulness of ALE is higher than PDP and ICE for all datasets and models. This indicates that the weight of each feature assigned by ALE corresponds more to the actual influence of that feature on the prediction of the model. This is also the case when forty or a hundred grid values or intervals are used.

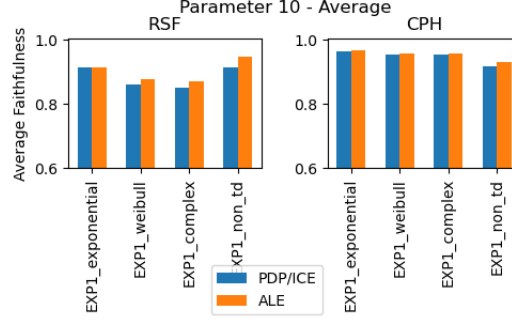


Figure 7.6: Average time-dependent faithfulness of PDP, ICE and ALE using ten grid values or intervals for RSF and CPH models trained on the four datasets of Experiment 1

However, we noticed for all interpretability methods that the faithfulness for the RSF models decreases as the number of grid values or intervals rises. In other words, the correspondence between the assigned weight by the interpretability methods and the actual influence of the features on the prediction of the survival model decreases as the number of grid values or intervals rises. This decrease is more extreme for ALE than for PDP and ICE. As a result, the faithfulness of ALE is lower than PDP and ICE for the RSF models trained on `EXP1_exponential` and `EXP1_weibull` when two hundred grid values or intervals are used, as displayed in Figure 7.7. A possible explanation for this is that, as we saw earlier, ALE incorrectly assigns more weight to noise variable $x^{(5)}$ as the number of intervals increases.

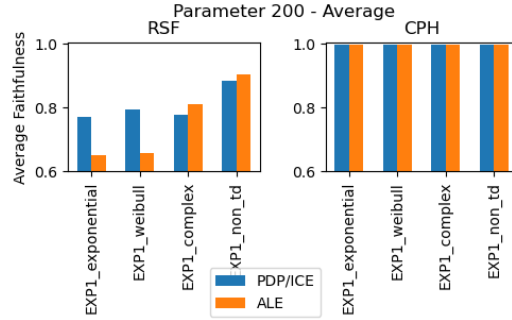


Figure 7.7: Average time-dependent faithfulness of PDP, ICE and ALE using two hundred grid values or intervals for RSF and CPH models trained on the four datasets of Experiment 1

7.1.3 Monotonicity

In Figure 7.8, the average monotonicity of the interpretability methods over time is displayed when ten grid values or intervals are used. We can see that the interpretability methods have perfect monotonicity for the CPH models. This is also the case when forty, a hundred or two hundred grid values or intervals are used. This indicates that the interpretability methods correctly assign a larger weight to more important features than to less important features in the CPH models. However, for the RSF models, we can see that ALE has higher monotonicity than PDP and ICE when ten grid values or intervals are used. However, when the number of grid values or intervals increases, this changes. The monotonicity of PDP and ICE stays relatively constant irrespective of the number of grid values, whereas that of ALE decreases as the number of intervals rises. For example, in Figure 7.9, the average monotonicity for forty grid values or intervals is displayed. We can see that PDP and ICE have higher monotonicity than

ALE for EXP1_exponential. When the number of grid values and intervals rises even more, this difference gets larger and starts to appear with the other datasets as well, as displayed in Figure 7.10. This indicates that ALE is more likely to distort feature priorities in RSF models than PDP and ICE as the number of intervals increases. We suspect that this is related to ALE assigning more weight to variable $x^{(5)}$ as the number of intervals increases, as we saw earlier.

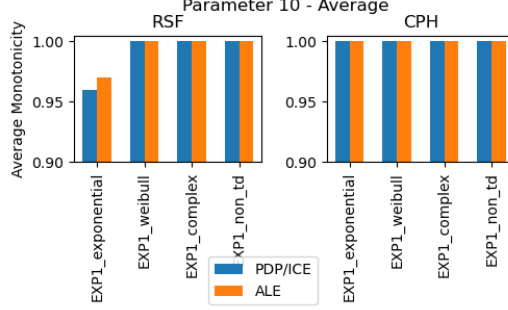


Figure 7.8: Average time-dependent monotonicity of PDP, ICE and ALE using ten grid values or intervals for RSF and CPH models trained on the four datasets of Experiment 1

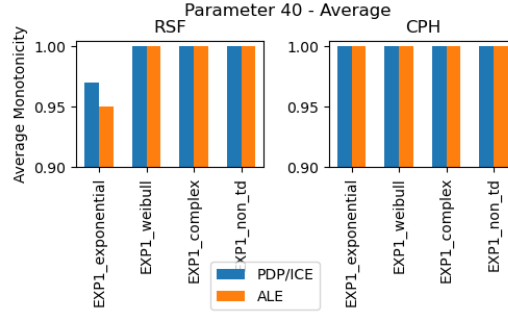


Figure 7.9: Average time-dependent monotonicity of PDP, ICE and ALE using forty grid values or intervals for RSF and CPH models trained on the four datasets of Experiment 1

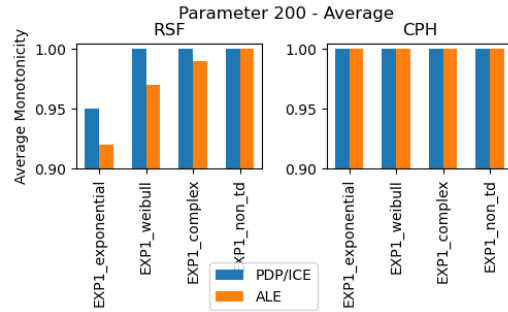


Figure 7.10: Average time-dependent monotonicity of PDP, ICE and ALE using two hundred grid values or intervals for RSF and CPH models trained on the four datasets of Experiment 1

7.2 Experiment 2

In the second experiment, we intend to determine the effect of feature correlation, created between variable $x^{(3)}$ and $x^{(4)}$, on the accuracy of the interpretability methods. Thus, we fit a CPH and RSF model to the training part (70%) of each of the six generated datasets for Experiment 2. Hence, we have six CPH and six RSF models. The performance of the models, expressed in the Brier score, is displayed in Figure 7.11. The corresponding integrated Brier scores are presented in Table 7.2. We can see in Figure 7.11 that the Brier score of RSF is higher than CPH in all cases. This is confirmed by the integrated Brier scores in Table 7.2. Therefore, we know that CPH outperforms RSF for all six datasets. Interestingly, the similarity between the Brier scores across different datasets indicates that feature correlation has little to no impact on the accuracy of the survival models.

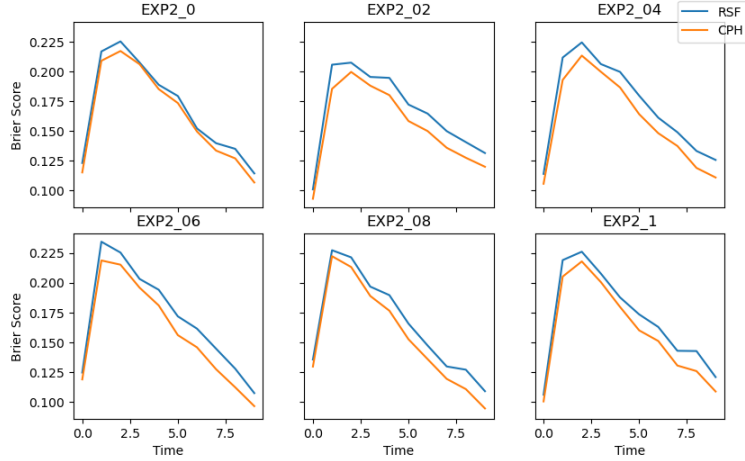


Figure 7.11: Time-dependent performance of the RSF and CPH models trained on the six datasets of Experiment 2 measured by Brier score

Dataset	RSF	CPH
EXP2_0	0.173459	0.167749
EXP2_02	0.171597	0.158801
EXP2_04	0.175838	0.163027
EXP2_06	0.175294	0.16206
EXP2_08	0.169522	0.158906
EXP2_1	0.17494	0.163784

Table 7.2: Integrated Brier score of the RSF and CPH models trained on the six datasets of Experiment 2

7.2.1 Ranking Plot

Figure 7.12 shows the weights assigned by the interpretability methods, computed using the test part (30%) of the six generated datasets, at different time points. In this case, the survival models are trained on EXP2_0, a dataset containing no feature correlation. Furthermore, the interpretability methods use ten grid values or intervals in their computation. We can see that the interpretability methods give similar weights to the variables in the CPH model. This was also the case for all other datasets containing different degrees of feature correlation.

Since the data for this experiment was synthetically generated, we know that each variable has a constant effect. This effect is highest for variable $x^{(1)}$, followed by $x^{(2)}$, $x^{(3)}$, $x^{(4)}$ and $x^{(5)}$, where $x^{(5)}$ represents noise. We can see in the bottom two graphs of Figure 7.12 that the interpretability methods assign more importance to $x^{(3)}$ than $x^{(2)}$ in the CPH model. It is unclear whether this is the actual distribution of feature importances in the underlying model or a mistake by the interpretability methods. However, as the number of grid values or intervals increases, this distortion of feature priorities is resolved. For example, in Figure 7.13, which shows the feature weights when forty grid values or intervals are used, we can see that variable $x^{(2)}$ gets more weight assigned than $x^{(3)}$.

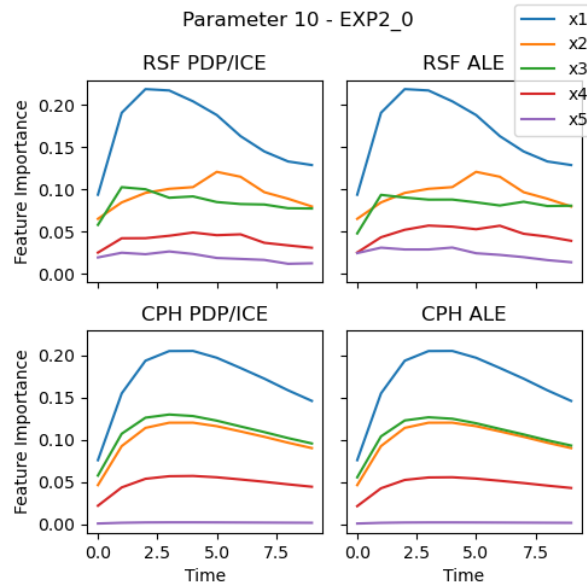


Figure 7.12: Feature weights of RSF and CPH trained on EXP2_0 assigned by PDP, ICE and ALE using ten grid values

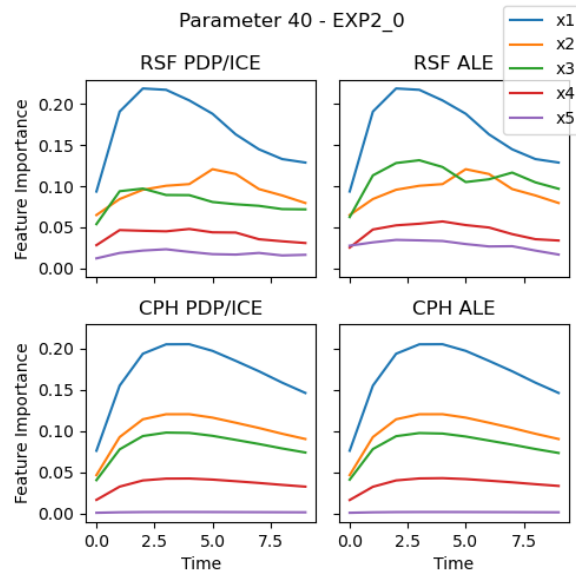


Figure 7.13: Feature weights of RSF and CPH trained on EXP2_0 assigned by PDP, ICE and ALE using forty grid values

Therefore, we suspect that the interpretability methods get more accurate for the CPH models as the number of grid values or intervals increases. This phenomenon is independent of the degree of feature correlation; it also occurred for the other datasets.

However, the degree of feature correlation seems to have little influence on the interpretability methods in the CPH models. For example, in the bottom two graphs of Figure 7.14, the assigned weights for the CPH model trained on EXP2_08, a dataset with a feature correlation of 0.8 between $x^{(3)}$ and $x^{(4)}$, is shown. If we compare this to the bottom two graphs of Figure 7.12, we can see that apart from a slight decrease in the importance of variable $x^{(2)}$ there is no significant change.

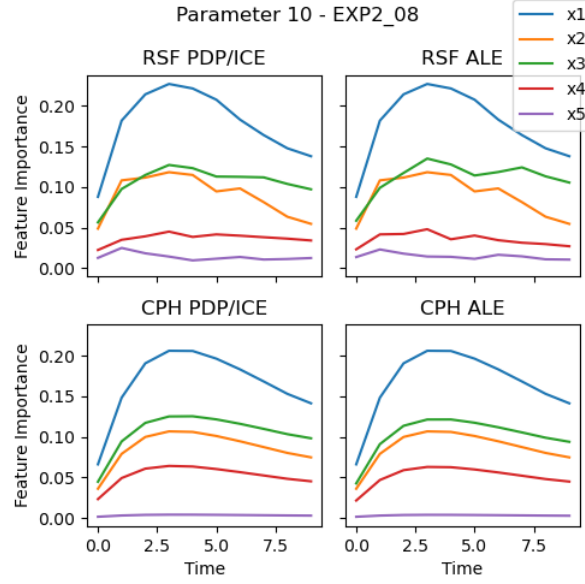


Figure 7.14: Feature weights of RSF and CPH trained on EXP2_08 assigned by PDP, ICE and ALE using ten grid values

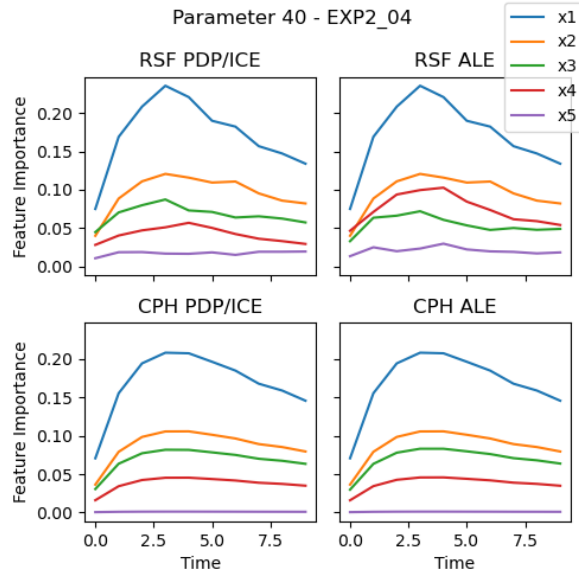


Figure 7.15: Feature weights of RSF and CPH trained on EXP2_04 assigned by PDP, ICE and ALE using forty grid values

In contrast, in the RSF models, the feature correlation seems to have a larger effect on ALE than on PDP and ICE. The assigned feature weights of PDP and ICE are relatively stable for different degrees of feature correlation, whereas ALE tends to distort feature priorities when the correlation is around 0.5. For example, in Figure 7.15, we can see that ALE incorrectly gives variable $x^{(4)}$ more weight than $x^{(3)}$ in the RSF model, whereas PDP and ICE do not. This distortion of feature priorities by ALE increases as the number of intervals rises. However, it is likely that this does not solely result from feature correlation. As we saw in Experiment 1, ALE tends to assign more weight to noise as the number of intervals increases. Apart from this, we also do not know whether the distortion of feature priorities results from an inaccurate survival model or an inaccurate interpretability method. However, as we concluded earlier from the Brier scores that feature correlation has little to no impact on the accuracy of the survival models, we suspect the latter.

7.2.2 Faithfulness

In Figure 7.16, the average faithfulness of PDP, ICE and ALE using ten grid values or intervals is displayed. The x-axis indicates the degree of feature correlation between $x^{(3)}$ and $x^{(4)}$ and the y-axis indicates the average faithfulness over time. For the CPH models, we can see that the average faithfulness of ALE is higher than PDP and ICE. In other words, the weights assigned by ALE correspond more to the actual influence of the features on the prediction of the model. This is also the case when forty, a hundred or two hundred grid values or intervals are used.

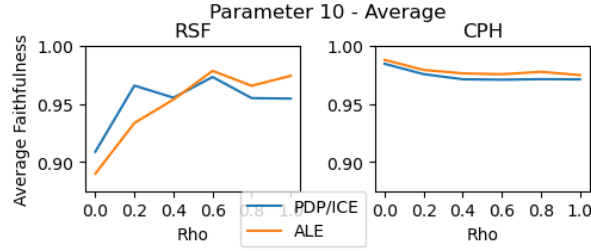


Figure 7.16: Average time-dependent faithfulness of PDP, ICE and ALE using ten grid values or intervals for RSF and CPH models trained on the six datasets of Experiment 2

For the RSF models, we noticed that the interpretability methods have similar faithfulness when the number of grid values or intervals is set to ten, as displayed in Figure 7.16. However, as the number of grid values and intervals increases there appear differences between the interpretability methods. The faithfulness of PDP and ICE stays approximately the same as the number of grid values increases. For example, in Figure 7.17, we can see the average faithfulness when two hundred grid values or intervals are used.

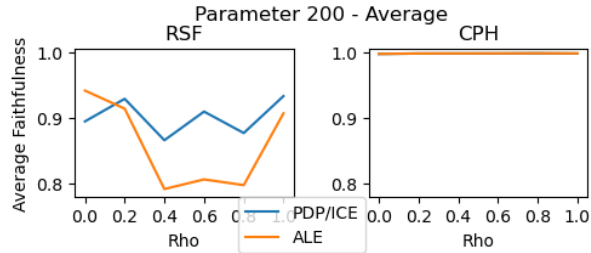


Figure 7.17: Average time-dependent faithfulness of PDP, ICE and ALE using two hundred grid values or intervals for RSF and CPH models trained on the six datasets of Experiment 2

Furthermore, we can see that the degree of feature correlation has little effect on the faithfulness of PDP and ICE. This indicates that the correspondence between the actual influence of features and their assigned weights by PDP and ICE stays approximately the same when the number of grid values and/or degree of feature correlation changes. However, the faithfulness of ALE decreases as the number of intervals increases. Especially, for models trained on datasets with a feature correlation between 0.4 and 0.8, as displayed in Figure 7.17. This indicates that the correspondence between the weights assigned by ALE and the actual influence of features decreases when the feature correlation is around 0.5 and the number of intervals increases.

7.2.3 Monotonicity

In terms of monotonicity, we found that the interpretability methods have perfect monotonicity for the CPH models irrespective of the feature correlation and the number of grid values or intervals. However, for the RSF models, we found some differences between the interpretability methods. The monotonicity of PDP and ICE stays approximately the same as the number of grid values increases. For example, in Figure 7.18 and 7.19, the average monotonicity is displayed when ten or two hundred grid values or intervals are used respectively.

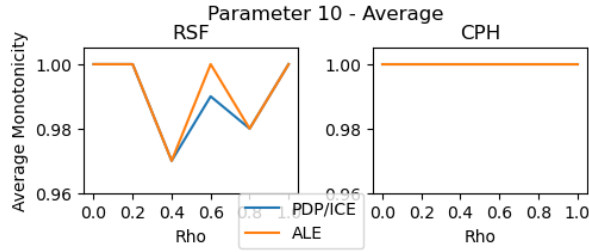


Figure 7.18: Average time-dependent monotonicity of PDP, ICE and ALE using ten grid values or intervals for RSF and CPH models trained on the six datasets of Experiment 2

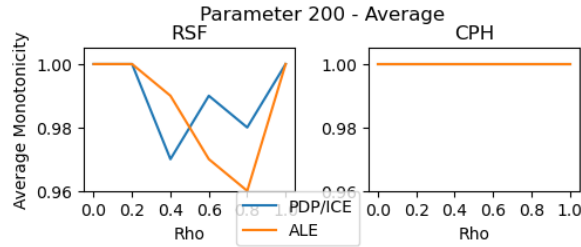


Figure 7.19: Average time-dependent monotonicity of PDP, ICE and ALE using two hundred grid values or intervals for RSF and CPH models trained on the six datasets of Experiment 2

By comparing the left plot in both figures, we can see that the monotonicity of PDP and ICE does not change. Furthermore, the degree of correlation has little effect on the monotonicity of PDP and ICE. This indicates that the likelihood of PDP and ICE distorting feature priorities is independent of the number of grid values used in the computation and the degree of feature correlation in the underlying data. In contrast, the monotonicity of ALE for models with a feature correlation between 0.4 and 0.8 decreases as the number of intervals increases. This can be seen when comparing the left plot in Figure 7.18 and 7.19. This indicates that ALE is more likely to distort feature priorities with a feature correlation between 0.4 and 0.8 as the number of intervals increases.

7.3 Experiment 3

In the third experiment, we compare the efficiency of the interpretability methods. Thus, we fit an RSF model to the dataset generated for Experiment 3. Then, the execution time of the interpretability methods is measured while varying the size of the input data and the number of grid values or intervals used in the computation. As explained in Chapter 5, the measurements only include the computation of the plot values, not the actual plotting itself. Furthermore, the hardware specifications of the machine used for measuring the execution times can be found in Appendix E.

7.3.1 1D-PDP

In Table 7.3, we can see the execution time of 1D-PDP for various combinations of the number of observations in the input data (500 until 2500) and the number of grid values (10 until 200) used in the computation. By looking at each column, we can determine the influence of the number of observations on the execution time. In each column, we see that a larger number of observations in the input data increases the execution time of 1D-PDP. However, this effect seems to be rather small; when the number of observations increases twofold, the execution time only increases by a small proportion. For example, when the number of observations increases from 500 to 1000 the execution time increases by 38% on average and when the observations rise from 1000 to 2000, the execution time increases by 55% on average.

	10	20	40	60	80	100	200
500	0.389	0.66175	1.29925	1.92575	2.559	3.2755	6.61425
1000	0.437251	0.87525	1.78975	2.6795	3.65425	4.515	9.15375
1500	0.55725	1.261	2.2875	4.0555	5.07075	5.82375	11.6035
2000	0.66925	1.4035	2.82925	4.1015	5.5495	6.894	13.854
2500	0.78	1.581	3.2005	5.08025	6.60875	9.05825	16.036

Table 7.3: Execution time (in seconds) of 1D-PDP with varying number of observations (500-2500) and number of grid values (10-200) used in the computation

When we look at the rows in Table 7.3, we can determine the influence of the number of grid values on the execution time. In every row, we see that an increase in the number of grid values increases the execution time. This effect is larger than the influence of the number of observations on the execution time; when the number of grid values doubles in size, the execution time increases approximately twofold. For example, when the grid values increase from 20 to 40, the execution time rises with 97%, from 40 to 80 with 105% and from 100 to 200 with 96% on average.

7.3.2 ICE

Table 7.4 describes the execution time of ICE for different combinations of the number of observations in the input data (500 until 2500) and the number of grid values (10 until 200) used in the computation. As one might expect, since the PDP and ICE algorithms are almost identical, we see the same relation between the execution time and the number of observations. When the number of observations doubles, the execution time increases with 39% (from 500 to 1000) and 54% (from 1000 to 2000) on average. We also see that the effect of the number of grid values is larger than the influence of the number of observations. For example, when the number of grid

values doubles, the execution time increases with 102% (from 20 to 40), 101% (from 40 to 80) and 95% (from 100 to 200) on average.

	10	20	40	60	80	100	200
500	0.313249	0.62925	1.26825	1.91	2.54325	3.293	6.53425
1000	0.443	0.8875	1.766	2.64875	3.6385	4.47425	8.99075
1500	0.56225	1.18225	2.42075	3.48125	4.7045	6.8975	11.5112
2000	0.712	1.36775	2.7685	4.10825	5.52025	6.935	13.8255
2500	0.78125	1.5735	3.16575	4.77975	6.579	8.079	17.116

Table 7.4: Execution time (in seconds) of ICE with varying number of observations (500-2500) and number of grid values (10-200) used in the computation

Interestingly, the execution time of PDP and ICE are similar. For example, PDP takes 16.036 seconds to compute with 2500 observations and 200 grid values, whereas ICE takes 17.116 seconds. This can be explained by the fact that the only difference in the computation of the PDP and ICE values is that PDP takes the average of the predictions whereas ICE does not. However, we suspect that actually plotting the ICE values will take longer than graphing the PDP values since ICE draws separate lines for each observation whereas PDP only needs to plot one line. Unfortunately, we did not measure the execution time including the plotting; we focused purely on the computation time of the values before they are plotted.

7.3.3 1D-ALE

Table 7.5 contains the execution time of 1D-ALE for various number of observations (500 until 2500) and number of intervals (10 until 200) used in the computation. Unlike PDP and ICE, the number of observations has little to no effect on the execution time. On average, the execution time increases by 3% when the observations rise from 500 to 1000 and 11% when the observations rise from 1000 to 2000. Interestingly, the influence of the observations decreases as the number of intervals gets larger. For example, in the left two columns, an increase in the number of observations always increases the execution time. In contrast, in the right two columns, an increase in the number of observations does not imply a higher execution time.

	10	20	40	60	80	100	200
500	0.343999	0.643751	1.2075	1.9465	2.37225	2.99225	6.14525
1000	0.381001	0.68425	1.285	1.889	2.49275	3.0295	6.06975
1500	0.42875	0.71725	1.34575	2.01225	2.49275	3.36075	7.727
2000	0.49325	0.7555	1.416	1.9535	2.548	3.22075	7.68275
2500	0.7655	0.795501	1.3975	2.008	2.811	3.23675	6.09525

Table 7.5: Execution time (in seconds) of 1D-ALE with varying number of observations (500-2500) and number of intervals (10-200) used in the computation

The relation between the execution time and the number of intervals is similar to 1D-PDP and ICE. Whenever the number of intervals doubles, from 20 to 40, 40 to 80 or 100 to 200,

the execution time increases by 85%, 91% and 112% respectively. Furthermore, with a small number of observations and number of grid values/intervals, the three interpretability methods have similar execution times. For example, 0.389, 0.313 and 0.344 for five hundred observations and ten grid values or intervals. However, compared to 1D-PDP and ICE, 1D-ALE scales better. As the number of observations and the number of grid values or intervals increase, 1D-ALE becomes increasingly faster than 1D-PDP and ICE. For example, with 1500 observations and 80 grid values/intervals, 1D-ALE is 2.03 times faster than 1D-PDP and 1.89 times faster than ICE. However, with 2500 observations and 200 grid values/intervals, 1D-ALE is 2.6 times faster than 1D-PDP and 2.8 times faster than ICE.

7.3.4 2D-PDP

In Table 7.6, we can see the execution time of 2D-PDP for different numbers of observations (500 until 2500) and numbers of grid values (5 until 40). It is important to note that we used smaller grid sizes than in the preceding measurements of 1D-PDP, ICE and 1D-ALE. The main reason for this choice was that the execution time of 2D-PDP was already high for a relatively small number of grid values. Therefore, it was not feasible to use the same number of grid values as in previous measurements.

	5	10	20	30	40
500	0.6515	2.70225	11.0708	25.576	45.7858
1000	0.9015	3.7095	15.3098	35.7798	61.9257
1500	1.108	4.63125	20.1963	43.4847	79.0197
2000	1.32525	5.5325	23.959	51.9077	92.8818
2500	1.5775	7.881	26.5565	60.3913	107.05

Table 7.6: Execution time (in seconds) of 2D-PDP with varying number of observations (500-2500) and number of grid values (5-40) used in the computation

When looking at the columns, we see that a rise in the number of observations increases the execution time. This effect is the same as for 1D-PDP when the observations go from 500 to 1000, namely an increase of 38% in execution time. However, when the observations go from 1000 to 2000, there is an increase of 50% whereas 1D-PDP has an increase of 55%. It is likely that this difference results from measurement errors.

By looking at the rows, we can determine the effect of the number of grid values on the execution time. In every row, we can see that an increase in grid values has a relatively large effect on the execution time. For example, when the grid values change from 5 to 10, the execution time increases by 332%, from 10 to 20 by 306% and from 20 to 40 by 300%. As one might expect, the effect of grid values for 2D-PDP is larger than for 1D-PDP. This can be explained by the fact that 1D-PDP goes over each grid value, but 2D-PDP needs to go over each combination of grid values for the two features. Therefore, the execution time does not grow linearly, like for 1D-PDP, but quadratically with the number of grid values.

7.3.5 2D-ALE

In Table 7.7, we can see the execution time of 2D-ALE for different numbers of observations (500 until 2500) and numbers of intervals (5 until 40). If we compare it with the execution times of

2D-PDP in Table 7.6, we can see that 2D-PDP is faster when computing with a small number of grid values/intervals. For example, the execution time of 2D-PDP is on average 1.5 times faster than 2D-ALE when there are five grid values/intervals. However, 2D-ALE scales better than 2D-PDP; the number of observations and intervals have a smaller effect on the execution time of 2D-ALE compared to 2D-PDP. When the number of observations increases from 500 to 1000, the execution time rises by 35% and from 1000 to 2000 the execution time increases by 21%. In contrast, for 2D-PDP this is 38% and 50% respectively. Additionally, when the number of intervals go from 5 to 10, the execution time increases by 278%, from 10 to 20 by 253% and from 20 to 40 by 140%. For 2D-PDP, each of these values was over 300%. As a result, 2D-ALE gets faster than 2D-PDP for larger numbers of grid values/intervals. For example, the execution time of 2D-ALE is on average 1.4 times faster than 2D-PDP when there are 40 grid values or intervals.

	5	10	20	30	40
500	1.47375	5.9825	16.8815	24.0745	25.2403
1000	1.56425	5.9125	22.1613	38.6725	45.4547
1500	1.6965	8.0525	23.0213	43.5537	61.052
2000	1.70225	6.14	25.6675	47.4025	70.1193
2500	2.271	6.13425	24.6715	50.0563	75.3467

Table 7.7: Execution time (in seconds) of 2D-ALE with varying number of observations (500-2500) and number of intervals (5-40) used in the computation

Chapter 8

Conclusions

In conclusion, we adapted PDP, ICE and ALE for survival analysis by incorporating the survival probability and the notion of time. Furthermore, we compared the accuracy and efficiency of the adapted feature summary methods by conducting three experiments.

8.1 Findings

In the first experiment, we have seen that PDP, ICE and ALE correctly explain variables with a time-dependent effect, provided that the underlying survival model can make use of such dependencies. However, ALE tends to assign more weight to noise as the number of intervals increases. Furthermore, the faithfulness and monotonicity of PDP and ICE exceed that of ALE as the number of grid values or intervals increases.

In the second experiment, we have seen that a moderate to high feature correlation has a negative effect on ALE. Especially when the number of intervals increases. In contrast, feature correlation has no significant effect on PDP and ICE. As a result, the faithfulness and monotonicity of PDP and ICE stay relatively constant independent of the number of grid values or degree of feature correlation. However, the faithfulness and monotonicity of ALE, especially for these moderate to high feature correlations, decrease as the number of intervals increases.

In the third experiment, we have seen that the execution time of 1D-PDP and ICE is similar. Furthermore, with a small number of observations and intervals, the execution time of 1D-ALE is also similar. However, 1D-ALE gets increasingly faster than 1D-PDP and ICE as the number of observations or the number of grid values/intervals increases. Apart from this, by comparing 2D-PDP and 2D-ALE we have seen that 2D-PDP is faster for smaller grid values, whereas 2D-ALE scales better for larger intervals.

Using these findings, we can address the research question: *What are the advantages and disadvantages of different feature summary methods for survival analysis?* The answer is that while PDP and ICE offer the advantage of greater accuracy, exhibiting robustness against feature correlation and changing method parameters, ALE stands out for its efficiency, demonstrating better scalability. Thus, when there is little to no feature correlation, ALE is the preferred method. In this case, it is important to keep the number of intervals relatively low to maintain high accuracy. On the other hand, with moderate to high feature correlation or when the degree of feature correlation is unknown, PDP and ICE are the preferred choices.

8.2 Limitations

A limitation of our research is the small number of interpretability methods that were compared. In this research, we only compared three feature summary methods: PDP, ICE and ALE. The choice to limit our comparison to these methods was primarily due to time constraints imposed by the scope and duration of the research project. However, we also had to select interpretability methods that give the same type of output. Otherwise, finding appropriate evaluation metrics would be hard or even infeasible. As explained in Chapter 2, formalising general evaluation metrics for different types of explanations remains an open problem.

Another limitation is the small number of survival models used in the experiments. This is mainly a limitation because the accuracy of the feature summary methods seems to depend on the underlying survival model. For example, as highlighted in Chapter 7, the interpretability methods performed better on the Cox Proportional Hazards than on the Random Survival Forest models. Therefore, even though our experiments yielded valuable insights, these findings may not be generalisable to other survival models.

The last limitation is the substantial amount of critical components we had to implement ourselves. This includes the faithfulness metric, monotonicity metric, and feature importance measure. This was mainly due to the absence of available open-source implementations. Therefore, given the limited time frame, this extra work prevented us from conducting more extensive experiments.

8.3 Future Work

Future studies could focus on addressing the aforementioned limitations of this research. For example, as an extension to this study, other interpretability methods could be adapted and compared for survival analysis. For example, surrogate intrinsically interpretable models which explain a black-box model by approximating it with an intrinsically interpretable model [5]. Apart from this, future research could incorporate a more extensive selection of survival models. This could provide a more comprehensive understanding of the strengths and limitations of the interpretability methods. In addition, in this research, we focused on two of the three goals of interpretability, namely accuracy and efficiency. Future studies could compare the interpretability methods based on the third goal, namely understandability.

Bibliography

- [1] General Data Protection Regulation (GDPR), 5 2016.
- [2] Worldwide Spending on AI-Centric Systems Forecast to Reach 154 Billion in 2023, 3 2023.
- [3] Amina Adadi and Mohammed Berrada. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access*, 6:52138–52160, 9 2018.
- [4] Daniel W. Apley and Jingyu Zhu. Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models. *Cornell University - arXiv*, 12 2016.
- [5] Diogo Teixeira Carvalho, Eduardo Pereira, and Jaime S. Cardoso. Machine Learning Interpretability: A Survey on Methods and Metrics. *Electronics*, 8(8):832, 7 2019.
- [6] T Justin Clark, Mike Bradburn, Sandy Love, and Douglas G. Altman. Survival Analysis Part I: Basic concepts and first analyses. *British Journal of Cancer*, 89(2):232–238, 7 2003.
- [7] David G. Cox. *Regression Models and Life-Tables*. Springer Science+Business Media, 1 1972.
- [8] Thomas H. Davenport and Ravi Kalakota. The potential for artificial intelligence in healthcare. *Future healthcare journal*, 6(2):94–98, 6 2019.
- [9] Radwa ElShawi, Youssef Sherif, Mouaz Al-Mallah, and Sherif Sakr. Interpretability in healthcare: A comparative study of local machine learning interpretability techniques. *Computational Intelligence*, 37(4):1633–1650, 11 2020.
- [10] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.
- [11] Brandon George, Samantha R. Seals, and Inmaculada Aban. Survival analysis and regression models. *Journal of Nuclear Cardiology*, 21(4):686–694, 5 2014.
- [12] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael A. Specter, and Lalana Kagal. Explaining Explanations: An Overview of Interpretability of Machine Learning. 10 2018.
- [13] Alex S. Goldstein, Adam Kapelner, Justin Bleich, and Emil Pitkin. Peeking Inside the Black Box: Visualizing Statistical Learning with Plots of Individual Conditional Expectation. *Cornell University - arXiv*, 9 2013.
- [14] Erika Graf, Claudia Schmoor, Willi Sauerbrei, and Martin Schumacher. Assessment and comparison of prognostic classification schemes for survival data. *Statistics in Medicine*, 18(17-18):2529–2545, 9 1999.
- [15] Brandon Greenwell. pdp: An R Package for Constructing Partial Dependence Plots. *R Journal*, 9(1):421, 1 2017.

- [16] Brandon Greenwell, Bradley C. Boehmke, and Andrew A. McCarthy. A Simple and Effective Model-Based Variable Importance Measure. *arXiv (Cornell University)*, 5 2018.
- [17] John L. Hennessy and David A. Patterson. *Computer Architecture*. Morgan Kaufmann, 6 edition, 11 2017.
- [18] Bernease Herman. The Promise and Peril of Human Evaluation for Model Interpretability, 11 2017.
- [19] Junyong In and Dong Ho Lee. Survival analysis: Part I — analysis of time-to-event. *Korean Journal of Anesthesiology*, 71(3):182–191, 5 2018.
- [20] Hemant Ishwaran, Udaya B. Kogalur, Eugene H. Blackstone, and Michael S. Lauer. Random survival forests. *The Annals of Applied Statistics*, 2(3), 11 2008.
- [21] E. L. Kaplan and Paul Meier. Nonparametric Estimation from Incomplete Observations. *Journal of the American Statistical Association*, 53(282):457–481, 6 1958.
- [22] Christiana Kartsonaki. Survival analysis. *Diagnostic histopathology*, 22(7):263–270, 7 2016.
- [23] Arash Khoda Bakhshi and Mohamed M. Ahmed. Utilizing black-box visualization tools to interpret non-parametric real-time risk assessment models. *Transportmetrica A: Transport Science*, 17(4):739–765, 8 2020.
- [24] Mateusz Krzyżiński, Mikołaj Spytek, Hubert Baniecki, and Przemysław Biecek. SurvSHAP(t): Time-dependent explanations of machine learning survival models. *Knowledge Based Systems*, 262:110234, 2 2023.
- [25] Changhee Lee, Jinsung Yoon, and Mihaela van der Schaar. Dynamic-DeepHit: A Deep Learning Approach for Dynamic Survival Analysis With Competing Risks Based on Longitudinal Data. *IEEE Transactions on Biomedical Engineering*, 67(1):122–133, 2020.
- [26] Jingyu Lin, Kan Li, and Sheng Luo. Functional survival forests for multivariate longitudinal outcomes: Dynamic prediction of Alzheimer’s disease progression. *Statistical Methods in Medical Research*, 30(1):99–111, 1 2021.
- [27] Pantelis Linardatos, Vasilis Papastefanopoulos, and Sotiris Kotsiantis. Explainable AI: A Review of Machine Learning Interpretability Methods. *Entropy*, 23(1):18, 12 2020.
- [28] Scott McKinney, Marcin Sieniek, Varun Godbole, Jonathan Godwin, Natasha Antropova, Hutan Ashrafian, Trevor Back, Mary Chesus, Greg S. Corrado, Ara Darzi, Mozziyar Etemadi, Florencia Garcia-Vicente, Fiona J. Gilbert, Mark D. Halling-Brown, Demis Hassabis, Sunny Jansen, Alan Karthikesalingam, Christopher B. Kelly, Dominic King, Joseph R. Ledsam, David Melnick, Hormuz Mostofi, Lily Peng, Joshua J. Reicher, Bernardino Romera-Paredes, Richard Sidebottom, Mustafa Suleyman, Daniel Tse, Kenneth C. Young, Jeffrey De Fauw, and Shravya Shetty. International evaluation of an AI system for breast cancer screening. *Nature*, 577(7788):89–94, 1 2020.
- [29] Christoph Molnar. *Interpretable Machine Learning*. Leanpub, 2019.
- [30] W. James Murdoch, Chandan Singh, Karl Kumbier, Reza Abbasi-Asl, and Bin Yu. Definitions, methods, and applications in interpretable machine learning. *Proceedings of the National Academy of Sciences of the United States of America*, 116(44):22071–22080, 10 2019.
- [31] Alexandr Oblizanov, Natalya Shevskaya, Anatoliy Kazak, Marina Rudenko, and Anna Dorofeeva. Evaluation Metrics Research for Explainable Artificial Intelligence Global Methods Using Synthetic Data. *Applied system innovation*, 6(1):26, 2 2023.

- [32] Ozan Ozyegen, Igor Ilic, and Mucahit Cevik. Evaluation of interpretability methods for multivariate time series forecasting. *Applied Intelligence*, 52(5):4727–4743, 1 2022.
- [33] Shankar Prinja, Nidhi Gupta, and Ramesh Verma. Censoring in clinical trials: Review of survival analysis techniques. *Indian Journal of Community Medicine*, 35(2):217, 4 2010.
- [34] Sebastian Pölsterl. scikit-survival: A Library for Time-to-Event Analysis Built on Top of scikit-learn, 2020.
- [35] Priya Ranganathan and C.S. Pramesh. Censoring in survival analysis: Potential for bias. *Perspectives in Clinical Research*, 3(1):40, 1 2012.
- [36] Qudrat E Alahy Ratul, Edoardo Serra, and Alfredo Cuzzocrea. Evaluating Attribution Methods in Machine Learning Interpretability. 12 2021.
- [37] Falco J. Bargagli Stoffi, Gustavo Cevolani, and Giorgio Gnecco. Simple Models in Complex Worlds: Occam’s Razor and Statistical Learning Theory. *Minds and Machines*, 32(1):13–42, 3 2022.
- [38] Julian Tritscher, Markus Ring, Daniel Schlr, Lena Hettinger, and Andreas Hotho. *Evaluation of Post-hoc XAI Approaches Through Synthetic Tabular Data*. Springer Science+Business Media, 9 2020.
- [39] Ping Wang, Yan Li, and Chandan K. Reddy. Machine Learning for Survival Analysis. *ACM Computing Surveys*, 51(6):1–36, 2 2019.
- [40] Xue Ying. An Overview of Overfitting and its Solutions. *Journal of physics*, 1168:022022, 2 2019.
- [41] Jianlong Zhou, Amir H. Gandomi, Fang Chen, and Andreas Holzinger. Evaluating the Quality of Machine Learning Explanations: A Survey on Methods and Metrics. *Electronics*, 10(5):593, 3 2021.

Appendix A

Interpretability Methods for Survival Analysis

The following implementations are inspired by existing code^{1,2,3} of the interpretability methods, discussed in Chapter 2, that are only suitable for classification and regression models.

```
[ ]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors
from sksurv.linear_model import CoxPHSurvivalAnalysis
from sksurv.ensemble import RandomSurvivalForest
from sksurv.metrics import brier_score, integrated_brier_score
import time
from scipy.stats import pearsonr
from sklearn.model_selection import train_test_split
from tabulate import tabulate
import math
```

A.1 Partial Dependence Plot

```
[ ]: def grid_values(X, feature, no_values):
    """
    Create a subset of the unique values.
    :param X: array of shape (n_instances, n_features) containing the
              covariates
    :param feature: integer to indicate the column index of the feature
                  of interest
    :param no_values: integer determining the size of the subset
    :return: array of shape (no_values,) containing a subset of the unique
            values of the feature of interest in X
    Note: if the request subset size (no_values) is larger than the number of
          unique values, the unique values are returned, and hence the
```

¹<https://github.com/scikit-learn/scikit-learn/tree/main>

²<https://github.com/blent-ai/ALEPython/tree/dev>

³<https://github.com/DanaJomar/PyALE/tree/master>

```

        subset size is smaller than no_values
    """
    return np.unique(np.quantile(X[:, feature],
                                np.linspace(0, 1, no_values), method="lower"))

```

```

[ ]: def pdp_1d(estimator, X, feature, t, no_values):
    """
    Compute the partial dependence values for the feature of interest at a
    specific time.
    :param estimator: trained survival model
    :param X: array of shape (n_instances, n_features) containing the
               covariates
    :param feature: integer to indicate the column index of the feature of
                   interest
    :param t: time of interest
    :param no_values: integer determining the number of unique values that
                     will be used in the computation
    :return: array of shape (no_values,) containing the subset of unique
            values used in the computation,
            array of shape (no_values,) containing the average survival
            probability at each unique value
    """
    unique_values = grid_values(X, feature, no_values)
    X_copy = np.copy(X)
    averaged_predictions = []
    for i in unique_values:
        X_copy[:, feature] = i
        prediction = estimator.predict_survival_function(X_copy, True)[: , t]
        averaged_predictions.append(np.mean(prediction, axis=0))

    return unique_values, averaged_predictions

```

```

[ ]: def pdp_2d(estimator, X, feature1, feature2, t, no_values):
    """
    Compute the partial dependence values for the features of interest at a
    specific time.
    :param estimator: trained survival model
    :param X: array of shape (n_instances, n_features) containing the
               covariates
    :param feature1: integer to indicate the column index of the first
                    feature of interest
    :param feature2: integer to indicate the column index of the second
                    feature of interest
    :param t: time of interest
    :param no_values: integer determining the number of unique values that
                     will be used in the computation
    :return: array of shape (no_values,) containing the subset of unique
            values for feature1 used in the computation,
            array of shape (no_values,) containing the subset of unique
            values for feature2 used in the computation,
            array of shape (no_values, no_values) containing the average

```

```

        survival probability at each combination of unique values of
        feature1 and feature2
    """
    unique_values1 = grid_values(X, feature1, no_values)
    unique_values2 = grid_values(X, feature2, no_values)
    X_copy = np.copy(X)
    averaged_predictions = []
    for i in range(0, len(unique_values1)):
        X_copy[:, feature1] = unique_values1[i]
        for j in range(0, len(unique_values2)):
            X_copy[:, feature2] = unique_values2[j]
            prediction = estimator.predict_survival_function(X_copy,
                                                            True)[: , t]
            averaged_predictions.append(np.mean(prediction, axis=0))
    averaged_predictions = np.reshape(averaged_predictions,
                                      (len(unique_values1),
                                       len(unique_values2))).T

    return unique_values1, unique_values2, averaged_predictions

```

```

[ ]: def pdp_plot(estimator, X, features, t, no_values=100):
    """
    Create 1D- or 2D-PDPs.
    :param estimator: trained survival model
    :param X: array of shape (n_instances, n_features) containing the
                covariates
    :param features: array of {int, (int, int)} containing the column indices
                of the features of interest for which to compute a PDP-plot
                (e.g. 0 for 1D-PDP and (0, 1) for 2D-PDP)
    :param t: time of interest
    :param no_values: integer determining the number of unique values that
                will be used in the computation
    :return: None
    """
    for feature in features:
        if type(feature) is int:
            unique_values,
            averaged_predictions = pdp_1d(estimator, X, feature, t, no_values)

            plt.figure()
            plt.xlabel(feature)
            plt.ylabel('Partial Dependence')
            plt.plot(unique_values, averaged_predictions)
            plt.show()
        elif type(feature) is tuple and len(feature) == 2:
            feature1 = feature[0]
            feature2 = feature[1]
            unique_values1, unique_values2,
            averaged_predictions = pdp_2d(estimator, X, feature1,
                                          feature2, t, no_values)

```

```

XX, YY = np.meshgrid(unique_values1, unique_values2)
levels = np.linspace(averaged_predictions.min(),
                     averaged_predictions.max(), 8)
fig, ax = plt.subplots()
ax.set_xlabel(feature1)
ax.set_ylabel(feature2)
contourf_ = ax.contourf(XX, YY, averaged_predictions,
                       levels=levels)
cbar = fig.colorbar(contourf_)
plt.show()
else:
    print('Feature ' + str(feature) + ' does not have the right ' +
          'format.')
    return NotImplementedError

```

A.2 Individual Conditional Expectation

```

[ ]: def ice(estimator, X, feature, t, no_values):
    """
    Compute the individual conditional expectation values for the feature of
    interest at a specific time.
    :param estimator: trained survival model
    :param X: array of shape (n_instances, n_features) containing the
               covariates
    :param feature: integer to indicate the column index of the feature of
                   interest
    :param t: time of interest
    :param no_values: integer determining the number of unique values that
                     will be used in the computation
    :return: array of shape (no_values,) containing the subset of unique
            values used in the computation,
            array of shape (n_instances, no_values) containing the survival
            probability of each instance at each unique value
    """
    unique_values = grid_values(X, feature, no_values)
    X_copy = np.copy(X)
    predictions = []
    for i in unique_values:
        X_copy[:, feature] = i
        prediction = estimator.predict_survival_function(X_copy, True)[: , t]
        predictions.append(prediction)

    return unique_values, np.ravel(predictions).reshape(len(unique_values),
                                                         -1).T

```

```

[ ]: def ice_plot(estimator, X, features, t, no_values=100):
    """
    Create ICE plot.
    :param estimator: trained survival model
    :param X: array of shape (n_instances, n_features) containing the

```

```

        covariates
:param features: array of {int} containing the column indices of the
        features of interest for for which to compute an ICE-plot
:param t: time of interest
:param no_values: integer determining the number of unique values that
        will be used in the computation
:return: None
"""
for feature in features:
    if type(feature) is int:
        unique_values, predictions = ice(estimator, X, feature, t,
                                         no_values)

        plt.figure()
        for prediction in predictions:
            plt.plot(unique_values, prediction, color='blue',
                     alpha=0.1, linewidth=0.5) #plot each individual line
        plt.xlabel(feature)
        plt.ylabel('Partial Dependence')
        plt.show()
    else:
        print('Feature ' + str(feature) + ' does not have the right ' +
              'format.')
        return NotImplementedError

```

A.3 Accumulated Local Effects

```

[ ]: def get_quantiles(X, feature, k):
    """
    Compute k unique quantiles for the feature of interest.
    :param X: array of shape (n_instances, n_features) containing the
        covariates
    :param feature: integer to indicate the column index of the feature of
        interest
    :param k: number of quantiles
    :return: array of shape (k+1,) containing the unique quantiles of the
        feature of interest.
    Note: if identical quantiles are present, these are removed, and hence
        the number of unique quantiles is less than k+1
    """
    return np.unique(np.quantile(X[:, feature],
                                np.linspace(0, 1, k+1), method="lower"))

```

```

[ ]: def get_indices(X, feature, quantiles):
    """
    Compute the bin each instance falls into based on the instance's feature
        value and the quantiles.
    :param X: array of shape (n_instances, n_features) containing the
        covariates
    :param feature: integer to indicate the column index of the feature of
        interest

```

```

:param quantiles: array of shape (n_quantiles,) containing the quantiles
                  of the feature of interest
:return: array of shape (n_instances,) containing for each instance the
        index of the bin it falls into
Note: the number of bins is the number of quantiles - 1
Note: first bin: [lower quantile, upper quantile],
      other bins: (lower quantile, upper quantile]
"""
return np.clip(np.digitize(X[:, feature], quantiles, right=True) - 1,
               0, None)

```

```

[ ]: def roll_replace(X, periods, axis, fill_value):
    """
    Shift the elements a number of periods along a given axis.
    :param X: array of shape (n_instances, n_features) containing the
              covariates
    :param periods: number of rows/columns the elements need to be shifted
    :param axis: 0 or 1 indicating the axis along which to shift the elements
    :param fill_value: replacement value for empty cells created by shifting
                      the elements
    :return: array of shape (n_instances, n_features) containing the shifted
            elements.
    """
    X_shift = np.roll(X, periods, axis)
    if axis == 0:
        X_shift[:periods, :] = fill_value
    else:
        X_shift[:, :periods] = fill_value
    return X_shift

```

```

[ ]: def ale_1d(estimator, X, feature, t, k):
    """
    Compute the accumulated local effects values for the feature of interest
    at a specific time.
    :param estimator: trained survival model
    :param X: array of shape (n_instances, n_features) containing the
              covariates
    :param feature: integer to indicate the column index of the feature of
                    interest
    :param t: time of interest
    :param k: number of intervals
    :return: array of shape (k,) containing the interval edges,
            array of shape (k,) containing the survival probability in each
            interval
    """
    quantiles = get_quantiles(X, feature, k)
    indices = get_indices(X, feature, quantiles)

    #uncentered ALE
    X_copy = np.copy(X)
    ale_values = np.array([0])

```



```

interval_sizes = np.array([0])
for interval in range(0, len(quantiles) - 1):
    X_interval = X_copy[indices == interval]

    #compute prediction when feature value is set to lower bound
    lower_bound = quantiles[interval]
    X_interval[:, feature] = lower_bound
    pred_low = estimator.predict_survival_function(X_interval, True)[:, t]

    #compute prediction when feature value is set to upper bound
    upper_bound = quantiles[interval + 1]
    X_interval[:, feature] = upper_bound
    pred_up = estimator.predict_survival_function(X_interval, True)[:, t]

    pred_dif = pred_up - pred_low
    pred_sum = np.sum(pred_dif)
    weighted_sum = pred_sum/len(X_interval) #divide by number of
                                                #instances in interval

    ale_values = np.append(ale_values, weighted_sum)
    interval_sizes = np.append(interval_sizes, len(X_interval))

ale_values_uncentered = np.cumsum(ale_values) #cumulatively add the ale
                                                #values of all intervals

#centered ALE
mean = 0
bottom = np.sum(interval_sizes)
for i in range(0, len(ale_values_uncentered) - 1):
    top = np.sum((ale_values_uncentered[i] +
                  ale_values_uncentered[i + 1]) /
                  2 * interval_sizes[i + 1])
    mean += top/bottom

ale_values_centered = ale_values_uncentered - mean

return quantiles, ale_values_centered

```

```

[194]: def ale_2d(estimator, X, feature1, feature2, t, k):
    """
    Compute the accumulated local effects values for the features of interest
    at a specific time.
    :param estimator: trained survival model
    :param X: array of shape (n_instances, n_features) containing the
               covariates
    :param feature1: integer to indicate the column index of the first
                     feature of interest
    :param feature2: integer to indicate the column index of the second
                     feature of interest
    :param t: time of interest
    :param k: number of intervals
    :return: array of shape (k,) containing the interval edges for the first

```

```

        feature of interest,
        array of shape (k,) containing the interval edges for the second
        feature of interest,
        array of shape (k,k) containing the survival probability in each
        interval cell
    """
    quantiles1 = get_quantiles(X, feature1, k)
    indices1 = get_indices(X, feature1, quantiles1)
    quantiles2 = get_quantiles(X, feature2, k)
    indices2 = get_indices(X, feature2, quantiles2)

    #uncentered ALE
    X_copy = np.copy(X)
    #masked in which every empty cell is set to True
    ale_values = np.ma.MaskedArray(
        np.zeros((len(quantiles1) - 1, len(quantiles2) - 1)),
        mask=np.ones((len(quantiles1) - 1, len(quantiles2) - 1)))
    interval_sizes = np.zeros((len(quantiles1) - 1, len(quantiles2) - 1))

    for interval1 in range(0, len(quantiles1) - 1):
        lower_bound1 = quantiles1[interval1]
        upper_bound1 = quantiles1[interval1 + 1]
        for interval2 in range(0, len(quantiles2) - 1):
            lower_bound2 = quantiles2[interval2]
            upper_bound2 = quantiles2[interval2 + 1]

            #compute interval cell
            X_interval = X_copy[(indices1 == interval1) &
                                (indices2 == interval2)]
            interval_sizes[interval1, interval2] = len(X_interval)

            if len(X_interval) > 0:
                #mark interval as non-empty
                ale_values.mask[interval1, interval2] = False

                #compute prediction when both features are set to upper bound
                X_interval[:, feature1] = upper_bound1
                X_interval[:, feature2] = upper_bound2
                pred_up_up = estimator.predict_survival_function(X_interval,
                                                                True)[: , t]

                #compute prediction when feature1 is set to lower bound and
                #feature2 is set to upper bound
                X_interval[:, feature1] = lower_bound1
                X_interval[:, feature2] = upper_bound2
                pred_low_up = estimator.predict_survival_function(X_interval,
                                                                True)[: , t]

                #compute prediction when featur1 is set to upper bound and
                #feature2 is set to lower bound
                X_interval[:, feature1] = upper_bound1

```

```

X_interval[:, feature2] = lower_bound2
pred_up_low = estimator.predict_survival_function(X_interval,
                                                  True)[: , t]

#compute prediction when both features are set to lower bound
X_interval[:, feature1] = lower_bound1
X_interval[:, feature2] = lower_bound2
pred_low_low = estimator.predict_survival_function(X_interval,
                                                  True)[: , t]

#compute ALE value for the interval cell
difference = (pred_up_up - pred_low_up) - (pred_up_low -
                                           pred_low_low)

sum_dif = np.sum(difference)
weighted_dif = sum_dif/len(X_interval)
ale_values[interval1, interval2] = weighted_dif

# replace empty interval cells with nearest neighbour value
indices_valid = np.where(~ale_values.mask)
indices_invalid = np.where(ale_values.mask)
if len(indices_invalid[0]) != 0: #if there are empty cells
    range1 = quantiles1.max() - quantiles1.min()
    range2 = quantiles2.max() - quantiles2.min()
    bins1_values = np.tile(quantiles1, (len(quantiles2), 1))
    bins2_values = np.tile(quantiles2, (len(quantiles1), 1)).T
    features_valid = np.append((quantiles1[indices_valid[0]] +
                               quantiles1[indices_valid[0] + 1]) /
                              (2 * range1),
                              (quantiles2[indices_valid[1]] +
                               quantiles2[indices_valid[1] + 1]) /
                              (2 * range2))
    features_valid = np.reshape(features_valid, (2, -1)).T
    features_invalid = np.append((quantiles1[indices_invalid[0]] +
                                 quantiles1[indices_invalid[0] + 1]) /
                                (2 * range1),
                                (quantiles2[indices_invalid[1]] +
                                 quantiles2[indices_invalid[1] + 1]) /
                                (2 * range2))
    features_invalid = np.reshape(features_invalid, (2, -1)).T

    nbrs = NearestNeighbors(n_neighbors=1,
                           algorithm="kd_tree").fit(features_valid)
    distances, indices = nbrs.kneighbors(features_invalid)
    ale_values[indices_invalid[0],
               indices_invalid[1]] = ale_values[indices_valid[0],
                                               [indices.flatten()],
                                               indices_valid[1],
                                               [indices.flatten()]]

#add zero column and row
ale_values = np.insert(ale_values, 0, np.zeros(len(quantiles1) - 1),

```

```

        axis = 1)
ale_values = np.insert(ale_values, 0, np.zeros(len(quantiles2)),
                        axis = 0)
interval_sizes = np.insert(interval_sizes, 0, np.zeros(
    len(quantiles1) - 1), axis = 1)
interval_sizes = np.insert(interval_sizes, 0, np.zeros(len(quantiles2)),
                            axis = 0)

#cumulatively add rows and columns
ale_values = np.cumsum(np.cumsum(ale_values, axis=0), axis=1)

#compute individual effect of feature1
difference = ale_values - roll_replace(ale_values, periods=1,
                                       axis=0, fill_value=0)
weighted_dif = interval_sizes * ((difference +
                                   roll_replace(difference, periods=1,
                                                axis=1, fill_value=0))/2)

sum_dif = np.sum(weighted_dif, axis=1)
weighted = np.where(np.sum(interval_sizes, axis=1) == 0, 0,
                    sum_dif / (np.sum(interval_sizes, axis=1)))
individual_effect1 = np.cumsum(weighted)

#compute individual effect of feature2
difference = ale_values - roll_replace(ale_values, periods=1,
                                       axis=1, fill_value=0)
weighted_dif = interval_sizes * ((difference +
                                   roll_replace(difference, periods=1,
                                                axis=0, fill_value=0))/2)

sum_dif = np.sum(weighted_dif, axis=0)
weighted = np.where(np.sum(interval_sizes, axis=0) == 0, 0,
                    sum_dif / (np.sum(interval_sizes, axis=0)))
individual_effect2 = np.cumsum(weighted)

#subtract individual effect
ale_values = ((ale_values - individual_effect2).T - individual_effect1).T

#centered ALE
average = np.sum(interval_sizes[1:, 1:] *
                  (ale_values[0:len(quantiles1) - 1,
                              0:len(quantiles2) - 1] +
                   ale_values[1:len(quantiles1),
                              1:len(quantiles2)] +
                   ale_values[0:len(quantiles1) - 1,
                              1:len(quantiles2)] +
                   ale_values[1:len(quantiles1),
                              0:len(quantiles2) - 1]) / 4)
ale_values_centered = ale_values - average / np.sum(interval_sizes)

return quantiles1, quantiles2, ale_values_centered.T

```

```
[ ]: def ale_plot(estimator, X, features, t, k=40):
    """
    Create 1D- or 2D-ALEs.
    :param estimator: trained survival model
    :param X: array of shape (n_instances, n_features) containing the
        covariates
    :param features: array of {int, (int, int)} containing the column indices
        of the features of interest for which to compute an ALE-plot
        (e.g. 0 for 1D-ALE and (0, 1) for 2D-ALE)
    :param t: time of interest
    :param k: number of intervals
    :return: None
    """
    for feature in features:
        if type(feature) is int:
            quantiles, ale_values = ale_1d(estimator, X, feature, t, k)

            plt.figure()
            plt.xlabel(feature)
            plt.ylabel('Accumulated Local Effects')
            plt.plot(quantiles, ale_values)
            plt.show()
        elif type(feature) is tuple and len(feature) == 2:
            quantiles1, quantiles2, ale_values = ale_2d(
                estimator, X, feature[0], feature[1], t, k)

            XX, YY = np.meshgrid(quantiles1, quantiles2)
            fig, ax = plt.subplots()
            ax.set_xlabel(feature[0])
            ax.set_ylabel(feature[1])
            im = ax.imshow(ale_values, origin="lower",
                           extent=[XX.min(), XX.max(), YY.min(), YY.max()])
            imc = ax.contour(XX, YY, ale_values, colors="black")
            ax.clabel(imc, imc.levels, inline="True", fontsize=10)
            cbar = fig.colorbar(im, ax=ax)
            plt.show()
        else:
            print('Feature ' + str(feature) + ' does not have the right ' +
                  'format.')
```

Appendix B

Simulated Data

```
[ ]: """
    Script for data simulation provided by Wieske de Swart
    """

import numpy as np
import pandas as pd
from numpy.random import default_rng

class SimulatedDataset:
    """
    Create a simulated dataset based on the multivariate joint model linear
    Based on the description in "Functional survival forests for multivariate
        longitudinal outcomes: Dynamic prediction
    of Alzheimer's disease progression" by Jeffrey Lin and "SurvSHAP(t):
        Time-dependent explanations of machine learning
    survival models" by Mateusz Krzyżiński.
    """
    def __init__(self, N, visit_times, h0="constant", c=0, c_x1="td",
                  data="EXP1", nr_datasets=1, seed=42):
        """
        :param N: number of subjects
        :param visit_times: array of time points at which variables and
            resulting hazard/ survival/ event are computed
        :param h0: ["constant", "weibull", "complex"] string to indicate the
            baseline hazard function
        :param c: intiger indicating the degree of feature correlation
        :param c_x1: ["td", "constant"] string to indicate the effect of
            feature x1
        :param data: ["EXP1", "EXP2"] string to indicate the type of dataset
        :param nr_datasets: number of datasets that should be generated
        :param seed: random seed
        """
        self.N = N
        self.visit_times = visit_times
        self.nr_times = len(visit_times)
```

```

self.rng = default_rng(seed=seed)
self.h0 = h0
self.c = c
self.c_x1 = c_x1
self.data = data

if nr_datasets == 1:
    self.df = self.create_dataframe()
    self.dflist = [self.df]
elif nr_datasets > 1:
    self.dflist = []
    for i in range(nr_datasets):
        self.dflist.append(self.create_dataframe())

def generate_var(self):
    """
    :return: set of variables with size [N, 1, nr_var]
    """
    x = np.zeros([self.N, 1, 5])
    x[:, 0, 0] = self.rng.binomial(1, 0.5, size=self.N)
    x[:, 0, 1] = self.rng.binomial(1, 0.5, size=self.N)
    if self.data == "EXP1":
        x[:, 0, 2] = self.rng.normal(10, 2, size=self.N)
        x[:, 0, 3] = self.rng.normal(20, 4, size=self.N)
    elif self.data == "EXP2":
        x[:, 0, 2:4] = self.rng.multivariate_normal([2, 2],
                                                    [[2, self.c],
                                                     [self.c, 2]],
                                                    size=self.N)
    else:
        print("data should be one of {EXP1, EXP2}, but was: ", data)
        return NotImplementedError
    x[:, 0, 4] = self.rng.normal(0, 1, size=self.N)
    return x

def hazard_function(self, h0_type, c_x1_type, t, x):
    """ Generate the measured and underlying longitudinal covariates
    :param h0_type: determines baseline hazard function
    :param c_x1_type: determines the effect of feature x1
    :param t: array with the visit times
    :param x: set of variables/ covariates
    :return: array of shape [N, nr_times] with the hazard for all
             subjects and time points
    """
    t = t[None, :]
    # determine baseline hazard
    if h0_type == "constant":
        h0 = 0.08 + 1e-18 * t
    elif h0_type == "weibull":
        h0 = 1.2 * 0.1 * t ** 0.2
    elif h0_type == "complex":

```

```

        h0 = np.exp(-17.8 + 6.5 * t - 11 * np.sqrt(t) * np.log(t+1e-18) +
                    9.5 * np.sqrt(t))
    else:
        print("h0 should be one of {constant, weibull, complex}, " +
              "but was: ", h0_type)
        return NotImplementedError

    # determine hazard function
    if c_x1_type == "td":
        c_x1 = -0.9 + 0.1 * t + 0.9 * np.log(t+1e-18)
    elif c_x1_type == "constant":
        c_x1 = 1
    else:
        print("c_x1 should be one of {td, constant}, but was: ",
              c_x1_type)
        return NotImplementedError
    h_exp = (c_x1 * x[:, :, 0] + 0.5 * x[:, :, 1] - 0.2 * x[:, :, 2] +
             0.1 * x[:, :, 3] + 1e-6 * x[:, :, 4])

    return h0 * np.exp(h_exp)

def generate_survival_model(self, x):
    """ Generate the hazard and survival function
    :param h0: baseline hazard
    :param bcov: array of shape [N, 1, nr_bcov] with baseline covariate(s)
    :param lcov: array of shape [N, nr_times, nr_lcov] with longitudinal
                  covariate(s)
    :return: hazard, survival, event_time and event
    """
    # survival submodel
    hazard = self.hazard_function(self.h0, self.c_x1, self.visit_times, x)
    # determine the survival based on the hazard:
    surv = np.exp(-np.cumsum(hazard, axis=1))
    u = self.rng.uniform(0, 1, size=[self.N, 1])
    # the event happens when the random probability u is greater than the
    # survival probability surv
    # the index of the event time can be found by counting the instances
    # before this is true
    time_idx = np.count_nonzero(surv > u, axis=1)
    # when all instances are nonzero the subject is censored at the last
    # time step ("end of study" censoring)
    cens_end = time_idx == self.nr_times
    event_time = self.visit_times[np.where(time_idx == self.nr_times,
                                           self.nr_times - 1, time_idx)]
    # add more censoring by generating random censoring times ("dropout")
    cens = self.rng.uniform(self.visit_times[1], 2*self.visit_times[-1],
                           size=self.N)
    cens_time = self.visit_times[np.searchsorted(self.visit_times, cens,
                                                  side="right") - 1]
    event = np.where(np.logical_or(cens_end, cens_time < event_time),
                     0, 1)

```



```

event_time = np.minimum(event_time, cens_time)
return hazard, surv, event_time, event

def create_dataframe(self):
    """
    Generate the covariates and survival model and combine all relevant
    information in a dataframe
    :return: dataframe with id, visit_time, hazard, survival, event_time,
            event and all covariates
    """
    x = self.generate_var()
    hazard, surv, event_time, event = self.generate_survival_model(x)

    df = pd.DataFrame({
        "id": np.repeat(np.arange(self.N), self.nr_times),
        "visit_time": np.tile(self.visit_times, self.N),
        "hazard": np.ravel(hazard),
        "survival": np.ravel(surv),
        "event_time": np.repeat(event_time, self.nr_times),
        "event": np.repeat(event, self.nr_times),
    })
    # add variables
    for i in range(x.shape[-1]):
        df["x_" + str(i + 1)] = np.repeat(x[:, 0, i], self.nr_times)

    # remove visit times that occur after the event time:
    df = df.loc[lamba df: df["visit_time"] < df["event_time"], :]
    return df

```

Appendix C

Evaluation Metrics for Interpretability Methods

```
[ ]: def feature_importance(y_values):  
    """  
    Compute the importance weight of a feature based on the values in its  
    PDP- or ALE-plot.  
    Based on the description in "A Simple and Effective Model-Based Variable  
    Importance" by Greenwell et al.  
    :param y_values: array of shape (no_y_values,) containing the y-values of  
    the PDP- or ALE-plot of the feature of interest  
    :return: integer indicating the feature importance of the feature of  
    interest  
    Note: a small importance value indicates less importance and vice versa  
    """  
    return np.sqrt(np.sum(np.square(y_values - np.mean(y_values))) /  
                    (len(y_values) - 1))
```

```
[ ]: def rank_values_t(estimator, X, t, parameter, plot='PDP/ICE'):  
    """  
    Compute the feature importance value of each feature at a specific time  
    point.  
    :param estimator: trained survival model  
    :param X: array of shape (n_instances, n_features) containing the  
    covariates  
    :param t: time of interest  
    :param parameter: integer indicating the number of grid values (PDP/ICE)  
    or number of intervals (ALE) used in the feature's plot  
    :param plot: ['PDP/ICE', 'ALE'] string to indicate the plot the feature  
    importance value is based on  
    :return: array of shape (n_features,) with the feature as first field,  
    and feature importance as second field.  
    Note: PDP and ICE are grouped together since they give the same result  
    """  
    X = X.to_numpy()  
    feature_columns = np.arange(X.shape[1]) #no_features = no_columns in X  
    rank_values = []
```

```

y_values = []
for f in feature_columns:
    if plot == 'PDP/ICE':
        _, y_values = pdp_1d(estimator, X, f, t, parameter)
    elif plot == 'ALE':
        _, y_values = ale_1d(estimator, X, f, t, parameter)
    else:
        print("plot should be one of {PDP/ICE, ALE}, but was: ", plot)
        return NotImplementedError
rank_value = feature_importance(y_values)
rank_values.append((f, rank_value))
return rank_values

```

```

[ ]: def ascending_ranking(ranking):
    """
    Sort the feature importance values in ascending order.
    :param ranking: array of shape (n_features,) with the feature as first
                    field, and feature importance as second field.
    :return: sorted array of shape (n_features,) with the feature as first
            field, and the feature importance as second field.
    """
    return sorted(ranking, key=lambda x: x[1], reverse=False)

```

```

[ ]: def faithfulness(estimator, X, ranking, t):
    """
    Compute the faithfulness metric.
    Based on the description in "Evaluation Metrics Research for Explainable
    Artificial Intelligence Global Methods Using Synthetic Data" by
    Oblizanov et al.
    :param estimator: trained survival model
    :param X: array of shape (n_instances, n_features) containing the
              covariates
    :param ranking: array of shape (n_features,) with the feature as first
                    field, and feature importance as second field.
    :param t: time of interest
    :return: integer indicating the correlation between the feature
            importance values and prediction changes when the feature
            is set to its mean value
    """
    X = X.to_numpy()
    base_survival = estimator.predict_survival_function(X, True)[: , t]
    delta_survival = np.array([])
    for (feature, importance) in ranking:
        X_faithfulness = np.copy(X)
        X_faithfulness[:, feature] = np.mean(X[:, feature])
        survival = estimator.predict_survival_function(X_faithfulness,
                                                         True)[: , t]

        delta_survival = np.append(delta_survival,
                                    np.mean(np.abs(survival - base_survival)))
    rank_values = [i for (f, i) in ranking]
    statistic, pvalue = pearsonr(rank_values, delta_survival)

```

```
return statistic
```

```
[ ]: def monotonicity(estimator, X, ranking, t):  
    """  
    Compute the monotonicity metric.  
    Based on the description in "Evaluation Metrics Research for Explainable  
        Artificial Intelligence Global Methods Using Synthetic Data" by  
        Oblizanov et al.  
    :param estimator: trained survival model  
    :param X: array of shape (n_instances, n_features) containing the  
        covariates  
    :param ranking: array of shape (n_features,) with the feature as first  
        field, and feature importance as second field.  
    :param t: time of interest  
    :return: integer indicating the proportion of features that received a  
        higher feature importance value compared to a feature with a  
        lower feature importance value that causes more prediction change  
    """  
    X = X.to_numpy()  
    ranking = ascending_ranking(ranking)  
    base_survival = estimator.predict_survival_function(X, True)[: , t]  
    delta_survival = np.array([])  
    X_monotonicity = np.copy(X)  
    for (feature, importance) in ranking:  
        X_monotonicity[:, feature] = np.mean(X[:, feature])  
        survival = estimator.predict_survival_function(X_monotonicity,  
                                                       True)[: , t]  
        delta_survival = np.append(delta_survival,  
                                   np.mean(np.abs(survival - base_survival)))  
    count = 0  
    for i in np.arange(len(delta_survival)-1):  
        for j in delta_survival[i+1:]: #values after d[i]  
            if delta_survival[i] > j:  
                count += 1  
    n_comparisons = len(delta_survival)*(len(delta_survival)-1)/2  
    return 1 - count/n_comparisons
```

Appendix D

Experiments

```
[ ]: def transform(df):  
    """  
    Transform the dataframe by:  
    - reducing the longitudinal data to the first measurement only  
    - splitting the dataframe into train data (70%) and test data (30%)  
    - splitting both the train and test data into two separate dataframes:  
      - X - shape = (n_samples, n_features) containing the covariates  
      - y - shape = (n_samples,) containing the binary event indicator as  
        first field, and time of the event/censoring as second field.  
    :param df: dataframe with id, visit_time, hazard, survival, event_time,  
              event and five covariates  
    :return: transformed dataframe  
    """  
    df = df.drop_duplicates(['id'])  
    X = df[['x_1', 'x_2', 'x_3', 'x_4', 'x_5']]  
    y = np.array([tuple(x) for x in df[['event', 'event_time']].values],  
                  dtype=[('event', 'bool_'), ('event_time', 'i4')])  
    X_train, X_test = train_test_split(X, test_size=0.3, random_state=42)  
    y_train, y_test = train_test_split(y, test_size=0.3, random_state=42)  
    return X_train, X_test, y_train, y_test  
  
[ ]: def brier_scores(estimator, X_test, y_train, y_test, times=np.arange(10)):  
    """  
    Compute the brier score at different time points and the integrated brier  
    score.  
    :param estimator: trained survival model  
    :param X_test: array of shape (n_test_samples, n_features) containing the  
                  covariates for the test data  
    :param y_train: array of shape (n_train_samples, ) containing the binary  
                  event indicator as first field, and time of the event/censoring  
                  as second field field for the training data.  
    :param y_test: array of shape (n_train_samples, ) containing the binary  
                  event indicator as first field, and time of the event/censoring  
                  as second field field for the test data.  
    :param times: times of interest  
    :return: array of shape (n_times, ) containing the brier score at each
```

```

        time, and integer indicating the integrated brier score.
    """
    survs = estimator.predict_survival_function(X_test)
    preds = np.asarray([[fn(t) for t in times] for fn in survs])
    return brier_score(y_train, y_test, preds, times)[1],
    ↪integrated_brier_score(y_train, y_test, preds, times)

```

D.1 Experiment 1

```

[ ]: #Generate the datasets with different baseline hazard functions

visit_times = np.arange(0, 10.1, 0.5)

#EXP1_exponential
sim_data_exp = SimulatedDataset(2000, visit_times, h0="constant")
df_exp = sim_data_exp.df
df_exp.to_csv("./dataset/simdata_N2000_EXP1_exponential.csv", index=False)
df_exp = pd.read_csv("./dataset/simdata_N2000_EXP1_exponential.csv")

#EXP1_weibull
sim_data_weib = SimulatedDataset(2000, visit_times, h0="weibull")
df_weib = sim_data_weib.df
df_weib.to_csv("./dataset/simdata_N2000_EXP1_weibull.csv", index=False)
df_weib = pd.read_csv("./dataset/simdata_N2000_EXP1_weibull.csv")

#EXP1_complex
sim_data_comp = SimulatedDataset(2000, visit_times, h0="complex")
df_comp = sim_data_comp.df
df_comp.to_csv("./dataset/simdata_N2000_EXP1_complex.csv", index=False)
df_comp = pd.read_csv("./dataset/simdata_N2000_EXP1_weibull.csv")

#EXP1_non_td
sim_data_non_td = SimulatedDataset(2000, visit_times, h0="complex",
                                    c_x1="constant")
df_non_td = sim_data_non_td.df
df_non_td.to_csv("./dataset/simdata_N2000_EXP1_non_td.csv", index=False)
df_non_td = pd.read_csv("./dataset/simdata_N2000_EXP1_non_td.csv")

```

```

[ ]: #Transform dataframes for survival models

datasets = [df_exp, df_weib, df_comp, df_non_td]
n_datasets = len(datasets)
dataset_types = ['EXP1_exponential', 'EXP1_weibull',
                 'EXP1_complex', 'EXP1_non_td']

X_train = np.zeros(n_datasets, dtype=np.ndarray)
X_test = np.zeros(n_datasets, dtype=np.ndarray)
y_train = np.zeros(n_datasets, dtype=np.ndarray)
y_test = np.zeros(n_datasets, dtype=np.ndarray)

```

```

for d in np.arange(n_datasets):
    X_train[d], X_test[d], y_train[d], y_test[d] = transform(datasets[d])

```

```
[ ]: #Train the survival models
```

```

model_types = ['RSF', 'CPH']
n_model_types = len(model_types)

models = np.zeros((n_model_types, n_datasets), dtype=np.ndarray)
n_models = len(models) * len(models[0])

for d in np.arange(n_datasets):
    models[0, d] = RandomSurvivalForest().fit(X_train[d], y_train[d])
    models[1, d] = CoxPHSurvivalAnalysis().fit(X_train[d], y_train[d])

```

```
[ ]: #Evaluate the models
```

```

normal_brier_scores = np.zeros((n_model_types, n_datasets), dtype=np.ndarray)
integrated_brier_scores = np.zeros((n_model_types, n_datasets))
end_time = 10

```

```
#Brier Score Plot
```

```

fig, axs = plt.subplots(1, n_datasets, figsize=(10, 3))
fig.tight_layout()
axs[0].set_ylabel('Brier Score')
for m in np.arange(n_model_types): #RSF or CPH
    for d in np.arange(n_datasets): #EXP1_exponential, EXP1_weibull,
                                    #EXP1_complex or EXP1_non_td
        normal_brier_scores[m, d], integrated_brier_scores[m, d] = (
            brier_scores(models[m, d], X_test[d], y_train[d], y_test[d]))
        axs[d].plot(np.arange(end_time), normal_brier_scores[m, d],
                    label=model_types[m])
        axs[d].set_ylim([0, 0.25])
        axs[d].set_title(dataset_types[d])
        axs[d].set_xlabel('Time')
fig.legend(model_types)
fig.show()

```

```
#Integrated Brier Score Table
```

```

table = [['Dataset', model_types[0], model_types[1]],
         [dataset_types[0], integrated_brier_scores[0, 0],
          integrated_brier_scores[1, 0]],
         [dataset_types[1], integrated_brier_scores[0, 1],
          integrated_brier_scores[1, 1]],
         [dataset_types[2], integrated_brier_scores[0, 2],
          integrated_brier_scores[1, 2]],
         [dataset_types[3], integrated_brier_scores[0, 3],
          integrated_brier_scores[1, 3]]]
print(tabulate(table, headers='firstrow', tablefmt='fancy_grid'))

```

```
[ ]: #Compute the rankings

parameters = [10, 40, 100, 200]
n_parameters = len(parameters)

method_types = ['PDP/ICE', 'ALE']
n_methods = len(method_types)

feature_names = np.array(['x1', 'x2', 'x3', 'x4', 'x5'])
n_features = len(feature_names)

rankings = np.zeros((n_model_types, n_datasets, end_time, n_methods,
                    n_features, n_parameters), dtype=tuple)

for par in np.arange(n_parameters):
    #Compute ranking at different time points
    for m in np.arange(n_model_types): #RSF or CPH
        for d in np.arange(n_datasets): #EXP1_exponential, EXP1_weibull,
            #EXP1_complex, EXP1_non_td
            for t in np.arange(end_time): #[0, 10)
                for p in np.arange(n_methods): #PDP/ICE or ALE
                    rankings[m, d, t, p, :, par] = rank_values_t(
                        models[m, d], X_test[d], t,
                        parameter=parameters[par],
                        plot=method_types[p])
```

```
[ ]: #Plot the rankings

for par in np.arange(n_parameters):
    for d in np.arange(n_datasets):
        fig, axs = plt.subplots(n_model_types, n_methods,
                                figsize=(5, 5), sharex=True, sharey=True)
        axs[0, 0].set_ylabel('Feature Importance')
        axs[1, 0].set_xlabel('Time')
        axs[1, 0].set_ylabel('Feature Importance')
        axs[1, 1].set_xlabel('Time')
        for m in np.arange(n_model_types):
            for p in np.arange(n_methods):
                for f in np.arange(n_features):
                    axs[m, p].plot(np.arange(end_time),
                                    [i for (f, i) in
                                     rankings[m, d, :, p, f, par]],
                                    label=feature_names[f])
                    axs[m, p].set_title(model_types[m]+' '+method_types[p])
        fig.legend(feature_names)
        fig.suptitle('Parameter '+str(parameters[par])+
                     ' - ' +dataset_types[d])
        fig.tight_layout()
        fig.show()
```



```

[ ]: #Faithfulness

for par in np.arange(n_parameters):
    faith_values = np.zeros((n_model_types, n_datasets, end_time, n_methods))
    average = np.zeros((n_model_types, n_datasets, n_methods))
    for t in np.arange(end_time): #[0, 10)
        fig, axs = plt.subplots(1, n_model_types, figsize=(5, 2))
        fig.tight_layout()
        axs[0].set_title(model_types[0])
        axs[0].set_ylabel('Faithfulness')
        axs[1].set_title(model_types[1])
        for m in np.arange(n_model_types): #RSF or CPH
            for p in np.arange(n_methods): #PDP/ICE or ALE
                for d in np.arange(n_datasets): #EXP1_exp, EXP1_weibull,
                                                #EXP1_complex, EXP1_non_td
                    faith_values[m, d, t, p] = faithfulness(
                        models[m, d], X_test[d],
                        rankings[m, d, t, p, :, par], t)
                    average[m, d, p] += faith_values[m, d, t, p]
                axs[m].plot(np.arange(n_datasets), faith_values[m, :, t, p],
                            label=method_types[p])
                axs[m].set_xticks(np.arange(n_datasets), dataset_types,
                                rotation='vertical')
        fig.legend(method_types, loc='lower center')
        fig.suptitle('Parameter '+str(parameters[par])+' - Time ' +str(t))
        fig.subplots_adjust(top=0.80)
        fig.show()
    average = average/end_time
    fig, axs = plt.subplots(1, n_model_types, figsize=(5, 2))
    fig.tight_layout()
    axs[0].set_title(model_types[0])
    axs[0].set_ylabel('Average Faithfulness')
    axs[1].set_title(model_types[1])
    for m in np.arange(n_model_types): #RSF or CPH
        for p in np.arange(n_methods): #PDP/ICE or ALE
            axs[m].plot(np.arange(n_datasets), average[m, :, p],
                        label=method_types[p])
            axs[m].set_xticks(np.arange(n_datasets), dataset_types,
                            rotation='vertical')
    fig.legend(method_types, loc='lower center')
    fig.suptitle('Parameter '+str(parameters[par])+' - Average')
    fig.subplots_adjust(top=0.80)
    fig.show()
    table = [['Dataset', model_types[0]+' '+method_types[0],
              model_types[0]+' '+method_types[1],
              model_types[1]+' '+method_types[0],
              model_types[1]+' '+method_types[1]],
             [dataset_types[0], average[0, 0, 0], average[0, 0, 1],
              average[1, 0, 0], average[1, 0, 1]],
             [dataset_types[1], average[0, 1, 0], average[0, 1, 1],
              average[1, 1, 0], average[1, 1, 1]],

```

```

[dataset_types[2], average[0, 2, 0], average[0, 2, 1],
average[1, 2, 0], average[1, 2, 1]],
[dataset_types[3], average[0, 3, 0], average[0, 3, 1],
average[1, 3, 0], average[1, 3, 1]]]
print(tabulate(table, headers='firstrow', tablefmt='fancy_grid'))

```

[]: *#Monotonicity*

```

for par in np.arange(n_parameters):
    mon_values = np.zeros((n_model_types, n_datasets, end_time, n_methods))
    average = np.zeros((n_model_types, n_datasets, n_methods))
    for t in np.arange(end_time): #[0, 10)
        fig, axs = plt.subplots(1, n_model_types, figsize=(5, 2),
                                sharex=True, sharey=True)

        fig.tight_layout()
        axs[0].set_title(model_types[0])
        axs[0].set_ylabel('Monotonicity')
        axs[1].set_title(model_types[1])
        for m in np.arange(n_model_types): #RSF or CPH
            for p in np.arange(n_methods): #PDP/ICE or ALE
                for d in np.arange(n_datasets): #EXP1_exp, EXP1_weibull,
                                                    #EXP1_complex, EXP1_non_td
                    mon_values[m, d, t, p] = monotonicity(
                        models[m, d], X_test[d],
                        rankings[m, d, t, p, :, par], t)
                    average[m, d, p] += mon_values[m, d, t, p]
                axs[m].plot(dataset_types, mon_values[m, :, t, p],
                            label=method_types[p])
                axs[m].set_xticks(np.arange(n_datasets), dataset_types,
                                rotation='vertical')

        fig.legend(method_types, loc='lower center')
        fig.suptitle('Parameter '+str(parameters[par])+' - Time ' +str(t))
        fig.subplots_adjust(top=0.80)
        fig.show()

    average = average/end_time
    fig, axs = plt.subplots(1, n_model_types, figsize=(5, 2),
                            sharex=True, sharey=True)

    fig.tight_layout()
    axs[0].set_title(model_types[0])
    axs[0].set_ylabel('Average Monotonicity')
    axs[1].set_title(model_types[1])
    for m in np.arange(n_model_types): #RSF or CPH
        for p in np.arange(n_methods): #PDP/ICE or ALE
            axs[m].plot(np.arange(n_datasets), average[m, :, p],
                        label=method_types[p])
            axs[m].set_xticks(np.arange(n_datasets), dataset_types,
                            rotation='vertical')

    fig.legend(method_types, loc='lower center')
    fig.suptitle('Parameter '+str(parameters[par])+' - Average')
    fig.subplots_adjust(top=0.80)
    fig.show()

```

```

table = [['Dataset', model_types[0]+' '+method_types[0],
          model_types[0]+' '+method_types[1],
          model_types[1]+' '+method_types[0],
          model_types[1]+' '+method_types[1]],
         [dataset_types[0], average[0, 0, 0], average[0, 0, 1],
          average[1, 0, 0], average[1, 0, 1]],
         [dataset_types[1], average[0, 1, 0], average[0, 1, 1],
          average[1, 1, 0], average[1, 1, 1]],
         [dataset_types[2], average[0, 2, 0], average[0, 2, 1],
          average[1, 2, 0], average[1, 2, 1]],
         [dataset_types[3], average[0, 3, 0], average[0, 3, 1],
          average[1, 3, 0], average[1, 3, 1]]]
print(tabulate(table, headers='firstrow', tablefmt='fancy_grid'))

```

D.2 Experiment 2

[]: *#Generate datasets with feature correlation*

```

visit_times = np.arange(0, 10.1, 0.5)
correlations = np.array([0, 0.2, 0.4, 0.6, 0.8, 1])

#EXP2_0
sim_data_00 = SimulatedDataset(2000, visit_times, h0="constant", c=0,
                                data="EXP2", c_x1="constant")
df_0 = sim_data_00.df
df_0.to_csv("./dataset/simdata_N2000_EXP2_0.csv", index=False)
df_0 = pd.read_csv("./dataset/simdata_N2000_EXP2_0.csv")

#EXP2_02
sim_data_02 = SimulatedDataset(2000, visit_times, h0="constant", c=0.2,
                                data="EXP2", c_x1="constant")
df_02 = sim_data_02.df
df_02.to_csv("./dataset/simdata_N2000_EXP2_02.csv", index=False)
df_02 = pd.read_csv("./dataset/simdata_N2000_EXP2_02.csv")

#EXP2_04
sim_data_04 = SimulatedDataset(2000, visit_times, h0="constant", c=0.4,
                                data="EXP2", c_x1="constant")
df_04 = sim_data_04.df
df_04.to_csv("./dataset/simdata_N2000_EXP2_04.csv", index=False)
df_04 = pd.read_csv("./dataset/simdata_N2000_EXP2_04.csv")

#EXP2_06
sim_data_06 = SimulatedDataset(2000, visit_times, h0="constant", c=0.6,
                                data="EXP2", c_x1="constant")
df_06 = sim_data_06.df
df_06.to_csv("./dataset/simdata_N2000_EXP2_06.csv", index=False)
df_06 = pd.read_csv("./dataset/simdata_N2000_EXP2_06.csv")

#EXP2_08

```

```

sim_data_08 = SimulatedDataset(2000, visit_times, h0="constant", c=0.8,
                               data="EXP2", c_x1="constant")
df_08 = sim_data_08.df
df_08.to_csv("./dataset/simdata_N2000_EXP2_08.csv", index=False)
df_08 = pd.read_csv("./dataset/simdata_N2000_EXP2_08.csv")

#EXP2_1
sim_data_1 = SimulatedDataset(2000, visit_times, h0="constant", c=1,
                              data="EXP2", c_x1="constant")
df_1 = sim_data_1.df
df_1.to_csv("./dataset/simdata_N2000_EXP2_1.csv", index=False)
df_1 = pd.read_csv("./dataset/simdata_N2000_EXP2_1.csv")

```

```

[ ]: #Transform dataframes for survival models

datasets = [df_0, df_02, df_04, df_06, df_08, df_1]
n_datasets = len(datasets)
dataset_types = ['EXP2_0', 'EXP2_02', 'EXP2_04',
                 'EXP2_06', 'EXP2_08', 'EXP2_1']

X_train = np.zeros(n_datasets, dtype=np.ndarray)
X_test = np.zeros(n_datasets, dtype=np.ndarray)
y_train = np.zeros(n_datasets, dtype=np.ndarray)
y_test = np.zeros(n_datasets, dtype=np.ndarray)

for d in np.arange(n_datasets):
    X_train[d], X_test[d], y_train[d], y_test[d] = transform(datasets[d])

```

```

[ ]: #Train the survival models

model_types = ['RSF', 'CPH']
n_model_types = len(model_types)

models = np.zeros((n_model_types, n_datasets), dtype=np.ndarray)
n_models = len(models) * len(models[0])

for d in np.arange(n_datasets):
    models[0, d] = RandomSurvivalForest().fit(X_train[d], y_train[d])
    models[1, d] = CoxPHSurvivalAnalysis().fit(X_train[d], y_train[d])

```

```

[ ]: #Evaluate models

normal_brier_scores = np.zeros((n_model_types, n_datasets), dtype=np.ndarray)
integrated_brier_scores = np.zeros((n_model_types, n_datasets))
end_time = 10

#Brier Score Plot
fig, axs = plt.subplots(2, int(n_datasets/2), figsize=(8, 5),
                       sharex=True, sharey=True)
fig.tight_layout()
axs[0, 0].set_ylabel('Brier Score')

```

```

axs[1, 0].set_ylabel('Brier Score')
axs[1, 0].set_xlabel('Time')
axs[1, 1].set_xlabel('Time')
axs[1, 2].set_xlabel('Time')
for m in np.arange(n_model_types): #RSF or CPH
    for d in np.arange(n_datasets): #EXP1_exponential, EXP1_weibull,
                                    #EXP1_complex or EXP1_non_td
        normal_brier_scores[m, d], integrated_brier_scores[m, d] = (
            brier_scores(models[m, d], X_test[d], y_train[d], y_test[d]))
        axs[math.floor(d/3), d % 3].plot(np.arange(10),
                                         normal_brier_scores[m, d],
                                         label=model_types[m])
        axs[math.floor(d/3), d % 3].set_title(dataset_types[d])
        axs[math.floor(d/3), d % 3].set_xticks(np.arange(0, end_time, 2.5),
                                              np.arange(0, end_time, 2.5))

fig.legend(model_types)
fig.show()

#Integrated Brier Score Table
table = [['Dataset', model_types[0], model_types[1]],
         [dataset_types[0], integrated_brier_scores[0, 0],
          integrated_brier_scores[1, 0]],
         [dataset_types[1], integrated_brier_scores[0, 1],
          integrated_brier_scores[1, 1]],
         [dataset_types[2], integrated_brier_scores[0, 2],
          integrated_brier_scores[1, 2]],
         [dataset_types[3], integrated_brier_scores[0, 3],
          integrated_brier_scores[1, 3]],
         [dataset_types[4], integrated_brier_scores[0, 4],
          integrated_brier_scores[1, 4]],
         [dataset_types[5], integrated_brier_scores[0, 5],
          integrated_brier_scores[1, 5]]]
print(tabulate(table, headers='firstrow', tablefmt='fancy_grid'))

```

```

[ ]: #Compute the rankings

method_types = ['PDP/ICE', 'ALE']
n_methods = len(method_types)

parameters = [10, 40, 100, 200]
n_parameters = len(parameters)

feature_names = np.array(['x1', 'x2', 'x3', 'x4', 'x5'])
n_features = len(feature_names)

rankings = np.zeros((n_model_types, n_datasets, end_time, n_methods,
                    n_features, n_parameters), dtype=tuple)

for par in np.arange(n_parameters):
    #Compute ranking at different time points
    for m in np.arange(n_model_types): #RSF or CPH

```

```

for d in np.arange(n_datasets): #0, 0.2, 0.4, 0.6, 0.8 or 1
    for t in np.arange(end_time): #[0, 10)
        for p in np.arange(n_methods): #PDP/ICE or ALE
            rankings[m, d, t, p, :, par] = rank_values_t(
                models[m, d], X_test[d], t,
                parameter=parameters[par],
                plot=method_types[p])

```

[]: *#Plot the rankings*

```

for par in np.arange(n_parameters):
    for d in np.arange(n_datasets):
        fig, axs = plt.subplots(n_model_types, n_methods, figsize=(5, 5),
                                sharex=True, sharey=True)
        axs[0, 0].set_ylabel('Feature Importance')
        axs[1, 0].set_xlabel('Time')
        axs[1, 0].set_ylabel('Feature Importance')
        axs[1, 1].set_xlabel('Time')
        for m in np.arange(n_model_types):
            for p in np.arange(n_methods):
                for f in np.arange(n_features):
                    axs[m, p].plot(np.arange(end_time),
                                    [i for (f, i) in
                                     rankings[m, d, :, p, f, par]],
                                    label=feature_names[f])
                    axs[m, p].set_title(model_types[m]+' '+method_types[p])
        fig.legend(feature_names)
        fig.suptitle('Parameter '+str(parameters[par])+
                     ' - ' +dataset_types[d])
        fig.tight_layout()
        fig.show()

```

[]: *#Faithfulness*

```

for par in np.arange(n_parameters):
    faith_values = np.zeros((n_model_types, n_datasets, end_time, n_methods))
    average = np.zeros((n_model_types, n_datasets, n_methods))
    for t in np.arange(end_time): #[0, 10)
        fig, axs = plt.subplots(1, n_model_types, figsize=(5, 2))
        fig.tight_layout()
        axs[0].set_title(model_types[0])
        axs[0].set_ylabel('Faithfulness')
        axs[0].set_xlabel('Rho')
        axs[1].set_title(model_types[1])
        axs[1].set_xlabel('Rho')
        for m in np.arange(n_model_types): #RSF or CPH
            for p in np.arange(n_methods): #PDP/ICE or ALE
                for d in np.arange(n_datasets): #0, 0.25, 0.5, 0.75 or 1
                    faith_values[m, d, t, p] = faithfulness(
                        models[m, d], X_test[d],
                        rankings[m, d, t, p, :, par], t)

```

```

        average[m, d, p] += faith_values[m, d, t, p]
        axs[m].plot(correlations, faith_values[m, :, t, p],
                    label=method_types[p])
        axs[m].set_xticks(correlations, correlations)
    fig.legend(method_types, loc='lower center')
    fig.suptitle('Parameter '+str(parameters[par])+' - Time ' +str(t))
    fig.subplots_adjust(top=0.80)
    fig.show()
average = average/end_time
fig, axs = plt.subplots(1, n_model_types, figsize=(5, 2))
fig.tight_layout()
axs[0].set_title(model_types[0])
axs[0].set_ylabel('Average Faithfulness')
axs[0].set_xlabel('Rho')
axs[1].set_title(model_types[1])
axs[1].set_xlabel('Rho')
for m in np.arange(n_model_types): #RSF or CPH
    for p in np.arange(n_methods): #PDP/ICE or ALE
        axs[m].plot(correlations, average[m, :, p], label=method_types[p])
        axs[m].set_xticks(correlations, correlations)
fig.legend(method_types, loc='lower center')
fig.suptitle('Parameter '+str(parameters[par])+' - Average')
fig.subplots_adjust(top=0.80)
fig.show()
table = [['Dataset', model_types[0]+' '+method_types[0],
        model_types[0]+' '+method_types[1],
        model_types[1]+' '+method_types[0],
        model_types[1]+' '+method_types[1]],
        [dataset_types[0], average[0, 0, 0], average[0, 0, 1],
        average[1, 0, 0], average[1, 0, 1]],
        [dataset_types[1], average[0, 1, 0], average[0, 1, 1],
        average[1, 1, 0], average[1, 1, 1]],
        [dataset_types[2], average[0, 2, 0], average[0, 2, 1],
        average[1, 2, 0], average[1, 2, 1]],
        [dataset_types[3], average[0, 3, 0], average[0, 3, 1],
        average[1, 3, 0], average[1, 3, 1]],
        [dataset_types[4], average[0, 4, 0], average[0, 4, 1],
        average[1, 4, 0], average[1, 4, 1]],
        [dataset_types[5], average[0, 5, 0], average[0, 5, 1],
        average[1, 5, 0], average[1, 5, 1]]]
print(tabulate(table, headers='firstrow', tablefmt='fancy_grid'))

```

```
[ ]: #Monotonicity
```

```

for par in np.arange(n_parameters):
    mon_values = np.zeros((n_model_types, n_datasets, end_time, n_methods))
    average = np.zeros((n_model_types, n_datasets, n_methods))
    for t in np.arange(end_time): #[0, 10)
        fig, axs = plt.subplots(1, n_model_types, figsize=(5, 2),
                                sharex=True, sharey=True)
        fig.tight_layout()

```

```

    axs[0].set_title(model_types[0])
    axs[0].set_ylabel('Monotonicity')
    axs[0].set_xlabel('Rho')
    axs[1].set_title(model_types[1])
    axs[1].set_xlabel('Rho')
    for m in np.arange(n_model_types): #RSF or CPH
        for p in np.arange(n_methods): #PDP/ICE or ALE
            for d in np.arange(n_datasets): #0, 0.25, 0.5, 0.75 or 1
                mon_values[m, d, t, p] = monotonicity(
                    models[m, d], X_test[d],
                    rankings[m, d, t, p, :, par], t)
                average[m, d, p] += mon_values[m, d, t, p]
            axs[m].plot(correlations, mon_values[m, :, t, p],
                        label=method_types[p])
            axs[m].set_xticks(correlations, correlations)
    fig.legend(method_types, loc='lower center')
    fig.suptitle('Parameter '+str(parameters[par])+' - Time ' +str(t))
    fig.subplots_adjust(top=0.80)
    fig.show()
average = average/end_time
fig, axs = plt.subplots(1, n_model_types, figsize=(5, 2),
                        sharex=True, sharey=True)

fig.tight_layout()
axs[0].set_title(model_types[0])
axs[0].set_ylabel('Average Monotonicity')
axs[0].set_xlabel('Rho')
axs[1].set_title(model_types[1])
axs[1].set_xlabel('Rho')
for m in np.arange(n_model_types): #RSF or CPH
    for p in np.arange(n_methods): #PDP/ICE or ALE
        axs[m].plot(correlations, average[m, :, p], label=method_types[p])
        axs[m].set_xticks(correlations, correlations)
fig.legend(method_types, loc='lower center')
fig.suptitle('Parameter '+str(parameters[par])+' - Average')
fig.subplots_adjust(top=0.80)
fig.show()
table = [['Dataset', model_types[0]+' '+method_types[0],
          model_types[0]+' '+method_types[1],
          model_types[1]+' '+method_types[0],
          model_types[1]+' '+method_types[1]],
          [dataset_types[0], average[0, 0, 0], average[0, 0, 1],
            average[1, 0, 0], average[1, 0, 1]],
          [dataset_types[1], average[0, 1, 0], average[0, 1, 1],
            average[1, 1, 0], average[1, 1, 1]],
          [dataset_types[2], average[0, 2, 0], average[0, 2, 1],
            average[1, 2, 0], average[1, 2, 1]],
          [dataset_types[3], average[0, 3, 0], average[0, 3, 1],
            average[1, 3, 0], average[1, 3, 1]],
          [dataset_types[4], average[0, 4, 0], average[0, 4, 1],
            average[1, 4, 0], average[1, 4, 1]],
          [dataset_types[5], average[0, 5, 0], average[0, 5, 1],

```



```

        average[1, 5, 0], average[1, 5, 1]])
print(tabulate(table, headers='firstrow', tablefmt='fancy_grid'))

```

D.3 Experiment 3

```

[ ]: #Generate data and train model

sim_data_eff = SimulatedDataset(10000, visit_times)
df_eff = sim_data_eff.df
df_eff.to_csv("./dataset/simdata_N5000_EXP3.csv", index=False)
df_eff = pd.read_csv("./dataset/simdata_N5000_EXP3.csv")

#Transform data to fit model
X_train, X_test, y_train, y_test = transform(df_eff)

#Train model
model = RandomSurvivalForest().fit(X_train, y_train)

[ ]: #1D-plots

no_instances = [500, 1000, 1500, 2000, 2500]
parameters = [10, 20, 40, 60, 80, 100, 200]

feature = 2 #we apply the interpretability methods on feature x3
feature_names = ['x_1', 'x_2', 'x_3', 'x_4', 'x_5']
no_repetitions = 4

#PDP
times_pdp = np.full((len(no_instances), len(parameters)), None)
for n in np.arange(len(no_instances)):
    for p in np.arange(len(parameters)):
        unique_values = len(np.unique(
            X_test.iloc[:no_instances[n], :].to_numpy(),
            [feature_names[feature]]))
        if unique_values >= parameters[p]:
            times = 0
            for r in np.arange(no_repetitions):
                begin = time.time()
                pdp_1d(model, X_test.iloc[:no_instances[n], :].to_numpy(),
                        feature, 0, parameters[p])
                end = time.time()
                times += end - begin
            times_pdp[n, p] = times/no_repetitions
table_pdp = [np.append(no_instances[0], times_pdp[0, :]),
              np.append(no_instances[1], times_pdp[1, :]),
              np.append(no_instances[2], times_pdp[2, :]),
              np.append(no_instances[3], times_pdp[3, :]),
              np.append(no_instances[4], times_pdp[4, :])]
print(tabulate(table_pdp,
               headers=['', parameters[0], parameters[1],

```

```

        parameters[2], parameters[3],
        parameters[4], parameters[5],
        parameters[6]], tablefmt='fancy_grid'))

#ICE
times_ice = np.full((len(no_instances), len(parameters)), None)
for n in np.arange(len(no_instances)):
    for p in np.arange(len(parameters)):
        unique_values = len(np.unique(
            X_test.iloc[:no_instances[n], :]
            [feature_names[feature]]))
        if unique_values >= parameters[p]:
            times = 0
            for r in np.arange(no_repetitions):
                begin = time.time()
                ice(model, X_test.iloc[:no_instances[n], :].to_numpy(),
                    feature, 0, parameters[p])
                end = time.time()
                times += end - begin
            times_ice[n, p] = times/no_repetitions

table_ice = [np.append(no_instances[0], times_ice[0, :]),
             np.append(no_instances[1], times_ice[1, :]),
             np.append(no_instances[2], times_ice[2, :]),
             np.append(no_instances[3], times_ice[3, :]),
             np.append(no_instances[4], times_ice[4, :])]
print(tabulate(table_ice,
               headers=['', parameters[0], parameters[1],
                       parameters[2], parameters[3],
                       parameters[4], parameters[5],
                       parameters[6]], tablefmt='fancy_grid'))

#ALE
times_ale = np.full((len(no_instances), len(parameters)), None)
for n in np.arange(len(no_instances)):
    for p in np.arange(len(parameters)):
        unique_values = len(np.unique(
            X_test.iloc[:no_instances[n], :]
            [feature_names[feature]]))
        if unique_values >= parameters[p]:
            times = 0
            for r in np.arange(no_repetitions):
                begin = time.time()
                ale_1d(model, X_test.iloc[:no_instances[n], :].to_numpy(),
                      feature, 0, parameters[p])
                end = time.time()
                times += end - begin
            times_ale[n, p] = times/no_repetitions

table_ale = [np.append(no_instances[0], times_ale[0, :]),
             np.append(no_instances[1], times_ale[1, :]),

```

```

        np.append(no_instances[2], times_ale[2, :]),
        np.append(no_instances[3], times_ale[3, :]),
        np.append(no_instances[4], times_ale[4, :]))
print(tabulate(table_ale,
               headers=['', parameters[0], parameters[1],
                       parameters[2], parameters[3],
                       parameters[4], parameters[5],
                       parameters[6]], tablefmt='fancy_grid'))

```

[]: *#2D-plots*

```

no_instances = [500, 1000, 1500, 2000, 2500]
parameters = [5, 10, 20, 30, 40]

feature1 = 2 #x3
feature2 = 3 #x4

#PDP
times_pdp = np.full((len(no_instances), len(parameters)), None)
for n in np.arange(len(no_instances)):
    for p in np.arange(len(parameters)):
        unique_values1 = len(np.unique(
            X_test.iloc[:no_instances[n], :]
            [feature_names[feature1]]))
        unique_values2 = len(np.unique(
            X_test.iloc[:no_instances[n], :]
            [feature_names[feature2]]))
        if (unique_values1 >= parameters[p] and
            unique_values2 >= parameters[p]):
            times = 0
            for r in np.arange(no_repetitions):
                begin = time.time()
                pdp_2d(model, X_test.iloc[:no_instances[n], :].to_numpy(),
                       feature1, feature2, 0, parameters[p])
                end = time.time()
                times += end - begin
            times_pdp[n, p] = times/no_repetitions

table_pdp = [np.append(no_instances[0], times_pdp[0, :]),
             np.append(no_instances[1], times_pdp[1, :]),
             np.append(no_instances[2], times_pdp[2, :]),
             np.append(no_instances[3], times_pdp[3, :]),
             np.append(no_instances[4], times_pdp[4, :])]
print(tabulate(table_pdp,
               headers=['', parameters[0], parameters[1],
                       parameters[2], parameters[3],
                       parameters[4]], tablefmt='fancy_grid'))

#ALE
times_ale = np.full((len(no_instances), len(parameters)), None)
for n in np.arange(len(no_instances)):

```

```

for p in np.arange(len(parameters)):
    unique_values1 = len(np.unique(
        X_test.iloc[:no_instances[n], :][
            feature_names[feature1]]))
    unique_values2 = len(np.unique(
        X_test.iloc[:no_instances[n], :][
            feature_names[feature2]]))
    if (unique_values1 >= parameters[p] and
        unique_values2 >= parameters[p]):
        times = 0
        for r in np.arange(no_repetitions):
            begin = time.time()
            ale_2d(model, X_test.iloc[:no_instances[n], :].to_numpy(),
                    feature1, feature2, 0, parameters[p])
            end = time.time()
            times += end - begin
        times_ale[n, p] = times/no_repetitions

table_ale = [np.append(no_instances[0], times_ale[0, :]),
             np.append(no_instances[1], times_ale[1, :]),
             np.append(no_instances[2], times_ale[2, :]),
             np.append(no_instances[3], times_ale[3, :]),
             np.append(no_instances[4], times_ale[4, :])]
print(tabulate(table_ale,
               headers=['', parameters[0], parameters[1],
                       parameters[2], parameters[3],
                       parameters[4]], tablefmt='fancy_grid'))

```

Appendix E

Hardware Specifications

Operating System: Microsoft Windows 10 Home

System Type: x64-based PC

CPU: Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz, 2201 MHz, 2 cores, 4 logical processors

RAM: 16GB, DDR3, 1600 MT/s

Environment: Jupyter Notebook

Kernel: ipykernel