Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Frida Svendal Aase

An Exploration of Shapley Values for Model Interpretability: Providing a Fair and Accurate Explanation of Black Box Models

Master's thesis in Applied Physics and Mathematics
Supervisor: Kjersti Aas
June 2023

NTNU
Norwegian University of
Science and Technology

Frida Svendal Aase

An Exploration of Shapley Values for Model Interpretability: Providing a Fair and Accurate Explanation of Black Box Models

Master's thesis in Applied Physics and Mathematics
Supervisor: Kjersti Aas
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Preface

This thesis, with course code TMA4900, completes my master's thesis in Applied Physics and Mathematics at the Norwegian University of Science and Technology (NTNU), Department of Mathematics. The work was completed in the Spring of 2023.

I would like to thank my supervisor Kjersti Aas for her excellent guidance and supervision with both my specialization project and master's thesis. Her feedback and advice have been invaluable. Thank you also to Ian Covert for answering our questions regarding the FastSHAP method. In addition, thank you to my friends and family for their support and for rooting for me, and to my classmates for keeping me company at university during the long work hours.

Trondheim, June 2023

Frida Svendal Aase

Abstract

Shapley values are popular for explainable artificial intelligence due to their solid theoretical foundation of correctness and fairness. However, the computation of the Shapley values is costly, and therefore, infeasible in many real-world problems. To reduce the cost, Shapley value estimators are used. The Shapley value estimation can be divided into two steps. The first step is the contribution function estimation. The contribution function is meant to capture how each feature affects the prediction of the black box model, and a common choice is the expected value of the model conditioned on a subset of the features being observed, which often is analytically intractable. Two estimators are considered in this thesis. The off-manifold method [1, 2] assumes feature independence, often leading to a bias for real-world data sets. The surrogate model [3] is a supervised machine learning model trained to approximate the contribution function. The second estimation step is to estimate the Shapley values given an estimate of the contribution function. We consider two estimators, KernelSHAP [2] and FastSHAP [4]. Out of the exponential number of feature combinations considered in the exact Shapley values, KernelSHAP reduces the cost by considering a small subset of the most important ones. FastSHAP is a machine learning model trained to estimate the Shapley values.

We perform an in-depth empirical study of the estimators based on computational cost and accuracy for simulated and real-world data sets. By clearly separating the estimation steps and evaluating the methods on simulated data sets where the ground truth is known, we provide new insights into the methods. We find that the off-manifold method is more accurate for smaller data sets, especially if the features are "nearly independent". In contrast, for larger data sets, the surrogate model is more accurate, especially if the features are "far from independent". The surrogate model is clearly faster, making it preferable in many real-world problems. Both of the Shapley value estimators are fast when the contribution function is given. The KernelSHAP method is more accurate than the FastSHAP method if enough feature combinations are considered, especially for smaller data sets. In practice, increasing the number of feature combinations in KernelSHAP requires increasing the number of estimates of the contribution function. Thus, if KernelSHAP is combined with the slower off-manifold method, increasing the accuracy will lead to slower computation, which must be repeated for every instance's prediction. FastSHAP has the advantage that after an initial training procedure, the full two step estimation procedure can be performed by a single model evaluation per instance of interest. For a real-world data set, we find that the relative performance of the methods aligns with the results from the simulation study.

Sammendrag

Shapley-verdier er en populær metode innenfor forklarbar kunstig intelligens på grunn av et solid teoretiske grunnlag for deres korrekthet og rettferdighet. Imidlertid er beregningen av Shapley-verdier kostbar og derfor ikke gjennomførbart i mange praktiske situasjoner. For å redusere kostnaden brukes Shapley-verdiestimater. Estimeringen av Shapley-verdier kan deles inn i to trinn. Det første trinnet er å estimere bidragsfunksjonen. Bidragsfunksjonen skal fange opp hvordan hver kovariat påvirker prediksjonen til maskinlæringsmodellen, og et vanlig valg er forventningsverdien til modellen betinget på at en delmengde av kovariatene er observert. Denne forventningsverdien er ofte analytisk utilgjengelig. To estimatorer betraktes i denne avhandlingen. Metoden "off-manifold" (norsk: av eller utenfor manifolden) [1, 2] antar uavhengighet mellom kovariatene, noe som ofte fører til skjevhet (engelsk: bias) for virkelige datasett. Surrogatmodellen [3] er en veiledet (engelsk: supervised) maskinlæringsmodell som er trent for å tilnærme bidragsfunksjonen. Det andre estimeringstrinnet er å estimere Shapley-verdiene gitt et estimat av bidragsfunksjonen. Vi betrakter to estimatorer, KernelSHAP [2] og FastSHAP [4]. Av de eksponentielt mange kombinasjonene av kovariater som betraktes i de eksakte Shapley-verdiene, reduserer KernelSHAP kostnaden ved å betrakte en liten delmengde av de viktigste kombinasjonene. FastSHAP er en maskinlæringsmodell som er trent for å estimere Shapley-verdiene.

Vi utfører en grundig empirisk studie av estimatorene basert på beregningskostnad og nøyaktighet for simulerte og virkelige datasett. Ved å tydelig separere estimeringstrinnene og evaluere metodene på simulerte datasett der sannheten er kjent, gir vi nye innsikter i metodene. Vi finner at off-manifold-metoden er mer nøyaktig for mindre datasett, spesielt hvis kovariatene er "nesten uavhengige". Derimot for større datasett, er surrogatmodellen mer nøyaktig, i hvert fall hvis kovariatene er "langt fra uavhengige". Surrogatmodellen er klart raskere, noe som gjør den foretrukket i mange situasjoner. Begge Shapley-verdiestimatorene er raske når bidragsfunksjonen er gitt. KernelSHAP-metoden er mer nøyaktig enn FastSHAP-metoden hvis tilstrekkelig mange kombinasjoner av kovariater betraktes, spesielt for mindre datasett. I praksis krever økning av antall kombinasjoner av kovariater i KernelSHAP økning av antall estimater av bidragsfunksjonen. Dermed, hvis KernelSHAP kombineres med den langsommere off-manifold-metoden, vil økning i nøyaktighet føre til tregere beregning, som må gjentas for hver enkelt observasjons prediksjon. FastSHAP har fordelen av at etter treningen av modellen er utført, kan hele den totrinns estimeringsprosedyren utføres med en enkelt modellevaluering per observasjon som forklares. Vi finner at ytelsen til metodene for et virkelighetsdatasett samsvarer med resultatene fra simulasjonsstudien.

Table of Contents

List of Figures

1 Introduction

Machine learning has emerged as an important tool in various fields. The complexity of machine models has increased significantly, leading to improved predictive abilities, but resulting in models that are difficult for humans to interpret. Model interpretability or explainability is essential in order to e.g. detect model biases, build trust in the models and ensure model robustness [5]. The need for interpretability has led to the development of the field **eXplainable Artificial Intelligence (XAI)**, where various methods are developed in order to interpret complex machine learning models. A report from the Finance Sector Union of Norway (Finansforbundet) [6], points out that XAI is crucial for the application of machine learning models in the Norwegian financial industry.

Within the field of XAI, there are several approaches. One approach is to stick to using models that are **intrinsically** interpretable. However, this would rule out the use of some models that have state-of-the-art performance. In contrast to this, one can perform **post hoc** interpretability, where a model is interpreted after training e.g. by analyzing the inputs and outputs of the model. Within the domain of post hoc interpretability, there are **model-agnostic** and **model-specific** methods. Model-agnostic methods aim at interpreting *any* kind of machine learning model, and can therefore only consider the inputs and outputs of the model, and must treat the model itself as a "black box". In contrast, model-specific methods aim at explaining a particular kind of machine learning model, e.g. a random forest or a convolutional neural network. Model-specific methods can consider the parameters, such as the weights in a neural network, and the structure of the model. Moreover, some interpretability methods aim at explaining the overall workings of a model. These are referred to as **global**, whereas other methods aim at explaining an individual prediction of the model for a particular observation, which is referred to as a **local** method.

In this thesis, the focus will be on using **Shapley values** [7] as a local model-agnostic explanation method [1]. The Shapley values can also be used for global interpretability. The Shapley values are a fair and correct way, defined through satisfying some desirable criteria [1, 2, 8], to attribute the prediction of a model to its features. Although the Shapley values arguably provide a fair and correct explanation, the computational cost of the Shapley values is exponential in the number of features in the model, and therefore, infeasible to compute in many situations. To apply the method, it is necessary to develop estimation methods with a relatively low computational cost and provide accurate estimates of the Shapley values. The Shapley value estimation procedure can be divided into two steps. In order to fully evaluate the estimation methods and determine where each method breaks down, each estimation step is considered separately. The first step is to estimate the **contribution function**, which is meant to capture how the black box model's prediction changes conditioned on different combinations of features being observed. The second step is the estimation of the Shapley values given the estimates of the contribution function.

A common choice of contribution function estimator is a Monte Carlo integral under the assumption of feature independence [1, 2]. In this thesis, this estimator will be referred to as the off-manifold estimate because the generated Monte Carlo samples will often lay off the data manifold [3]. The feature independence assumption and the method's high computational cost are drawbacks. To incorporate the dependence of the features in the contribution function estimate, Aas, Jullum and Løland [8] propose parametric and non-parametric methods. However, due to the computational cost of these methods, they do not scale to more complex higher dimensional problems. Another strategy, proposed by Frye et al. [3] and Jethani et al. [9], uses supervised and generative machine learning models, neural networks, to learn to estimate the contribution function. Due to the high representation ability of neural networks, this strategy is promising [3, 9, 10], but suffer from drawback like the need for developing robust strategies for evaluating and tuning the hyperparameters of the model [10]. Another drawback of this method is that it is somewhat paradoxical to use a black box machine learning model as part of explaining another black box machine learning model.

Moving on to the second estimation step, Lundberg and Lee [2] propose both model-agnostic and model-specific Shapley value estimators. One of their proposed methods is KernelSHAP, which is a model-agnostic Shapley value estimation method that has become popular through the SHAP library [2]. As previously mentioned, the computation of the exact Shapley values is exponential in the number of features. This is because exponentially many combinations of features are considered.

KernelSHAP exploits that out of the exponentially many combinations, the contribution of some combinations is negligible compared to others. Therefore, it approximates the full problem by choosing a small subset of the most important combinations, reducing the computational cost significantly. A more recent Shapley value estimator, FastSHAP, was introduced by Jethani et al. [4]. FastSHAP is a machine learning model, a neural network, trained to approximate the Shapley values. Again, like for the surrogate model, FastSHAP raises the question of whether a non-interpretable machine learning model is suitable for explaining another machine learning model. FastSHAP is a method with a higher initial computational cost, the training of the model, but the cost is amortized across the number of explanations to be explained. After the initial training has been performed, any number of observations can be explained at a low cost, corresponding to a single model evaluation. This is desirable in practical applications where explanations must be provided within a short time frame. In contrast, in the KernelSHAP method, the computation related to generating an explanation must be repeated for every instance that is to be explained. Therefore, especially when combined with the slower off-manifold contribution function estimator, it is less suitable in situations where a fast generation of explanations is important.

Both KernelSHAP and FastSHAP were introduced with a default choice of contribution function estimators. In previous work, such as [2, 4, 11], the full two step estimation procedure is evaluated as a whole. To fully evaluate the estimation in terms of accuracy and computational cost, we distinguish the two steps from each other and evaluate each step on its own. By separating the steps, we hope to provide new insight into the properties of each method, determine where in the full estimation procedure the error in the final approximation originates, and how the computational cost is distributed between the steps. In this thesis, we consider the off-manifold and surrogate estimates of the contribution function. In addition, fixing the estimate of the contribution function, the KernelSHAP and FastSHAP estimates are evaluated. The surrogate model and FastSHAP are recent methods, and in [3] and [4] the methods are tested on some real-world data sets. However, as novelty work, we performed an in-depth study of the methods' performance on simulated data sets with varying empirical properties where the ground truth Shapley values are known.

As previously mentioned, in the off-manifold estimate, feature independence is assumed. Feature independence is rarely observed in real-world problems, therefore, in many practical situations, the assumption will not hold. To investigate the effect of falsely assuming independence, we test the method on simulated data sets where we vary the degree of linear correlation between the features. Moreover, the surrogate and FastSHAP models consist of neural networks which require sufficiently large data sets to yield high-precision predictions. Therefore, in the simulated data sets, the number of training observations is varied in order to investigate whether this affects the accuracy of the models' predictions. Moreover, the data generating distribution is varied in the simulations to determine how the "difficulty" of the data affects the estimators. The Shapley value estimators are most needed in high-dimensional problems since the computation of the exact Shapley values is exponential in the number of features in the model. It is, therefore, interesting to determine whether the accuracy of the estimators varies based on the number of features in the black box model. Based on the empirical properties of the simulated data sets, we outline the relative performance of the methods and present results that may indicate a priori which method is to be preferred in real-world applications based on the properties of the data set. Finally, the estimation methods are tested and evaluated on a real-world data set.

The outline of this thesis is as follows. Some concepts from machine learning and statistics are presented as background material in Section 2. In Section 3, we describe the origin of the Shapley values and the theory of the Shapley values as an explanation method. Moreover, the Shapley value and contribution function estimation methods are described in detail. We outline the full estimation procedures and discuss the computational cost of the methods. The methods are tested on simulated data, and the results from this study are presented in Section 4. Then, the results of applying the methods to a real-world data set are presented in Section 5. In all our experiments, we use our implementation of the methods, which can be found on GitHub. The implementation of the surrogate and FastSHAP models are adaptations of the FastSHAP implementation in TensorFlow [4]. The results are summarized and discussed in Section 6 before we conclude and point out some future work in Sections 7.1 and 7.2, respectively.

Some of the sections of this thesis are, with minor alterations, copied from my specialization

project, Aase [12]. In Chapter 2, Sections 2.1.1, 2.2.1 and 2.4 are taken from [12]. In Sections 2.2.1 and 2.4, some information has been added and minor alterations have been made e.g. to improve the notation. In Chapter 3, Sections 3.1 to 3.3 and 3.5.1 have, with minor alterations, been copied from [12]. Lastly, in Chapter 5, Sections 5.1 and 5.2 are copied from [12] with minor modifications.

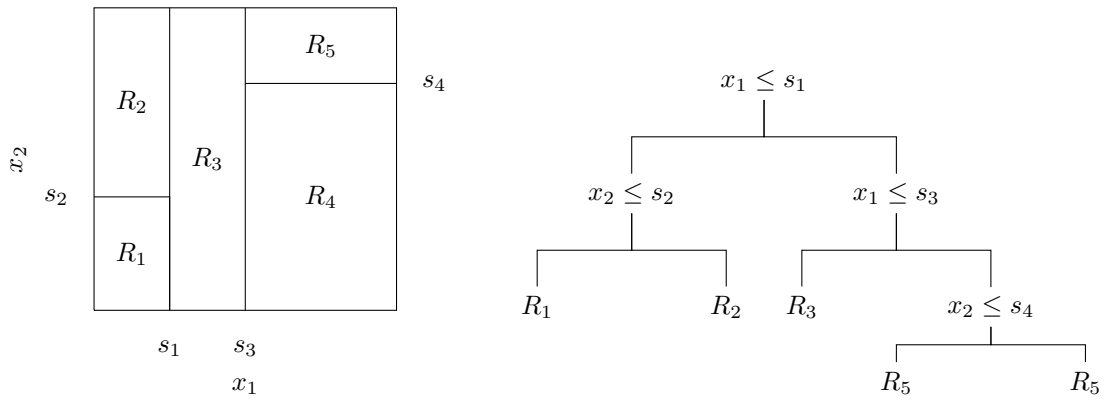


Figure 1: An example of the splitting region of a CART with two features x_1 and x_2 is shown on the left. The corresponding tree representation is shown on the right. These illustrations are based on similar illustrations in Chapter 9 of Hastie, Tibshirani and Friedman [13].

2 Background Material

This section will present some relevant topics from statistics and machine learning. A general overview of the field of machine learning can be found in my specialization project [12] and is not given in this report. The majority of Sections 2.1.1, 2.2.1 and 2.4 are taken from my specialization project [12] and are included in this report for completion.

2.1 An Intrinsically Interpretable Machine Learning Model

In the following section, an intrinsically interpretable machine learning model is presented. The model will also be an important building block in the XGBoost model that will be described in Section 2.2.1.

2.1.1 Classification and Regression Tree (CART)

A tree is a model f that divides the feature space into disjoint (high-dimensional) rectangles or boxes $R_1, R_2, \ldots, R_M$ and predicts the same value c_m for all instances in a rectangle. The splitting rules can be represented by a tree, for an illustration see Figure 1. Mathematically, this can be expressed as

$$f(\boldsymbol{x}) = \sum_{m=1}^{M} c_m \mathbb{1}\{\boldsymbol{x} \in R_m\},$$

where

$$\mathbb{1}\{\boldsymbol{x} \in R_m\} = \begin{cases} 1, & \boldsymbol{x} \in R_m, \\ 0, & \boldsymbol{x} \notin R_m, \end{cases}$$

is the indicator function. There are several tree algorithms. The one that will be considered here is the one presented by Lewis [14], known as CART, an abbreviation for Classification And Regression Tree. As the name suggests, trees can be used for both regression and classification, although the specifics of the algorithm are different in the two cases. In this section, we follow the notation of Hastie, Tibshirani and Friedman [13].

Splitting the whole feature space in an optimal composition of rectangles is a complex problem. Therefore, in practice, simplifying the search space into only considering recursive binary partitions is necessary. This means that the only possible choice at each step is to split one of the existing rectangles into two. For instance, adding a rectangle in the middle of one of the existing rectangles is not allowed. The algorithm works by first splitting the feature space into two rectangles and proceeds by dividing one of them into two, and so forth. To consider all possible combinations of

such splits is not feasible. Therefore, one greedily approaches the problem. This means that at every split, one chooses the split giving the most improvement at the current time. One does not consider whether it would be possible to achieve a better solution in e.g. ten steps from now. This greedy search solves the problem of how to divide the feature space into disjoint rectangles.

It remains to find the feature and corresponding feature value for which splitting a currently existing rectangle into two will give the largest improvement. To do this, a loss function is used. In machine learning, a loss function usually measures the model's fit to the data. The loss function measures the amount by which the model's predictions differ from the response variables in the training data. This is also the case for determining the splits in CART. Given the loss function, one can determine the best value to predict within each rectangle. There are many different choices of loss functions. It depends on the nature of the problem, whether it is classification or regression. In addition, some loss functions have analytical minimizers, whereas others can only be numerically approximated. Non-convex functions are generally hard to optimize numerically. Thus, the loss function must be chosen with care. The details of CART will be treated for one such choice, the sum of squared losses in the regression setting.

The sum of squares loss is given as $\sum_{i=1}^{n}(y_i - f(\boldsymbol{x_i}))^2$, which is minimized by $f(\boldsymbol{x_i}) = \frac{1}{n}\sum_{i=1}^{n} y_i$. Thus, in each rectangle, the average value of the response variables (of the instances inside the rectangle) is predicted, i.e. $\hat{c}_m = \frac{1}{n_m}\sum_{\boldsymbol{x}_i \in R_m} y_i$, for $m = 1, 2, \ldots, M$, where n_m is the number of instances in rectangle R_m. Given the split of the feature space, determining the best value to predict is easy. However, the problem of splitting the feature space is harder to solve. Mathematically, if there are two rectangles R_1 and R_2 the splitting problem can be expressed as

$$\min_{j,s}\left\{\min_{c_1}\sum_{\boldsymbol{x}_i \in R_1(j,s)}(y_i - c_1)^2 + \min_{c_2}\sum_{\boldsymbol{x}_i \in R_2(j,s)}(y_i - c_2)^2\right\},$$

where the solutions of the inner two problems are the averages within each rectangle. Considering a feature j, determining the point s to split is easy as we can scan through the values of the feature and choose the optimal value as the one with the lowest loss. Then, to find the best split at a certain time, it is possible to iterate through all the features and find the best splitting point for each feature. Having the best splits for each feature, the final split is chosen as the split over the feature reducing the loss the most.

One thing that remains is to choose how many splits to perform and thereby determine the tree's topology. If there are too many splits, for instance, if there is one rectangle for each instance in the data, the model likely overfits the data. A common choice of stopping criteria is to stop when the change in the loss is reduced by very little in each iteration. However, it is possible that the current split leads to little improvement, whilst the next might have a big impact. Therefore, it is necessary to use another strategy. The way CART does this is to build a large tree and then simplify it, whilst maintaining a sufficiently high accuracy, rather than stopping the tree building at a predefined point. The "simplification" that is performed is called cost-complexity-pruning. In this procedure, one specifies a minimum number of instances that should be contained per node. Thus, the feature space cannot be split into too small regions. The large tree T_0 one starts with is built until the next split divides the space into two regions where at least one contains fewer instances than the predefined minimum number of allowed instances. Then, the idea is to consider subtrees T of T_0 found by joining some internal nodes of the tree and simultaneously minimizing the complexity of the subtree and the error rate of the corresponding predictor. A hyperparameter controls the amount of pruning and can be determined, for instance, by cross-validation. For more details on cost-complexity-pruning and other details of the CART-algorithm, such as the treatment of categorical variables, see Chapter 9 of Hastie, Tibshirani and Friedman [13].

2.1.1.1 Interpretability of Trees

To demonstrate the interpretability of trees, a practical example will be considered. Let's say the goal is to classify a person as having lung cancer or not. As features in the model, we include age and smoking. Figure 2 shows a possible tree model for this example. Trees are thought to be good at mimicking how humans make decisions. First, what is thought to be the most important feature

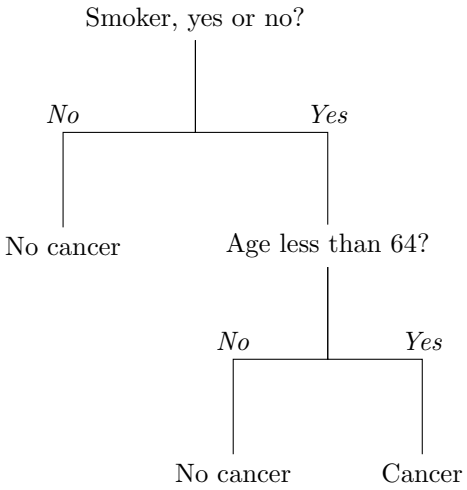


Figure 2: A simple tree where the first split is done based on whether or not an individual smokes. If not, the model predicts that the individual does not have cancer. If yes, the model then considers whether the person is above or below 64 years old. If so, the individual is predicted to have cancer; if not, the model predicts no cancer. Note that this is a very simplified example not based on real-world data. It is just meant as an illustration of the interpretation of a tree model.

is considered. Then, after evaluating this factor, the second most important factor is considered, given the conclusion based on the first factor, and so on. Trees are often even easier for laypeople to understand than linear regression models, which are generally considered interpretable when there are few features in the model. To remain interpretable, the trees cannot be too deep. The depth of the tree refers to how many levels of splits there are in the tree.

2.2 Some Examples of "Black Box" Machine Learning Models

In this section, two examples of non-interpretable, "black box" machine learning models are described. We start by introducing boosting [15] and XGBoost [16] in the following section. Then, in Section 2.2.2, feed-forward neural networks are described.

2.2.1 Gradient Boosting and XGBoost

Boosting is a machine learning method that composes weak learners to form a strong learner. In classification settings, a **weak learner** is defined as a machine learning model with accuracy slightly better than pure guessing. This weak learner can, for instance, be a decision tree. A **strong learner** is defined as a model with high accuracy, where what is considered a "high" accuracy depends on the problem. The idea of **boosting** is to "boost" or improve a set of weak learners to form a strong learner. In practice, many boosting methods are based on training weak learners iteratively on a training set weighted based on the accuracy of the weak learner in the previous iteration. Instances that were misclassified in the previous iteration get a higher weight, and the weak learner is then forced to focus on learning these instances. Since the weights are adapted during the training, the method is referred to as "adaptive". The final model, or strong learner, is a weighted average of the weak learners, weighted by their accuracy. The first adaptive boosting algorithm was the AdaBoost [15] algorithm that won the prestigious Gödel prize.

This idea of boosting as a weighted average of weak learners trained on a data set adapted iteratively to "force" the weak learners to focus on the misclassified points provides the intuition behind the method and is how the first boosting algorithms worked. However, it was later discovered that these boosting algorithms could be seen as a minimization problem. This point of view has allowed the boosting method to be generalized and robustified. Specifically, boosting can be seen as performing gradient descent, which is a numerical optimization method that will be treated in Section 2.2.2.3,

in function space. This procedure is known as **gradient boosting** and was discovered by Friedman [17]. It has been important for the further development of the boosting method. The specifics of formulating and proving the equivalence of boosting from both perspectives is quite tedious and can, for instance, be found in Chapter 10 of Hastie, Tibshirani and Friedman [13]. In the following, we focus on gradient boosting and specifically the implementation available in the open-source XGBoost library [16], available in various programming languages. XGBoost is an abbreviation for eXtreme Gradient Boosting. XGBoost has demonstrated state-of-the-art performance in many machine learning competitions, especially for tabular data [16]. XGBoost can be used for both regression and classification.

2.2.1.1 The Details of XGBoost

To express the mathematical details of the gradient boosting technique exploited by XGBoost, we follow the notation of Chen and Guestrin [16]. This paper was published after the XGBoost-library had become popular and gave the theoretical details of the method.

The XGBoost model is a composition, or **ensemble**, of trees and can be expressed as

$$\hat{y}_i = \sum_{k=1}^{K} f_k(\boldsymbol{x}_i), \tag{1}$$

where $\hat{y}_i$ is the prediction of the model for a feature observation $\boldsymbol{x}_i$ with corresponding true response variable y_i. The index i is in $\{1, 2, \dots n\}$, and n is the number of observations in the available data set. Moreover, f_k is a CART, which was described in Section 2.1.1, and K denotes the total number of CARTs. To improve upon other boosting versions, the XGBoost method not only considers the model's loss but also considers a regularization term $\omega(f_k)$ penalizing the complexity of the tree f_k. Then, the objective function used by XGBoost can be expressed as

$$\text{obj}(\theta) = \sum_{i=1}^{n} \mathcal{L}(y_i, \hat{y}_i) + \sum_{k=1}^{K} \omega(f_k), \tag{2}$$

where $\mathcal{L}(y_i, \hat{y}_i)$ is a loss function measuring the similarity between the true response variable y_i and the prediction $\hat{y}_i$. The ensemble in XGBoost is trained iteratively, where at iteration t, the ensemble model is given as

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(\boldsymbol{x}_i), \quad \hat{y}_i^{(0)} = 0, \tag{3}$$

where the tree $f_t(\boldsymbol{x}_i)$ is added to the previous model $\hat{y}_i^{(t-1)}$ in iteration t. Thus, the objective function at iteration t is

$$\text{obj}^{(t)} = \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t)}\right) + \sum_{i=1}^{t} \omega(f_i) = \sum_{i=1}^{n} \mathcal{L}\left(y_i, \hat{y}_i^{(t-1)} + f_t(\boldsymbol{x}_i)\right) + \omega(f_t) + \text{constant}. \tag{4}$$

One of the advantages of gradient boosting, in contrast to additive boosting algorithms such as AdaBoost [15], is that any loss function, which has a defined gradient and Hessian, can be used, which allows for generalizing the model. However, the objective in (4) cannot be analytically minimized for any loss function. Therefore, Chen and Guestrin [16] proposes to use a second-order Taylor expansion of the loss function as an approximation. Accordingly, at every iteration, the approximate objective becomes

$$\text{obj}^{(t)} = \sum_{i=1}^{n} \left[\mathcal{L}\left(y_i, \hat{y}_i^{(t-1)}\right) + g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i) \right] + \omega(f_t) + \text{constant} \tag{5}$$

where the gradient g_i and Hessian h_i of the loss function are defined as

$$g_i = \partial_{\hat{y}_i^{(t-1)}} \mathcal{L}(y_i, \hat{y}_i^{(t-1)}), \quad h_i = \partial^2_{\hat{y}_i^{(t-1)}} \mathcal{L}(y_i, \hat{y}_i^{(t-1)}). \tag{6}$$

The expression $\mathcal{L}(y_i, \hat{y}_i^{(t-1)}), i = 1, 2, \ldots, n$ does not depend on the tree f_t that is added to the model. And since this is the quantity we minimize with respect to, the term acts as a constant in the optimization problem, and we can equivalently minimize the objective

$$\widetilde{\text{obj}}^{(t)} = \sum_{i=1}^{n} \left[g_i f_t(\boldsymbol{x}_i) + \frac{1}{2} h_i f_t^2(\boldsymbol{x}_i) \right] + \omega(f_t). \tag{7}$$

This illustrates the advantage of XGBoost; the objective function only depends on the gradient and the Hessian of the loss, allowing any loss function, that is two times differentiable, to be used.

To fully write down the explicit formula for the iterative updates, we need to define the complexity penalty term in the objective. First, we explicitly write down the definition of the tree model as $f_t(x) = w_{q(x)}, w \in R^T, q : R^d \to \{1, 2, \cdots, T\}$, where w is the vector containing scores on leaves, T is the number of leaves in the tree and q is a function assigning each data point to the corresponding leaf. Then, the definition of the complexity term used in XGBoost is

$$\omega(f) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2, \tag{8}$$

where γ is a penalty parameter on the number of leaves in the tree and λ is an L2-regularization penalty parameter on the scores vector w. Denoting by $I_j = \{i | q(\boldsymbol{x}_i) = j\}$ the set of indices of data points assigned to the jth leaf, the objective function in (7) can be written as

$$\begin{aligned} \widetilde{\text{obj}}^{(t)} &= \sum_{i=1}^{n} \left[g_i w_{q(\boldsymbol{x}_i)} + \frac{1}{2} h_i w_{q(\boldsymbol{x}_i)}^2 \right] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} \left[\Big(\sum_{i \in I_j} g_i \Big) w_j + \frac{1}{2} \Big(\sum_{i \in I_j} h_i + \lambda \Big) w_j^2 \right] + \gamma T, \end{aligned} \tag{9}$$

where the second equality is possible because all instances in the same leaf get the same score.

Defining $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$, the objective takes the form

$$\widetilde{\text{obj}}^{(t)} = \sum_{j=1}^{T} \left[G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma T. \tag{10}$$

The score in each leaf w_j does not depend on any other $w_i, i \neq j$, so each term in the sum can be optimized for the corresponding w_j while keeping the structure $q(\boldsymbol{x})$ constant. This yields the optimal scores $w_j^* = -\frac{G_j}{H_j + \lambda}$. Inserting this in the objective gives the corresponding best objective reduction as

$$\widetilde{\text{obj}}^{*} = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T. \tag{11}$$

Now it remains to find the best tree structure. Because the space of all possible tree structures is very large, it is infeasible to search through the whole space in practice. Therefore, XGBoost greedily optimizes one level of the tree at a time by splitting one leaf into two leaves and evaluating the score improvement to decide if the split is sufficiently good to proceed with it. At each level, the algorithm searches through the space of all possible splits of one leaf into two and proceeds with the one with the biggest improvement in the overall score. To evaluate the gain of the split, consider

$$\text{gain} = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma. \tag{12}$$

The term in brackets is the combined score of the proposed leaves and the original leaf. Specifically, the first term inside the brackets is the score of the new leaf to the left, indexed by L. The second term is the score of the new leaf to the right, indexed by R. The score of the original leaf is the third term in the brackets. The last term in the gain of the split is a regularizing term $-\gamma$ corresponding to the penalty of adding a single leaf node to the tree. Accordingly, this expression

is positive only if the gain of adding the leaf is greater than γ. Thus, this objective penalizes too complex tree structures, thereby preventing overfitting. The splitting procedure that has been described here, will at any time enumerate all possible such splits on all features and choose the best one. Chen and Guestrin [16] refer to this as an "exact greedy" splitting procedure. This is one of the splitting procedures that can be chosen in the XGBoost library. It is expensive to consider all possible splits over all the features when the features are continuous. Therefore, several other, less computationally expensive, approximative algorithms for splitting are presented in [16] and implemented in the corresponding library.

Splitting over Categorical Features

As of version 1.5, XGBoost has support for categorical features. Moreover, from version 1.6, the optimal partitioning technique, which optimality was proven by Fisher [18], has been incorporated in the XGBoost library to handle splitting for categorical variables. In our experiments, XGBoost version 1.7.3 is used, meaning that support for categorical variables is available.

2.2.1.2 XGBoost in the Context of XAI

Since the models built by boosting methods are a composition of weak learners, their interpretability is very low. With CART models as the underlying weak learners, the final model will typically be a combination of hundreds of CARTs. Therefore, the transparency of calculating each prediction is low. However, due to their state-of-the-art performance on many machine learning problems, it is desirable to use XGBoost models. This makes such models a good example to illustrate the use of XAI. Although some techniques are specifically designed to explain the predictions of an XGBoost model, as seen in e.g. [19], one might also use model-agnostic methods, in which the role of the XGBoost-model is to act as a black box model. This is the theme of Chapter 3, where two model-agnostic explanation methods will be presented. In the following section, another example of a machine learning black box model is presented.

2.2.2 Feed-Forward Neural Networks

Deep learning models such as feed-forward neural networks are perhaps the most famous example of non-interpretable machine learning models. Deep learning models can consist of millions of parameters and are very complex. They have a high representation capacity and therefore perform well for many real-world problems. In this section, feed-forward neural networks are reviewed. Generally, neural networks are a very flexible and complex type of machine learning model. In this section, we try to give a brief outline of the method and provide some detail on topics that are especially relevant to this thesis. Neural networks are building blocks in the FastSHAP and surrogate methods, which are investigated in this thesis and will be presented in Sections 3.4 and 3.5.2, respectively. Deep neural networks are examples of non-interpretable machine learning black box models and, therefore, also illustrate the need for XAI. The outline of this section is as follows. Firstly, the general feed-forward neural networks are presented. Then, some choices of loss functions, activation functions, and optimization methods for neural networks are explained in more detail. Moreover, some other aspects of neural network training are described. Lastly, the universal approximation theorem is presented, which gives a theoretical foundation for why the class of neural networks is useful for finding good approximators in a wide range of applications. To start, some mathematical details of a feed-forward neural network are given.

To train a neural network, a training data set is necessary. In this thesis, tabular data is considered, therefore consider a training data set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ of feature observations $\boldsymbol{x}_i \in \mathbb{R}^q$ and corresponding response variables y_i. It is assumed that the response variables were generated by a function $f(\boldsymbol{x})$ and that the response variables are noisy observations of this function evaluated at the features. In machine learning the goal is to approximate $f(\boldsymbol{x})$ by a function that is commonly denoted $\hat{f}(\boldsymbol{x})$. The function $\hat{f}(\boldsymbol{x})$ is learned from the data. A neural network is an example of such a function.

The components of a deep neural network can be visualized using a graph. Each input in the model,

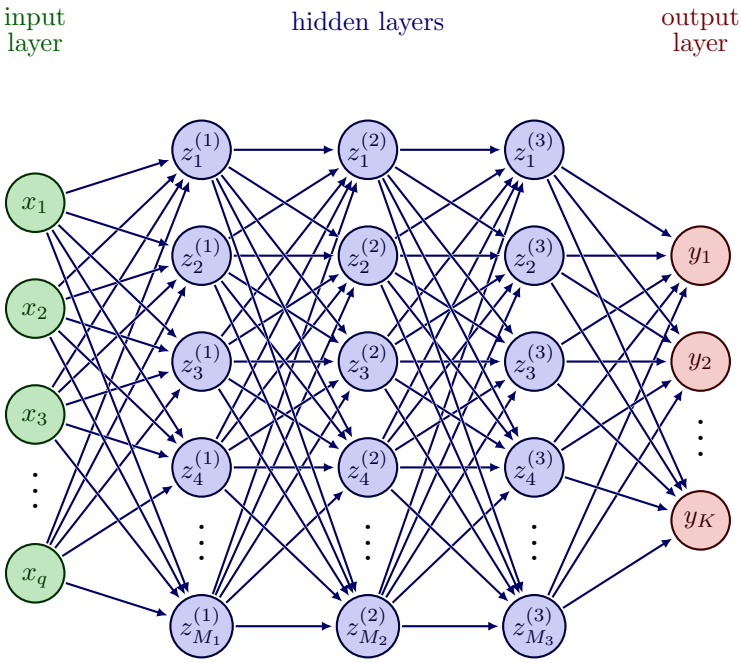


Figure 3: An illustration of a fully connected feed-forward neural network with three hidden layers. The illustration has been adapted from here [20]. In this example, each observation consists of q features represented by the input layer. The hidden layers consist of M_1, M_2, and M_3 nodes, and the output layer consists of K nodes, where each node represents one of the K classes in a classification problem. In a regression problem, $K = 1$, and the output node represents the predicted value of the response variable.

corresponding to a feature, and output, corresponding to the response variable, can be represented by a node in the graph. The inputs make up a "layer" in the network, and likewise, so do the outputs. Between the layer of inputs and the layer of outputs, there are layers of nodes called hidden layers. The nodes in the hidden layers receive information from the previous layer and send information to the next layer. More details on this will follow. An illustration of a feed-forward neural network is seen in Figure 3. The information contained in a neural network is propagated through the layers. In a feed-forward neural network, a layer uses information only from previous layers. The information contained in one layer is transferred to the next by applying a non-linear function, called an activation function, commonly denoted by $\sigma(\cdot)$, to an affine transformation of the elements in each layer. Some examples of activation functions will be given in Section 2.2.2.1.

To mathematically express the information flow between the layers in a neural network, some notation must be introduced. Let the total number of layers in the deep neural network be denoted $L+1$, where layer 0 is the input layer, and layer L is the output layer. In the input layer, there are q nodes corresponding to the q components of the feature vector $\boldsymbol{x}$, and in the output layer, there are K nodes, where K is the number of classes in a classification problem. In a regression setting $K = 1$, and there is only one output node. Let the number of nodes in the lth layer be denoted M_l. Then $M_0 = q$ and $M_L = K$. In the most elementary structure of feed-forward neural networks, all the nodes in the previous layers are connected to all nodes in the next layer. An illustration of this is shown in Figure 3. Let $z_j^{(l-1)}$ denote the jth node in the $l-1$th layer and $z_m^{(l)}$ the mth node in the lth layer. The nodes are connected by weight and bias terms. The weight connecting the nodes $z_j^{(l-1)}$ and $z_m^{(l)}$ is denoted w_{jm}^l. In total, there are $M_{l-1} \cdot M_l$ weights connecting these two layers. In addition per node in the lth layer, there is a bias term b_m. We introduce the following matrix and vectors,

$$\boldsymbol{W}_l = \begin{pmatrix} w_{1,1}^l, & w_{1,2}^l, & \dots, & w_{1,M_{l-1}}^l \\ w_{2,1}^l, & w_{2,2}^l, & \dots, & w_{2,M_{l-1}}^l \\ \vdots & \vdots & \ddots & \vdots \\ w_{M_l,1}^l, & w_{M_l,2}^l, & \dots, & w_{M_l,M_{l-1}}^l \end{pmatrix}, \qquad \boldsymbol{b}_l = \begin{pmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_{M_l}^l \end{pmatrix}, \qquad \boldsymbol{z}_l = \begin{pmatrix} z_1^l \\ z_2^l \\ \vdots \\ z_{M_l}^l \end{pmatrix}. \tag{13}$$

Moreover, let the activation function between layer $l-1$ and layer l be denoted σ_l. Using the values of the nodes in the previous layer, the values of the nodes in the next are computed as $\sigma_l(\boldsymbol{W}_l \boldsymbol{z}_{l-1} + \boldsymbol{b}_l)$. The neural network model $\hat{f}$ can then be expressed as a composition of the activation functions applied to affine transformations of the inputs of each layer as

$$\hat{f}(\boldsymbol{x}) = \sigma_L(\boldsymbol{W}_L \sigma_{L-1}(\dots \sigma_2(\boldsymbol{W_2} \sigma_1(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2) \dots) + \boldsymbol{b}_L), \tag{14}$$

where evaluating the activation functions at a vector means element-wise evaluation. It is common to use the symbol $\boldsymbol{\theta}$ to denote a vector that contains all the parameters of the neural network model. This includes all weights and bias terms. Then, the neural network model can be expressed as $\hat{f}(\boldsymbol{x}; \boldsymbol{\theta})$ where it is emphasized that the model is parametric and that it is the parameters $\boldsymbol{\theta}$ of the model that are learned during training. As mentioned, there are many possible activation functions that can be used in a neural network. In the following section, some examples of activation functions are given.

2.2.2.1 Activation Functions

One on the most popular activation function is the Rectified Linear Unit (**ReLU**) activation function, which is given as

$$\boldsymbol{\sigma}^{\text{ReLU}}(\boldsymbol{z}) = \max\{0, \boldsymbol{z}\}. \tag{15}$$

The function is evaluated element-wise on the vector $\boldsymbol{z}$. The ReLU activation function is mostly used between the input and hidden layers, or between two hidden layers. The activation function that acts between the last hidden layer and the output layer must be chosen with care taking into account the nature of the response variable. Some activation functions are suited for classification problems and others for regression problems. For regression problems, it is common to use the **identity function** as the activation between the last hidden layer and the output layer. An activation function that is common for classification functions is the **softmax** activation function. For class $k = 1, 2, \dots, K$, the kth element of the softmax function is given as

$$\sigma^{\text{softmax}}(\boldsymbol{z})_k = \frac{e^{z_k}}{\sum_{l=1}^{K} e^{z_l}}, \qquad k = 1, \dots, K, \tag{16}$$

for $\boldsymbol{z} \in \mathbb{R}^K$.

2.2.2.2 Loss Functions

The loss function is a function that takes in the model's prediction $\hat{y}_i$ and compares it to the true response variable y_i. The loss function encaptures the similarity between the predictions of the model and the truth. If $\hat{y}_i$ deviates a lot from y_i, the loss assigned by the loss function will be large, and if $\hat{y}_i$ is similar to y_i, the assigned loss will be small. In the training of a neural network, the loss function is used to steer the training in the right direction. More formally, the goal of the training procedure is to minimize the expected value of the loss function. In practice, it is common to train the model on a data set and use another data set to evaluate how well the model performs on unseen data. This is done in order to prevent overfitting. The choice of loss function depends on the nature of the problem. Typically, different loss functions will be used for classification and regression problems. In regression problems, a common loss function is the mean squared error

$$\mathcal{L}^{\text{MSE}} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2, \tag{17}$$

where n is the number of observations in the data set that the loss is evaluated over. In classification problems with K classes, a typical loss function is the categorical cross-entropy, which is defined as

$$\mathcal{L}^{\text{CE}} = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log(\hat{y}_{ik}), \tag{18}$$

where the true response variables are one-hot-encoded such that $y_{ik} \in \{0, 1\}$ and the predicted probabilities $\hat{y}_{ik} \in (0, 1)$ for all K classes. This loss function evaluates the loss over all the classes

for each observation. As mentioned, the goal of the training procedure is to minimize the expected value of the loss function. In general, the loss function will be a complex function. Therefore, it is necessary to use approximative optimization strategies to search for the minimizer(s) of the loss function. Some commonly used optimizers are presented in the following section.

2.2.2.3 Optimizers

Precisely, the goal of training a neural network is to find the parameters of the model that minimizes the loss function and generalizes well across other samples from the same distribution. In general, neural networks are used to approximate non-convex functions, and therefore, it will generally be infeasible to find the global minimizer of the loss function. It should be noted that based on the training data, it is undesirable to find the global optimum of the loss function since this is likely to produce a model that overfits the training data set and performs poorly on observations that are not in the training data set. As a consequence, when training a neural network, algorithms that approach a local minimum of the loss function are used.

Gradient Descent

A commonly used optimization strategy is the gradient descent algorithm. Given the current best estimates of the model, the gradient descent method moves in the direction of the steepest descent, given by the negative gradient of the loss function, in order to move towards the local minimizer of the loss function. Let $\boldsymbol{\theta}^{\text{prev}}$ denote the previous parameter estimates of the model. Then, the next parameter estimate of the gradient descent method is

$$\boldsymbol{\theta}^{\text{current}} = \boldsymbol{\theta}^{\text{prev}} - \alpha \nabla_{\boldsymbol{\theta}}^{\text{prev}} \mathcal{L}, \tag{19}$$

where the parameter α is called the learning rate and the notation $\nabla_{\boldsymbol{\theta}}^{\text{prev}} \mathcal{L}$ means the gradient of $\mathcal{L}$ with respect to the parameters $\boldsymbol{\theta}$ evaluated at the previous parameter estimates $\boldsymbol{\theta}^{\text{prev}}$. A large value of the learning rate α means that the updates will be large, which could lead to faster convergence, but it may also result in updates that overshoot the minimizer by moving too far, and thereby the estimates will not converge to the local minimizer. On the other hand, a small value of α ensures that the updates will not overshoot the minimizer, however, it can lead to small updates which cause the convergence of the method to be too slow. Since generally the loss function will consist of a composition of many functions, the computation of its gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}$ is complicated. The method that is commonly used for handling this computation is called backpropagation, details on backpropagation can be seen in e.g. Guilhoto [21].

Stochastic Gradient Descent

To both reduce the computational complexity of updating the parameters of the model and to improve the convergence of the optimizer, more sophisticated optimization techniques are used. In gradient descent, the gradient of the loss function is computed using the whole training data set. If the training data set is large, this may be very computationally expensive. Therefore, it makes sense to consider estimates of the gradient of the loss function computed using a subset of the full training data set. One optimization method that does this is stochastic gradient descent, which is based on the Robbins-Monro algorithm [22], where a stochastic estimate of the gradient is used instead of the full gradient. Typically, the loss function that is used is such that the full loss can be expressed as a sum over the loss of each observation in the training data set, i.e.,

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_i(\boldsymbol{\theta}),$$

where the total number of observations in the training data set is n and $\mathcal{L}_i$ is the loss associated with observation i. If the loss function is of this form, its gradient also satisfies this property, i.e.,

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}).$$

The stochastic gradient descent estimate of the gradient is to consider the gradient evaluated at a randomly drawn observation from the training data set as an approximation of the full gradient,

which is expressed as

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \approx \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}), \qquad i \sim \{1, 2, \ldots, n\}, \tag{20}$$

where the notation $i \sim \{1, 2, \ldots, n\}$ means that the index i was randomly drawn from the full set of indices $\{1, 2, \ldots, n\}$ and the probability of drawing each observation is uniform over the set. The stochastic gradient descent update is similar to the gradient descent update (19), but with the gradient approximation (20).

A compromise between gradient descent, where all the n training observations are considered, and stochastic gradient descent where a single observation is considered, is to consider a **batch** consisting of B randomly drawn observations in each update of the gradient. The strategy of using batches often performs better than stochastic gradient descent. Therefore, this is a popular technique used in neural network training. When using stochastic gradient descent or batches of observations, one can also specify the number of times to iterate through the whole training data set. This is referred to as the number of **epochs**.

Adam

Kingma and Ba [23] introduced the "Adam"-method (Adaptive Moment Estimation) which is a more sophisticated stochastic optimization method than stochastic gradient descent. Adam has been very successful and is commonly used for minimizing stochastic loss functions, such as the loss in a deep learning model. Adam builds on the idea of stochastic gradient descent by utilizing a subsample of the full training data set to update the gradient of the loss function. In addition, Adam considers the running averages of both the gradient and second-order moments of the loss function in the previous iteration. The resulting update is a combination of the gradient approximation and these moments. Empirical results have shown that Adam performs well compared with other optimization methods [23].

Algorithm 1: Adam Optimization

Input: Step size α

Input: Exponential decay rates for the moment estimates $\beta_1, \beta_2 \in [0, 1)$

Input: Stochastic objective function with parameters $\boldsymbol{\theta}$: $f(\boldsymbol{\theta})$

Input: Initial parameter values $\boldsymbol{\theta}_0$

Input: A small positive constant to prevent division by 0: ε. A typical value is $\varepsilon = 10^{-8}$

Output: Resulting parameter estimates $\boldsymbol{\theta}_t$

Initialize first moment vector: $\boldsymbol{m}_0 \leftarrow 0$

Initialize second moment vector: $\boldsymbol{v}_0 \leftarrow 0$

Initialize timestep: $t \leftarrow 0$

while *not converged* **do**

  $t \leftarrow t + 1$

  Calculate the gradient of the objective w.r.t. current parameter values: $\boldsymbol{g}_t \leftarrow \nabla_{\boldsymbol{\theta}} f_t(\boldsymbol{\theta}_{t-1})$

  Update the biased first moment estimate: $\boldsymbol{m}_t \leftarrow \beta_1 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_1) \cdot \boldsymbol{g}_t$

  Update the biased second moment estimate: $\boldsymbol{v}_t \leftarrow \beta_2 \cdot \boldsymbol{v}_{t-1} + (1 - \beta_2) \cdot \boldsymbol{g}_t \odot \boldsymbol{g}_t$

  Compute the bias-corrected first moment estimate: $\hat{\boldsymbol{m}}_t \leftarrow \frac{\boldsymbol{m}_t}{1 - \beta_1^t}$

  Compute the bias-corrected second moment estimate: $\hat{\boldsymbol{v}}_t \leftarrow \frac{\boldsymbol{v}_t}{1 - \beta_2^t}$

  Update parameters $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha \cdot \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \varepsilon}$

end

We follow the notation of Kingma and Ba [23] and denote the stochastic scalar objective function as $f(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ are the parameters of the function. In the neural network setting, f is the loss function $\mathcal{L}$. The goal is to minimize the expected value of this function. The objective function must

be differentiable with respect to the parameters $\boldsymbol{\theta}$. Denote the gradient of f at time step t in the iteration scheme by $g_t = \nabla_{\boldsymbol{\theta}} f_t(\boldsymbol{\theta})$ evaluated at the current parameter estimates $\boldsymbol{\theta}_t$. The algorithm considers exponential moving averages $\boldsymbol{m}_t$ of the gradient, and the element-wise squared gradient, $\boldsymbol{v}_t$, which are estimates of the mean, the first moment, and the uncentered variance, the second moment, of the gradient. The method uses hyperparameters $\beta_1, \beta_2 \in [0, 1)$ in order to control the rate of the exponential decay of the moving averages. The outline of the Adam procedure is given in Algorithm 1. The algorithm takes as inputs a step size α, the exponential decay rates β_1, β_2, the objective function $f(\boldsymbol{\theta})$, initial values of the parameters $\boldsymbol{\theta}_0$, and a hyperparameter ε that prevents division by 0. According to Kingma and Ba [23], the method requires little hyperparameter tuning, and good default values for the hyperparameters are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.99$, and $\varepsilon = 10^{-8}$. The algorithm's output is the final parameter estimates $\boldsymbol{\theta}_t$. In Steps 1-2 the first and second moments, denoted $\boldsymbol{m}_0$ and $\boldsymbol{v}_0$, respectively, are initialized to 0. The counter of the current time step is set to 0 in Step 3. Then, Steps 5-11 are repeated until convergence. In Step 5, the iteration index is increased. Next, the gradient of the objective function $\boldsymbol{g}_t$ at the previous values of the parameters $\boldsymbol{\theta}_{t-1}$ is computed. Then, in Steps 7 and 8, respectively, the biased estimates of the first and second moments are updated. Steps 9 and 10 compute the bias-corrected first and second moment, respectively. The final step of the current iteration is to update the current estimate of the parameters $\boldsymbol{\theta}_t$ according to the update rule in Line 11 of the algorithm. In Step 11, the notation $\sqrt{\boldsymbol{v}_t}$ means element-wise square root. After this step, the algorithm either proceeds to the next iteration or finishes and returns the current parameter estimates if the convergence criterion is met. The convergence criteria can e.g. be to end the iterations if the update has stagnated and the parameter estimates change (very) little between iterations. In Algorithm 1, it has not been specified what the batch size is. However, in the iterative scheme, Steps 5-11 are repeated for each sample in the batch that is considered.

2.2.2.4 Validation Sets, Adaptive Learning Rates, and Early Stopping

In this section, some other topics related to training neural networks are considered.

Validation Sets and Batches

To avoid overfitting, it is common to evaluate the loss of the model during training on another data set than the training data set. This data set is called the **validation data set**. In some settings, evaluating the loss on the whole validation data set can be too computationally expensive. Then it is common to consider only a batch of the validation data set each time the loss is evaluated. A batch means that only a subset of the validation set is used when evaluating the loss at a current time point during training. The deep learning model uses the loss evaluated on the validation data set/batch to guide the learning of the parameters of the model. Typically, the validation loss is calculated at the end of an epoch to evaluate if the model has improved. This steers the model to learn parameter values that generalize well over unseen samples.

Adaptive Learning Rates

In the update of the parameters in the optimizers in Section 2.2.2.3, the learning rate or step size α is important for determining the convergence rate of the methods. Intuitively, considering the simplified setting of minimizing a convex loss function with a unique global minimizer, if the current estimates of the parameters are far from the minimizer of the loss function, it makes sense to use a large value of the learning rate to allow for fast convergence. If the parameters are close to the minimizer, it makes sense to use a low learning rate in order to avoid overshooting the minimizer. This illustrates that the learning rate should not be kept constant during the whole training process. It should be adapted. A common adaptation scheme is to reduce the learning rate by a certain factor, e.g. 0.5, if there has been no improvement in the loss for the last few epochs, e.g. five epochs. Often this is combined with the validation batch strategy previously discussed.

Early Stopping

Another common strategy used in the training of deep learning models is early stopping. This is used to avoid overfitting and unnecessary training resulting in no improvement of the model.

Rather than specifying a low number of epochs, which could lead to underfitting, one can use early stopping. The idea of early stopping is to monitor a metric, such as the validation loss, and stop training the model if this metric no longer improves. At a given time, it could be that the metric does not improve, but if the training were continued, there would be significant improvements after a few iterations. Therefore, an option is to specify several that there should have been no improvement for several epochs before stopping.

There exists a wide range of methods related to the training, architecture, optimization, and so forth of neural networks that have not been discussed here. In the previous sections, the focus has been to provide an overview of the techniques that will be relevant to this thesis. To close off this section about neural networks, a brief insight into the universal approximation theorem is provided, which gives insight into why neural networks can provide good approximations of a wide range of functions.

2.2.2.5 The Universal Approximation Theorem

Neural networks have been very successful in many machine learning competitions. This is due to the expressive power of deep neural networks. The expressive power of artificial neural networks is rooted in the universal approximation theorem. The universal approximation theorem states that a sufficiently large neural network *can* represent a wide range of interesting functions if the weights of the neural network are chosen correctly [24, 25]. The theorem does not provide a way to choose the weights but gives a theoretical foundation for the potential power of neural networks as function approximators. In practice, the training of neural networks relies on e.g. successful optimization strategies to find good parameter estimates, such as Adam, which was discussed in Section 2.2.2.3. This finishes the section on neural networks, and in the following section, the focus is multivariate probability distributions with analytically known conditional distributions. These will be useful in the simulation study that will be presented in Section 4.

2.3 Some Multivariate Probability Distributions with Known Conditional Distributions

The methods that will be introduced in Section 3 will be tested on simulated data because the true Shapley values are rarely known for real-world data sets and complex black box machine learning models. In the simulation study, we need to simulate data from a distribution where the conditional distribution of some of the features in the model, given the observation of the others, is known. Therefore, two multivariate distributions which have this property will be presented.

2.3.1 The Multivariate Normal Distribution

Let $\boldsymbol{x} \in \mathbb{R}^q$ denote a random vector with q components that follows a multivariate normal distribution with expected value $\boldsymbol{\mu} \in \mathbb{R}^q$ and covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{q \times q}$. This is denoted as $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. The probability density function of $\boldsymbol{x}$ is then

$$f_q(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^q |\Sigma|}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^\top \Sigma^{-1} (\boldsymbol{x} - \boldsymbol{\mu})\right), \tag{21}$$

where $|\Sigma|$ denotes the determinant of Σ and Σ^{-1} denotes the inverse of Σ. Let $\boldsymbol{x}_1 \in \mathbb{R}^{q_1}$ denote the first q_1 components of $\boldsymbol{x}$ and let $\boldsymbol{x}_2 \in \mathbb{R}^{q_2}$ denote the last q_2 components of $\boldsymbol{x}$, such that $q_1 + q_2 = q$. This corresponds to the partitioning $\boldsymbol{x} = \left(\boldsymbol{x}_1^\top, \quad \boldsymbol{x}_2^\top\right)^\top$. Moreover, partition the vector of expected values $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$ in accordance with the partitioning of $\boldsymbol{x}$ as follows

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1, \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix}.$$

Then the conditional distribution of $\boldsymbol{x}_1$ given the observation $\boldsymbol{x}_2 = \boldsymbol{x}_2^*$ is also multivariate normally distributed, $(\boldsymbol{x}_1 \mid \boldsymbol{x}_2 = \boldsymbol{x}_2^*) \sim \mathcal{N}(\overline{\boldsymbol{\mu}}, \overline{\boldsymbol{\Sigma}})$, where

$$\overline{\boldsymbol{\mu}} = \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\left(\boldsymbol{x}_2^* - \boldsymbol{\mu}_2\right), \tag{22}$$

$$\overline{\boldsymbol{\Sigma}} = \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21}. \tag{23}$$

The multivariate normal distribution is fully determined by its expected value and covariance matrix. Therefore, knowing these two quantities is sufficient for determining the conditional distribution. By computing (22) and (23), the conditional distributions are known and can be simulated from by using e.g. the available functionality for generating numbers from a multivariate normal distribution in the NumPy [26] library in Python.

2.3.2 The Multivariate Burr Distribution

The probability density function of the q-dimensional Burr distribution [27] is given as

$$f_q(\boldsymbol{x}) = \frac{\Gamma(\zeta + q)}{\Gamma(\zeta)} \left(\prod_{j=1}^{q} b_j r_j\right) \frac{\prod_{j=1}^{q} x_j^{b_j - 1}}{\left(1 + \sum_{j=1}^{q} r_j x_j^{b_j}\right)^{\zeta + q}}, \tag{24}$$

where $\zeta, b_1, b_2, \ldots, b_q$ and $r_1, r_2, \ldots, r_q$ are the parameters of the distribution. The distribution is defined for $x_j > 0, j = 1, 2, \ldots, q$. The multivariate Burr is a compound Weibull distribution with the Gamma distribution as a compounder [27]. It can be regarded as a special case of the Pareto IV distribution [28].

The multivariate Burr distribution, like the normal distribution, has the desirable property that any marginal and conditional distribution of it is also a (multivariate) Burr distribution, a proof is given in Takahasi [27]. The conditional density $f(x_1, x_2, \ldots, x_{\tilde{q}} | x_{\tilde{q}+1} = x_{\tilde{q}+1}^*, x_{\tilde{q}+2} = x_{\tilde{q}+2}^*, \ldots, x_q = x_q^*)$ is a $\tilde{q}$-dimensional Burr density with parameters $\tilde{\zeta}, \tilde{b}_1, \tilde{b}_2, \ldots, \tilde{b}_{\tilde{q}}$ and $\tilde{r}_1, \tilde{r}_2, \ldots \tilde{r}_{\tilde{q}}$ where

$$\begin{aligned} \tilde{\zeta} &= \zeta + q - \tilde{q}, \\ \tilde{b}_j &= b_j, && \text{for} \quad j = 1, 2, \ldots, \tilde{q}, \\ \tilde{r}_j &= \frac{r_j}{1 + \sum_{j=\tilde{q}+1}^{q} r_j (x_j^*)^{b_j}}, && \text{for} \quad j = 1, 2, \ldots, \tilde{q}. \end{aligned} \tag{25}$$

The Burr distribution allows for heavy-tailed, skewed marginal distributions, in addition to non-linear correlation structures [29] between the variables in $\boldsymbol{x}$. These properties are often found in real-world data, and the normal distribution often behaves "too nicely" compared to real-world data. Therefore, the Burr distribution is useful to check the accuracy of a method on more realistic simulated data. There are preexisting R-packages [30] where the random generation of samples from the multivariate Burr distribution is implemented. We omit details of the simulation as this goes beyond the scope of this thesis, but have replicated the simulation procedure in Python based on the R-package [30].

2.4 Monte Carlo Integration

Many statistical applications involve computing a possibly high-dimensional integral. Some examples are computing a posterior probability in Bayesian statistics, marginalizing over a joint density in frequentist statistics, and computing moments, like the expected value and variance, of a probability distribution. Many high-dimensional integrals are analytically intractable and must therefore be approximated. One way to approximate such integrals when the integrand is a function of random variables is **Monte Carlo integration.**

The idea behind Monte Carlo integration is to use samples $\boldsymbol{x}_1^*, \boldsymbol{x}_2^*, \ldots, \boldsymbol{x}_M^* \in \mathbb{R}^q$ obtained from a distribution $f(\boldsymbol{x})$. The samples are obtained independently. Let $\boldsymbol{x}$ denote a random variable and let $\boldsymbol{x} \sim f(\boldsymbol{x})$. Moreover, define $h : \mathbb{R}^q \to \mathbb{R}$ as a function of $\boldsymbol{x}$. The goal is to estimate the expected

value of $h(\boldsymbol{x})$. The strong law of large numbers states that the Monte Carlo approximation $\hat{\mu}_{\text{MC}}$ of $\mathbb{E}[h(\boldsymbol{x})]$ converges towards $\mathbb{E}[h(\boldsymbol{x})]$. For the Monte Carlo integral, the strong law of large numbers can be written as

$$\hat{\mu}_{\text{MC}} = \frac{1}{M}\sum_{m=1}^{M} h(\boldsymbol{x}_{\boldsymbol{m}}^*) \xrightarrow{\text{a.s.}} \int h(\boldsymbol{x}) f(\boldsymbol{x}) d\boldsymbol{x} = \mathbb{E}[h(\boldsymbol{x})] =: \mu \quad \text{as } M \to \infty, \tag{26}$$

where $\xrightarrow{\text{a.s.}}$ means "converges almost surely", which is the strongest type of convergence in probability theory. Mathematically, an infinite sequence of random variables $\{y_m\}_{m\in\mathbb{N}}, y_m \in \mathbb{R}$ **converges almost surely** to the random variable $y \in \mathbb{R}$ if $P\left(\lim_{m\to\infty} y_m = y\right) = 1$, i.e. the event that the sequence converges to y has probability one.

Under the assumption that the second moment is finite $\mathbb{E}[h(\boldsymbol{x})^2] < \infty$, the variance of the Monte Carlo estimator [31] is

$$\text{Var}[\hat{\mu}_{\text{MC}}] = \frac{1}{M}\,\mathbb{E}[(h(\boldsymbol{x}) - \mu)^2] =: \frac{\sigma^2}{M}.$$

Then, using a similar Monte Carlo estimate, the variance σ^2 can be estimated by

$$\hat{\sigma}^2_{\text{MC}} = \frac{1}{M-1}\sum_{i=1}^{M} \left(h(\boldsymbol{x}_i^*) - \hat{\mu}_{\text{MC}}\right)^2.$$

By the central limit theorem, $\hat{\mu}_{\text{MC}}$ is approximately normally distributed: $\sqrt{M-1}(\hat{\mu}_{\text{MC}} - \mu) \sim \mathcal{N}(0, \sigma^2) \approx \mathcal{N}(0, \hat{\sigma}^2_{\text{MC}})$, where the approximate distribution still holds when replacing the variance σ^2 by the maximum likelihood estimator $\hat{\sigma}^2_{\text{MC}}$. This shows that the error δ_{MC} in the Monte Carlo estimate grows as

$$\delta_{\text{MC}} = \mathcal{O}(M^{-\frac{1}{2}}), \tag{27}$$

which is quite slow compared to the best deterministic quadrature methods [31]. However, as the number of dimensions grows, the number of nodes required in the deterministic quadrature methods grows exponentially. The Monte Carlo sampling, on the other hand, does not systematically explore the whole support of the integrand. Therefore, it is less affected by the dimensionality increasing [31]. The increasing number of dimensions means that it might be necessary to increase the number of samples M, and hence, the computational cost. However, the increase is not as severe as the exponential growth in the number of quadrature nodes in the deterministic methods. Moreover, quadrature methods often make assumptions about the smoothness of the integrand, limiting their area of use. No smoothness assumptions are made about the function h in the Monte Carlo integral (26). Therefore, it can be applied to also non-smooth functions.

2.4.1 Monte Carlo Integration Using the Empirical Distribution Function

Let $\boldsymbol{x}_1^*, \boldsymbol{x}_2^*, \ldots, \boldsymbol{x}_M^* \in \mathbb{R}^q$ be observations that are assumed to have been obtained independently of each other from the same distribution. Let the observations be realizations of the random variable $\boldsymbol{x}$ that follows the cumulative distribution function (cdf) $F_{\boldsymbol{x}}(\boldsymbol{\xi})$, where $F_{\boldsymbol{x}}(\boldsymbol{\xi}) = \text{P}(\boldsymbol{x} \leq \boldsymbol{\xi}) = \text{P}(x_1 \leq \xi_1, x_2 \leq \xi_2, \ldots, x_q \leq \xi_q)$. In many practical situations, the cdf is unknown, and therefore it is desirable to estimate it. An appropriate estimator of the cdf is the **empricial distribution function** (edf) defined as

$$\hat{F}(\boldsymbol{\xi}) = \sum_{i=1}^{M} \mathbb{1}_{\boldsymbol{x}_i^* \leq \boldsymbol{\xi}}, \quad \text{where} \quad \mathbb{1}_{\boldsymbol{x}_i^* \leq \boldsymbol{x}} = \begin{cases} 1, & \boldsymbol{x}_i^* \leq \boldsymbol{\xi}, \\ 0, & \text{otherwise}, \end{cases}$$

is the indicator function. By the vector inequalities $\boldsymbol{x}_i^* \leq \boldsymbol{\xi}$, it is meant that $x_{i1}^* \leq \xi_1, x_{i2}^* \leq \xi_2, \ldots, x_{iq}^* \leq \xi_q$.

In machine learning we typically have many observations assumed to be drawn independently from the same cdf. Thus, we can estimate the edf using this data set. Having estimated the edf from the data set, we can draw new samples from it. These samples can be used to estimate the expected value of a function of the samples using Monte Carlo integration defined in (26). In practice, one

does not have to estimate the edf and then sample from this distribution. Drawing samples from the data set, with an equal probability of drawing each instance, will yield samples that implicitly follow the edf because more frequently observed values will be drawn more often as there are more observations containing these values.

3 The Shapley Value and Shapley Value Estimators

Shapley value is a term that originates from game theory. It was introduced by Shapley [7]. In the context of game theory, its purpose is to find each player's contribution to the total payoff in a cooperative game. How does this translate to the interpretable machine learning setting? The idea is to consider the prediction for an instance as a game where the features are the players and the prediction is the payoff. The outline of this chapter is as follows. In the next section, Section 3.1, the Shapley value will be defined, and it will be argued why the Shapley value can be seen as a fair evaluation of the contribution of each player. Then, in Section 3.2, we will describe how the Shapley value can be used as a local model-agnostic explanation method. As will be explained, the computation of the Shapley values is expensive. Therefore, in practice, it is necessary to consider estimators of it. Two approximation methods for estimating the Shapley values will be presented in Sections 3.3 and 3.4. An important step in estimating the Shapley values is estimating the contribution function, discussed in Section 3.5. Combining the two steps of the estimation procedure, the full estimation methods are summarized in Section 3.6. The computational cost of the estimation procedures is discussed in Section 3.7. Lastly, we end the chapter by briefly discussing how Shapley values can be used for global interpretability in Section 3.8. In this chapter, the majority of Sections 3.1, 3.2, 3.3 and 3.5.1 are taken from my specialization project [12].

3.1 The Shapley Value and Cooperative Game Theory

The Shapley value originates from cooperative game theory. Assume that there are q players participating in a cooperative game, all with the goal of maximizing their profits. As the players are allowed to collaborate, their contribution to the different coalitions of players must be determined in order to decide on the payout of each player. The Shapley value is one way of assigning the payout among the players. Denote the set of all q players by $\mathcal{Q} = \{1, 2, \dots, q\}$. Consider now a subset $\mathcal{S} \subseteq \mathcal{Q}$ consisting of $|\mathcal{S}|$ players. A contribution function $v(\mathcal{S})$ is used to calculate the gain earned by a coalition. To find the gain for each player in a coalition, the Shapley value compares the gain of coalitions including and excluding the player. If two coalitions, where the only difference is that player j is included in one of them and not the other, have different gains, then the difference in the gain is the contribution of that player. Considering all the possible coalitions of the players, the Shapley value is the weighted average of the difference in gain between all coalitions including and excluding player j. Since the permutation of players in a set $\mathcal{S}$ does not matter, the weights account for this, and the Shapley value for player $j = 1, 2, \dots, q$ is calculated as

$$\phi_j(v) = \phi_j = \sum_{\mathcal{S} \subseteq \mathcal{Q} \setminus \{j\}} \frac{|\mathcal{S}|!\,(q - |\mathcal{S}| - 1)!}{q!} \left(v\left(\mathcal{S} \cup \{j\}\right) - v(\mathcal{S})\right). \tag{28}$$

The Shapley value assigns to player j the weighted average of the difference between the contribution function of sets including and excluding player j. The empty set $\emptyset$ is also included in the sum. This means that the case where the player chooses not to collaborate with anyone is also included. A fixed payout that is not associated with any of the players is defined as $\phi_0(v) = \phi_0 = v(\emptyset)$. Although this is often zero in collaborative games, it can in general be non-zero.

The Shapley value is "fair" in the sense that it is the only distribution of the payout satisfying the following four properties. Proof of this can be found in [7].

Property 1: Efficiency. The total worth of the game is distributed over the players, which mathematically means that

$$\sum_{j=0}^{q} \phi_j = v(\mathcal{Q}).$$

Property 2: Symmetry. If two players $i \in \mathcal{Q}$ and $j \in \mathcal{Q}$, contribute equally to all possible coalitions, meaning that $v(\mathcal{S} \cup \{i\}) = v(\mathcal{S} \cup \{j\})$ for all $\mathcal{S}$ where neither of the two players participates, then they must have same Shapley value:

$$\phi_i = \phi_j.$$

Property 3: Dummy player. If a player j does not change the worth of the payout for any coalition $\mathcal{S} \subseteq \mathcal{Q} \setminus \{j\}$, meaning $v(\mathcal{S} \cup \{j\}) = v(\mathcal{S})$, then the value distributed to the player must be zero, i.e.

$$\phi_j = 0.$$

Property 4: Linearity. Consider two coalition games that are described, respectively, by the contribution functions v and w. Then, if the two games are combined, the distributed gain of each player in the combined game equals the sum of the game from the two games individually:

$$\phi_j(v + w) = \phi_j(v) + \phi_j(w), \quad j = 1, 2, \ldots, q.$$

In addition, if the contribution function is scaled by a real number a, then so is the Shapley value of each player, expressed mathematically as

$$\phi_j(av) = a\phi_j(v).$$

3.2 The Shapley Value in the Context of Explainable Artificial Intelligence

In the context of XAI, the Shapley value is used to explain the prediction of a machine learning model for an instance of interest. In their paper introducing the Shapley value as an explanation method, Štrumbelj and Kononenko [1] write that "its advantage over existing general methods is that all subsets of input features are perturbed, so interactions and redundancies between features are taken into account". This is a very desirable property because features will often interact in practice.

In order to compute the Shapley values, the contribution function must be specified. The contribution function of a subset $\mathcal{S}$ of the features should calculate the expected prediction for the instance of interest $(\boldsymbol{x}^*, y^*)$ when only the features in the subset $\mathcal{S}$ are known to the model. Denote by the black box model $\hat{f}_y(\boldsymbol{x})$. If the underlying problem is a regression problem, $\hat{f}_y(\boldsymbol{x})$ is the model's predicted value. Whereas in the classification setting, $\hat{f}_y(\boldsymbol{x})$ is the predicted probability of belonging to class y. One choice of contribution function $v(\mathcal{S})$ is then to use the expected prediction of the model, conditional on the feature values of the instance of interest in the subset $\mathcal{S}$. Denote the set of features that are not included in $\mathcal{S}$ by $\mathcal{S}^c$, where superscript c denotes the complement of the set $\mathcal{S}$, then a common choice of contribution function is

$$v(\mathcal{S}) = v_{\boldsymbol{x}^*, y^*}(\mathcal{S}) = \mathop{\mathbb{E}}_{p(\boldsymbol{x}_{\mathcal{S}^c} | \boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})} [\hat{f}_y(\boldsymbol{x}) | \boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}], \tag{29}$$

where the notation $v_{\boldsymbol{x}^*, y^*}$ is used to emphasize that this is the value of the contribution function for the subset $\mathcal{S}$ for the instance of interest $(\boldsymbol{x}^*, y^*)$. Moreover, throughout this thesis the notation $\mathop{\mathbb{E}}_{p(\boldsymbol{x})} [f(\boldsymbol{x})]$ means the expectation of $f(\boldsymbol{x})$ with respect to the probability density function $p(\boldsymbol{x})$ Analytical computation of (29) involves computing an integral in $q - |\mathcal{S}|$ dimensions for each subset $\mathcal{S}$. In situations with a high number of features q, this is generally intractable.

When applying the Shapley value to explain the prediction of a black box machine learning model, the features are considered as the players in the game. The "prize" that will be distributed among the players is the prediction of the black box model for the instance of interest. In other words, the final prediction is distributed among the players. In accordance with the first property from above, the "Efficiency"-property, the distribution is additive. That is, the gain of each feature sums up to the prediction and we can interpret each Shapley value as the contribution of the feature value to the final prediction. This gives us a way to explain the prediction of the model. The "Efficiency"-property with contribution function (29) then takes the concrete form

$$\hat{f}_y(\boldsymbol{x}^*) = \phi_0 + \sum_{j=1}^{q} \phi_j^*, \tag{30}$$

where $\phi_0 = \mathbb{E}[\hat{f}_y(\boldsymbol{x})]$, $v(\mathcal{Q}) = \hat{f}_y(\boldsymbol{x}^*)$ and ϕ_j^* is the Shapley value of feature j for the prediction of the instance of interest $(\boldsymbol{x}^*, y^*)$. Since ϕ_0 is the mean prediction of the model, it does not depend

on the instance of interest, and therefore it is denoted as ϕ_0, without the asterisk superscript. That is, the Shapley values of the features add up to the difference between the global average prediction and the prediction of the instance. Or in other words, the features *explain* the deviation of that particular prediction from the global average prediction, and the explanation is the Shapley values.

The Shapley values are here expressed as an additive feature attribution method. It is the only additive attribution method that satisfies the above four properties. These properties are also desirable for an explanation, and the resulting explanation can be seen as correct. To provide an understanding of why the Shapley values provide a fair and correct explanation, each of the four properties given in 3.1 will be interpreted in the setting where the Shapley value is used to explain a prediction of a black box model. The interpretation of the "Efficiency"-property is that this property ensures that the part of the prediction that cannot be explained by the global mean prediction is fully distributed and explained by the features. In addition, it ensures that the Shapley values can be compared across predictions of different instances of interest. The interpretation of the "Symmetry"-property is that if two features when combined with any other subset of features, contribute equally to the resulting prediction of the black box model, they will have the same Shapley value. If the Shapley values did not satisfy this property, then the explanation would be inconsistent and untrustworthy. The interpretation of the third property, the "Dummy Player"-property, is that if a feature never affects a prediction, regardless of which feature subset it is combined with, it has a Shapley value of 0. That a feature that does not contribute to the prediction is assigned no contribution to the final explanation is reasonable, and therefore an explanation method should satisfy this property. If a black box model consists of a sum of simple ensemble models, e.g. a random forest, then the simple models can be interpreted and explained individually, and the explanation of the black box model is the combined explanation of the ensemble methods. This is ensured by the "Linearity"-property of the Shapley values. It is sensible that any explanation should satisfy these four properties. Since the Shapley values are the only additive distribution method that satisfies all these four properties [7], the resulting explanations from the Shapley values are the only truthful additive explanation method [2, 1, 8].

The computation of the Shapley values (28) grows exponentially in the number of features, as 2^q, and it is, therefore, generally infeasible to compute for a high number of features. Therefore, in practice, it is necessary to use approximations that can be computed in polynomial rather than exponential time. The estimation of the Shapley values can be divided into two steps. The first step is to estimate the contribution function (29). This will be denoted as **contribution function estimation**. Given the estimate of the contribution function, the next step is to estimate the Shapley values (28). In order to separate the two steps from each other, we refer to this simply as **Shapley value estimation**, although the precise term would be "Shapley value estimation for a given contribution function estimate". Because the estimation is performed in two steps, different combinations of contribution function estimators and Shapley values estimators are possible. In Section 3.3 and Section 3.4, two Shapley value estimators are presented. In these sections, it is assumed that the contribution function, or an estimate thereof, is given. Then, two methods for estimating the contribution function are presented in Section 3.5.

3.3 KernelSHAP

The KernelSHAP method was proposed by Lundberg and Lee [2]. Because some detail was missing in the original paper, Aas, Jullum and Løland [8] published a more detailed explanation of the method. The description of the KernelSHAP method given here closely follows that given by Aas, Jullum and Løland [8] in their paper. In the original paper on KernelSHAP, Lundberg and Lee [2] propose methods both for estimating the Shapley values (28) for a given contribution function $v(\mathcal{S})$ and for estimating the contribution function as defined in (29). In this section, the Shapley value estimation for a given contribution function estimate is treated. Accordingly, by the "KernelSHAP" estimate, this is what will be referred to. In Section 3.5, the estimation of the contribution function (29) is treated, meanwhile it is assumed that this is given.

In their paper, Lundberg and Lee [2] define the Shapley values as the optimal solution of a weighted least squares (WLS) problem. Before introducing the KernelSHAP method, the WLS problem with the Shapley values as the minimizer will be presented, which will lead to a new formula for

computing the Shapley values. This will make it more clear where the idea of the KernelSHAP method originated. To find the Shapley values $\phi_0, \phi_1^*, \ldots, \phi_q^*$ of the instance of interest $(\boldsymbol{x}^*, y^*)$ for a contribution function $v_{\boldsymbol{x},y}$ one can minimize the following weighted least squares problem

$$\sum_{\mathcal{S} \subseteq \mathcal{Q}} k(\mathcal{Q}, \mathcal{S}) \left[v_{\boldsymbol{x}^*,y^*}(\mathcal{S}) - \left(\phi_0 + \sum_{j \in \mathcal{S}}^{q} \phi_j^* \right) \right]^2, \tag{31}$$

with respect to $\phi_0, \phi_1^*, \ldots, \phi_q^*$, where the Shapley kernel weights are defined as

$$k(\mathcal{Q}, \mathcal{S}) = \frac{q-1}{\binom{q}{|\mathcal{S}|} |\mathcal{S}| (q - |\mathcal{S}|)}. \tag{32}$$

The WLS problem can further be rewritten using matrix-vector notation. Denote by $\boldsymbol{Z}$ the $2^q \times (q+1)$ matrix representing all combinations of inclusion/exclusion of the features and an extra column which is necessary for the computation of the Shapley value ϕ_0 that is not associated with any of the features. The first column of this matrix has 1 in all rows, and element $j+1$ of row l is 1 if feature j is included in the combination the row represents, and 0 otherwise. Let the vector $\boldsymbol{v}_{\boldsymbol{x}^*,y^*}$ contain $v_{\boldsymbol{x}^*,y^*}(\mathcal{S})$ for all subsets $\mathcal{S} \subseteq \mathcal{Q}$. Note that we also consider the sets $\emptyset$ and $\mathcal{Q}$ for which the Shapley kernel weights (32) are infinite. This can in practice be solved by setting them equal to a large positive constant c, e.g. $c = 10^6$ [8]. Moreover, let $\boldsymbol{W}$ be the $2^q \times 2^q$ matrix with the Shapley kernel weights $k(\mathcal{Q}, \mathcal{S})$ on the diagonal. In both $\boldsymbol{v}_{\boldsymbol{x}^*,y^*}$ and $\boldsymbol{W}$ the element in each row must correspond to the same subset $\mathcal{S}$ as in that row in $\boldsymbol{Z}$. Lastly, let $\boldsymbol{\phi}_{\boldsymbol{x}^*,y^*} = \left(\phi_0, \quad \phi_1^*, \quad \ldots, \quad \phi_q^*\right)^\top$. Then the weighted least squares problem in (31) can be expressed as

$$(\boldsymbol{v}_{\boldsymbol{x}^*,y^*} - \boldsymbol{Z}\boldsymbol{\phi}_{\boldsymbol{x}^*,y^*})^\top \boldsymbol{W} (\boldsymbol{v}_{\boldsymbol{x}^*,y^*} - \boldsymbol{Z}\boldsymbol{\phi}_{\boldsymbol{x}^*,y^*}),$$

which has minimizer

$$\boldsymbol{\phi}_{\boldsymbol{x}^*,y^*} = (\boldsymbol{Z}^\top \boldsymbol{W} \boldsymbol{Z})^{-1} \boldsymbol{Z}^\top \boldsymbol{W} \boldsymbol{v}_{\boldsymbol{x}^*,y^*}. \tag{33}$$

These are the exact Shapley values, equivalent to (28). As previously mentioned, the number of inclusions/exclusions of features 2^q grows exponentially in the number of features q. This makes the computation of the Shapley values expensive, and an approximation desirable. We now present an approximator that is based on the Shapley values as the minimizer (33) of the WLS problem.

3.3.1 The KernelSHAP Approximation

To reduce the computational complexity, the KernelSHAP method [2] is introduced. Lundberg and Lee [2] propose to use the probability distribution following the Shapley kernel weights to select a subset $\mathcal{D}$ of the rows of $\boldsymbol{Z}$ and use only these to approximate the minimizer (33). By drawing from this distribution, with replacement, the rows with the highest kernel Shapley weight, which contribute most to the computation of the Shapley values, are used. The probability distribution is calculated by dividing each weight by the sum of all the weights such that they sum to one and are normalized. To do this in practice, it is necessary to assume that all weights are finite, i.e. the weights corresponding to the empty set and the full set $\mathcal{Q}$ cannot be included. Thus, the sampling occurs first while omitting these rows, and the rows corresponding to the empty set and full set are manually included since they have infinite weight. Let $\boldsymbol{Z}_\mathcal{D}$, $\boldsymbol{v}^\mathcal{D}_{\boldsymbol{x}^*,y^*}$, and $\boldsymbol{W}_\mathcal{D}$ be the rows/elements of $\boldsymbol{Z}$, $\boldsymbol{v}$, and $\boldsymbol{W}$ respectively, that corresponds to the subset $\mathcal{D}$. In addition, the rows and elements corresponding to the coalition of all the features $\mathcal{Q}$ and the coalition of no features $\emptyset$ are added to $\boldsymbol{Z}_\mathcal{D}$, $\boldsymbol{v}^\mathcal{D}_{\boldsymbol{x}^*,y^*}$, and $\boldsymbol{W}_\mathcal{D}$. We will refer to $\mathcal{D}$ as the subset of all feature combinations including $\emptyset$ and $\mathcal{Q}$. As before, the value of the Shapley kernel weights can be set to a large constant c for these two rows. Thus, an approximation to the minimizer in (33) is

$$\boldsymbol{\phi}^\mathcal{D}_{\boldsymbol{x}^*,y^*} = (\boldsymbol{Z}_\mathcal{D}^\top \boldsymbol{W}_\mathcal{D} \boldsymbol{Z}_\mathcal{D})^{-1} \boldsymbol{Z}_\mathcal{D}^\top \boldsymbol{W}_\mathcal{D} \boldsymbol{v}^\mathcal{D}_{\boldsymbol{x}^*,y^*} = \boldsymbol{R}_\mathcal{D} \boldsymbol{v}^\mathcal{D}_{\boldsymbol{x}^*,y^*}, \tag{34}$$

where the matrix $\boldsymbol{R}_\mathcal{D} = (\boldsymbol{Z}_\mathcal{D}^\top \boldsymbol{W}_\mathcal{D} \boldsymbol{Z}_\mathcal{D})^{-1} \boldsymbol{Z}_\mathcal{D}^\top \boldsymbol{W}_\mathcal{D}$ does not depend on the instance of interest. It can therefore be used to generate explanations for several instances of interest. Only $\boldsymbol{v}^\mathcal{D}_{\boldsymbol{x}^*,y^*}$ depends on the instance of interest and must be recomputed for every instance that one wishes to explain the prediction for. It should be noted that the computation of $\boldsymbol{v}^\mathcal{D}_{\boldsymbol{x}^*,y^*}$ with contribution function

as defined in (29) is often analytically intractable and in many practical situations expensive to approximate, something we will get back to in Section 3.5. Therefore, even though $\boldsymbol{R}_{\mathcal{D}}$ does not have to be recomputed for each instance, explaining the prediction for a new instance will be computationally expensive in general.

Covert and Lee [32] find evidence of variance reduction in the KernelSHAP estimate when using **paired sampling**. For each coalition $\mathcal{S}$ in $\mathcal{D}$ that is considered, the compliment $\mathcal{S}^c$ is also used. Covert and Lee [32] find that in some cases paired sampling will reduce the computational time of KernelSHAP by as much as nine times compared to without paired sampling. Therefore, this is a possible improvement of the method that we have incorporated as a hyperparameter in our implementation of the method.

To summarize, the KernelSHAP method reduces the computational cost of computing the Shapley values from exponential 2^q in the total number of features q to considering only $|\mathcal{D}|$ terms. To give more details on the computational cost of the full Shapley value estimation method, the cost of computing the contribution function (29) must also be considered. Therefore, we consider this in Section 3.7 after the full estimation procedures have been presented. In the following section, we present an alternative Shapley value estimator. The method amortizes the computational cost of estimating the Shapley values over the instances to be explained.

3.4 FastSHAP

Jethani et al. [4] presents an alternative estimation method to KernelSHAP, which they call FastSHAP. The idea of the method is to train a machine learning model to predict the Shapley values. The model is trained on the data set the black box model is trained on. This model can then be used to calculate the Shapley values for any instance, without requiring any additional estimation. This amortizes the cost of estimating the Shapley value over the instances to be explained; even though the initial training of the machine learning model is expensive, it is made up for because explaining a new instance simply requires the evaluation of the machine learning model, a neural network, in a forward pass. In this section, we focus on introducing the theoretical foundation of the FastSHAP method and present both an algorithmic overview of the method and a more in-depth algorithm of the method.

The intuitive idea when the goal is to train a machine learning model to learn the Shapley values is that it would require a large data set of true Shapley values in order to train the machine learning model. This is not possible, since the estimation of the Shapley values is still an ongoing problem in research, and exactly the topic we are considering. Thus, it is unrealistic to require a data set of exact Shapley values or good approximations thereof. However, Jethani et al. [4] present a loss function with global optimizer that converges almost surely to the true Shapley values in the joint distribution of $\boldsymbol{x}$ and y. Therefore, one can train a machine learning model on the feature observations in the original data set with this loss function, and given a large enough sample and a complex enough model, one can obtain a good estimate of the true contribution function. Thus, the training of this model does not require access to the true Shapley values. In general, in XAI, the explanations can be calculated on the training data set, the test data set, or a combination of the two. However, since the FastSHAP model is a machine learning model, it makes sense to train the model on the training data set and evaluate it on the validation and test data sets in order to prevent overfitting the training data set.

Jethani et al. [4] introduce FastSHAP in the classification setting, therefore, like the authors, we present FastSHAP as a method to explain a classification machine learning model. Let the possible classes the response variable y can belong to be denoted by $1, 2, \ldots, K$. Let $\hat{f}(\boldsymbol{x})$ denote a machine learning model that outputs a probability distribution over the K classes. As in the previous section, let $\hat{f}_y(\boldsymbol{x})$ denote the predicted probability of a single class $y \in \{1, 2, \ldots, K\}$. The full model $\hat{f}(\boldsymbol{x})$ gives a prediction for all K classes, and the model $\hat{f}_y(\boldsymbol{x})$ only gives the prediction for a single class y. For now, it is assumed that the contribution function $v_{\boldsymbol{x},y}(\mathcal{S})$ is given. In Section 3.5.2, the details of how the contribution function is estimated in the original version of the FastSHAP method are given. Like with KernelSHAP, we refer to "FastSHAP" as an estimator of the Shapley values for a given estimate of the contribution function. As a starting point, Jethani

et al. [4] exploit that the weighted least squares characterization of the Shapley values (31) can be rewritten by defining a probability distribution over the subsets $\mathcal{S}$, which will be denoted by $p(\mathcal{S})$. The probability distribution is the Shapley kernel weights (32) normalized to sum to 1 so that they are a true probability distribution. Thus, the distribution $p(\mathcal{S})$ is proportional to the Shapley kernel weights $k(\mathcal{Q}, \mathcal{S})$ defined in (32), which is denoted by $p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$. It should be noted that Jethani et al. [4] does not define the probability distribution for $\mathcal{S} = \emptyset$ and $\mathcal{S} = \mathcal{Q}$, therefore there is not a problem related to infinite weights. Thus, for a given contribution function $v_{\boldsymbol{x},y}$ the Shapley values of the instance of interest $\boldsymbol{\phi}^* = \boldsymbol{\phi}_{\boldsymbol{x}^*,y^*} = \boldsymbol{\phi}(v_{\boldsymbol{x}^*,y^*})$ are the solution to the optimization problem

$$\boldsymbol{\phi}(v_{\boldsymbol{x}^*,y^*}) = \underset{\boldsymbol{\phi}_{\boldsymbol{x}^*,y^*}}{\arg\min} \underset{p(\mathcal{S})}{\mathbb{E}} \left[\left(v_{\boldsymbol{x}^*,y^*}(\mathcal{S}) - v_{\boldsymbol{x}^*,y^*}(\emptyset) - \sum_{j \in \mathcal{S}}^{q} \phi_j^* \right)^2 \right]. \tag{35}$$

Recalling that $v_{\boldsymbol{x}^*,y^*}(\emptyset) = \phi_0$, and because $p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$ it is clear that this optimization problem is equivalent to the weighted least squares optimization problem (31).

As briefly mentioned, the idea of FastSHAP is to train a machine learning model on the feature observations $\{\mathbf{x}_i\}_{i=1}^n$ from the original training data set with a specific loss function which ensures that the machine learning model learns the Shapley values rather than predicting the response variables y, which is the most common predictive task in machine learning. We follow the notation of Jethani et al. [4] and denote the machine learning model by $\boldsymbol{\phi}^{\text{fast}}(\mathbf{x}, y; \boldsymbol{\theta}) : \mathcal{X} \times \mathcal{Y} \mapsto \mathbb{R}^q$, where $\boldsymbol{\theta}$ are the parameters of the model. The model outputs the Shapley values for all $1, 2, \ldots, q$ features and all $1, 2, \ldots, K$ classes, not only the true class. Therefore, to get the predicted Shapley values for feature j of an observation $(\boldsymbol{x}^*, y^*)$, one has to evaluate the model at the true value of the response variable y^* and take the jth element of the output. Let now $\boldsymbol{x}$ and y be random variables, and let the instance of interest $(\boldsymbol{x}^*, y^*)$ be an observation of $\boldsymbol{x}$ and y. Jethani et al. [4] prove that if the predictions of the model $\boldsymbol{\phi}^{\text{fast}}(\mathbf{x}, y; \boldsymbol{\theta})$ are forced to satisfy the "Efficiency"-constraint (30), and if $\boldsymbol{\phi}^{\text{fast}}(\mathbf{x}, y; \boldsymbol{\theta})$ belongs to a sufficiently expressive class of functions, then the global optimizer of the loss function they introduce converges almost surely in the joint distribution of $\boldsymbol{x}$ and y to the true Shapley values of $(\boldsymbol{x}^*, y^*)$. The loss function they present is

$$\mathcal{L}^{\text{fast}}(\boldsymbol{\theta}) = \underset{p(\boldsymbol{x})}{\mathbb{E}} \underset{\text{Unif}(y)}{\mathbb{E}} \underset{p(\mathcal{S})}{\mathbb{E}} \left[\left(v_{\boldsymbol{x},y}(\mathcal{S}) - v_{\boldsymbol{x},y}(\emptyset) - \sum_{j \in \mathcal{S}}^{q} \phi_j^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta}) \right)^2 \right], \tag{36}$$

where $\text{Unif}(y)$ denotes a uniform distribution over the K classes of y. In practice, this means assuming that our data set is sufficiently large and that all the observations are from the same distribution, a sufficiently large neural network, which can approximate any continuous function by the universal approximation theorem, can learn to approximate the Shapley value function. Note that the innermost expectation in (36) is the loss function (35), which has the exact Shapley values as the minimizer. Thus, the loss function (36) in FastSHAP is the expectation of (35) over $\boldsymbol{x}$ and y. Now that the loss function of the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\mathbf{x}, y; \boldsymbol{\theta})$ has been introduced, we proceed in the following section by describing how the efficiency property (30) is enforced in the FastSHAP framework. Then, the algorithm for training the FastSHAP model, including the efficiency enforcement, will be described in Section 3.4.2.

3.4.1 Enforcing the "Efficiency"-Property of the Shapley Values

Jethani et al. [4] present two methods for ensuring that the Shapley values estimated by the FastSHAP model satisfy the "Efficiency"-property (30). The first is to adjust the estimates by using *additive efficiency normalization* [33], which corresponds to adding a term to the estimated values:

$$\boldsymbol{\phi}_{\text{eff}}^{\text{fast}}(\boldsymbol{x}^*, y^*; \boldsymbol{\theta}) = \boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}^*, y^*; \boldsymbol{\theta}) + \frac{1}{q} \left(v_{\boldsymbol{x}^*,y^*}(\mathcal{Q}) - v_{\boldsymbol{x}^*,y^*}(\emptyset) - \sum_{j=1}^{q} \boldsymbol{\phi}_j^{\text{fast}}(\boldsymbol{x}^*, y^*; \boldsymbol{\theta}) \right)^2. \tag{37}$$

The term that is added to the estimate $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}^*, y^*; \boldsymbol{\theta})$ is called the *efficiency gap*. The efficiency gap can be added to the Shapley value estimates both during the training of the neural network

and during inference, i.e., the term can be added any time the network makes a prediction. The efficiency gap is zero if the current estimates satisfy the "Efficiency"-property and positive if they don't. Jethani et al. [4] prove that adding this term is guaranteed to make the estimates closer to the true Shapley values. The other proposed strategy is to alter the loss function (36) with a penalty $\gamma > 0$ on the efficiency gap. As $\gamma \to \infty$, the "Efficiency"-property is guaranteed to hold [4]. The loss function (36) with the efficiency gap added as a regularizing term is

$$\mathcal{L}_{\text{eff}}^{\text{fast}}(\boldsymbol{\theta}) = \mathcal{L}^{\text{fast}}(\boldsymbol{\theta}) + \gamma \cdot \left(v_{\boldsymbol{x}^*, y^*}(\mathcal{Q}) - v_{\boldsymbol{x}^*, y^*}(\emptyset) - \sum_{j=1}^{q} \phi_j^{\text{fast}}(\boldsymbol{x}^*, y^*; \boldsymbol{\theta}) \right)^2, \tag{38}$$

where the penalty parameter γ must be specified by the user, and can therefore be seen as a hyperparameter of the method.

In their empirical studies, Jethani et al. [4] explore the effect of different combinations of how to enforce the efficiency constraint. In summary, they find that enforcing the additive efficiency normalization both during training and inference results in more accurate Shapley value estimates than either not enforcing it at all or only enforcing it during inference. Moreover, adding the regularizing term to the loss function, like in (38), with penalty parameter $\gamma = 0.1$, yields less accurate estimates of the Shapley values. Therefore, in their experiments, the authors use additive efficiency normalization both during training and inference and no regularization term. These are also the default choices in the implementations of FastSHAP in PyTorch and TensorFlow [4].

3.4.2 Training the Shapley Value Estimator

In this section, the algorithm for training the Shapley value estimator $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ is given. Two versions of this algorithm are given. Algorithm 2 gives an overview of the training process, and Algorithm 3 gives more details, including techniques guaranteed to reduce the variance of the estimate [4].

The feature observations $\{\boldsymbol{x}_i\}_{i=n}^n$ from the data set the black box model was trained on are given as inputs to Algorithm 2. Moreover, a given contribution function $v_{\boldsymbol{x},y}$, or an estimate thereof, is required in the algorithm. Recall that the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ is a neural network. Therefore, a learning rate α to use in the update of the neural network's parameters is required and input in the algorithm. By default, in the FastSHAP implementations in PyTorch and TensorFlow, the optimizer in the neural network is the Adam-optimizer that was described in Section 2.2.2.3, thus α is the learning rate that will be used in the Adam-algorithm 1. Algorithm 2 outputs the Shapley value estimator $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$, that predicts the Shapley values of an instance $(\boldsymbol{x}^*, y^*)$ for all features $j = 1, 2, \ldots, q$ and all classes $y = 1, 2, \ldots, K$.

The first step of Algorithm 2 is to initialize a value of the machine learning model that will predict the Shapley values $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$. Then, until the neural network training converges, Steps 3-13 of the algorithm are repeated. The convergence criterion of training a neural network can be specified in different ways. In the FastSHAP implementations in PyTorch and TensorFlow, an adaptive learning rate is used in addition to early stopping based on a validation batch. In Appendix A, the implementation details of the method are given, including the convergence criteria. Until convergence is met the following steps are repeated.

In Step 3, a feature observation $\boldsymbol{x}'$ is sampled from the data set of feature observations $\{\boldsymbol{x}_i\}_{i=n}^n$ given as input to the algorithm. Next, a feature coalition $\mathcal{S}'$ is drawn from the probability distribution $p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$ that is defined via the Shapley kernel weights (32). In Step 5, the loss $\mathcal{L}$ is initialized to 0. Then, iterating through the classes $y = 1, 2, \ldots, K$ in the for-loop in lines 6-11, facilitates the learning of the Shapley values for all K classes. In Step 7, using the current model estimate $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$, the Shapley values of the sample $\boldsymbol{x}'$ corresponding to class y is predicted. If the boolean indicator corresponding to additive efficiency normalization is set to true (this is checked in Step 8), the current estimate $\hat{\boldsymbol{\phi}}$ is updated by adding the term corresponding to the efficiency gap defined in (37) in Step 9. Next, in Step 11, the loss function (36) evaluated at the current estimate for class y is added to the previous value of the loss. This is done to evaluate the total loss over all the classes $y = 1, 2, \ldots, K$. After Steps 7-11 have been repeated for all K

Algorithm 2: FastSHAP training

Input: Data set $\{\boldsymbol{x}_i\}_{i=n}^n$ consisting of feature observations only

Input: Contribution function (estimator) $v_{\boldsymbol{x},y}$

Input: Learning rate α

Output: Shapley value predictor $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$

initialize $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$

while *not converged* **do**

  sample $\boldsymbol{x}' \sim \{\boldsymbol{x}_i\}_{i=n}^n$

  sample $\mathcal{S}' \sim p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$

  $\mathcal{L} \leftarrow 0$

  for $y = 1, 2, \ldots, K$ **do**

    predict $\hat{\boldsymbol{\phi}} \leftarrow \boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}', y; \boldsymbol{\theta})$

    if *additive efficiency normalization* **then**

      set $\hat{\boldsymbol{\phi}} \leftarrow \hat{\boldsymbol{\phi}} + \frac{1}{q}\left(v_{\boldsymbol{x}',y}(\mathcal{Q}) - v_{\boldsymbol{x}',y}(\emptyset) - \sum_{j=1}^q \hat{\phi}_j\right)^2$

    end

    calculate $\mathcal{L} \leftarrow \mathcal{L} + \left(v_{\boldsymbol{x}',y}(\mathcal{S}') - v_{\boldsymbol{x}',y}(\emptyset) - \sum_{j \in \mathcal{S}'} \hat{\phi}_j\right)^2$

  end

  update $\boldsymbol{\theta} \leftarrow$ `ADAM_Update` $(\mathcal{L})$

end

classes, the parameters $\boldsymbol{\theta}$ of the neural network are updated according to the Adam scheme given in Algorithm 1. This is the final step performed before starting the next iteration, or if the training has converged, the algorithm finishes and returns the current model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$.

Notice that in the definition of the loss function $\mathcal{L}^{\text{fast}}(\boldsymbol{\theta})$ in (36), the expectation is taken over the probability distribution of $\boldsymbol{x}$, a uniform distribution of y and the Shapley kernel distribution $p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$ over $\mathcal{S}$. In practice, this is approximated by using the empirical distribution over $\boldsymbol{x}$, which is standard procedure in machine learning. Moreover, all the K classes are considered in each iteration of the training, thus, the consideration of all the classes is deterministic. Lastly, the expectation over the distribution $p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$ is imitated by considering a randomly selected subset of all the coalitions drawn with probability $p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$.

The full training procedure of FastSHAP is more complicated and involves several variance-reducing techniques that are common in deep learning and some that are specific to the Shapley value estimation problem. The full algorithm is given in Algorithm 3. Some of the steps in the algorithm are strictly necessary parts of the FastSHAP method, however, others are hyperparameters/variations of the neural network that can be tuned like in any other deep learning task. Throughout the description of the algorithm, it will be clarified which parts are standard techniques in deep learning, and which are FastSHAP specific. However, it should be mentioned that the FastSHAP-specific parts are built into the neural network, so the whole procedure is performed through the neural network.

The first three inputs of Algorithm 3 are the same as the inputs of Algorithm 2. These three inputs are the training data set of feature observations $\{\boldsymbol{x}_i\}_{i=1}^n$, the contribution function, or an estimate thereof, $v_{\boldsymbol{x},y}(\mathcal{S})$, and the learning rate α of the optimizer in the neural network. As additional input parameters, Algorithm 3 takes the batch size B to use in the neural network, details about using batches in the training of a neural network can be found in Section 2.2.2.4. The FastSHAP method allows for considering several feature coalitions $\mathcal{S}$ per instance $\boldsymbol{x}$ in a batch. In the exact Shapley values, all possible feature coalitions are considered per instance. Therefore, it is sensible that considering several coalitions per instance should better enable the model to learn to estimate

Algorithm 3: FastSHAP training in-depth

Input: Data set $\{\boldsymbol{x}_i\}_{i=1}^n$ consisting of feature observations only

Input: Contribution function (estimator) $v_{\boldsymbol{x},y}$

Input: Learning rate α

Input: Batch size B

Input: Number of different coalitions to consider per instance n_{coals}

Input: Penalty parameter γ

Output: Shapley value predictor $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$

initialize $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$

while *not converged* **do**

  set $\mathcal{R} \leftarrow 0, \quad \mathcal{L} \leftarrow 0$

  for b = *1, 2, ...* B **do** `// B is the batch size`

    sample $\boldsymbol{x}' \sim \{\boldsymbol{x}_i\}_{i=1}^n$ `// Randomly draw an instance from data set`

    for y = *1, 2, ...,* K **do** `// K is the number of classes`

      predict $\hat{\boldsymbol{\phi}} \leftarrow \boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}', y; \boldsymbol{\theta})$

      calculate $\mathcal{R} \leftarrow \mathcal{R} + \left(v_{\boldsymbol{x}',y}(\mathcal{Q}) - v_{\boldsymbol{x}',y}(\emptyset) - \sum_{j=1}^q \hat{\phi}_j\right)^2$ `// Pre-normalization by adding regularizing term to the loss`

      if *additive efficiency normalization* **then**

        set $\hat{\boldsymbol{\phi}} \leftarrow \hat{\boldsymbol{\phi}} + \frac{1}{q}\left(v_{\boldsymbol{x}',y}(\mathcal{Q}) - v_{\boldsymbol{x}',y}(\emptyset) - \sum_{j=1}^q \hat{\phi}_j\right)^2$

      end

      for m = *1, 2, ...,* n_{coals} **do** `//` n_{coals} `is the number of coalitions` $\mathcal{S}$ `to consider per sample` $\boldsymbol{x}'$

        if *paired sampling* **and** $b \mod 2 = 0$ **then**

          set $\mathcal{S}' \leftarrow (\mathcal{S}')^c$

        else

          sample $\mathcal{S}' \sim p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$

        end

        calculate $\mathcal{L} \leftarrow \mathcal{L} + \left(v_{\boldsymbol{x}',y}(\mathcal{S}') - v_{\boldsymbol{x}',y}(\emptyset) - \sum_{j \in \mathcal{S}'} \hat{\phi}_j\right)^2$

      end

    end

  end

  update $\boldsymbol{\theta} \leftarrow$ `ADAM_Update` $\left(\frac{\mathcal{L}}{B \cdot K \cdot n_{\text{coals}}} + \gamma \frac{\mathcal{R}}{B \cdot K}\right)$

end

the Shapley values. Therefore, the desired number of coalitions to consider per instance n_{coals} is given as an input to the algorithm. Thus, within each batch, n_{coals} coalitions $\mathcal{S}$ are considered per observation $\boldsymbol{x}$. Lastly, the penalty parameter γ in the regularized loss (38) must be provided as input. As mentioned, $\gamma = 0$ by default in the algorithm because Jethani et al. [4] provide empirical evidence that using the regularized loss (38) reduces the accuracy of the predictions. The output of Algorithm 3 is the neural network that predicts Shapley values $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$.

The first step of Algorithm 3 is to initialize the value of $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$. Next, Steps 3-22 are repeated until convergence. The first step per iteration is to set the current value of the loss (36) and regularizing term, which is the efficiency gap defined in (38), to zero. Then the training procedure is performed batch-wise by iterating through the batches that the training data has been divided into. The for-loop in lines 4-21 of the algorithm contains the steps that are repeated per sample in a batch. Using batches is a common neural network strategy and not specific to FastSHAP. The size of the batches can be seen as a hyperparameter of the neural network. The first step performed within a batch is to sample a training observation from the data set, as seen in step 5. The samples are drawn without replacement. Thus, in the span of an epoch, all the samples in the training data set are considered once. For each sample, the algorithm considers the predictions of the Shapley values for each of the K classes by iterating through the classes in the for-loop in Steps 6-20. The first step of the for-loop, Step 7 of the algorithm, is to predict the Shapley values using the current model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$. Then, the regularizing term for that prediction is added to the current value of the regularizing term in Step 8. Step 8 is FastSHAP specific, however, as mentioned in Section 3.4.1, Jethani et al. [4] generally find that efficiency regularization decreases the accuracy of the estimated Shapley. Therefore, by default, the penalty parameter γ that controls the degree of regularization is set to zero, and no regularization is performed. However, Jethani et al. [4] find that additive efficiency normalization (37) yields more accurate estimates, and therefore, by default, this is applied in Step 10 of Algorithm 3. In the implementations, however, the user can choose to omit it if desired. The additive efficiency normalization is also a FastSHAP-specific step.

The for-loop in Lines 12-19 handles the sampling of coalitions $\mathcal{S}$. Per feature observation $\boldsymbol{x}'$, a total of n_{coals} different coalitions are randomly drawn according to the Shapley kernel probability distribution $p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$ where $k(\mathcal{Q}, \mathcal{S})$ are as in (32). Since paired sampling has been found to improve the accuracy of other Shapley value estimators [32], it is included in FastSHAP as well. Jethani et al. [4] provide results showing that this improves the accuracy of the estimates. If paired sampling is desired, rather than drawing a new coalition in each iteration, at every second iteration, the complement of the previous coalition $\mathcal{S}^c$ will be used in the following iteration. The loss per coalition for sample $\boldsymbol{x}'$ and class y is added to the total loss in Step 18 of the algorithm. Thus, the loss that is considered for each sample $\boldsymbol{x}'$ will be over all classes $y = 1, 2, \ldots, K$ and all n_{coals} coalitions. The full for-loop in Steps 12-19 is FastSHAP-specific since the loss function (36) that is used is chosen because its global minimizer is the Shapley values. Therefore, this is the key step to ensuring that the neural network will learn the Shapley values rather than predicting the response variables, which is the most common task in machine learning. Moreover, the coalition sampling that is performed in the if-else statement in Lines 13-17, a total of n_{coals} times, is FastSHAP specific. To empirically learn the minimizer of the loss function (36), taking the expectation over $p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$ must be approximated. Therefore, for all instances in a batch and all K classes, the loss over n_{coals} coalitions $\mathcal{S}$ is considered as an approximation of the expectation. After all $y = 1, 2, \ldots, K$ classes have been considered, the iteration for that sample ends, and the next sample in the current batch is processed. After a whole batch has been processed in this manner, the parameters $\boldsymbol{\theta}$ of the FastSHAP predictive model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ are updated in Step 22 by using the Adam optimizer, which is given in Algorithm 1. By default, FastSHAP uses the Adam optimization scheme given in Algorithm 1. In general, the choice of an optimization strategy is a hyperparameter of the neural network and can be changed. However, the specific loss that is numerically minimized in Step 22 is FastSHAP specific.

The algorithm either continues by considering a new batch, repeating steps 4-22, either until the total number of epochs has been reached or until the convergence criterion is met. By default in the Python implementations of FastSHAP in PyTorch and TensorFlow, an adaptive learning rate and early stopping based on the loss of a validation batch is used. However, there are other strategies that can be used to evaluate the convergence of a neural network. Therefore, the convergence criteria can be seen as a hyperparameter of the neural network.

This completes the description of training the Shapley value prediction model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ and the FastSHAP method. As previously mentioned, estimating the contribution function (29) is an important part of estimating the Shapley values (28). Therefore, two methods for estimating the contribution function (29) are presented in the following section.

3.5 Estimating the Contribution Function

To be able to estimate the Shapley values (28), it is necessary to compute the contribution function $v_{\boldsymbol{x}^*,y^*}(\mathcal{S})$ defined in (29) for all feature coalitions $\mathcal{S} \subseteq \mathcal{Q}$. In this section, two estimators of the contribution function are presented. The first method is based on Monte Carlo integration (26) and assumes feature independence, an assumption that rarely holds in practice. This estimator was originally used in the KernelSHAP method [2] that was presented in Section 3.3. This estimate will be called the "off-manifold" estimate like in [3] to make it easier to distinguish the two methods from each other. It is called the off-manifold estimate because the Monte Carlo samples generated when falsely assuming independence will generally be far from the true distribution of the data and lie off the data manifold. The other estimation method trains a surrogate machine learning model to predict the value of the contribution function. The surrogate model is a supervised machine learning method. We follow the supervised procedure of Frye et al. [3], which is the proposed default contribution function estimator in the FastSHAP method [4]. The estimate resulting from the predictions of the surrogate model will be referred to as the "on-manifold" estimate because they generally will lay on the data manifold [3]. Throughout the whole section, we estimate the contribution function (29). It should be noted that there are two problems with computing (29) in practice. First, the conditional distributions $p(\boldsymbol{x}_{\mathcal{S}^c}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})\forall\mathcal{S} \subseteq \mathcal{Q}$ are generally unknown and analytically intractable in most real-world situations. Moreover, given a conditional distribution, or an estimate thereof, for many interesting machine learning models, calculating the expected value in (29) is not generally feasible, especially when the number of features q is high since this corresponds to a $q - |\mathcal{S}|$ dimensional integral of a complex expression. Therefore, estimation procedures are necessary.

3.5.1 The Off-Manifold Estimate via Monte Carlo Integration

As mentioned, the off-manifold estimate is to estimate the contribution function (29) by Monte Carlo integration (26) under the assumption of feature independence [2, 1]. Under the assumption of feature independence, the following rewrite of the contribution function (29) holds

$$\begin{aligned} v_{\boldsymbol{x}^*,y^*}(\mathcal{S}) &= \mathop{\mathbb{E}}_{p(\boldsymbol{x}_{\mathcal{S}^c}|\boldsymbol{x}_{\mathcal{S}}=\boldsymbol{x}^*_{\mathcal{S}})}[\hat{f}_y(\boldsymbol{x})|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}] = \mathop{\mathbb{E}}_{p(\boldsymbol{x}_{\mathcal{S}^c}|\boldsymbol{x}_{\mathcal{S}}=\boldsymbol{x}^*_{\mathcal{S}})}[\hat{f}_y(\boldsymbol{x}_{\mathcal{S}^c}, \boldsymbol{x}_{\mathcal{S}})|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}] \\ &= \int \hat{f}_y(\boldsymbol{x}_{\mathcal{S}^c}, \boldsymbol{x}^*_{\mathcal{S}}) p(\boldsymbol{x}_{\mathcal{S}^c}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}) d\boldsymbol{x}_{\mathcal{S}^c} \overset{\text{independence}}{=} \int \hat{f}_y(\boldsymbol{x}_{\mathcal{S}^c}, \boldsymbol{x}^*_{\mathcal{S}}) p(\boldsymbol{x}_{\mathcal{S}^c}) d\boldsymbol{x}_{\mathcal{S}^c}, \end{aligned}$$

where the independence assumption leads to the assumption that the conditional distribution $p(\boldsymbol{x}_{\mathcal{S}^c}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}) = p(\boldsymbol{x}_{\mathcal{S}^c})$, where $p(\boldsymbol{x}_{\mathcal{S}^c})$ is the marginal distribution of the features in $\mathcal{S}^c$. Under this assumption, the contribution function can be estimated by Monte Carlo integration with respect to the empirical distribution of the features, discussed in Section 2.4. This yields the off-manifold estimate

$$v^{\text{off}}_{\boldsymbol{x}^*,y^*}(\mathcal{S}) = \frac{1}{M}\sum_{m=1}^{M} \hat{f}_y(\boldsymbol{x}^m_{\mathcal{S}^c}, \boldsymbol{x}^*_{\mathcal{S}}), \tag{39}$$

where $\boldsymbol{x}^m$ for $m = 1, 2, \ldots, M$ is a randomly drawn instance from the data set and M is the total number of Monte Carlo samples. The feature values in $\boldsymbol{x}^m_{\mathcal{S}^c}$ are sampled from the data set. In practice, we randomly choose an instance in the data set and extract the features in $\mathcal{S}^c$ from it. This sampling will implicitly follow the empirical distribution of the features in $\mathcal{S}^c$ since more common feature values are more likely to be drawn because there are more instances in the data with these feature values. If the independence assumption were not made, then we would have to sample in a way that considers how the features in $\mathcal{S}^c$ depend on the features in $\mathcal{S}$. By simply sampling a point in the data and extracting the values of some of the features, no such relationship is considered, and only the empirical distribution of the feature in $\mathcal{S}^c$ is taken into account. The

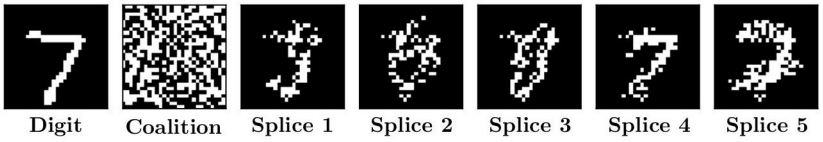


Figure 4: Some examples of unrealistic digits that were generated by splicing observations together, creating the Monte Carlo samples in the off-manifold estimate. The examples of Monte Carlo samples are captioned "Splice 1"-"Splice 5" and the original instance is the observation of the digit 7 at the left-hand side of the figure. The coalition $\mathcal{S}$ used to create the splices is shown in the second position from the left. This figure is taken from [3].

Monte Carlo samples can be drawn from the training data set, the test data set, or both of these data sets combined. In our experiments that are presented in Section 4 and 5, it is ensured that the same data set is used in all estimates to ensure that the conditions of the experiments are as similar as possible.

For $\mathcal{S} = \mathcal{Q}$ since $\mathcal{Q}^c = \emptyset$, the estimate is

$$\hat{v}^{\text{off}}_{\boldsymbol{x}^*,y^*}(\mathcal{Q}) = \frac{1}{M}\sum_{m=1}^{M} \hat{f}_y(\boldsymbol{x}^m_\emptyset, \boldsymbol{x}^*_\mathcal{Q}) = \frac{1}{M}\sum_{m=1}^{M} \hat{f}_y(\boldsymbol{x}^*_\mathcal{Q}) = \hat{f}_y(\boldsymbol{x}^*),$$

which is the black box model's prediction for the instance of interest. Thus, this can be computed separately by evaluating the black box model instead of estimating it using Monte Carlo integration. For $\mathcal{S} = \emptyset$, note that $v(\emptyset) = \mathbb{E}[\hat{f}_y(\boldsymbol{x})]$. This is the global average prediction of the model. Thus, it can be estimated as the mean prediction in the data set and only has to be computed once, since it does not depend on the instance of interest. For the other $|\mathcal{Q}| - 2$ subsets, M samples are used. Thus in total, to estimate the contribution function for all feature coalitions, $(|\mathcal{Q}| - 2) \cdot M$ Monte Carlo samples are used in the off-manifold estimate. To simplify, we round the total number of samples off to $|\mathcal{Q}| \cdot M$. This corresponds to $|\mathcal{Q}|$ evaluations of the black box model per instance of interest. The number of model evaluations is typically the size that is used to characterize the computational cost of the different estimates. Examples of this can be found in among others [2, 4, 3].

As previously mentioned, in the off-manifold estimate (39) of the contribution function (29), the features in $\mathcal{S}$ are assumed independent of the features in $\mathcal{S}^c$. In practice, the features are often correlated (and thus not independent), and this assumption can lead to unrealistic Monte Carlo samples with feature combinations that in reality could not have occurred. A good example of this is given for the MNIST data set [34] in Frye et al. [3], which can be seen in Figure 4. The MNIST data set consists of images of the handwritten digits 0 to 9. By combining the observations together under the assumption of feature independence, here resulting in the assumption of independent pixels, to create Monte Carlo samples, unrealistic digits are created. Although this thesis does not treat image data, this example is included because it serves as a good visual illustration of how incorrectly assuming feature independence can result in unrealistic Monte Carlo samples. For samples like these, the off-manifold method may "evaluate the [black box] model outside its domain of validity, where it is untrained and potentially wildly misbehaved. This garbage-in-garbage-out problem is the clearest reason to avoid the off-manifold approach" [3]. Especially deep learning models are known to be sensitive to distributional shifts [3], which means that evaluating the models outside the region they are trained on can lead to spurious and misleading final explanations.

Recall from Section 2.4 that the variance of the Monte Carlo integral decreases as M increases. By reducing the variance, the error (27) in the Monte Carlo integral decreases. However, in our case, because $v^{\text{off}}_{\boldsymbol{x}^*,y^*} \overset{\text{a.s.},M\to\infty}{\longrightarrow} \mathbb{E}_{p(\boldsymbol{x}_{\mathcal{S}^c})}[\hat{f}_y(\boldsymbol{x}^c_\mathcal{S}, \boldsymbol{x}^*_\mathcal{S})] \neq \mathbb{E}_{p(\boldsymbol{x}_{\mathcal{S}^c}|\boldsymbol{x}_\mathcal{S}=\boldsymbol{x}^*_\mathcal{S})}[\hat{f}_y(\boldsymbol{x}^c_\mathcal{S}, \boldsymbol{x}^*_\mathcal{S})] = v_{\boldsymbol{x}^*,y^*}$, in general, there is a bias in the estimate. Therefore, even though the variance of the estimate can be reduced by increasing the number of Monte Carlo samples, the bias term remains, which corresponds to an error in the estimate.

In general, it is desirable to calculate the Shapley value for dependent features. The exact Shapley

value incorporates the dependence structure between features. Therefore, to maintain this, it is necessary to use another estimation method to approximate the contribution function, that takes into account the dependence structure of the features. For instance, Aas, Jullum and Løland [8] propose several methods for estimating the contribution function (29) without the assumption of independent features. However, these methods do not scale to higher dimensional problems. Another contribution function estimation method that does assume feature independence and scales to higher dimensional problems is the surrogate model [3], which will be presented in the following section. Similar to the FastSHAP model, the method trains a machine learning model to learn the contribution function and amortizes the estimation of the contribution function across the number of instances to explain. Therefore, it can provide estimates fast by evaluating the surrogate model after the initial computationally costly phase has been performed. This makes the surrogate model useful in many practical situations.

3.5.2 The On-Manifold Estimate via a Supervised Surrogate Model

As a default choice of an estimator of the contribution function (29) in FastSHAP, Jethani et al. [4] consider a surrogate model. The surrogate model is a supervised machine learning model that is trained to learn the contribution function (29). This results in a method with a high initial computational cost, but after the initial model training, a new explanation can be given with a simple model evaluation. Thus, the method amortizes the computational cost of estimating the contribution function over the number of instances to explain. Training a predictive model to either directly learn the contribution function (29) or learn the conditional distributions $p(\boldsymbol{x}_{\mathcal{S}^c}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})$ has been done previously by Frye et al. [3] and Jethani et al. [9]. In this thesis, the surrogate model is trained in accordance with the supervised method of Frye et al. [3] like in the FastSHAP method [4], that was introduced in Section 3.4. The surrogate model is a machine learning model, a neural network, which has high representation abilities. Details on neural networks can be found in Section 2.2.2. Unlike the off-manifold estimate from the previous section that assumes feature independence, a sufficiently large neural network can learn interactions between the features. The estimate of the surrogate model will, therefore, more closely follow the correct distribution and lie **on** the data manifold [3]. Frye et al. [3] provide evidence that the surrogate model outperforms the off-manifold estimator for some real-world data sets. We now present the details of the method.

Frye et al. [3] introduce their surrogate model in the classification setting. Although the method can be generalized to the regression setting [3], we here give the details for the classification setting. Recall that FastSHAP [4] also was introduced in the classification setting, which makes the surrogate model and FastSHAP compatible. KernelSHAP, as presented in Section 3.3, can be used both in the classification and regression setting. The combination of KernelSHAP and the surrogate model is therefore also possible. The surrogate model aims at learning the expected value of the black box machine learning model conditional on having observed some of the feature values (29). To replicate the evaluation of the black box model on a subset of the features $\mathcal{S}$, the idea is to replace the feature values in $\mathcal{S}^c$ with a [`mask`]-value. Supervised by the black box model's predictions for the unmasked, original feature observations, the surrogate model tries to learn to replicate these predictions accurately, but it only observes the masked vector of features. The minimizer of the loss function that is used in the training of the surrogate model is the contribution function (29). Therefore, the surrogate model can learn the contribution function if provided with a sufficiently large data set to train on. This is the intuition behind the method, and the details will be given in the following sections.

We use similar notation to Jethani et al. [4] in the description of the surrogate model. The surrogate model takes as input a masking $\boldsymbol{m}(\boldsymbol{x}, \mathcal{S})$ of the features such that for $j \in \mathcal{S}^c$ the feature value x_j is replaced by a value that is not in the support of $\hat{f}(\boldsymbol{x})$. Recall that the full black box model that makes predictions for all K classes is denoted $\hat{f}(\boldsymbol{x})$, and as before, $\hat{f}_y(\boldsymbol{x})$ denotes the model that makes prediction for a single class $y \in \{1, 2, \ldots, K\}$. Specifically, the jth component of the masking is defined as

$$m_j(\boldsymbol{x}, \mathcal{S}) = \begin{cases} x_j, & j \in \mathcal{S}, \\ [\texttt{mask}], & j \notin \mathcal{S}. \end{cases} \tag{40}$$

The [`mask`]-value must be chosen by the user, and can therefore be seen as a hyperparameter of

the method, Frye et al. [3] use a value that has not been observed in the data set. We stick to the notation of Jethani et al. [4] and denote the surrogate model as $\hat{v}^{\text{on}}(y|\boldsymbol{m}(\boldsymbol{x}, \mathcal{S}); \boldsymbol{\beta})$, where $\boldsymbol{\beta}$ are the parameters of the surrogate model. According to Jethani et al. [4], the parameters of the surrogate model are learned by minimizing

$$\mathcal{L}^{\text{surr}}(\boldsymbol{\beta}) = \mathop{\mathbb{E}}_{p(\boldsymbol{x})} \mathop{\mathbb{E}}_{p(\mathcal{S})} \left[D_{\text{KL}} \left(\hat{f}(\boldsymbol{x}) \| \hat{v}^{\text{on}}(y \mid \boldsymbol{m}(\boldsymbol{x}, \mathcal{S}); \boldsymbol{\beta}) \right) \right], \tag{41}$$

where the Kullback–Leibler divergence $D_{\text{KL}}(p_1 \| p_2)$ between two probability distributions p_1 and p_2 is defined as

$$D_{\text{KL}}(p_1 \| p_2) = \mathop{\mathbb{E}}_{p_1(\boldsymbol{x})} \left[\log \left(\frac{p_1(\boldsymbol{x})}{p_2(\boldsymbol{x})} \right) \right]. \tag{42}$$

Since the surrogate model $\hat{v}^{\text{on}}(y \mid \boldsymbol{m}(\boldsymbol{x}, \mathcal{S}); \boldsymbol{\beta})$ was introduced for classification problems, $\hat{f}(\boldsymbol{x})$ is a probability density function, accordingly so is $\hat{v}^{\text{on}}(y \mid \boldsymbol{m}(\boldsymbol{x}, \mathcal{S}); \boldsymbol{\beta})$, and using the Kullback-Leibler divergence is unproblematic. Keeping the probability density function p_1 fixed, minimizing the Kullback–Leibler divergence (42) is equivalent to minimizing the categorical cross-entropy between the distributions. The categorical cross-entropy is defined as

$$H(p_1, p_2) = - \mathop{\mathbb{E}}_{p_1(\boldsymbol{x})} [\log(p_2(\boldsymbol{x}))] = D_{\text{KL}}(p_1 \| p_2) - \mathop{\mathbb{E}}_{p_1(\boldsymbol{x})} [\log(p_1(\boldsymbol{x}))], \tag{43}$$

where the second term only depends on p_1. Therefore, minimization with respect to p_2 of the categorical cross-entropy and the Kullback-Leibler divergence is equivalent. As cited in [4], Covert, Lundberg and Lee [35] show that the global optimizer of (41), which is the surrogate model, is equivalent to marginalizing out features from $\hat{f}(\boldsymbol{x})$ with their conditional distribution, thus,

$$\hat{v}^{\text{on}}(y \mid \boldsymbol{m}(\boldsymbol{x}^*, \mathcal{S}); \boldsymbol{\beta}) = \mathop{\mathbb{E}}_{p(\boldsymbol{x}_{\mathcal{S}^c} | \boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})} [\hat{f}(\boldsymbol{x}) | \boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}}], \tag{44}$$

which is exactly the contribution function (29).

Note that in Frye et al. [3], where the supervised surrogate model was introduced, the proposed loss function was the mean squared error, rather than the Kullback-Leibler divergence as in (41). However, both loss functions have (44) as the minimizer, which is the contribution function (29). However, for classification problems, it is more common to use the categorical cross-entropy loss than the mean squared error, therefore we stick to the practice of Jethani et al. [4] and use the loss function (41). This also makes it easier to compare our results to those of Jethani et al. [4] since we use the same loss function as them.

In summary, to estimate the contribution function, a machine learning model can be trained on masked feature observations from the original data set with (41) as the loss function. The response variables used in the training are the predictions of the black box model, not the response variables in the original data set. In practice, it is necessary to randomly sample subsets $\mathcal{S}$ determining which features to mask according to (40) and samples $\boldsymbol{x}$ from the data set to train the model on, in order to simulate taking the expectation over $p(\boldsymbol{x})$ and $p(\mathcal{S})$ in (41). The samples of $\boldsymbol{x}$ are drawn from the available data set, and therefore, follow the empirical distribution of the features in the data set. Moreover, the subsets $\mathcal{S}$ are drawn from the Shapley kernel probability distribution $p(\mathcal{Q}, \mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$ where $k(\mathcal{Q}, \mathcal{S})$ is defined in (32).

The details of training the surrogate model are given in Algorithm 4. The [`mask`]-value used to fill in the missing feature values should not be in the support of the black box model $f(\boldsymbol{x})$. In addition, for the surrogate model to learn that some features have been masked, the masking locations can be provided to the model. This is done by providing a masking net which for each observation is the same dimension as $\boldsymbol{x}$. The elements of the mask net are 0 if the corresponding feature is in $\mathcal{S}^c$ and 1 if the feature is in $\mathcal{S}$. Whether to append this to the observations the model is trained on is determined by a boolean parameter `append_mask`, which is given as an input in Algorithm 4. The other input parameters of the algorithm are the [`mask`]-value, the feature observations from the original data set $\{\boldsymbol{x}\}_{i=1}^n$ that will be used to train the model on, and the corresponding predictions of the black box model for these feature observations $\{y_i^{\text{bb}}\}_{i=1}^n$. These are used as the response variables in the training. The algorithm outputs the surrogate model that predicts the value of the contribution function for a masked feature vector $\hat{v}^{\text{on}}(y|\boldsymbol{m}(\boldsymbol{x}, \mathcal{S}); \boldsymbol{\beta})$. The first step of the

Algorithm 4: Surrogate model training

Input: Mask value: [`mask`]

Input: Boolean determining whether to append masking to the data set that the surrogate model is trained on: `append_mask`

Input: Feature observations: $\{\boldsymbol{x}_i\}_{i=1}^n$

Input: Black box model's predictions, which are denoted $\{y_i^{\text{bb}}\}_{i=1}^n$ in the algorithm

Output: Contribution function estimator: $\hat{v}^{\text{on}}(y|\boldsymbol{m}(\boldsymbol{x}, \mathcal{S}); \boldsymbol{\beta})$

Initialize $\hat{v}^{\text{on}}(y|\boldsymbol{m}(\boldsymbol{x}, \mathcal{S}); \boldsymbol{\beta})$

while *not converged* **do**

  $\boldsymbol{x}' \sim \{\boldsymbol{x}_i\}_{i=1}^n$ sample a feature observation from the training data given as input with corresponding response y'_{bb}

  sample $\mathcal{S}' \sim p(\mathcal{S}) \propto k(\mathcal{Q}, \mathcal{S})$

  create masked vector $\boldsymbol{x}'_{\mathcal{S}'}$ with elements $(x'_{\mathcal{S}'})_j = \begin{cases} x'_j, & j \in \mathcal{S}', \\ [\texttt{mask}], & j \notin \mathcal{S}' \end{cases}$

  if `append_mask` **then** `// If specified, the model is also given the coalition` $\mathcal{S}'$ as an input to train on

    $\boldsymbol{x}'_{\mathcal{S}'} \leftarrow \left(\boldsymbol{x}'_{\mathcal{S}'}, \quad \mathcal{S}'\right)$

  end

  $\mathcal{L} \leftarrow 0$

  for *y = 1, 2, . . . , K* **do**

    predict $\hat{v} \leftarrow \hat{v}^{\text{on}}(y|\boldsymbol{x}'_{\mathcal{S}'}; \boldsymbol{\beta})$

    calculate $\mathcal{L} \leftarrow \mathcal{L} + H(\hat{v}, y'_{\text{bb}})$

  end

  update $\boldsymbol{\beta} \leftarrow$ `Adam_Update`$(\mathcal{L})$

end

algorithm is to initialize the surrogate model $\hat{v}^{\text{on}}(y|\boldsymbol{m}(\boldsymbol{x},\mathcal{S});\boldsymbol{\beta})$. Then, until convergence, Steps 3-14 are repeated. In Step 3, a sample $\boldsymbol{x}'$ is drawn randomly from the data set given as input to the model. It is paired with the corresponding prediction of the black box model y'_{bb}. In addition, in Step 4, a feature coalition $\mathcal{S}$ is drawn according to $p(\mathcal{Q},\mathcal{S}) \propto k(\mathcal{Q},\mathcal{S})$ defined in (32). In Step 5, a masked vector based on the sample $\boldsymbol{x}'$ and coalition $\mathcal{S}$ is created in accordance with (40). Then, if the user has specified to append the mask net to the training data, this is done in Step 7 of the algorithm. Then, the loss is initialized to 0 in Step 9. Next, for the K classes $y = 1, 2, \ldots, K$, the current estimate of the model is predicted in Step 11. The categorical cross-entropy loss H (43) of the prediction is added to the loss in Step 12. The loss is added because the loss over all K classes must be considered. After the loss over all K classes has been computed, the parameters $\boldsymbol{\beta}$ of the surrogate model are updated according to the Adam optimization scheme given in Algorithm 1 in Step 14.

3.6 Full Procedures for Estimating the Shapley Values

In the previous sections, two contribution function estimators and two Shapley value estimators have been presented. The results of investigating the accuracy of both the contribution function estimates and the Shapley value estimates on simulated data will be presented in Section 4. Since there are two methods for performing each of the two steps in the estimation procedure, there are in total four ways to finally estimate the Shapley values. These are as follows.

(A) **KernelSHAP-Off-Manifold**: The original KernelSHAP version involves estimating the contribution function by the off-manifold estimate (39). Then, estimating the Shapley values by the KernelSHAP approximation (34).

(B) **KernelSHAP-Surrogate**: Estimating the contribution function by a surrogate model trained according to Algorithm 4, and estimating the Shapley values by the KernelSHAP approximation (34).

(C) **FastSHAP-Off-Manifold**: Combining the off-manifold estimate of the contribution function (39) with a machine learning model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ trained to predict the Shapley values according to Algorithm 2. The FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ is trained with the off-manifold estimate as the input of the contribution function estimator in Algorithm 2.

(D) **FastSHAP-Surrogate**: The original version of FastSHAP, using the surrogate model trained according to Algorithm 4 to predict the value of contribution function, and the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ to predict the Shapley values. The FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ is trained with the surrogate model as the input of the contribution function estimator in Algorithm 2.

3.7 The Computational Cost of the Estimators

In many practical applications, the computational cost related to an estimation method is an important factor in choosing a method. We will first consider the computational cost of the off-manifold method, the surrogate model, the KernelSHAP estimate, and the FastSHAP separately. Then, based on the individual cost of each estimator, we outline the cost of the full estimation procedures, defined in Section 3.6. For the estimators we consider, it is difficult to give exact expressions for the computational cost, therefore, we give a simplified overview.

- **Off-manifold estimate**: To calculate the off-manifold estimate (39), one must create $M \cdot |\mathcal{Q}| = M \cdot 2^q$ Monte Carlo samples if all $\mathcal{Q}$ feature coalitions are considered. In addition, one must evaluate the black box model $|\mathcal{Q}| = 2^q$ times. This computation must be repeated for each instance we want to explain.
- **Surrogate model**: The cost of the surrogate model can be split into two parts. These are as follows:

- *Training*: the surrogate model must be trained according to Algorithm 4. The training corresponds to a high initial cost that amortizes across the instances to be explained.
- *Providing an estimate*: In order to compute the value of the contribution function for all feature coalitions $\mathcal{S}$ in $\mathcal{Q}$ for an instance of interest, the surrogate model must be evaluated once for all the $|\mathcal{Q}| = 2^q$ coalitions. This must be repeated for all instances to be explained.

- **KernelSHAP**: The KernelSHAP estimate (34) needs access to the value of the contribution function, or an estimate thereof, for all feature coalitions $\mathcal{S}$ in $\mathcal{D}$. We assume here that it already has been computed and stored in a vector $\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$. Then, the computational cost of KernelSHAP can be divided into two parts as follows:
  - *Initialization*: The matrix $\boldsymbol{R}_{\mathcal{D}}$ in (34) does not depend on the instance of interest and can therefore be precomputed and stored. It must only be computed once. Hence, the initialization cost is amortized over the instances to be explained.
  - *Providing an estimate*: To provide the explanation for an instance, one must compute the matrix-vector product $\boldsymbol{R}_{\mathcal{D}}\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$. This must be repeated for all instances to be explained.

- **FastSHAP**: The computational cost of the FastSHAP model is similar to that of the surrogate model. It divides into two parts as follows:
  - *Training*: The FastSHAP model must be trained according to Algorithm 2. This is an initial cost that amortizes over the instances to be explained. The model can be used to explain any number of instances after the initial training. In the training, a contribution function estimator must be provided as input in Algorithm 2. It is assumed that this is given.
  - *Providing an estimate*: To provide an explanation with FastSHAP, the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ must be evaluated once for every instance of interest $\boldsymbol{x}^*$.

Correspondingly, the outline of the computational cost of the full estimation procedures is as follows:

- **KernelSHAP-Off-Manifold**: The cost divides into two parts, which are:
  - *Initialization*: The initialization corresponds to that of KernelSHAP, which means that the matrix $\boldsymbol{R}_{\mathcal{D}}$ in (34) must be computed.
  - *Providing an estimate*: To provide an explanation for an instance of interest one must first compute the off-manifold estimate for all feature coalitions in $\mathcal{D}$, which is stored in the vector $\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$. This corresponds to generating $M \cdot |\mathcal{D}|$ Monte Carlo samples and evaluating the black box model $|\mathcal{D}|$ times. Then, one must compute the matrix-vector product $\boldsymbol{R}_{\mathcal{D}}\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$.

- **KernelSHAP-Surrogate**: The computation of the KernelSHAP-Surrogate method is as follows:
  - *Initialization*: The surrogate model must be trained according to Algorithm 4. In addition, the matrix $\boldsymbol{R}_{\mathcal{D}}$ in (34) must be computed.
  - *Providing an estimate*: Firstly, the surrogate model must be evaluated once for all feature coalitions in $\mathcal{D}$. The values of the surrogate model's predictions are stored in the vector $\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$. Then, the matrix-vector product $\boldsymbol{R}_{\mathcal{D}}\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$ must be computed.

- **FastSHAP-Off-Manifold**: The cost divides as follows:
  - *Initialization*: The FastSHAP model must be trained according to Algorithm 2. The off-manifold estimator is given as the input of the contribution function estimator in Algorithm 2. When training the model, the off-manifold estimate must be computed for all feature coalitions $\mathcal{S}$ that are considered per instance in a batch.

- *Providing an estimate*: The FastSHAP-Off-Manifold model must be evaluated for each instance of interest $\boldsymbol{x}^*$.

- **FastSHAP-Surrogate**: The cost is divided into two parts as follows:
  - The surrogate model must be trained according to Algorithm 4. Then, with the surrogate model as input, the FastSHAP model must be trained according to Algorithm 3. The surrogate model is evaluated for all feature coalitions per instance in a batch during the training of the model.
  - *Providing an estimate*: The FastSHAP-Surrogate model must be evaluated for each instance of interest $\boldsymbol{x}^*$.

In the experiments that will be presented in Sections 4 and 5, for the different estimation methods, the **central processing unit (CPU) time** it takes to provide an estimate will be given. The CPU time is the usage time of a central processing unit to complete a computation. It provides a more accurate estimate of the computation time than a wall clock because it will not be influenced by other processes running simultaneously. In Python, the function `process_time` in the time library [36] can be used to get the CPU time of a process. All experiments presented in Sections 4 and 5 have been run on the same laptop with an Intel(R) Core(TM) i7-8550U CPU. The CPU time reported in the experiments is the total for the eight threads in the CPU.

3.8 Shapley Values for Global Interpretability

In the theoretical description of the Shapley values, we have described the method as a local explanation method. Local refers to the method explaining the prediction of the black box model for a single instance of interest. The Shapley values attribute the prediction of the black box model to its features. Therefore, if the Shapley values for all observations in a data set are computed, these can be aggregated to provide *global Shapley values*. The global Shapley value of a feature is the average of the absolute Shapley values of that feature over the instances in the data set. The global Shapley values indicate which variable is the most important for the model's predictions overall. If a feature has a high global Shapley value, it has been assigned a high, in absolute value, attribution on average for the observations in the data set. It has, therefore, on average, been important for the black box model's predictions. In practice, because the computational cost of the Shapley values is high, the global Shapley values can be approximated by considering a subset of the full data set rather than the whole data set.

4 Simulation Study

In many real-world problems, the exact Shapley values are unknown due to the contribution function (29) being analytically intractable and the computational complexity of computing the Shapley values (28) that is exponential 2^q in the number of features q. Since the exact Shapley values are generally unknown in practical situations, checking the accuracy of the Shapley value estimates is difficult. Therefore, it is necessary to simulate data where either the Shapley values are known, or a good estimate thereof can be computed using a preexisting method known to be accurate. The Shapley value estimates can then be compared to this estimate. In the simulations that will be presented in this section, we will not have access to the exact Shapley values. However, we will be able to estimate them accurately. Therefore, we can evaluate the accuracy of the Shapley value estimators presented in Sections 3.3 and 3.4 and the contribution function estimators presented in Section 3.5.

Recall that the computation of the Shapley values can be seen as a two step estimation procedure. The first step is to estimate the contribution function, and the second is to estimate the Shapley values for a given contribution function. In Section 3.5, two estimators of the contribution function were presented. To evaluate these, it is necessary to be able to compute the true value of the contribution function (29). If the conditional distributions $p(\boldsymbol{x}_{\mathcal{S}^c}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$ are known, the Monte Carlo integral (26) using the black box model evaluated at samples from this distribution is an unbiased estimator of the contribution function. Therefore, this estimate can be used as the true value, or more precisely, a sufficiently good estimate, of the contribution function. To investigate the accuracy of the estimation methods, simulated data where the conditional distributions $p(\boldsymbol{x}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$ are analytically given is used. Then, fixing an estimate of the contribution function, the exact weighted least squares formulation (33) of the Shapley values can be used to obtain a ground truth estimate of the Shapley values. The Shapley value estimators from Sections 3.3 and 3.4 will be evaluated against this.

In Section 2.3, two multivariate distributions where the conditional densities $p(\boldsymbol{x}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$ are known were introduced. These are the multivariate normal distribution and the multivariate Burr distribution. Both distributions will be used to simulate data, and the performance of the methods will be evaluated for these simulated data sets. In general, the multivariate normal distribution has many desirable properties, but it "behaves too nicely" to replicate real-world data in many cases. Therefore, the Burr distribution is also used because it can have heavier tails, be skewed, and have non-linear correlation structures. Hence, it more accurately represents a complex real-world data set.

The outline of this chapter is as follows. In Section 4.1, the procedure we follow in the experiments is described in detail. Then, in Section 4.2, the evaluation metrics that we will use to measure the accuracy of our method are presented. Next, the simulation models that we use are presented in Section 4.3. Finally, the evaluation of the contribution function estimators is presented in Section 4.4, and in Section 4.5, the Shapley value estimators are evaluated. In all the experiments, we use our own implementation of the estimation methods. The implementation can be found on GitHub. Some details on the hyperparameters of the estimation methods used in the simulation experiments are given in Appendix A.

4.1 Experimental Design

We perform experiments using several simulated data sets. In each simulation experiment, the following steps are repeated.

1. Generate n observations of feature variables $\boldsymbol{x} \in \mathbb{R}^q$ according to one of the models that will be described in Section 4.3. Set 20 % aside to form a test data set.[1] Then, of the remaining 80 % of the observations, let 80 % be the training data set, and the other 20 % be the validation data set. To investigate the effect of the size of the available data set, this is performed for

[1] In the experiments, we only used 100 test observations from the test data set. It was, therefore, unnecessary to generate additional observations in the test data set.

data set sizes $n = 2{,}000, 10{,}000$ and 100,000, which corresponds to training data sets of sizes $n_{\text{train}} = 1{,}280, 6{,}400$ and 64,000, respectively. In addition, for one of the simulation models, the experiments are also performed for $n = 200{,}000$, corresponding to $n_{\text{train}} = 128{,}000$. It will later be argued why this is only done for one of the simulation models.

All estimates, both of the contribution function and the Shapley values, are calculated using the training data set. This is to make the conditions of each experiment as equal as possible. In addition, since a machine learning model is trained both to estimate the contribution function using the surrogate model and to predict the Shapley values using the FastSHAP predictor $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$, it makes sense to stick to the machine learning practice of using separate training, validation, and test data sets. This is to avoid typical machine learning problems like overfitting to the training data set. The surrogate model and FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ are trained on the training data set, and the validation data set is used to evaluate the model at the end of an epoch, as is common when training neural networks, see Section 2.2.2.4 for details on this.

It is interesting to investigate if the performance of the estimators varies depending on the size of the data set because, in real-world applications, the number of available observations can vary greatly. Therefore, empirical results ranking the methods in different situations can make it easier to decide which method to use based on the properties of the available data set.

2. For all simulated feature observations, compute the corresponding response variables. We will return to this in Section 4.3, where the details of the simulation models are given. The response variable is either 0 or 1, corresponding to a classification data set.

3. In all experiments, an XGBoost classifier with default hyperparameter settings is used as the black box machine learning model that will be explained. The XGBoost model is trained on the training data set. It outputs the predicted probabilities of belonging to class 0 and class 1. For reproducibility, the most important hyperparameters in the XGBoost model are listed. They are as follows. The number of boosted trees `n_estimators` is set to 100, the learning rate `eta` is set to 0.3, the maximum depth `max_depth` of each tree is set to 6, and the L2-regularization penalty parameter λ is set to 1. Moreover, the minimum loss reduction required to proceed with a new partition on a leaf of the tree `gamma` is set to 0, i.e., the model will keep building trees even if there was no improvement in the last step. The hyperparameter `min_child_weight`, determining the minimum weight needed inside a child to keep partitioning, is set to 1. This parameter corresponds to the minimum number of instances needed in each node if linear regression mode is used. In the theoretical description of the XGBoost model in Section 2.2.1, we outlined the method and its working mechanisms. However, the full XGBoost library has more hyperparameters and functionality than we covered. In this thesis, we use the XGBoost model as an example of a complex black box model. The focus of this thesis is the contribution function and Shapley value estimators. Therefore, we do not provide more details on XGBoost and its hyperparameters here, but refer to the documentation of the XGBoost Python package [16] for more details.

4. Draw 100 test observations from the test data set. The contribution function and Shapley value estimates will be computed for these observations.

5. Perform either step (a) or step (b) below:

   (a) **Evaluate the contribution function estimators:** When evaluating the contribution function estimators, the number of features in the simulation model q is kept constant and equal to 10. The contribution function estimates will be computed for all $2^q = 2^{10} = 1024$ feature coalition $\mathcal{S}$, and the accuracy of the estimates is calculated over all the coalitions. For each simulated data set, the following steps are performed.

   - Train the surrogate model using the training and validation data sets according to Algorithm 4. Next, for the 100 test observations, use the surrogate model to predict the contribution function estimate for all coalitions $\mathcal{S}$. The hyperparameters and architecture of the surrogate model are given in Appendix A.

- Compute the off-manifold estimate (39) with the number of Monte Carlo samples M equal to 100, 300, and 1,000. The number of Monte Carlo samples used in the off-manifold estimate is varied to investigate how this affects the method's accuracy. Repeat this for all coalitions $\mathcal{S}$ of the features for all 100 test observations.
- Compute the unbiased Monte Carlo estimate (26) of the contribution function using the analytical formula for the conditional probability distributions $p(\boldsymbol{x}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})$ of the simulation models. This will be used as the ground truth value of the contribution function. Repeat this for all coalitions $\mathcal{S}$ for the 100 test observations.

(b) **Evaluate the Shapley value estimators:** When evaluating the Shapley value estimates, *the contribution function estimate is kept fixed* for all Shapley value estimation methods, as well as for estimating the ground truth Shapley values. In our experiments, the number of features to consider must be sufficiently high to replicate a real-world data set, but sufficiently low such that the exact Shapley values (33) can be computed for a given estimate of the contribution function. To investigate how the number of features q affects the performance of the methods, the experiments are performed for $q = 10, 11, \ldots, 16$ feature variables. Since the KernelSHAP estimate (34) uses only a subset of size $|\mathcal{D}|$ out of the total of 2^q feature coalitions, it is especially interesting to investigate the effect of the number of features on the estimate. In each experiment, we perform the following steps.

- Train the FastSHAP prediction model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ using the training and validation data sets according to Algorithm 2. Use $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ to predict the Shapley values of the 100 test observations. Repeat this for $n_{\text{coals}} = 4, 32$, and 64, where n_{coals} is the hyperparameter of the model that determines the number of feature coalitions $\mathcal{S}$ to consider per observation in a batch, as outlined in the more detailed Algorithm 3. The other hyperparameters and architecture of the FastSHAP model are given in Appendix A.
- Compute the KernelSHAP estimate (39) with the number of coalitions $|\mathcal{D}|$ equal to $50, 60, 70, \ldots, 1{,}000$ for the 100 test observations. We do not use paired sampling in KernelSHAP.[2]
- Compute the exact Shapley values (33) for the 100 test observations and use this as the ground truth to evaluate all the estimates against.

4.2 Evaluation Metric

To evaluate the accuracy of the estimation methods, an evaluation metric must be specified. Here, we specify both the evaluation metric that will be used to evaluate the contribution function estimates and the evaluation metric that will be used to evaluate the Shapley value estimates.

The mean absolute error (MAE) is used to evaluate the accuracy of the contribution function estimates. The MAE is computed across all 2^q coalitions $\mathcal{S}$ of the q features. For an observation $(\boldsymbol{x}^*, y^*)$, denote the vector containing the exact value of the contribution function for all 2^q feature coalitions $\mathcal{S}$ by $\boldsymbol{v}_{\boldsymbol{x}^*, y^*} = \left(v^*_0, \quad v^*_1, \quad \ldots, v^*_{2^q}\right)^\top \in \mathbb{R}^{2^q}$. Correspondingly, denote the vector containing the estimate thereof by $\hat{\boldsymbol{v}}_{\boldsymbol{x}^*, y^*} = \left(\hat{v}^*_0, \quad \hat{v}^*_1, \quad \ldots, \hat{v}^*_{2^q}\right)^\top \in \mathbb{R}^{2^q}$. Then, the MAE between the exact and estimated values is

$$\text{EM}_1 = \text{MAE}(\boldsymbol{v}_{\boldsymbol{x}^*, y^*}, \hat{\boldsymbol{v}}_{\boldsymbol{x}^*, y^*}) = \frac{1}{2^q} \sum_{s=1}^{2^q} |v^*_s - \hat{v}^*_s|. \tag{45}$$

The accuracy of the Shapley value estimators will be calculated using the mean Euclidean distance averaged over n test observations. Denote the exact Shapley values of the ith observation $(\boldsymbol{x}^*_i, y^*_i)$, $i \in \{1, 2, \ldots, n\}$ by $\boldsymbol{\phi}_{\boldsymbol{x}^*_i, y^*_i} = \left(\phi^*_{i1}, \quad \phi^*_{i2}, \quad \ldots, \quad \phi^*_{iq}\right)^\top \in \mathbb{R}^q$ and the estimated Shapley values by $\hat{\boldsymbol{\phi}}_{\boldsymbol{x}^*_i, y^*_i} = \left(\hat{\phi}^*_{i1}, \quad \hat{\phi}^*_{i2}, \quad \ldots, \quad \hat{\phi}^*_{iq}\right)^\top \in \mathbb{R}^q$. The Euclidean between the estimate and the ground

[2] In [4], paired sampling is found to improve both KernelSHAP and FastSHAP. However, the relative performance using paired sampling in both methods is similar to the performance when it is used in neither method. Therefore, we only provide results when it is not used.

truth for the ith observation is defined as $\ell_2(\boldsymbol{\phi}_{\boldsymbol{x}_i^*,y_i^*}, \hat{\boldsymbol{\phi}}_{\boldsymbol{x}_i^*,y_i^*}) = \frac{1}{q}\sqrt{\sum_{j=1}^{q}(\phi_{ij}^* - \hat{\phi}_{ij}^*)^2}$. Accordingly, we define the following metric across n test observations

$$\text{EM}_2 = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{q}\sqrt{\sum_{j=1}^{q}(\phi_{ij}^* - \hat{\phi}_{ij}^*)^2}. \tag{46}$$

4.3 Simulation Models

In the experiments, a tabular data set consisting of a set of features and a response variable is generated. Two different simulation models are considered for the features, which will be presented in the following paragraphs, and the response variable is calculated in the same way for both feature models. Because FastSHAP is proposed in the classification setting, the response variable in the simulated data set must be categorical. For $i = 1, 2, \ldots, n$, the response variable y_i for feature observation $\boldsymbol{x}_i \in \mathbb{R}^q$ is determined by using a latent variable z_i in the following way

$$z_i = e^{\beta_0 + \boldsymbol{\beta}^\top \boldsymbol{x}_i + \varepsilon_i}, \qquad y_i = \begin{cases} 1, & z_i \geq 0.5, \\ 0, & z_i < 0.5, \end{cases} \tag{47}$$

for a bias term β_0 and coefficients $\boldsymbol{\beta} = \begin{pmatrix}\beta_1, & \beta_2, & \ldots, & \beta_q\end{pmatrix}^\top$. A noise term $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$ is added to each latent variable z_i to replicate real-world data better since real data often arise from noisy measurements. The specific values of the coefficient $\beta_1, \beta_2, \ldots, \beta_q$ and the variance of the noise term σ^2 are changed in the experiments depending on both the parametric distribution of the features and the specific parameters of the distribution. This is because these choices affect the range of the feature values. The coefficients must be adjusted based on the range of the features to ensure that class 1 is the minority class when calculating the response variables by (47). In addition, the variance σ^2 must be adjusted based on the range of the feature values in order to ensure that the noise term is of a reasonable magnitude compared to the feature values. We specify the values of the coefficients in $\boldsymbol{\beta}$ and the variance of the noise terms σ^2 in the following sections for each simulation model. The bias term β_0 equals -1 in all the simulation models.

4.3.1 Simulation Model 1: The Multivariate Normal Distribution

In the first simulation model, the observations of the features $\boldsymbol{x} \in \mathbb{R}^q$ are sampled from a multivariate normal distribution with mean vector $\boldsymbol{\mu} \in \mathbb{R}^q$ and covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{q\times q}$. In the computation of the contribution function (29), the expectation is taken over the conditional probability densities $p(\boldsymbol{x}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$. In the case of the multivariate normal distribution, the conditional densities $p(\boldsymbol{x}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$ are analytically known, and are uniquely defined by its mean vector $\overline{\boldsymbol{\mu}}$ according to (22) and covariance matrix $\overline{\boldsymbol{\Sigma}}$ as in (23). Drawing Monte Carlo samples directly from $p(\boldsymbol{x}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}_{\mathcal{S}}^*)$ and taking the average of the prediction of the black box model evaluated at the Monte Carlo samples will give an unbiased estimate of the contribution function (29) regardless of the covariance matrix $\boldsymbol{\Sigma}$ used to sample the original feature observations. Both the surrogate model, given in Algorithm 4, and the off-manifold estimate (39) will be checked against the unbiased Monte Carlo estimate. Note that when providing the unbiased Monte Carlo estimate, the samples are drawn directly from the multivariate distribution, whereas in the off-manifold estimate, the Monte Carlo samples are drawn from the simulated training data set. The same data set is used in the off-manifold and surrogate methods to provide as similar conditions as possible.

We want to investigate how the degree of correlation affects the accuracy of the simulations. In particular, this is interesting because feature independence is assumed in the off-manifold estimate (39). Three different correlation matrices $\boldsymbol{R} \in \mathbb{R}^{q\times q}$ are used in the simulations of the feature observations. These are as follows.

1. The identity matrix, which corresponds to independent features. In this simulation, all the features have variance equal to 1 such that the correlation and covariance matrices are equal.

2. A correlation matrix where some of the covariates are positively correlated and some covariates are negatively correlated. The matrix is of the form

$$\boldsymbol{R}_{\text{some\_corr}} = \begin{pmatrix} 1 & 0.1 & 0.8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.1 & 1 & -0.1 & 0 & 0 & 0 & 0 & -0.9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.8 & -0.1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0.9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -0.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -0.5 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.9 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.9 & 0 & 0 & 0 & 0 & 1 & 0.4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{48}$$

The matrix $\boldsymbol{R}_{\text{some\_corr}}$ is of dimension 16×16, corresponding to $q = 16$ features. When the number of features q is less than 16, the submatrix corresponding to the first q features is used. In this simulation experiment, some of the features have variance different from 1, specifically, $\text{Var}(\boldsymbol{x}_2) = 1.44$, $\text{Var}(\boldsymbol{x}_4) = 0.64$ and $\text{Var}(\boldsymbol{x}_q) = 4, \forall q$. The relation between the correlation matrix $\boldsymbol{R}_{\text{some\_corr}}$ and the corresponding covariance matrix $\boldsymbol{\Sigma}_{\text{some\_corr}}$ is

$$\boldsymbol{R}_{\text{some\_corr}} = (\text{diag}(\boldsymbol{\Sigma}_{\text{some\_corr}}))^{-\frac{1}{2}} \boldsymbol{\Sigma}_{\text{some\_corr}} \text{diag}((\boldsymbol{\Sigma}_{\text{some\_corr}}))^{-\frac{1}{2}}, \tag{49}$$

where $\text{diag}(\boldsymbol{\Sigma}_{\text{some\_corr}})$ is a diagonal matrix with the variance of the features as elements along the diagonal. Moreover, by raising a matrix to a power, we mean that the operation is performed elementwise. Hence, the distribution is determined by the covariance matrix $\boldsymbol{\Sigma}_{\text{some\_corr}}$ using (48) and (49).

3. A correlation matrix where all the off-diagonal elements are 0.9 and the diagonal elements are 1 such that all the covariates are heavily correlated. Let this matrix be denoted $\boldsymbol{R}_{\text{high\_corr}}$. In this case, the variance of all the covariates is 1 in the simulation. Thus, the correlation and covariance matrices are equal such that the covariance matrix is $\boldsymbol{\Sigma}_{\text{high\_corr}} = \boldsymbol{R}_{\text{high\_corr}}$.

In all the simulations, the mean vector $\boldsymbol{\mu} \in \mathbb{R}^q$ is the vector with all elements equal to 0.

When simulating from the multivariate normal distribution, the coefficients $\boldsymbol{\beta}$ in the expression the response variable is determined by (47) are as follows

$$\begin{aligned} &\begin{pmatrix} \beta_1, & \beta_2, & \ldots, & \beta_q \end{pmatrix}^\top \\ &= \begin{pmatrix} 2, -1.5, 1, -2.3, -1.3, 0.4, 0.7, -0.3, 0.9, -0.1, 0.25, 0.3, -0.1, 2, -1.5, -0.1, 0.45 \end{pmatrix}^\top. \end{aligned}$$

The variance of the noise term in (47) is $\sigma^2_{\text{indep}} = 1$, $\sigma^2_{\text{some\_corr}} = 4$ and $\sigma^2_{\text{some\_corr}} = 1$ for the case of independent, somewhat correlated and heavily correlated features, respectively.

4.3.2 Simulation Model 2: The Multivariate Burr Distribution

The second simulation model is similar to the previous, except that the features now follow the multivariate Burr distribution (24), which also has analytically known conditional probability densities $p(\boldsymbol{x}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})$ with parameters defined in (25). The parameter ζ in the multivariate Burr probability density function (24) affects the degree of linear correlation between the variables. As $\zeta \to \infty$, the correlation between the features goes towards zero, whereas for low values of ζ, the covariates will tend to be heavily correlated. Therefore, to compare the performance of the methods in situations where the degree of correlation between the features varies, two values of ζ are considered. These values are ζ equal to 2 and 7, where the features

have a relatively high correlation for $\zeta = 2$ and a relatively low correlation for $\zeta = 7$. For the other parameters in the Burr distribution (24), the values are kept constant in all simulations. The values of the parameters are $b_1, b_2, \dots b_q = 2, 4, 6, 2, 4, 6, 2, 4, 6, 6, 4, 2, 6, 4, 2, 6$ and $r_1, r_2, \dots, r_q = 1, 3, 5, 1, 3, 5, 1, 3, 5, 5, 3, 1, 5, 3, 1, 5$. If $q < 16$, the first q values are used. The value of ζ also affects the tails of the Burr distribution. For lower values of ζ, the tails are heavier, corresponding to more extreme "outlier" observations, which generally means that the simulated data is more difficult to learn. For higher values of ζ the tails are less heavy, and the data is relatively easier to learn. It is interesting to investigate how the estimators perform in situations where the "difficultness" of the data varies. Generally, the data generated from the Burr distribution will be more difficult to learn than data generated from the normal distribution.

For the case with $\zeta = 2$, the coefficients $\boldsymbol{\beta}$ in the expression determining the response variables (47) are

$$\begin{aligned}&\big(\beta_1, \quad \beta_2, \quad \dots, \quad \beta_q\big)^\top \\ &= \big(3, -1.5, 1, -1.7, -1.6, -1, 1.7, -2, 0.9, -0.1, -1.9, 1.3, -0.1, 1.8, -1.5, -0.1\big)^\top,\end{aligned}$$

and the variance of the noise term σ^2 in (47) is equal to 0.75^2. In the case with $\zeta = 7$, the coefficients $\boldsymbol{\beta}$ are

$$\begin{aligned}&\big(\beta_1, \quad \beta_2, \quad \dots, \quad \beta_q\big)^\top \\ &= \big(0.1, -1.5, 1.7, -1.7, 1., 2, 0.7, -2, 0.9, -0.1, 0.25, -1.5, -0.1, 2, -1.5, -0.1\big)^\top,\end{aligned}$$

and the variance of the noise term σ^2 equals 0.5^2.

4.4 Evaluating the Contribution Function Estimators

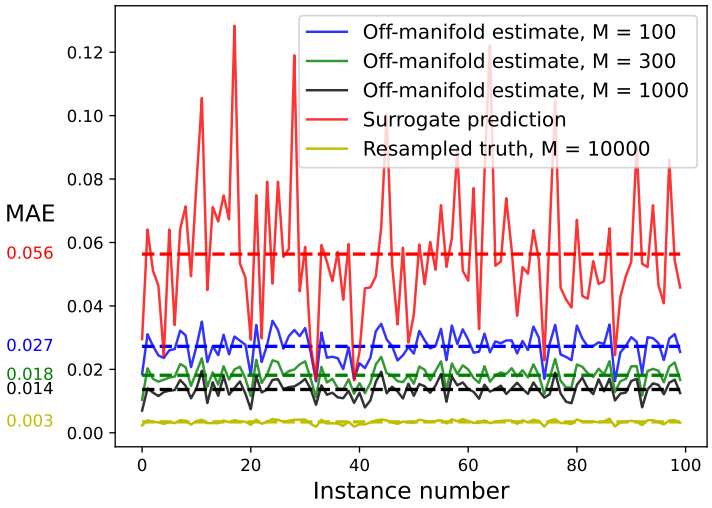
In this section, the accuracy of the off-manifold estimate (39) and surrogate model, trained according to Algorithm 4, are evaluated for the two different simulation models presented in Section 4.3. Firstly, the results when the features follow a multivariate normal distribution are presented in Section 4.4.1. Then, the results from the second simulation model, where the features are multivariate Burr distributed, are presented in Section 4.4.2.

4.4.1 Simulation Model 1: Multivariate Normally Distributed Features

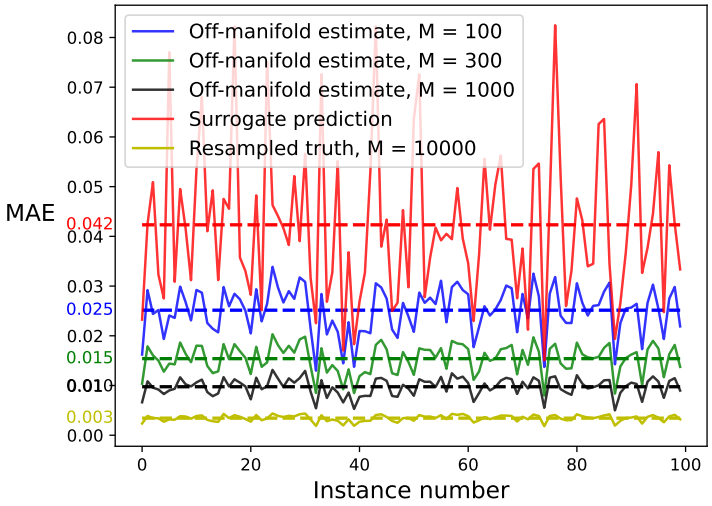
Independent Features

In this paragraph, the accuracy of the contribution function estimates for independent features drawn from the multivariate normal distribution is investigated. The experiment investigates the effect of varying the size of the training data set n_{train}. For each value of n_{train}, the surrogate model is trained, and the MAE (45) of its predictions for all possible feature coalitions $\mathcal{S}$ is calculated for 100 test observations is calculated. Moreover, for each value of n_{train}, the off-manifold estimate is calculated for all feature coalitions $\mathcal{S}$ for the same 100 test observations for three different values of the number of Monte Carlo samples; $M = 100, 300$ and 1,000. The MAE of the three off-manifold estimates is calculated and plotted in the same plot as the MAE of the surrogate model's predictions for each value of n_{train}. Because an unbiased Monte Carlo integral with samples drawn directly from the conditional distributions $p(\boldsymbol{x}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})$ is used as the true value of the contribution function, it is investigated how large the variations that are caused by resampling the truth are, compared to the MAE of the off-manifold and surrogate model estimates.

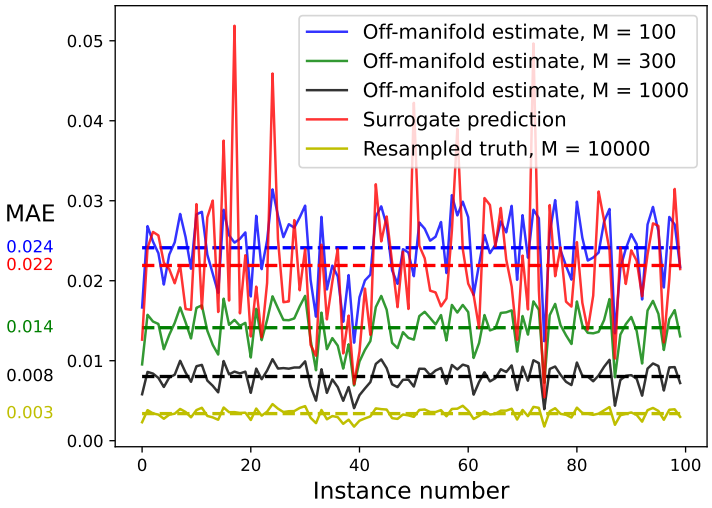
The results of the experiment are shown for $n_{\text{train}} = 1{,}280$, 6,400 and 64,000 in Figures 5a, 5b and 5c, respectively. As shown in Figures 5a and 5b, respectively, when 1,280 and 6,400 training observations are used, the MAE of the surrogate model is the highest with a significant margin. A notable difference between the smaller data sets and when $n_{\text{train}} = 64{,}000$ is that the surrogate model has higher accuracy than the off-manifold estimate using $M = 100$ Monte Carlo samples when the larger training data set is used, as shown in Figure 5c. Having a larger training data set



(a) $n_{\text{train}} = 1{,}280$.



(b) $n_{\text{train}} = 6{,}400$.



(c) $n_{\text{train}} = 64{,}000$.

Figure 5: **Independent Multivariate Normally Distributed Features**.
In the plot, MAEs resulting from the different estimates are shown in different colors. In dashed lines, the mean MAE of each estimation method is plotted in the same color as the MAE of the estimation method. Next to the dashed line, on the y-axis, the value of the mean MAE of each method is written. The "method" with the lowest MAE is the other realization of the estimate of the true value.

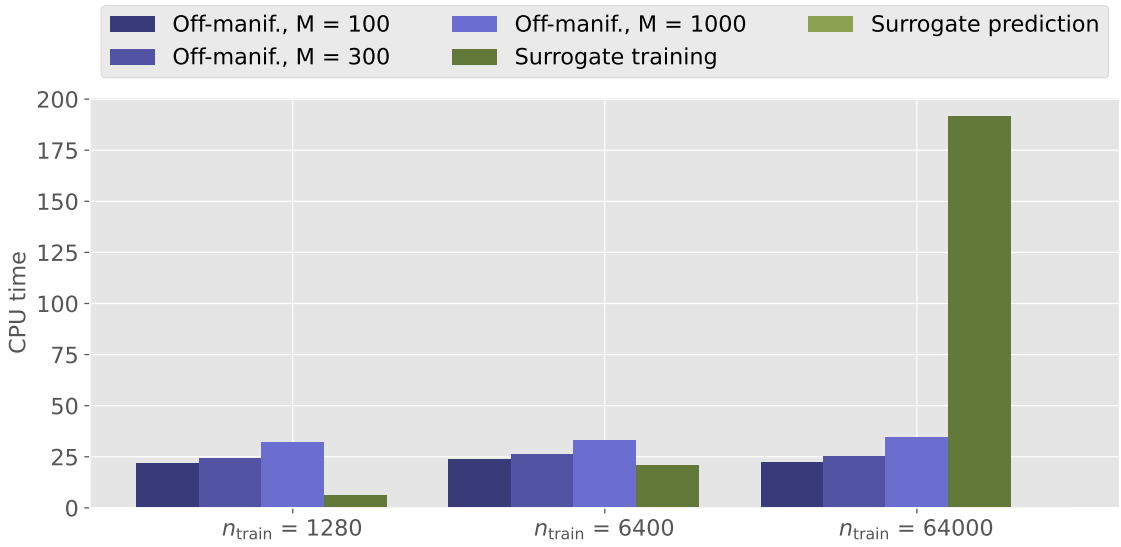


Figure 6: **Independent Multivariate Normally Distributed Features.**
The CPU time of computing the contribution function estimates is shown in the bar chart.

increases the accuracy of the surrogate model's predictions from an average MAE of 0.056 when $n_{\text{train}} = 1{,}280$ and 0.042 with 6,400 training observations to 0.022 with 64,000 training observations. Thus, this example shows that the accuracy of the surrogate model increases when the size of the training data set increases. This is a sensible result since, in general, having a larger training data set will result in a machine learning model that has better predictive abilities.

Moreover, the variability that is observed when resampling the true value in all plots in Figure 5 is much lower than the error in the estimates. Recall from Section 2.4 that the variance of the Monte Carlo estimate decreases as a function of increasing the number of samples M in the estimate, and for an unbiased estimate, the error decreases as in (27). Since the number of samples used to sample the ground truth is very high, $M = 10{,}000$, this explains the low variability. Clearly, the large error in the surrogate model's predictions is not due to noise arising from sampling the true value. The figure shows that for all three values of n_{train}, the MAE of the off-manifold estimate decreases as the number of Monte Carlo samples increases. In this example, the features are truly independent, and the "off-manifold" estimate is not truly off the manifold. Thus, the off-manifold estimate is equivalent to the sampling scheme that provides the true value of the contribution function. The only difference is that in the off-manifold estimate, the Monte Carlo samples are drawn from the training data set, whereas, in the estimate of the true value, the Monte Carlo samples are drawn directly from a multivariate normal distribution with mean and covariance matrix computed according to the formulas for the conditional multivariate normal distribution, (22) and (23), respectively. This explains the low error in the off-manifold estimates for higher values of M.

In addition to creating the plots of the MAEs shown in Figure 5, the CPU time of the computation of each estimate has been recorded. The CPU time was briefly described in Section 3.7. For the off-manifold estimate with $M = 100, 300$ and 1,000 Monte Carlo samples, the average CPU time of computing the off-manifold estimate (39) for all feature coalitions $\mathcal{S} \in \mathcal{Q}$, where $\mathcal{Q}$ is the set containing all 2^q feature coalitions, is reported. The average is taken across 100 test observations. For the surrogate model, recall that we split the process into two phases, as described in Section 3.7. The first phase is the training of the surrogate model. This initial cost is only performed once per black box model and is amortized across the instances to be explained. The second phase is to generate the estimate of the surrogate model for all feature coalitions $\mathcal{S} \in \mathcal{Q}$ for each new instance to be explained. This corresponds to evaluating the surrogate model 2^q times. We report the average CPU time of evaluating the surrogate models for all $\mathcal{S} \in \mathcal{Q}$, where the average is taken across 100 test observations. We refer to the first phase as "surrogate training" and the second phase as "surrogate prediction".

The corresponding CPU time of the experiment is shown in Figure 6. The figure shows that the CPU time of computing the off-manifold estimate increases as a function of the number of Monte Carlo samples M. However, as shown in Figure 5, the estimate's accuracy also increases. Therefore, the increased accuracy must be weighed against the increase in computation time. Note also that the CPU time of the off-manifold estimate is approximately unaffected by varying values of n_{train}. Moreover, for n_{train} equal to 1,280 and 6,400, the training of the surrogate model is faster than computing a single off-manifold estimate for all values of M. When n_{train} equals 64,000, the training of the surrogate model is significantly higher than computing an off-manifold estimate for all values of M. However, if more than around ten instances are to be explained, the computation time of training the surrogate model will be lower than computing the off-manifold estimate for all the instances. As the figure shows, the surrogate prediction time, which is so small that it is not visible in the bar chart, is negligible compared to the computation time of the off-manifold estimate and the training of the surrogate model. Thus, using the trained surrogate model, the values of the contribution function for all feature coalitions $\mathcal{S} \in \mathcal{Q}$ can be computed almost instantaneously. In summary, the surrogate model is faster when explaining more than a few instances. In many practical applications, it is desirable to be able to provide new explanations instantaneously after a model is put into a production system. Therefore, it is more important to be able to provide explanations instantaneously than to have a low initialization cost. In these situations especially, it is natural to prefer the surrogate model, although one has to weigh the lower computation time against a higher error in some cases.

Somewhat Correlated Features

The same experiments that were performed for independent features are repeated for the case when the features are simulated from a multivariate normal distribution with the covariance matrix $\mathbf{\Sigma}_{\text{some\_corr}}$ defined in (49). The MAE (45) of each estimate is shown for the three values of n_{train} in Figure 7. First, notice that the mean MAE of the surrogate estimate decreases as a function of an increasing number of training observations n_{train}. It is 0.088, 0.063, and 0.032 for $n_{\text{train}} = 1{,}280, 6{,}400$ and 64,000 as shown in Figures 7a, 7b and 7c, respectively. Unlike for the case with independent features, the mean MAE of the surrogate estimate is lower than the mean MAE of the off-manifold estimates for all values of n_{train} and all values of the number of Monte Carlo samples M. The error arising from resampling the truth is much smaller than the error in all estimates, and therefore is not the cause of the estimate's error.

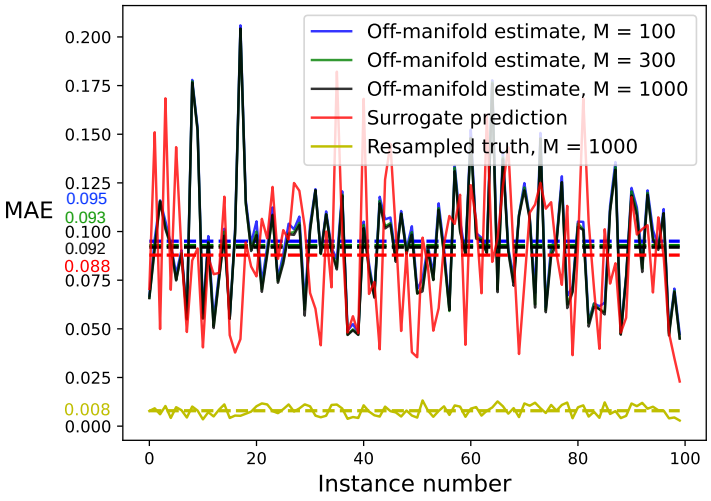
The figure shows that the MAE of the off-manifold estimate for all three values of M is approximately equal for all the values of n_{train}. The bias in the estimate can explain this. The bias is caused by falsely assuming feature independence. Recall that the standard deviation in the Monte Carlo integral grows as $\mathcal{O}(M^{-\frac{1}{2}})$ according to (27). Therefore, increasing M reduces the variance of the Monte Carlo estimate. However, since the estimate is biased, the bias term remains, and increasing M does not reduce the mean MAE of the estimate below a certain threshold value.

The results in Figure 7 demonstrate that the surrogate model can outperform the off-manifold estimate in settings with some correlation between the features. Thus, based on this example, it seems reasonable that in real-world problems, where feature independence is rare, the false assumption of feature independence is severe in terms of the accuracy of the resulting estimate of the contribution function. Based on the accuracy of the estimates, for this example, the estimate of the surrogate model should be preferred. The CPU times for this experiment are of similar magnitudes as for the case with independent features shown in Figure 6, and therefore, omitted here. However, the surrogate model should also be preferred based on the CPU time as long as more than a few instances are to be explained.

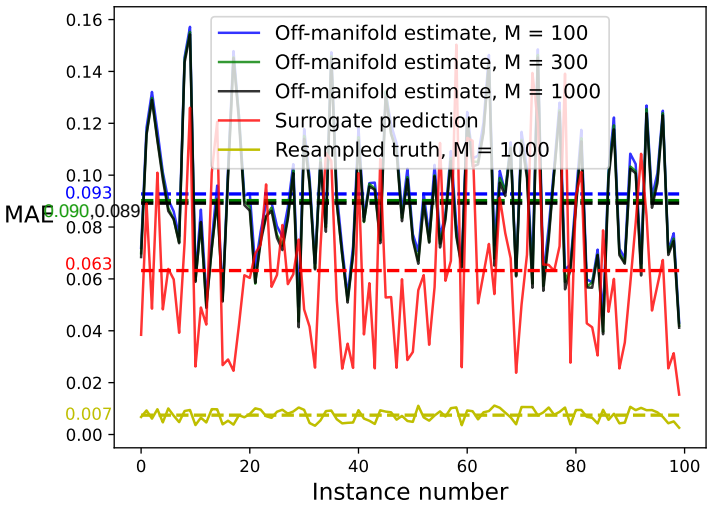
Highly Correlated Features

The last covariance matrix that is used is $\mathbf{\Sigma}_{\text{high\_corr}}$ with diagonal elements of 1 and off-diagonal elements of 0.9. Such a correlation matrix is not common in real-world problems, however, high correlation between pairs of features can occur. Therefore, it serves as a demonstration of how the methods perform in cases with a very high correlation between the features.

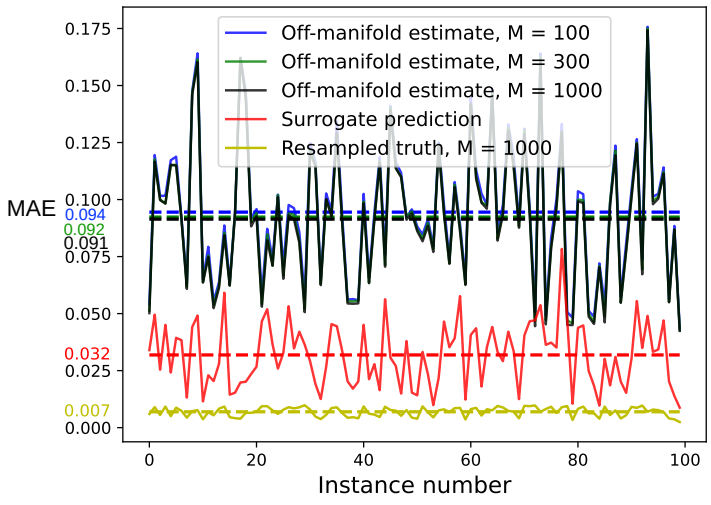
The results with this correlation matrix are shown in Figure 8. Like for the case with somewhat correlated features shown in Figure 7, the surrogate model outperforms the off-manifold method for



(a) $n_{\text{train}} = 1{,}280$.

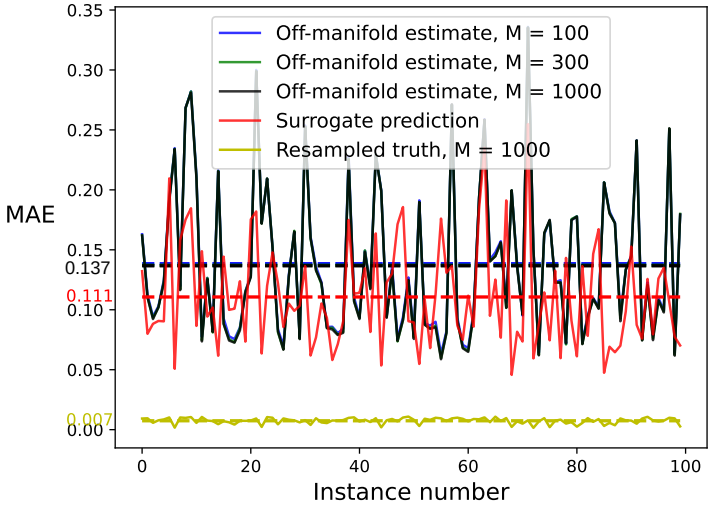


(b) $n_{\text{train}} = 6{,}400$.

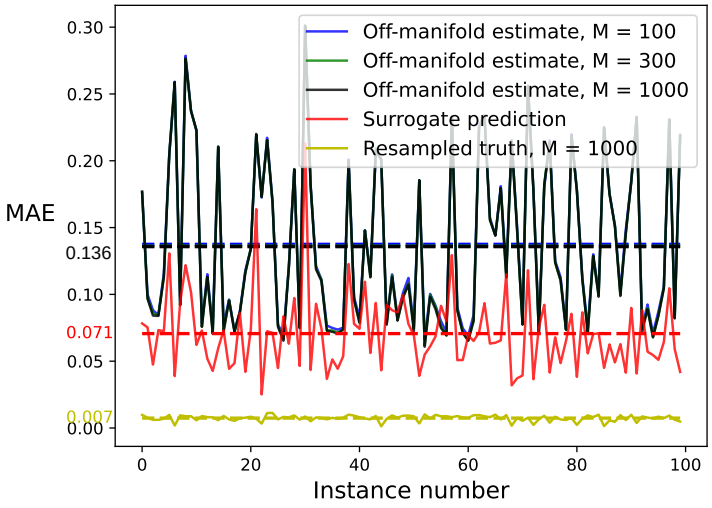


(c) $n_{\text{train}} = 64{,}000$.

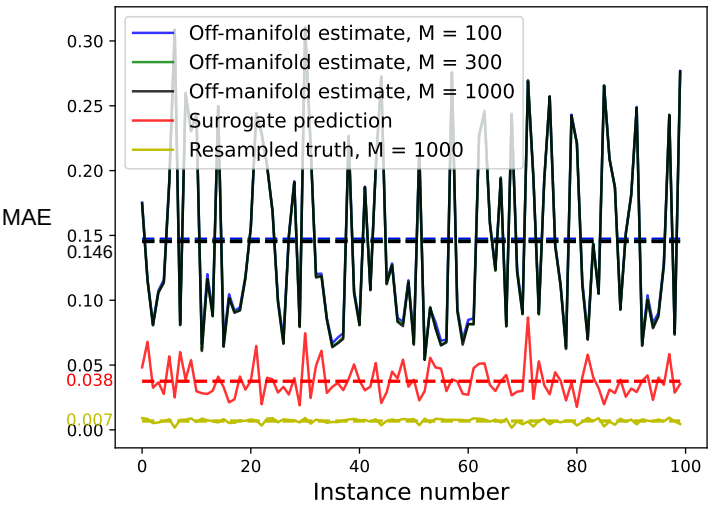
Figure 7: **Somewhat Correlated Multivariate Normally Distributed Features**. The MAE of the surrogate and the three off-manifold estimates using different numbers of Monte Carlo samples are plotted in different colors. In dashed lines, the mean MAE of each estimation method is plotted in the same color as the MAE of the estimation method. Next to the dashed line, on the y-axis, the value of the mean MAE of each method is written. In each example, the ground truth has been resampled and the MAE between the two realizations is plotted. The resampled truth is plotted in yellow and is the "method" with the lowest MAE for all values of n_{train}.



(a) $n_{\text{train}} = 1{,}280$.



(b) $n_{\text{train}} = 6{,}400$.



(c) $n_{\text{train}} = 64{,}000$.

Figure 8: **Highly Correlated Multivariate Normally Distributed Features**.
In the plot, the MAEs of the contribution function estimates are plotted in different colors. In dashed lines, the mean MAE of each estimation method is plotted in the same color as the MAE of the estimation method, with the value of the mean MAE of each method written next to it. In this simulation example, all three off-manifold estimates are nearly identical, and therefore, the graphs of their MAEs are overlapping. The "method" with the lowest MAE is the resampled ground truth.

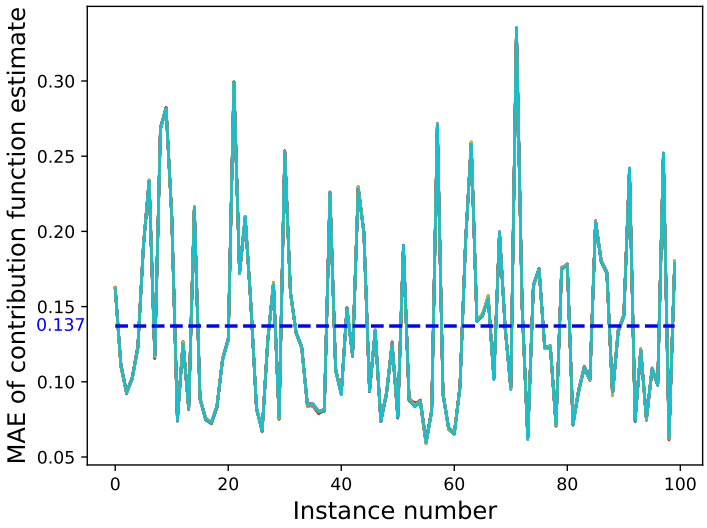
all combinations of the number of training observations n_{train} and the number of Monte Carlo M. Moreover, again like for the case with somewhat correlated features, the MAE of the off-manifold estimates is almost constant for all values of M, which like before, can be explained by the bias in the estimate. The off-manifold estimates are so similar that they overlap in the plots. Notice that the mean MAE of the off-manifold estimate is even higher relative to the surrogate estimate when the features are heavily correlated than when they are somewhat correlated. In this example as well, the variation in the estimate of the true value of the contribution function is small. It can not explain the relatively much higher error in all the estimates. Also, in this case, the CPU times are of similar magnitude as for the case with independent features, and therefore, omitted here. In summary, for the examples with normally distributed features, it is clear that the surrogate model outperforms the off-manifold method when the features are correlated, especially for large training data sets.

Other Results

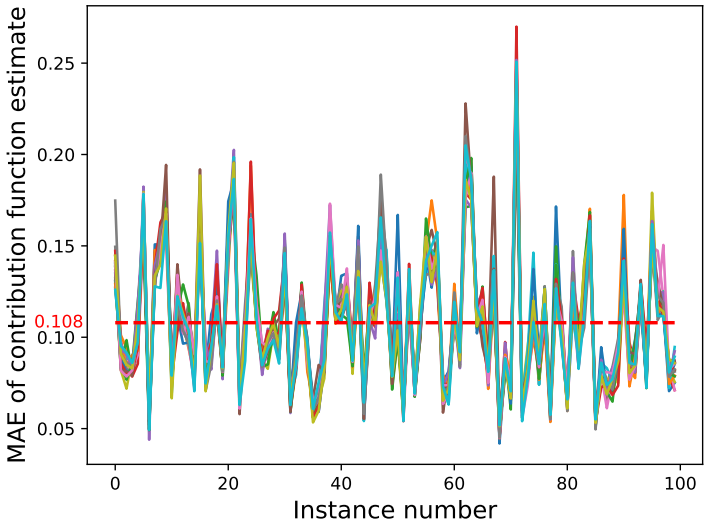
In Figures 5, 7 and 8, the MAE of all estimates is spiky. This means that the MAE in the estimate of the contribution function is higher for some instances and lower for others. It is, therefore, interesting to investigate why this is the case. For the off-manifold estimates, this could be caused by randomly drawing Monte Carlo samples from the training data set. Therefore, the computation of the off-manifold estimates is repeated several times with different seeds in order to investigate if this could explain the large variation in the MAE of the off-manifold estimates for different instances. In the surrogate model, the random seed can affect e.g. the initial distribution of the weights in the training of the neural network and can therefore affect the performance of the resulting model. Therefore, the surrogate model is retrained and evaluated for the same instances using different seeds to investigate the stability of the model.

Hence, in order to investigate the effect of randomly drawing Monte Carlo samples in the off-manifold estimate and the effect of randomness in the surrogate model, we reestimate both of them several times. This is done for $n_{\text{train}} = 1{,}280$ and $6{,}400$ training observations and the off-manifold estimate is calculated using $M = 300$ Monte Carlo samples. The truth is sampled with 1,000 Monte Carlo samples. This is considered sufficient since the MAE is significantly larger based on our previously described experiments. For $n_{\text{train}} = 1{,}280$, the covariance matrix $\boldsymbol{\Sigma}_{\text{high\_corr}}$ is used to simulate the features, and the MAEs of all the reruns are shown in Figures 9a and 9b for the off-manifold and surrogate estimate, respectively. In both cases, the MAEs resulting from the different reruns are so similar that the graph of the MAEs overlaps. The fluctuations arising from re-estimation with another seed are relatively small compared to the spikes in the MAE. Hence, it does not explain why some test observations have a higher error in the estimate of the contribution function. The same experiments have been performed for $n_{\text{train}} = 6{,}400$ with somewhat correlated features, the results of which are shown in Figures 9c and 9d for the off-manifold and surrogate estimates, respectively. As before, the MAEs resulting from the reruns are so similar that their graphs overlap, and the fluctuations arising from resampling do not explain the large differences in MAE between instances.

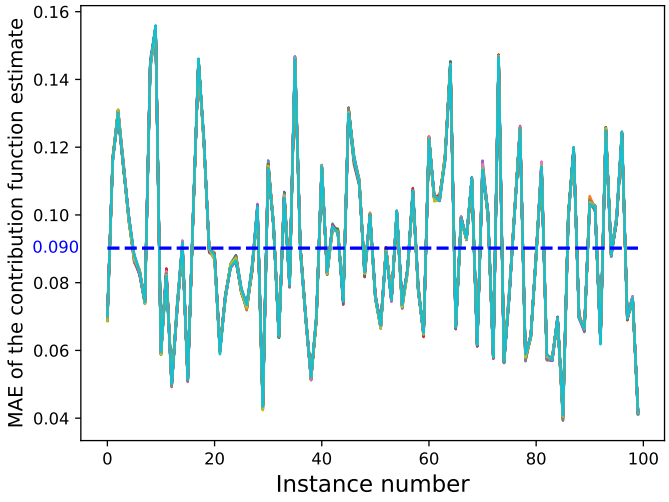
Another hypothesis for why the estimates are higher for some instances than others is that the observations with a higher error are outliers. The Mahalanobis distance [37] measures the distance from an observation to the center of a normal distribution and thereby detects outliers. The observations that are very far from the center are likely in the regions where there are few training observations, and therefore, the black box model has not been trained on such observations. However, we did not find a correspondence between the Mahalanobis distance to the center and the observations with a high error. In our experiments, the off-manifold estimate and surrogate model can have a high error for different observations. This indicates that the methods fail in different scenarios. For the off-manifold estimate, it is plausible that although an observation is not an outlier, the Monte Carlo samples used in the off-manifold estimate are outliers since the samples might consist of combinations of feature values that have never occurred in the training data. If so, the off-manifold estimate will rely on evaluating the black box model outside the region it has been trained on. Frye et al. [3] refer to this problem as "garbage-in-garbage-out", meaning that if the samples are unrepresentative of the training data and true conditional distributions, the predictions of the black box model for these observations are invalid. This might lead to a high estimation error.



(a) Off-manifold, $n_{\text{train}} = 1{,}280$, $\mathbf{\Sigma}_{\text{high\_corr}}$.

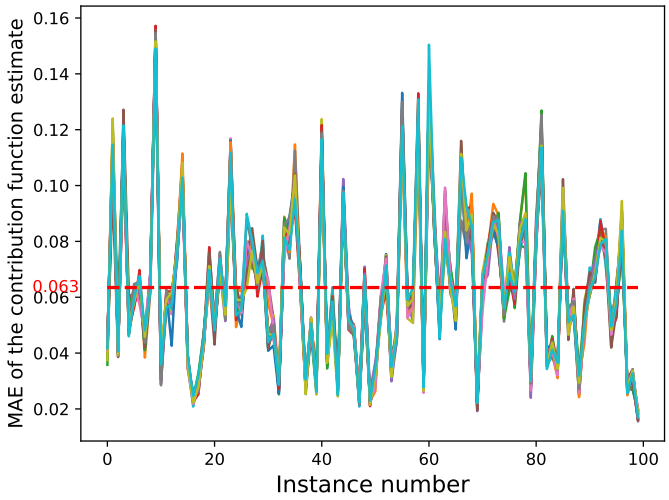


(b) Surrogate, $n_{\text{train}} = 1{,}280$, $\mathbf{\Sigma}_{\text{high\_corr}}$.



(c) Off-manifold, $n_{\text{train}} = 6{,}400$, $\mathbf{\Sigma}_{\text{some\_corr}}$.

Figure 9



(d) Surrogate, $n_{\text{train}} = 6{,}400$, $\boldsymbol{\Sigma}_{\text{some\_corr}}$.

Figure 9: (Continued) In the plot, the MAEs resulting from reestimating the contribution function with both the off-manifold and surrogate method for 100 test observations with a new seed in each run are plotted in different colors. In total 20 different seeds are used per example. In the caption of each plot, the details of each simulation are specified.

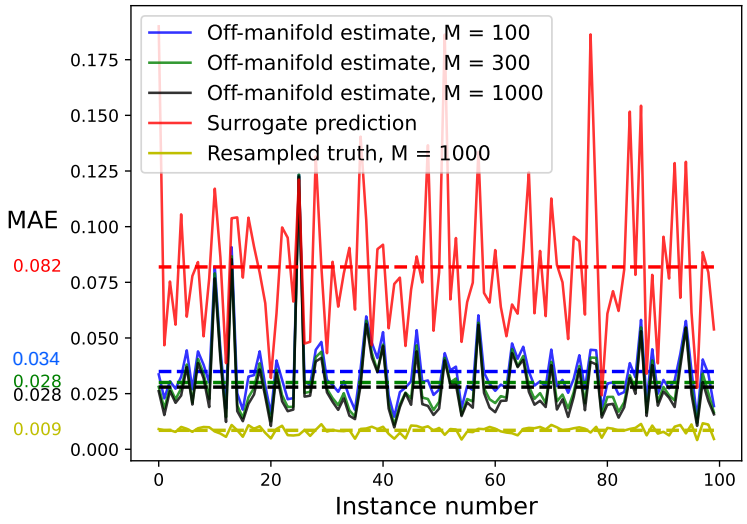
4.4.2 Simulation Model 2: Multivariate Burr Distributed Features

In this section, the results of performing similar experiments to those presented in Section 4.4.1, this time using the multivariate Burr distribution (24) are presented. The simulations of the features are generated as described in Section 4.3.2. As before, to evaluate the accuracy of the contribution function estimate, the MAE over all possible coalitions $\mathcal{S}$ of the q features is calculated. In total there are $2^q = 2^{10} = 1024$ coalitions of the $q = 10$ features. In all experiments, the true value of the contribution function (29) is calculated by Monte Carlo integration (26) sampling directly from the conditional distributions $p(\boldsymbol{x}|\boldsymbol{x}_{\mathcal{S}} = \boldsymbol{x}^*_{\mathcal{S}})$ with parameters as in (25) which yields an unbiased estimate.

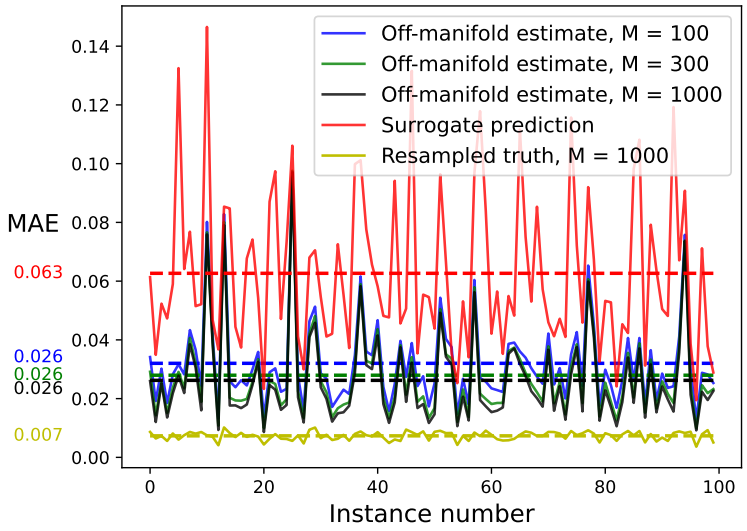
For the parameter ζ equal to 2 and 7, the off-manifold estimate is computed for $M = 100, 300$ and 1,000 Monte Carlo samples. In addition, the surrogate model is trained and used to predict the values of the contribution function. All simulations are repeated for $n_{\text{train}} = 1{,}280$, 6,400, 64,000, and 128,000 training observations to investigate how the size of the training data set affects the estimates. Since data generated from the Burr distribution generally is more challenging to learn than data from the normal distribution, we also consider a larger training data set than in the cases with the normal distribution. For each case, the average empirical correlation between the features, which is denoted $\hat{\rho}_{\text{avg}}$, in the simulated training data set is calculated. Lastly, for each combination of n_{train} and ζ, the truth is resampled and the MAE between the two realizations are plotted to investigate how large the error arising from this is compared to the error in the estimates. In both realizations, the truth is estimated with $M = 1{,}000$ Monte Carlo samples. The results of the simulations are shown in Figures 10 and 11 for $\zeta = 7$ and $\zeta = 2$, respectively.

Lower Correlation and Lighter Tails

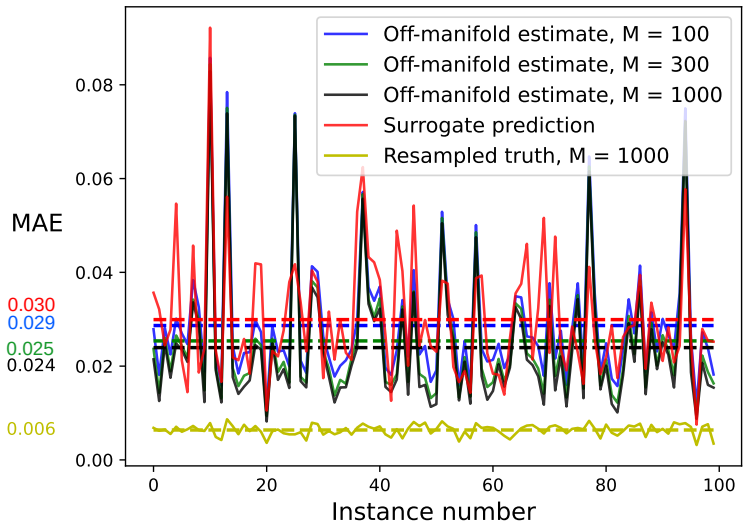
In Figure 10, the simulation results for $\zeta = 7$ are shown. This corresponds to an average empirical linear correlation of between 0.11 and 0.12 between the features in the training data set. The correlation between all feature pairs is approximately equal to the average correlation. Notice that for $n_{\text{train}} = 1{,}280$, 6,400 and 64,000, the off-manifold method outperforms the surrogate model for all values of the number of Monte Carlo M, as shown in Figures 10a, 10b and 10c, respectively. For $n_{\text{train}} = 1{,}280$ and 6,400, the surrogate model performs significantly worse than the off-manifold method for all values of M. However, when $n_{\text{train}} = 64{,}000$, the mean MAE of the surrogate estimate is almost the same as the off-manifold estimate with $M = 100$ Monte Carlo samples. For the largest training data set with $n_{\text{train}} = 128{,}000$, the surrogate model outperforms the off-



(a) $\hat{\rho}_{\text{avg}} \approx 0.12, n_{\text{train}} = 1{,}280$.

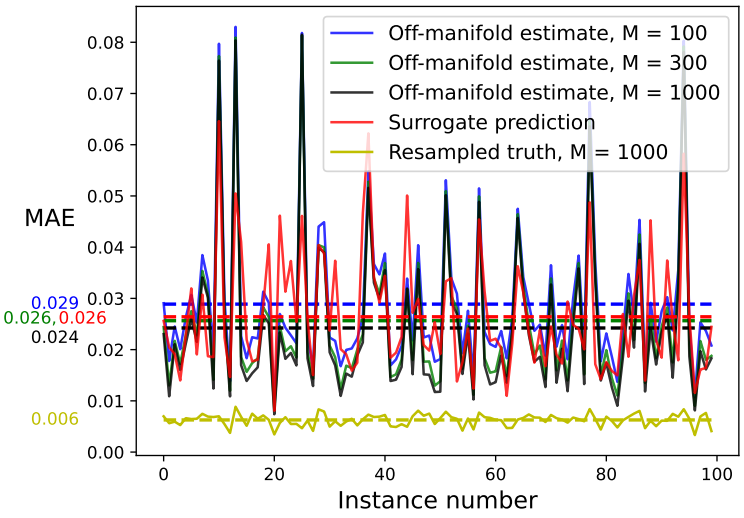


(b) $\hat{\rho}_{\text{avg}} \approx 0.11, n_{\text{train}} = 6{,}400$.



(c) $\hat{\rho}_{\text{avg}} \approx 0.11, n_{\text{train}} = 64{,}000$.

Figure 10: **Burr Distributed Features with $\zeta = 7$.**



(d) $\hat{\rho}_{\text{avg}} \approx 0.11, n_{\text{train}} = 128{,}000$.

Figure 10: (Continued) **Burr Distributed Features with** $\zeta = 7$.
In the plot, MAEs resulting from the different estimates are shown in different colors. In dashed lines, the mean MAE of each estimation method is plotted in the same color as the MAE of the corresponding estimation method. Next to the dashed line, on the y-axis, the value of the mean MAE of each method is written. The "method" with the lowest MAE is the other realization of the estimate of the true value. Each plot is captioned with the empirical average correlation $\hat{\rho}_{\text{avg}}$ in the simulated training data set and the number of training observations n_{train}.

manifold estimate with $M = 100$ Monte Carlo Samples, as shown in Figure 10d. Moreover, the average MAE of the surrogate model is, in this case, only slightly higher than that of the off-manifold estimate with $M = 300$ Monte Carlo samples. These results show that as the number of training samples n_{train} increases, the surrogate estimate obtains a lower average MAE. Notice also that in all cases, the off-manifold estimates for $M = 300$ and 1,000 have circa the same mean MAE, whereas the mean MAE of the off-manifold estimate with $M = 100$ is somewhat larger. Therefore, in this context, it seems like it is necessary to use at least $M = 300$ Monte Carlo samples in order to obtain the best possible estimate with the off-manifold estimate.

As the plots show, the mean MAE of the off-manifold estimate stops improving significantly when increasing the number of Monte Carlo samples from 300 to 1,000. This can be explained by the bias in the estimate caused by the false assumption of feature independence. Therefore, further increasing the number of Monte Carlo samples in the off-manifold estimate will likely not improve the mean MAE in the estimate. This is consistent with the results from the simulations with the multivariate normal distribution using the correlation matrices $\mathbf{\Sigma}_{\text{some\_corr}}$ and $\mathbf{\Sigma}_{\text{high\_corr}}$ shown in Figures 7 and 8, respectively. When there is some correlation between the features, the mean MAE off-manifold estimate does not seem to improve by increasing the number of Monte Carlo samples beyond a certain threshold value. There seems to be a value that is the lowest possible attainable mean MAE in the off-manifold estimate, which is explained by the bias in the estimate.

The surrogate model, on the other hand, can improve further if there are sufficiently many training observations. In practice, the number of training observations can usually not be increased. However, it illustrates how the methods perform based on the properties of the data set. It is also reasonable to believe that by tuning the hyperparameters of the surrogate model, both by tuning the hyperparameters in the neural network and the hyperparameters specific to the surrogate model, the surrogate model can yield more accurate estimates. In the simulations with somewhat correlated features in the multivariate normal distribution, the surrogate model outperformed the off-manifold estimates as shown in Figure 7. Whereas for this example with the Burr distribution, the off-manifold estimate outperformed the surrogate model. As discussed in Section 2.3.2, the multivariate Burr distribution has several properties that cause the data simulated from it to be harder to learn than data simulated from the multivariate normal distribution. This can explain

why the surrogate model outperforms the off-manifold method when there is only some correlation between the features in the case of multivariate normally distributed features, but not when they are multivariate Burr distributed.

Higher Correlation and Heavier Tails

In Figure 11, the simulation results when using the multivariate Burr distribution with the parameter $\zeta = 2$ corresponding to an average empirical linear correlation of the features between 0.37 and 0.38 is shown. In these simulations, it is the surrogate model that outperforms the off-manifold method for all values of the number of training observations n_{train} and all values of the number of Monte Carlo samples M, although only slightly for $n_{\text{train}} = 1{,}280$ as shown in Figure 11a. This shows that when the correlation between the features is relatively high, the false assumption of feature independence made in the off-manifold estimate has a more severe impact on the mean MAE of the method. Moreover, in this case, the off-manifold estimates are almost identical regardless of the number of Monte Carlo samples M. The bias term in the MAE of the method caused by falsely assuming feature independence cannot be reduced by increasing the number of Monte Carlo samples. It should also be noted that when $\zeta = 2$ in the multivariate Burr distribution, the distribution's tails will be heavier than when $\zeta = 7$ while keeping all other parameters in the distribution constant. When the tails are heavier, there will be more extreme "outlier" observations, and the data will be harder to learn. Therefore, the data used in the simulations with results shown in Figure 11 is harder to learn than the data used to get the results shown in Figure 10. Thus, this example shows that even when the data arise from a heavy-tailed distribution, the surrogate model can perform well compared to the off-manifold method, also when there are few training observations, as long as the average correlation between the features is sufficiently high.

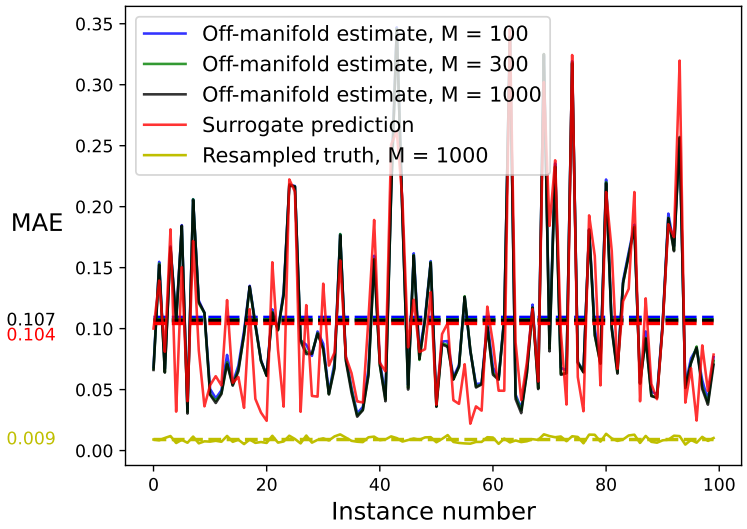
4.5 Evaluating the Shapley Value Estimators

In this section, the KernelSHAP and FastSHAP Shapley value estimators will be evaluated. For details on the methods, see Sections 3.3 and 3.4, respectively. Jethani et al. [4] find that the relative accuracy of FastSHAP compared to other methods, including KernelSHAP, is similar regardless of the choice of contribution function estimator. In their experiments, they use the same contribution function estimate in the ground truth Shapley values and all Shapley value estimates to distinguish the error arising in each step. We adopt this methodology in the experiments in this section, and therefore, the estimate of the contribution function will be kept constant in the computation of the KernelSHAP estimate (34), the training of the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ and the computation of the exact Shapley values (33). Because the computation time of the off-manifold estimate is high, the surrogate model trained according to Algorithm 4 is used in all experiments.

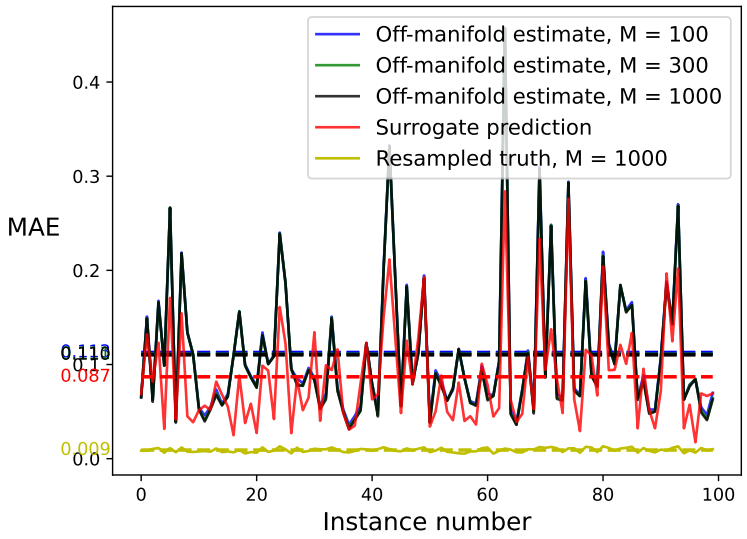
To be clear, in this section, what we use as the ground truth Shapley values, which we refer to as the "exact Shapley values", is the exact solution of the weighted least squares problem (33) given the surrogate model's predictions as the values of the contribution function. To compute (33), the values of the contribution function, which in the experiments are the surrogate model's predictions, must be given for all 2^q feature coalition. Throughout this section, it is assumed that this is given. Moreover, what we refer to as KernelSHAP, is the KernelSHAP estimate (34), also using the surrogate model's estimate of the contribution function. Lastly, we refer to the FastSHAP estimate as the predictions of the model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ trained with the surrogate model as the input in Algorithm 3 of the contribution function estimator.

In the simulations, we will vary the number of features in the model q and the number of observations in the training data set n_{train}. Because this changes the training data set, we have to train a new black box model and surrogate model for each combination of q and n_{train}. This should be kept in mind when the simulation results are compared between the training data sets.

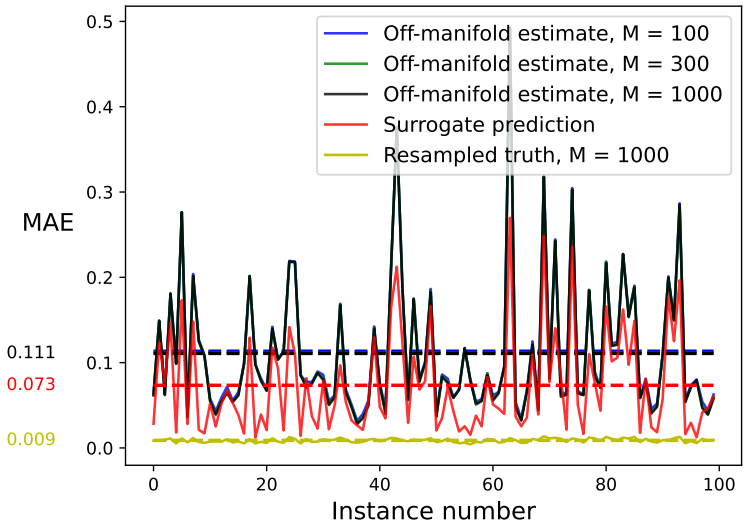
As before, the simulations are repeated for three sizes of the training data set, these are $n_{\text{train}} = 1{,}280$, 6,400, and 64,000. Moreover, for each value of n_{train}, the experiments are repeated for the number of features q equal to 10, 11, ..., 16. The KernelSHAP estimate (34) is computed with the number of feature coalitions $|\mathcal{D}|$ equal to 100, 300, 500 and 1,000. Moreover, the FastSHAP model is trained with the hyperparameter n_{coals} equal to 4, 32, and 64, see Algorithm 3 and



(a) $\hat{\rho}_{\text{avg}} \approx 0.38, n_{\text{train}} = 1{,}280$.

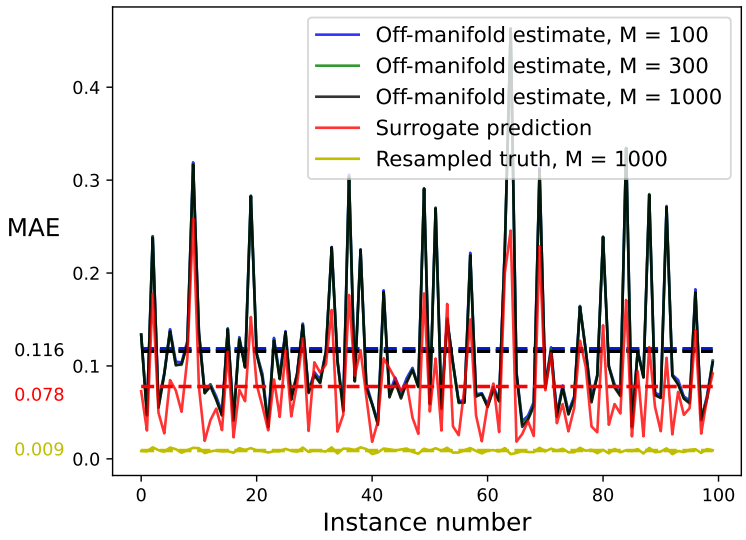


(b) $\hat{\rho}_{\text{avg}} \approx 0.37, n_{\text{train}} = 6{,}400$.



(c) $\hat{\rho}_{\text{avg}} \approx 0.37, n_{\text{train}} = 64{,}000$.

Figure 11: **Burr Distributed Features with** $\zeta = 2$.



(d) $\hat{\rho}_{\text{avg}} \approx 0.37, n_{\text{train}} = 128{,}000$.

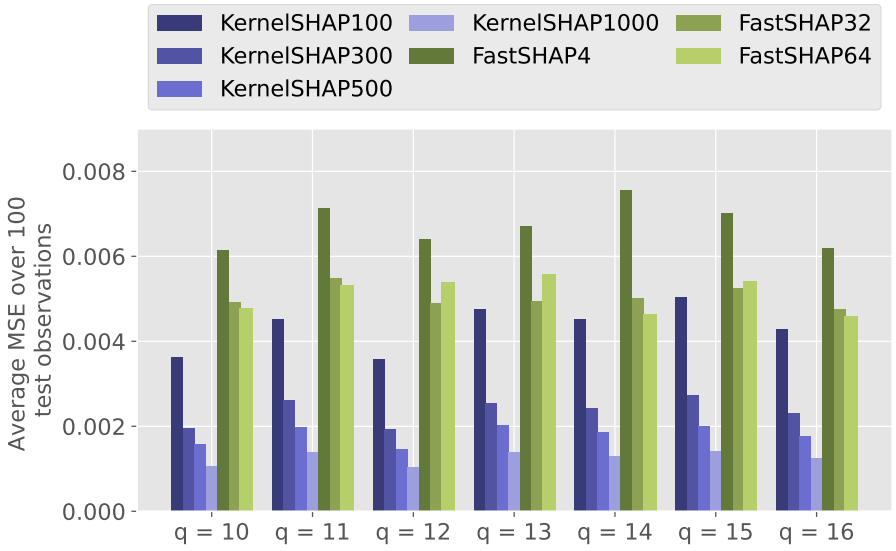
Figure 11: (Continued) **Burr Distributed Features with $\zeta = 2$.**
In the plot, MAEs resulting from the different estimates are shown in different colors. In dashed lines, the mean MAE of each estimation method is plotted in the same color as the MAE of the corresponding estimation method. Next to the dashed line, on the y-axis, the value of the mean MAE of each method is written. The "method" with the lowest MAE is the other realization of the estimate of the true value. Each plot is captioned with the empirical average correlation $\hat{\rho}_{\text{avg}}$ in the simulated training data set and the number of training observations n_{train}.

Section 3.4.2 for details regarding this hyperparameter. In the first experiments, only these values of $|\mathcal{D}|$ and n_{coals} will be considered. However, other values of $|\mathcal{D}|$ and n_{coals} will be considered later in Sections 4.5.4 and 4.5.5, respectively. Throughout this section, for simplicity, the notation "KernelSHAP$|\mathcal{D}|$" will be used to denote the KernelSHAP estimate computed with $|\mathcal{D}|$ feature coalitions, e.g. KernelSHAP100 when $|\mathcal{D}| = 100$. In addition, the notation "FastSHAPn_{coals}" will be used to denote the FastSHAP model trained with the specific value of the hyperparameter n_{coals}, e.g. FastSHAP4 when $n_{\text{coals}} = 4$. First, in Section 4.5.1, we consider the multivariate normally distributed features with the three covariance matrices defined in Section 4.3.1. Then, in Section 4.5.2, the results related to the multivariate Burr distribution are presented.

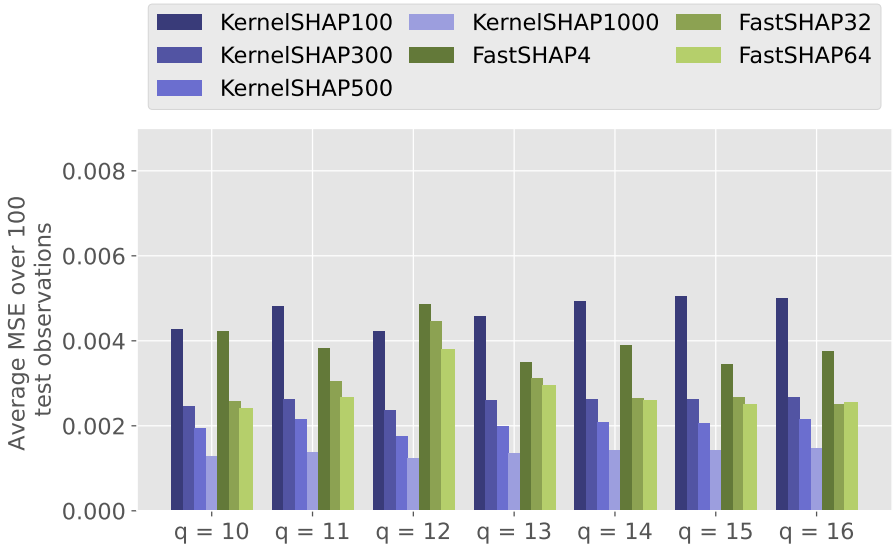
4.5.1 Simulation Model 1: The Multivariate Normal Distribution

Independent Features

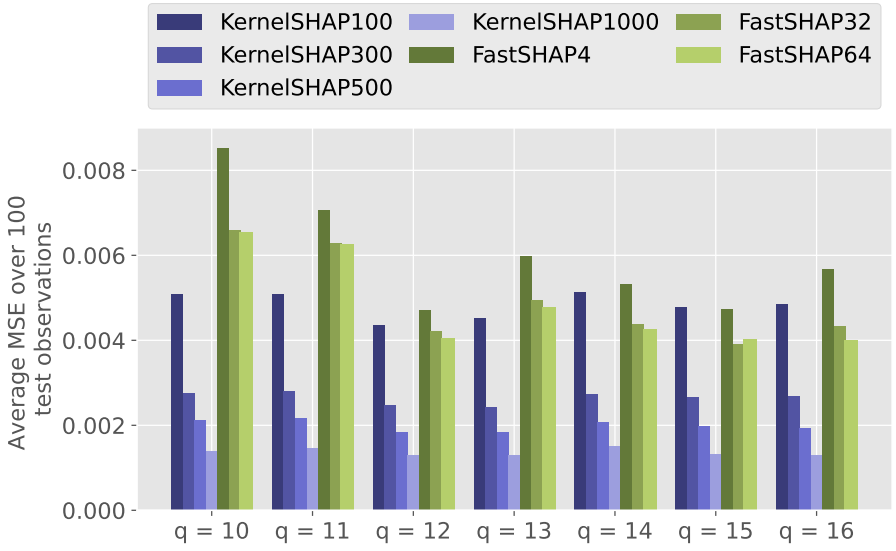
The error of the KernelSHAP and FastSHAP estimates are shown for independently multivariate normally distributed features in Figures 12a, 12b and 12c for n_{train} equal to 1,280, 6,400 and 64,000, respectively. Firstly, note that for all values of n_{train}, the error in the KernelSHAP estimate decreases when $|\mathcal{D}|$ increases. In addition, the error in the predictions of the FastSHAP model is smaller for $n_{\text{coals}} = 32$ and 64 than $n_{\text{coals}} = 4$, also for all values of n_{train} and q. The improvement between FastSHAP64 and FastSHAP32 is much smaller than between FastSHAP32 and FastSHAP4 in most cases, and the relative performance of FastSHAP32 and FastSHAP64 varies in the examples. Since these two models are separate neural networks, in which the training is affected by randomness, this indicates that after sufficiently many coalitions are considered, the error in the model's predictions stops decreasing. In Section 4.5.5, we try to determine more precisely the effect of the hyperparameter by considering also other values of it. Increasing $|\mathcal{D}|$ and n_{coals} increases the computational complexity of KernelSHAP and FastSHAP, respectively, therefore, this must be weighed against the increased accuracy of the estimates, which we will discuss more later on in Section 4.5.6.



(a) $n_{\text{train}} = 1{,}280$.



(b) $n_{\text{train}} = 6{,}400$.



(c) $n_{\text{train}} = 64{,}000$.

Figure 12: **Independent Multivariate Normally Distributed Features**.
The error in the KernelSHAP and FastSHAP estimates evaluated using the metric (46).

When $n_{\text{train}} = 1{,}280$, KernelSHAP outperforms FastSHAP for all values of the number of features q, for all values of n_{coals}, and for all values of the number of feature coalitions $|\mathcal{D}|$. This might indicate that the training data set with $n_{\text{train}} = 1{,}280$ training observations is too small for the FastSHAP models to successfully learn to estimate the Shapley values, at least relative to KernelSHAP. For the larger data sets, the performance of the FastSHAP models is at best similar to KernelSHAP300. Thus, KernelSHAP clearly outperforms FastSHAP when the number of feature coalitions $|\mathcal{D}|$ considered in KernelSHAP is sufficiently high. Comparing the cases with $n_{\text{train}} = 6{,}400$ and $n_{\text{train}} = 64{,}000$, the FastSHAP models perform worse for the larger data set. This indicates that the size of the training data set is not the only factor that determines the relative performance of FastSHAP compared to KernelSHAP. However, since the FastSHAP model is a neural network, it is clear that access to a sufficiently large training data set is necessary for FastSHAP to perform well. When considering this, we must keep in mind that the black box model and surrogate model differ in the two cases, which can cause some dissimilarity between the examples, and might explain why the performance of FastSHAP is worse for the larger training data set.

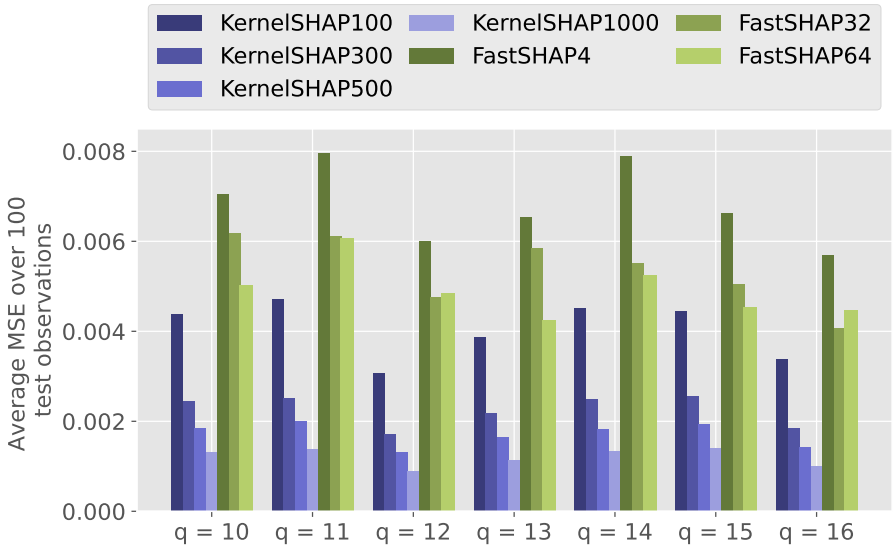
It is interesting to note that for all three values of n_{train}, the plots show that the average error over the 100 test observations is quite similar for all values of q, with an exception of the FastSHAP models when $n_{\text{train}} = 64{,}000$ and q equal to 10 and 11. For KernelSHAP, this might seem counter-intuitive since the fixed values of $|\mathcal{D}|$ that are used in the KernelSHAP estimates make out a much smaller proportion of the total number of feature coalitions 2^q for larger values of q. However, as argued in Doumard et al. [11], "this is probably due to the fact that usually, the more features there are, the less influence amplitude each individual feature has in the prediction". Therefore, it might be reasonable to assume that the difficulty of estimating the Shapley values of a model is related to the number of *significant* features in the model, not the full number of features the model is trained on.

Somewhat Correlated Features

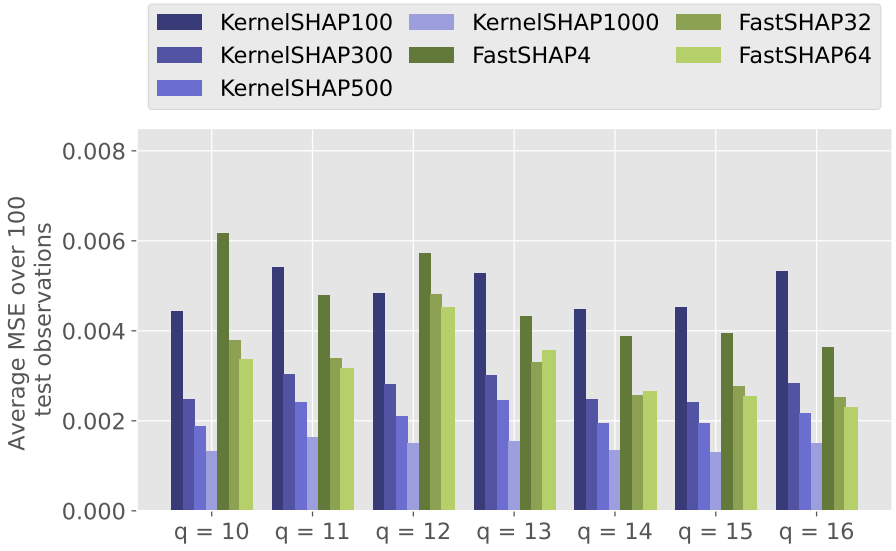
In Figures 13a, 13b and 13c, the error of the estimates are shown for somewhat correlated features with n_{train} equal to 1,280, 6,400 and 64,000, respectively. As in the case with independent features, for all values of n_{train} and q, the error in the KernelSHAP estimate decreases when $|\mathcal{D}|$ increases. In addition, the error in the predictions of the FastSHAP model is smaller for $n_{\text{coals}} = 32$ and 64 than $n_{\text{coals}} = 4$, also for all values of n_{train}. Increasing n_{coals} from 32 to 64 does not necessarily lead to an increase in the accuracy of the FastSHAP model, which may indicate that using $n_{\text{coals}} = 32$ is sufficient in this case. More results regarding this parameter will be presented in Section 4.5.5. When $n_{\text{train}} = 1{,}280$, KernelSHAP100 outperforms all FastSHAP models for all values of q, which was also the case for independent features as shown in Figure 12a. From this, it seems like 1,280 training observations is too few for the FastSHAP model to successfully learn to estimate the Shapley values. In the examples in Figure 13, it seems like the performance of FastSHAP is better relative to KernelSHAP when $n_{\text{train}} = 64{,}000$ and $n_{\text{train}} = 6{,}400$. The FastSHAP models perform quite similarly relative to KernelSHAP in these two cases, as shown in Figures 13b and 13c. Moreover, in these cases, the best FastSHAP models are either with $n_{\text{coals}} = 32$ or 64. They are competitive with KernelSHAP300, or in the best cases KernelSHAP500. In total, in the training of FastSHAP, $n_{\text{train}} \cdot n_{\text{epochs}} \cdot n_{\text{coals}}$ combinations of instances $\boldsymbol{x}$ and feature coalitions $\mathcal{S}$ are processed, where n_{epochs} is the number of epochs in the training procedure. Thus in the span of an epoch, the "efficient size" of the training data set is $n_{\text{train}} \cdot n_{\text{coals}}$, which may explain why the FastSHAP32 and FastSHAP64 models do not seem to perform better for the larger data set with n_{train} equal to 64,000 than when n_{train} equals 6,400. It might be that because each instance is considered n_{coals} times, the data set with 6,400 training samples is sufficiently large for the FastSHAP32 and FastSHAP64 models to learn to estimate the Shapley values.

Highly Correlated Features

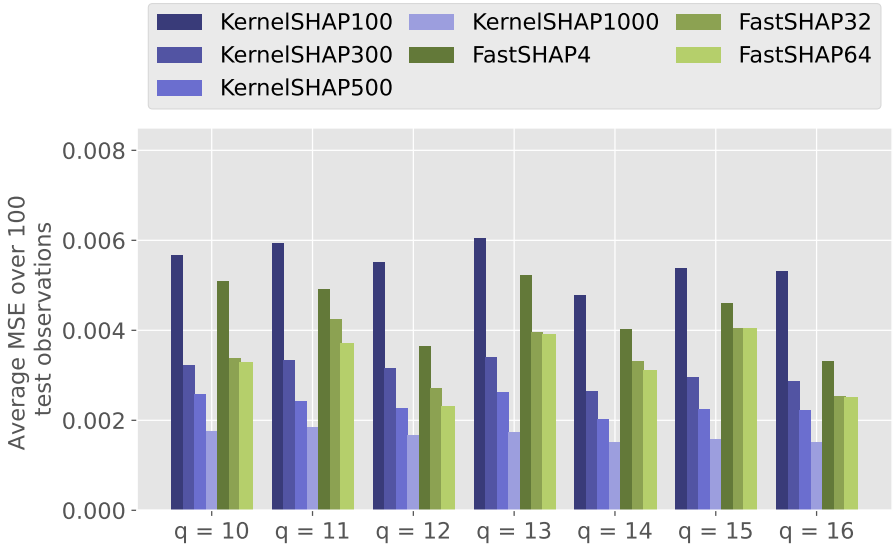
For the case with highly correlated multivariate normally distributed features, the error of the estimation methods is shown in Figures 14a, 14b and 14c for n_{train} equal to 1,280, 6,400 and 64,000, respectively. Strikingly, for all three values of n_{train}, the KernelSHAP estimates for all values of $|\mathcal{D}|$ have a much higher error in the case with $q = 14$ features, than any other value of q. For now, we will treat this case as an "outlier" and disregard it, however, in Section 4.5.7



(a) $n_{\text{train}} = 1{,}280$.

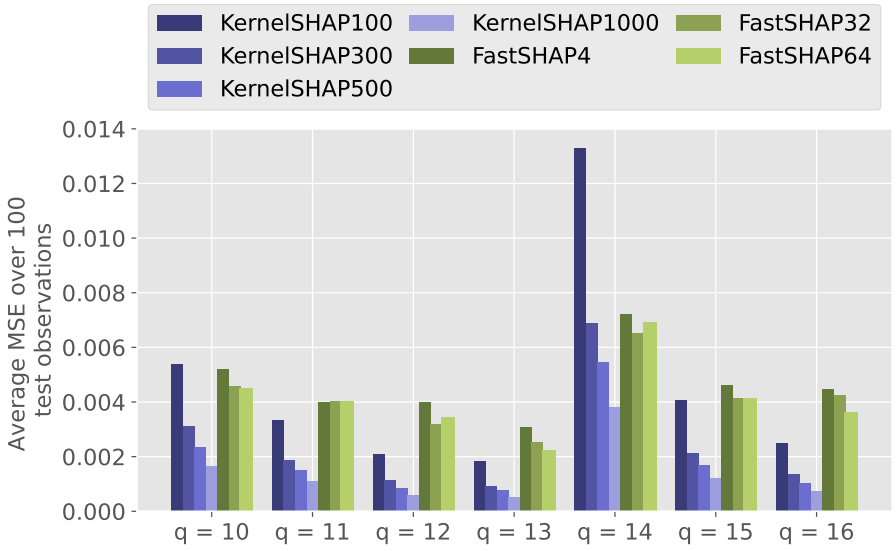


(b) $n_{\text{train}} = 6{,}400$.

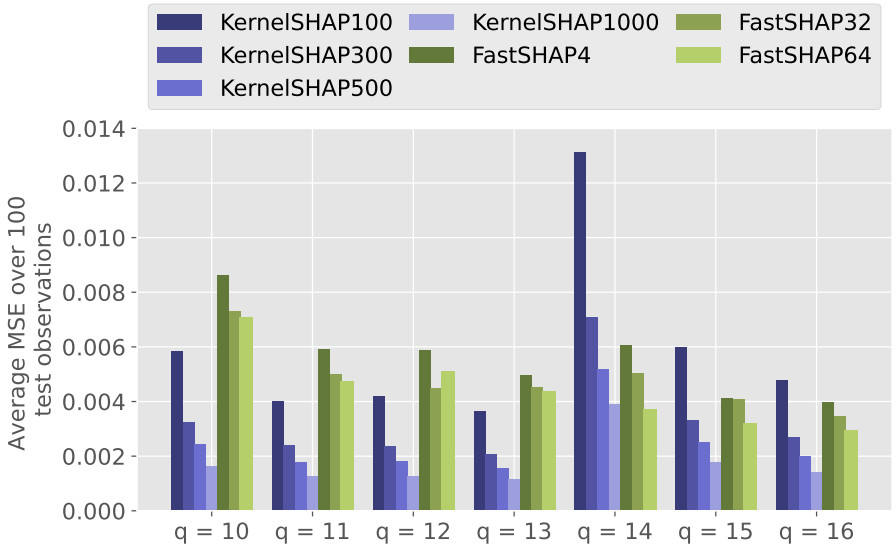


(c) $n_{\text{train}} = 64{,}000$.

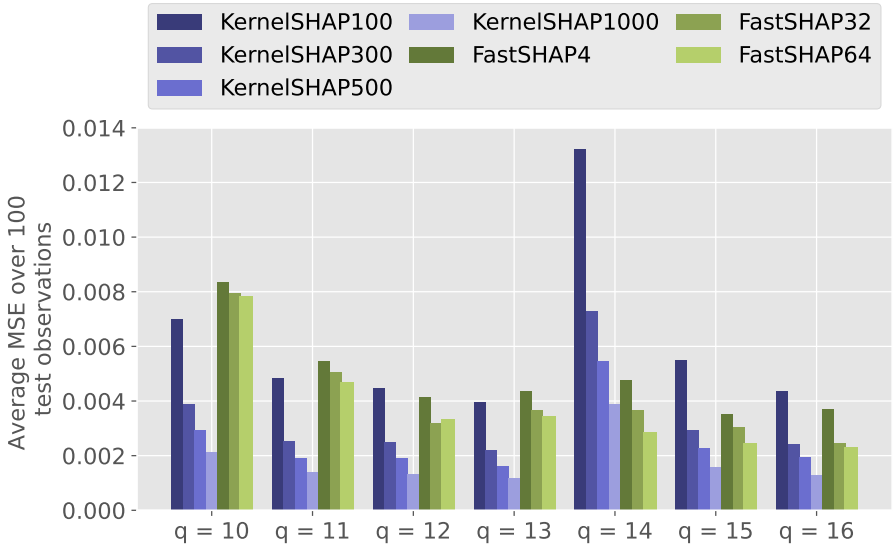
Figure 13: **Somewhat Correlated Multivariate Normally Distributed Features.** The average error of the KernelSHAP and FastSHAP estimators over 100 test observations.



(a) $n_{\text{train}} = 1{,}280$.



(b) $n_{\text{train}} = 6{,}400$.



(c) $n_{\text{train}} = 64{,}000$.

Figure 14: **Highly Correlated Multivariate Normally Distributed Features.** The error of the Shapley value estimators evaluated using the metric (46).

we will get back to this case and try to distinguish the causes of why the KernelSHAP method behaves differently in this case. As shown in Figure 14a, when the number of training observations n_{train} equals 1,280, the FastSHAP32 and FastSHAP64 models are performing similarly to KernelSHAP100. KernelSHAP1000, which is the best estimate, is significantly better for all values of q. The performance of FastSHAP32 and FastSHAP64 is for some values of q close to the performance of KernelSHAP300 when n_{train} equals 6,400 and 64,000. However, also in these examples, none of the FastSHAP models are competitive with KernelSHAP1000, with the exception of the cases with $q = 14$ which we will analyze in more detail in Section 4.5.7.

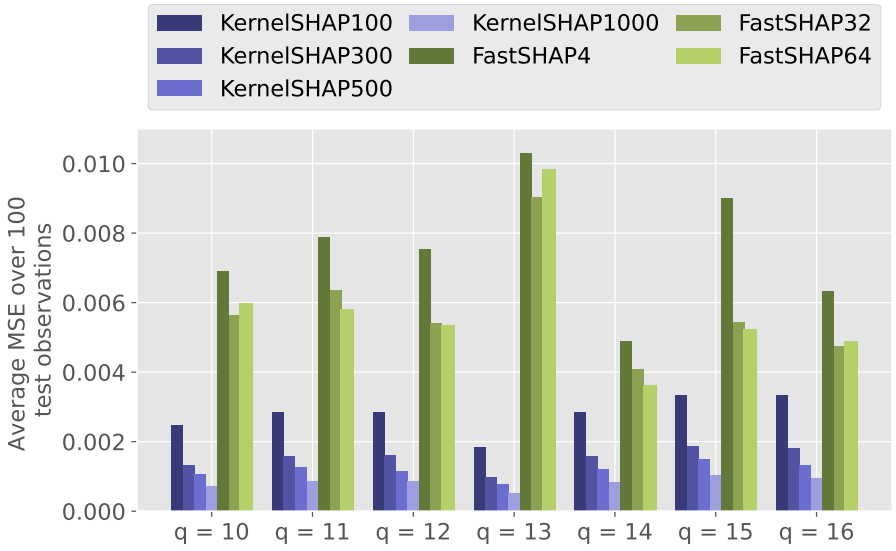
In summary, based on Figures 12, 13 and 14 it is clear that increasing the number of feature coalitions $|\mathcal{D}|$ increases the accuracy of KernelSHAP. Moreover, increasing the hyperparameter n_{coals} from 4 to 32 in the training of the FastSHAP model in accordance with Algorithm 3, increases the accuracy of the method. However, the relative performance of FastSHAP32 and FastSHAP64 varies in the examples, which can indicate that using $n_{\text{coals}} = 32$ is sufficient, which we discuss in more detail in Section 4.5.5. In general, it seems like the error in the estimation methods is not determined by the number of features q in the training data set. There is no clear trend of how the number of features affects the error in the estimates. Especially for KernelSHAP, this is interesting, since the fixed values of the number of feature coalitions $|\mathcal{D}|$ used in the estimates make out a much smaller proportion of the total number of feature coalitions 2^q as the dimension q increases. As previously mentioned, a possible explanation of this is that if the number of features is increased, the amplitude of the Shapley values that each feature is assigned will be smaller, therefore the number of features significant to the model does not necessarily increase when the total number of features increases.

Overall, it also seems like using more than around 300 feature coalitions in KernelSHAP generally means that KernelSHAP outperforms FastSHAP, whereas, for lower values, FastSHAP is more competitive. Using $|\mathcal{D}|$ equal to 1000 feature coalitions in KernelSHAP makes it significantly more accurate than the FastSHAP models in all the examples, except the "outlier" examples shown in Figure 14 with $q = 14$. However, the computational cost of KernelSHAP is increased by increasing $|\mathcal{D}|$, which must be weighed against the increase in accuracy, which we will get back to in Section 4.5.6. Lastly, the results indicate that a training data set with 1,280 training observations is too small for training the FastSHAP models to provide high accuracy results, even with a higher value of n_{coals}. Since the FastSHAP model is a neural network, it seems counterintuitive its performance is not significantly better in the case with n_{train} equal to 64,000 than with 6,400. However, per epoch in the training of the model, each instance is processed n_{coals} times. Therefore, $n_{\text{coals}} \cdot n_{\text{trian}}$ combinations of instances $\boldsymbol{x}$ and feature coalitions $\mathcal{S}$ are considered per epoch, which means that the "efficient" size of the training data set is larger, which can explain why having more than 6,400 training observations does not increase FastSHAP's accuracy in these examples. As previously discussed, the normal distribution has several properties that make data generated from it relatively easy to learn. Therefore, we have also tested the methods on a non-normal distribution and will present the results from this in the following section.

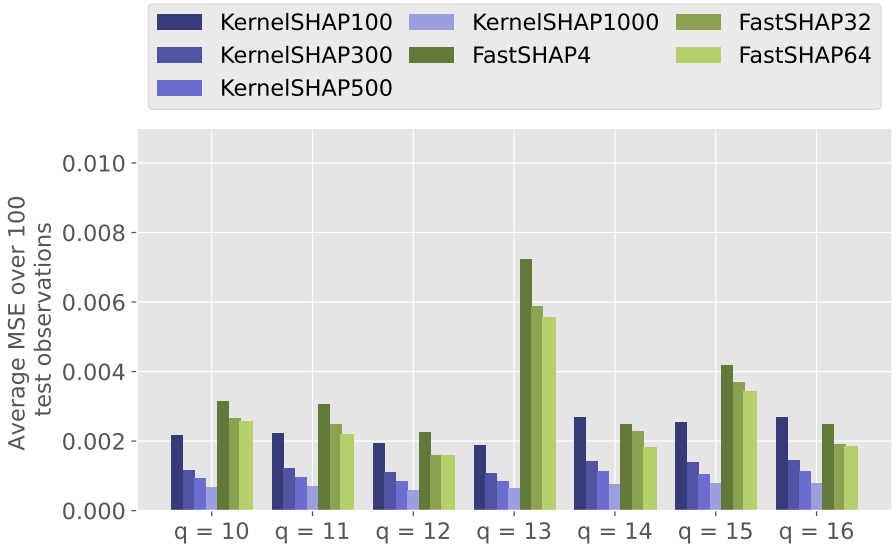
4.5.2 Simulation Model 2: The Multivariate Burr Distribution

Lower Correlation and Lighter Tails

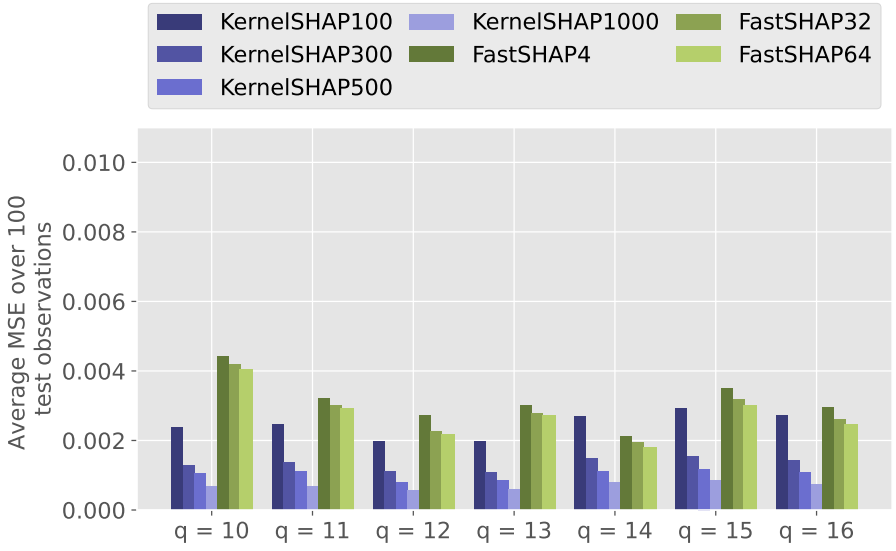
Firstly, the results concerning the data set simulated from the multivariate Burr distribution (24) with parameter $\zeta = 7$, as described in Section 4.3, are presented. For n_{train} equal to 1,280, 6,400 and 64,000, the accuracy of the FastSHAP and KernelSHAP estimates are shown in Figures 15a, 15b and 15c, respectively. As before, we consider KernelSHAP with the number of feature coalitions $|\mathcal{D}|$ equal to 100, 300, 500, and 1,000. In addition, FastSHAP is considered for three values of the hyperparameter n_{coals}. These are n_{coals} equal to 4, 32, and 64. For all values of n_{train} and q, as for the examples shown in Figures 12, 13 and 14 with the normal distributions, the KernelSHAP estimate improves as a function of the number of feature coalitions $|\mathcal{D}|$. Overall, the performance of FastSHAP32 and FastSHAP64 is better than FastSHAP4. However, the relative performance of FastSHAP32 and FastSHAP64 varies, which can indicate that using $n_{\text{coals}} = 32$ is sufficient. Noticeably, as shown in Figure 15a, the error in the estimates of all three FastSHAP models is much higher than all KernelSHAP estimates when $n_{\text{train}} = 1{,}280$ for all values of the



(a) $n_{\text{train}} = 1{,}280$.

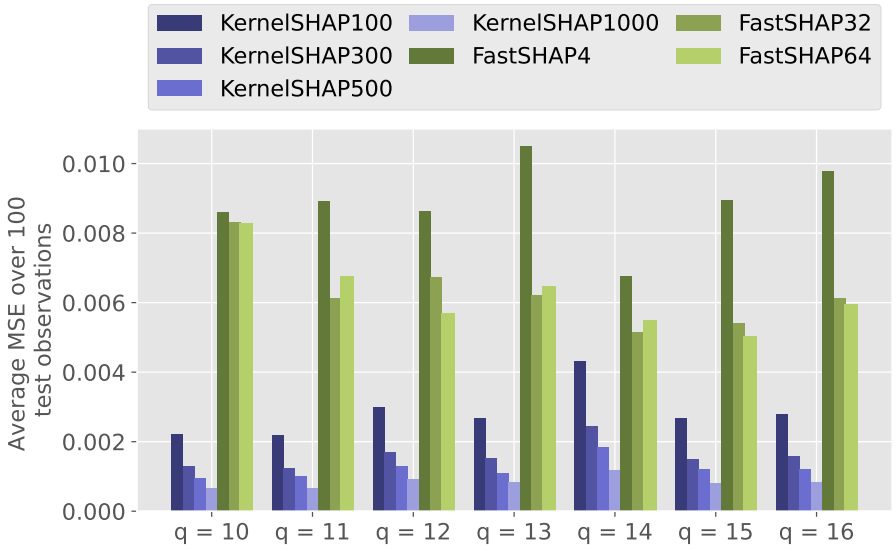


(b) $n_{\text{train}} = 6{,}400$.

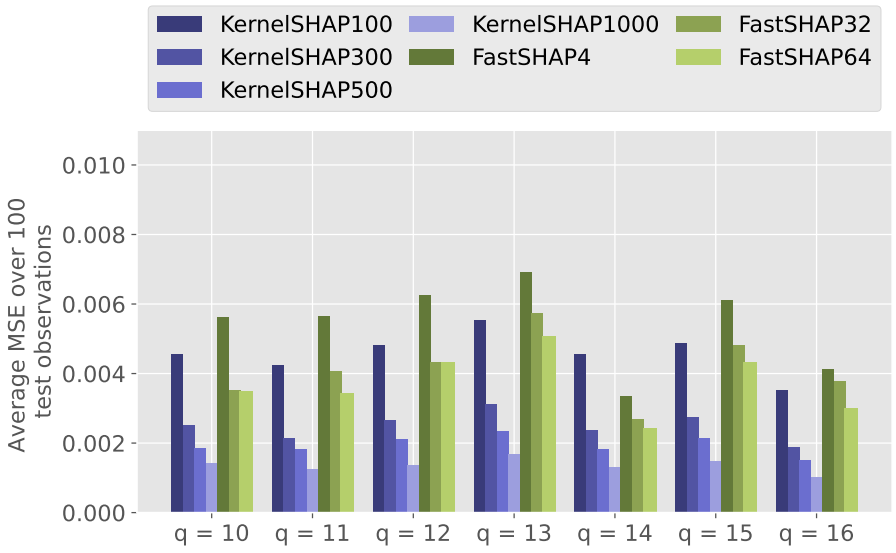


(c) $n_{\text{train}} = 64{,}000$.

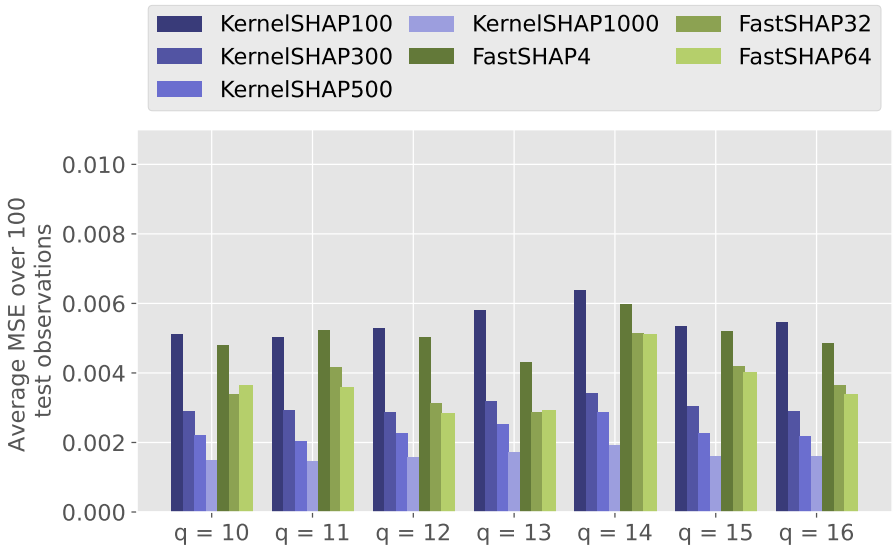
Figure 15: **Burr Distributed Features with $\zeta = 7$.**
The error (46) of the Shapley value estimators over 100 test observations.



(a) $n_{\text{train}} = 1{,}280$.



(b) $n_{\text{train}} = 6{,}400$.



(c) $n_{\text{train}} = 64{,}000$.

Figure 16: **Burr Distributed Features with $\zeta = 2$.**
The error of the Shapley value estimators averaged over 100 test observations.

number of features q. This shows that the training data set consisting of only 1,280 observations is too small for the FastSHAP model to learn to approximate the Shapley values well. On the other hand, the error in the KernelSHAP estimate is of a similar magnitude for all values of n_{train}, as shown in Figure 15, which shows that the size of the training data set n_{train} does not directly affect KernelSHAP in these examples. For n_{train} equal to 1,280 and 6,400, the FastSHAP4 and FastSHAP32 seem to fail more severely in the case with $q = 13$ features. This seems to be an "outlier" in the experiments. Looking aside from this example, the number of features in the data set generally does not have a direct influence on the accuracy of either FastSHAP or KernelSHAP. For this example, the performance of FastSHAP relative to KernelSHAP is similar when n_{train} equals 6,400 and when n_{train} equals 64,000. In both cases, the FastSHAP models' performance is worse than or similar to KernelSHAP100 for all values of the number of features q. Therefore, it seems like having a 6,400 rather than 1,280 training observations is important for FastSHAP to be more accurate. However, it seems like for the largest data set with 64,000 training observations, the model does not improve further. Clearly, in practical situations, it is not possible to increase the size of the training data set. The point of varying the number of training observations is to investigate empirically when each method is suitable depending on the properties of the original data set.

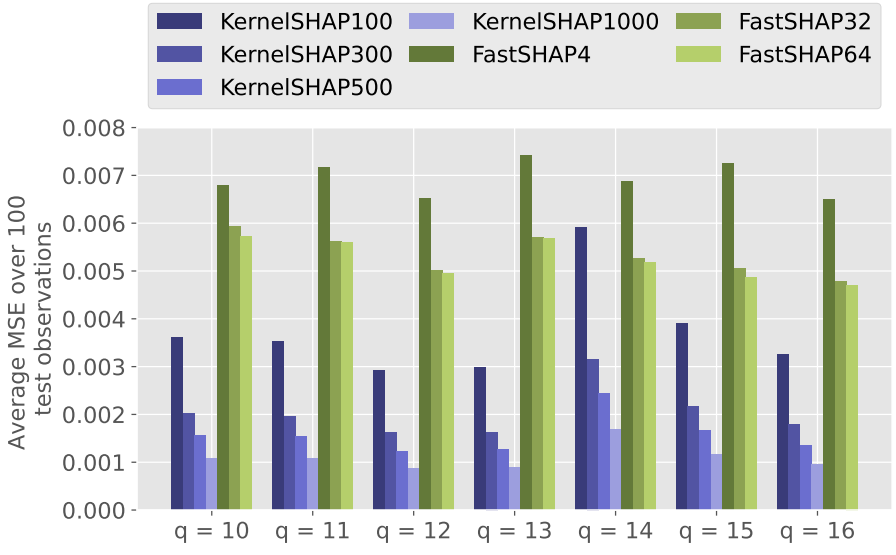
Higher Correlation and Heavier Tails

Recall that a lower value of the parameter ζ in the multivariate Burr distribution will lead to heavier tails and more extreme outlier observations. This means that generally, the data simulated from a Burr distribution with a lower value of ζ will be more difficult to learn. Therefore, the experiments are repeated with ζ equal to 2 in order to investigate if the heavier tails of the distribution affect the performance of the Shapley value estimation methods. In Figure 16, the accuracy of the Shapley value estimators evaluated according to the metric (46) is shown for the case with multivariate Burr distributed (24) features with the parameter $\zeta = 2$. As shown in Figure 16a, the error of the estimates of all the FastSHAP models is large compared to the error of KernelSHAP when $n_{\text{train}} = 1{,}280$, for all values of $|\mathcal{D}|$ in KernelSHAP. Hence, for the data set with $n_{\text{train}} = 1{,}280$, it is clear that KernelSHAP is to be preferred. This is in accordance with the results in Figure 15a. Compared to the examples shown in Figures 12a, 13a and 14a where the features were multivariate normally distributed, the relative performance of FastSHAP compared with KernelSHAP is worse in the case of the Burr distribution with $\zeta = 2$ and 7, as shown in Figures 16a and 15a. This may indicate that a non-normal distribution is more difficult for the FastSHAP model to successfully learn. The KernelSHAP weighted least squared (34) approximation seems less affected by this, based on these examples.

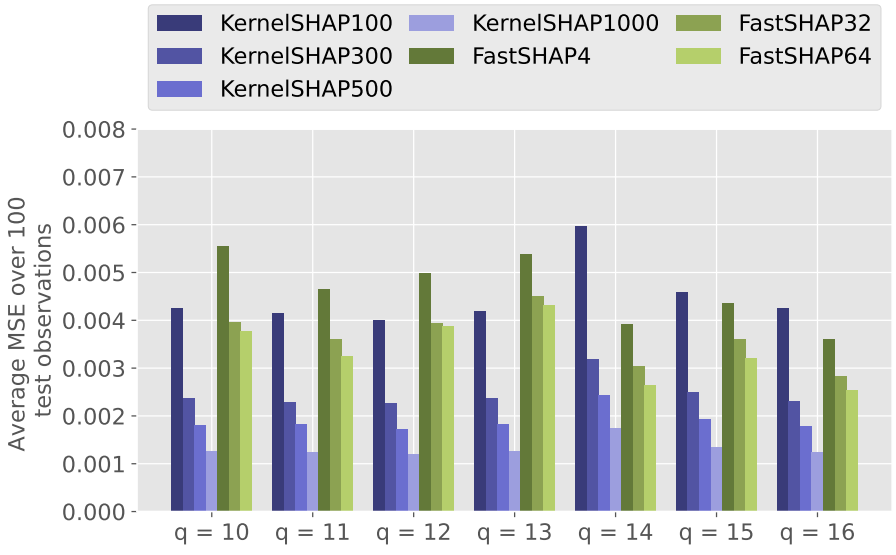
For the two larger data sets, the FastSHAP32 and FastSHAP64 models perform comparatively to KernelSHAP300 for some values of q, and better than KernelSHAP100 for all values of q. The FastSHAP4 model is outperformed in most cases by KernelSHAP100 when n_{train} equals 6,400, and when n_{train} equals 64,000, the FastSHAP4 model performs similarly to KernelSHAP100. However, even for the two larger data sets, KernelSHAP1000 clearly outperforms all the FastSHAP models. Moreover, also in this case, for all values of n_{train}, there is no clear trend between the number of features q and the accuracy of the estimates, which as previously mentioned can be explained by the number of features in a model not necessarily increasing the number of *significant* features in the model. In order to further investigate the relative accuracy of FastSHAP and KernelSHAP depending on the properties of the training data set, we will in the following section aggregate the results across the five data sets we have considered so far to distinguish the overall tendencies from the elements of randomness such as the initialization of the weights in the neural network in FastSHAP.

4.5.3 Evaluation Aggregated over the Simulation Models

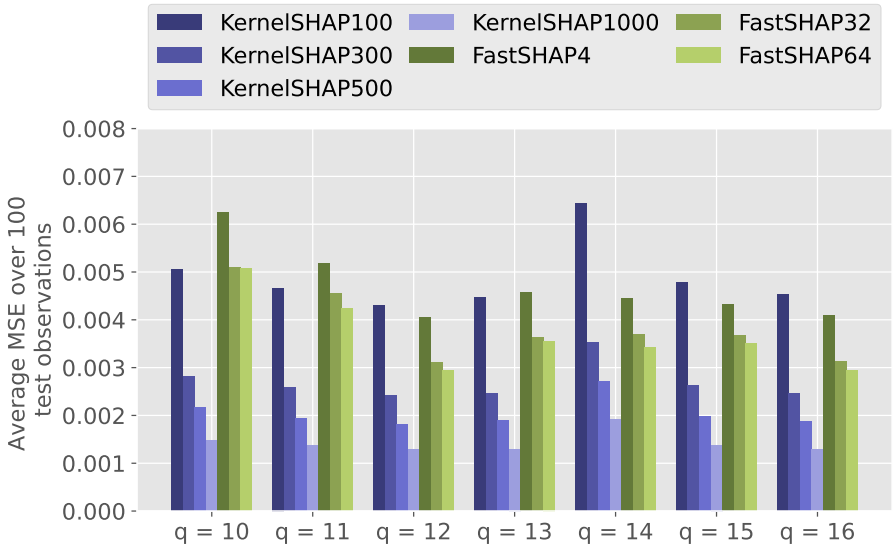
In the evaluation of the Shapley value estimators presented in Sections 4.5.1 and 4.5.2, there is some discrepancy in the relative performance of KernelSHAP and FastSHAP depending on e.g. the size of the training data set. Therefore, in order to investigate further the relative performance of FastSHAP and KernelSHAP depending on the nature of the data set, the results are aggregated across the five data sets from the previous section. Specifically, for each value of the number



(a) $n_{\text{train}} = 1{,}280$.



(b) $n_{\text{train}} = 6{,}400$.



(c) $n_{\text{train}} = 64{,}000$.

Figure 17: **Error Aggregated over the Simulated Data Sets.**
The error of the Shapley value estimators aggregated over the data sets simulated from the Burr and normal distributions.

of features q, the error according to the metric (46) is averaged over the five data sets for the KernelSHAP estimates with $|\mathcal{D}|$ equal to 100, 300, 500 and 100, and FastSHAP with n_{coals} equal to 4, 32, and 64.

The aggregated results are shown in Figures 17a, 17b and 17c for n_{train} equal to 1,280, 6,400 and 64,000, respectively. From the aggregated results, it seems clear that when the number of training observations is small, as illustrated for n_{train} equal to 1,280 in Figure 17a, FastSHAP overall performs significantly worse than KernelSHAP, even for smaller values of $|\mathcal{D}|$. The aggregated results with n_{train} equal to 6,400 and 64,000 show that in these cases FastSHAP32 and FastSHAP64 are competitive to KernelSHAP100 for all values of q. However, KernelSHAP clearly outperforms FastSHAP when the number of feature coalitions $|\mathcal{D}|$ is greater than a 500. The number of features q does not seem to directly affect the accuracy of either FastSHAP or KernelSHAP directly. This behavior is similar to that seen in [11], where an empirical study of several explanation methods, including KernelSHAP, is presented. Our results show that also FastSHAP behaves similarly. It might be that the "dimension" of the Shapley value approximation problem should be measured based on the number of Shapley values that are found to be significant when explaining a model rather than based on the number of features in the original data set. In practice, however, the number of significant Shapley values is not known a priori, and therefore, the dimension of the estimation problem will be unknown. It is still an important finding since it demonstrates that the methods' behavior are affected by other factors than the empirical properties of the data set. We will get back to this in Section 4.5.7, where we go into more detail on the examples with $q = 14$ shown in Figure 14, where the error of the KernelSHAP estimates was much higher than in all the other examples.

In this section, we have evaluated the methods purely based on accuracy. However, the computational cost of the methods is important in many practical applications. Therefore, in Section 4.5.6, the results of evaluating the methods based on computational time are presented. Before the results of evaluating the methods in terms of computational cost are presented, KernelSHAP's accuracy as a function of the number of feature coalitions $|\mathcal{D}|$ is analyzed in more detail in the following section. In addition, in Section 4.5.5, the accuracy of the FastSHAP model is evaluated as a function of the hyperparameter n_{coals}.

4.5.4 The Accuracy of KernelSHAP as a Function of the Number of Feature Coalitions $|\mathcal{D}|$

In the previous section, we saw that the KernelSHAP method's accuracy increased as a function of the number of feature coalitions $|\mathcal{D}|$. However, we only considered four values of $|\mathcal{D}|$, which where $|\mathcal{D}|$ equal to 100, 300, 500 and 1,000. Therefore, to further investigate how the value of $|\mathcal{D}|$ affects the estimate, we will give an example where the error in the KernelSHAP estimate is plotted for $|\mathcal{D}| = 50, 60, 70, \ldots, 2000$. In this example we have simulated $n_{\text{train}} = 6{,}400$ observations of $q = 16$ features following the multivariate normal distribution with correlation matrix $\boldsymbol{\Sigma}_{\text{some\_corr}}$. It should be noted that based on our experiments, the convergence behaviour of KernelSHAP is similar for the other simulation models, other values of the number of features q, and other values of the number of training observations n_{train}. Therefore, we only include this example here.

The error in the KernelSHAP estimates (34) is shown for this example in Figure 18. As shown in the plot, the accuracy of the KernelSHAP estimate overall increases for increasing values of the number of coalitions $|\mathcal{D}|$. The minor discrepancies from the trend of the decreasing accuracy are caused by the coalitions in $\mathcal{D}$ being randomly drawn from the set of all possible coalitions. In most of the examples in the previous section, shown in Figures 12 to 16, KernelSHAP300 outperformed all three FastSHAP models, and KernelSHAP500 and KernelSHAP1000 outperformed the FastSHAP models in all examples, with the exception of when $q = 14$ in the examples shown in Figure 14. Figure 18 shows that the accuracy of KernelSHAP has not converged when $|\mathcal{D}|$ equals 500, meaning that even before the error has converged, the KernelSHAP method outperforms FastSHAP in terms of accuracy. Therefore, based on our simulation experiments, it seems clear that when using e.g. 1,000 feature coalitions, the KernelSHAP method will provide significantly more accurate approximations than FastSHAP. However, the computational cost of the KernelSHAP method increases as a function of $|\mathcal{D}|$. Therefore, the increased computational cost must be weighed

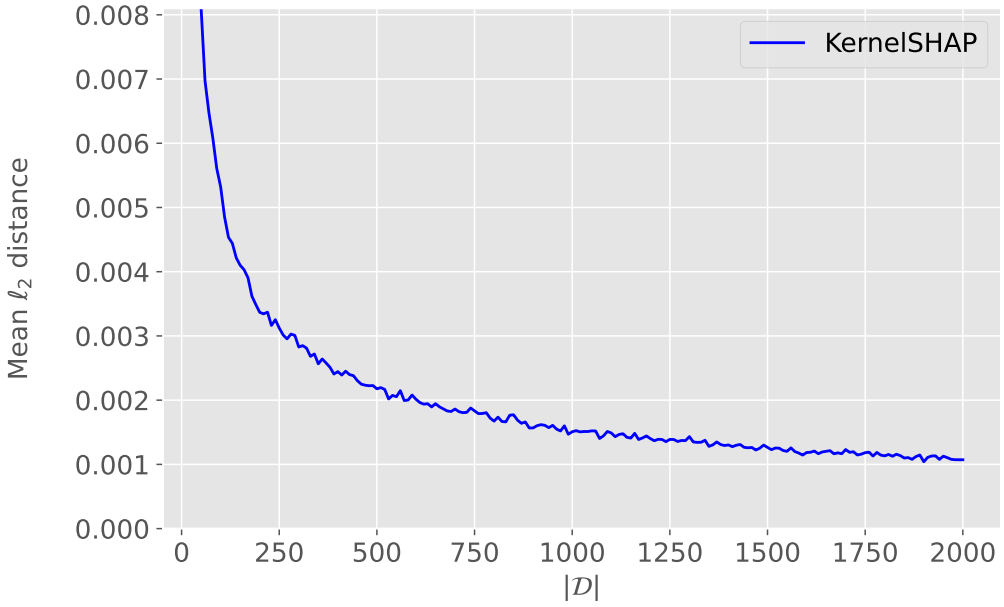


Figure 18: **Multivariate Normal,** $\Sigma_{\text{some\_corr}}$**,** $q = 16$ **and** $n_{\text{train}} = 6{,}400$
The error in the KernelSHAP estimate is plotted as a function of the number of coalitions $|\mathcal{D}|$.

against the decrease in the error of the estimate. In Section 4.5.6, more results concerning the computational cost of the methods will be presented.

4.5.5 The Accuracy of FastSHAP as a Function of the Hyperparameter n_{coals}

As explained in Section 3.4.2, in Algorithm 3, the training algorithm of the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$, the hyperparameter n_{coals} is important. It determines the number of feature coalitions $\mathcal{S}$ to consider per instance $\boldsymbol{x}$ in a batch. Recall that in the duration of an epoch, the whole training data set is processed once. For the FastSHAP model, the loss will be evaluated for $n_{\text{coals}} \cdot n_{\text{train}}$ feature coalitions per epoch in the training process, where n_{train} is the size of the training set. Therefore, setting $n_{\text{coals}} > 1$ ensures that several coalitions $\mathcal{S}$ are considered per instance in the batch. This is beneficial since in the exact Shapley values (33), all 2^q coalitions are considered per instance $\boldsymbol{x}$. Hence, this should facilitate the model's learning of the Shapley values. This was also the case in the examples in Figure 12 to 16, where the FastSHAP32 and FastSHAP64 models performed better than the FastSHAP4 model. However, the relative performance of FastSHAP32 and FastSHAP64 differed in the examples, which may indicate that increasing the value of the hyperparameter beyond a certain threshold is not necessarily beneficial. Therefore, we have performed a more detailed study of the effect of the hyperparameter and present the results here. In the training of the FastSHAP model, the loss will be evaluated $n_{\text{coals}} \cdot n_{\text{train}}$ per epoch, therefore, the time complexity of training the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ increases when increasing n_{coals}.

In order to investigate the effect of changing this hyperparameter, the mean error (46) over 100 test observations is computed for FastSHAP models trained with $n_{\text{coals}} = 1$, 4, 16, 32, 48, 64, 128, 256, and 512. In addition, the CPU time of training the FastSHAP models is determined. As an example, we use the simulation model described in Section 4.3.1 for $q = 16$ and somewhat correlated multivariate normally distributed features. In Figure 19, the error according to the evaluation metric (46) over 100 test observations and CPU training times are plotted as a function of n_{coals} for $n_{\text{train}} = 1{,}280$. This is repeated for n_{train} equal to 6,400 and 64,000, and the corresponding results are shown in Figures 20 and 21, respectively.

For all three values of n_{train}, the plots are very similar. The accuracy of the FastSHAP model increases as a function of increasing n_{coals} until the accuracy seems to have converged when n_{coals} is greater than or equal to 32. There are some fluctuations in the accuracy when $n_{\text{coals}} \geq 32$. This

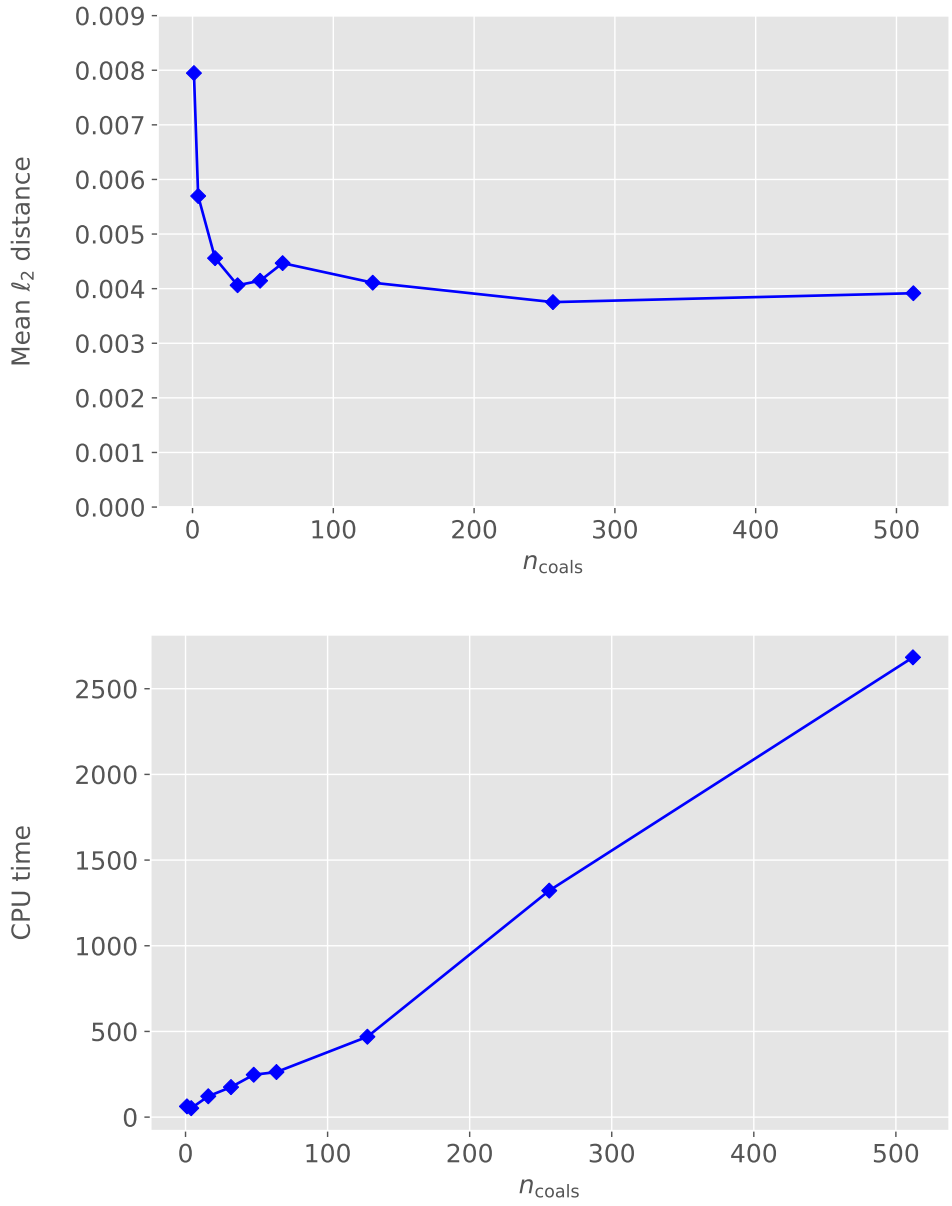


Figure 19: **Multivariate Normal, $\boldsymbol{\Sigma}_{\text{some\_corr}}$, $q = 16$ and $n_{\text{train}} = 1{,}280$.**
The upper panel shows the accuracy of the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ as a function of n_{coals}.
The lower panel shows the CPU time of the training of the model as a function of n_{coals}.

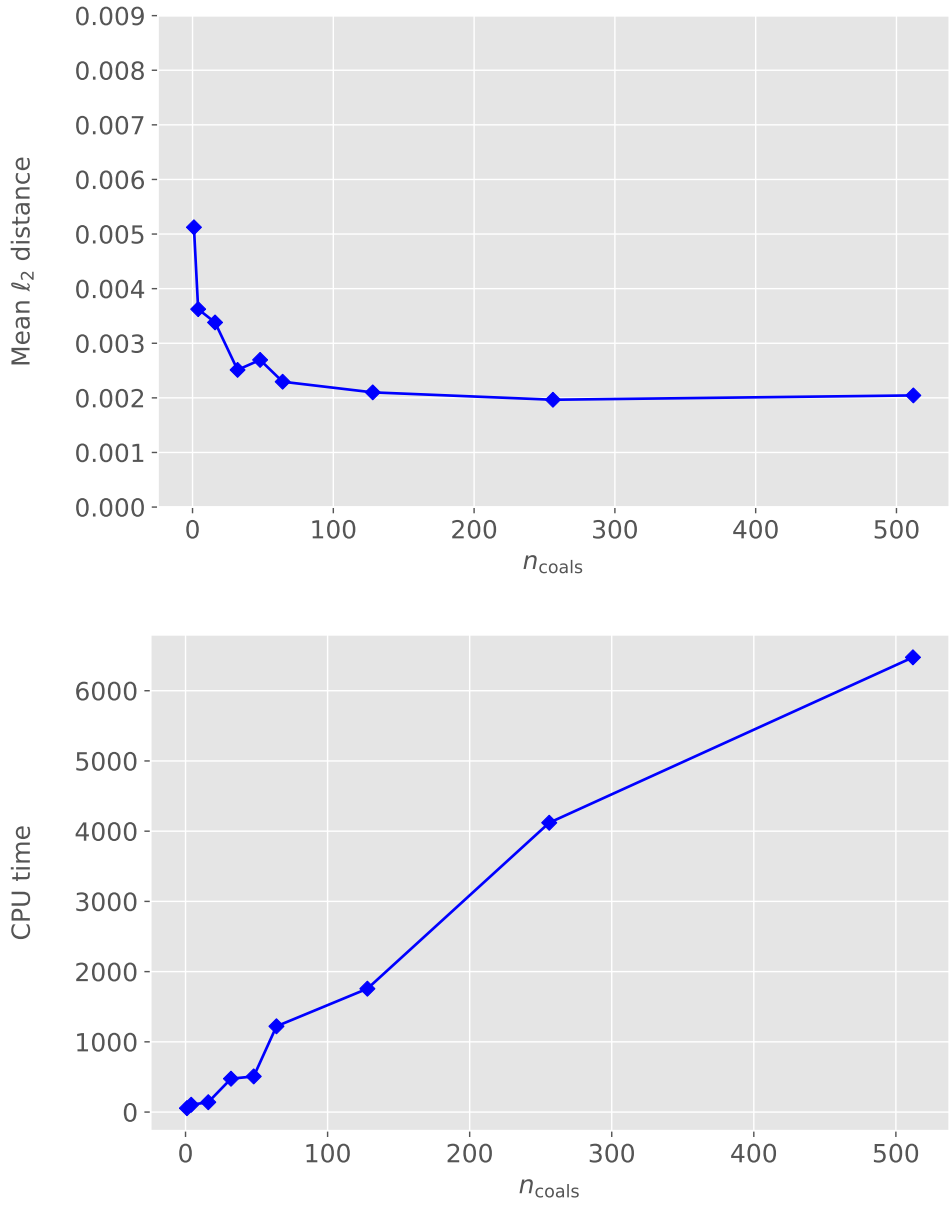


Figure 20: **Multivariate Normal, $\boldsymbol{\Sigma}_{\text{some\_corr}}$, $q = 16$ and $n_{\text{train}} = 6{,}400$.**
The upper panel shows the accuracy of the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ as a function of n_{coals}, while the lower panel shows the CPU time of the training of the model as a function of n_{coals}.

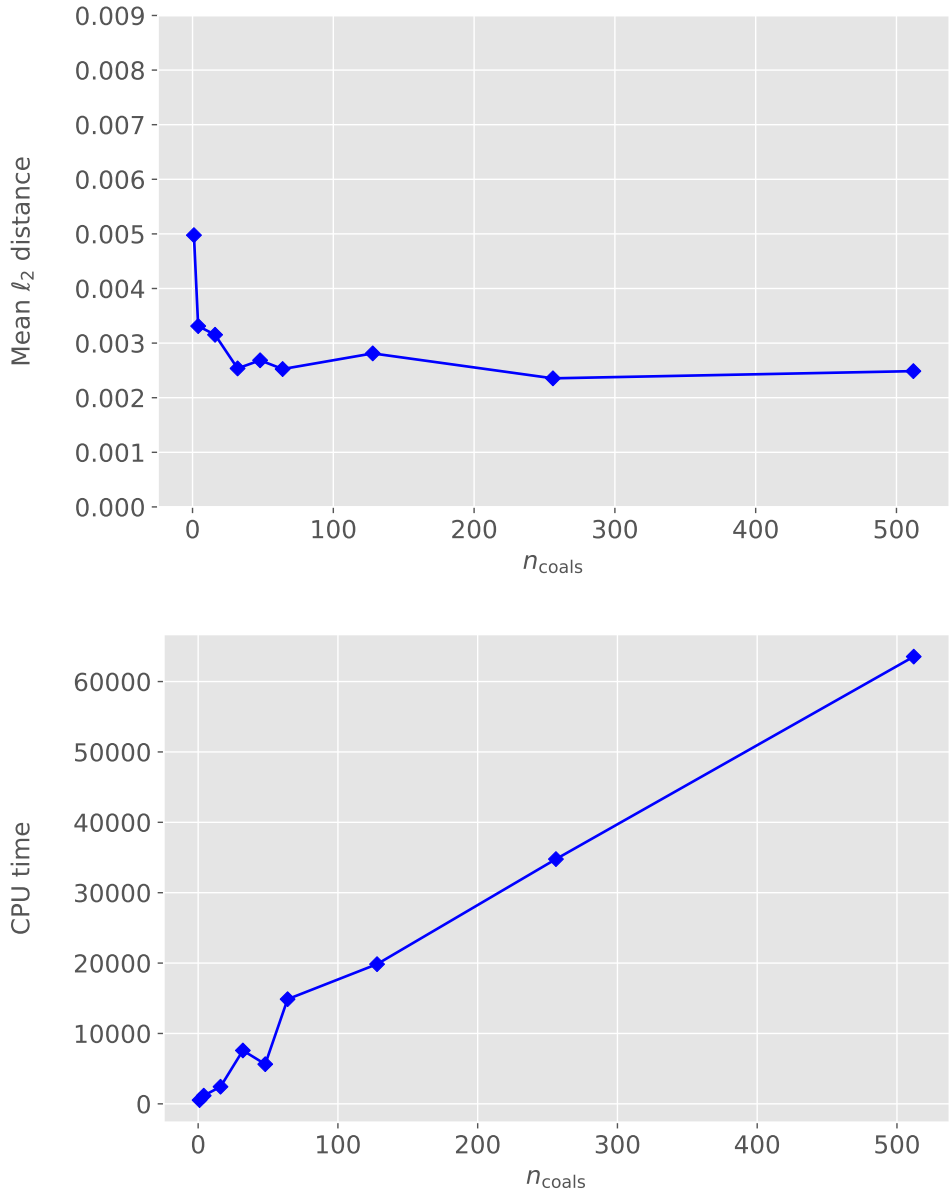


Figure 21: **Multivariate Normal,** $\Sigma_{\text{some\_corr}}$**,** $q = 16$ **and** $n_{\text{train}} = 64{,}000$**.**
At the top panel, the accuracy of the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ is shown as a function of n_{coals} for $n_{\text{train}} = 64{,}000$. In the lower plot, the corresponding CPU time of the training of the models is shown.

might be caused by elements of randomness in the FastSHAP model, e.g. the random initialization of the weights in the neural network. In [3], the surrogate models are retrained ten times, and the final results are the average of these models, which also provides uncertainty estimates and should stabilize the effect of such elements of randomness. Therefore, in practice, similar considerations may also be taken for the FastSHAP model. Since n_{coals} is a hyperparameter of the model, it can be tuned, and the models can be evaluated using e.g. cross-validation. As the lower plots of Figures 19, 20 and 21 show, the CPU time training the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ increases approximately linearly as a function of n_{coals}. The increased computation time must be weighed against the increase in the accuracy when determining the value of n_{coals}. Clearly, there is no benefit in increasing n_{coals} beyond the value for which the accuracy of the methods has converged, since this will only increase the computational cost of the method. In summary, based on our results, it seems that using $n_{\text{coals}} > 1$ is beneficial, e.g. $n_{\text{coals}} = 32$, which coincides with the results of Jethani et al. [4], where the FastSHAP method was introduced.

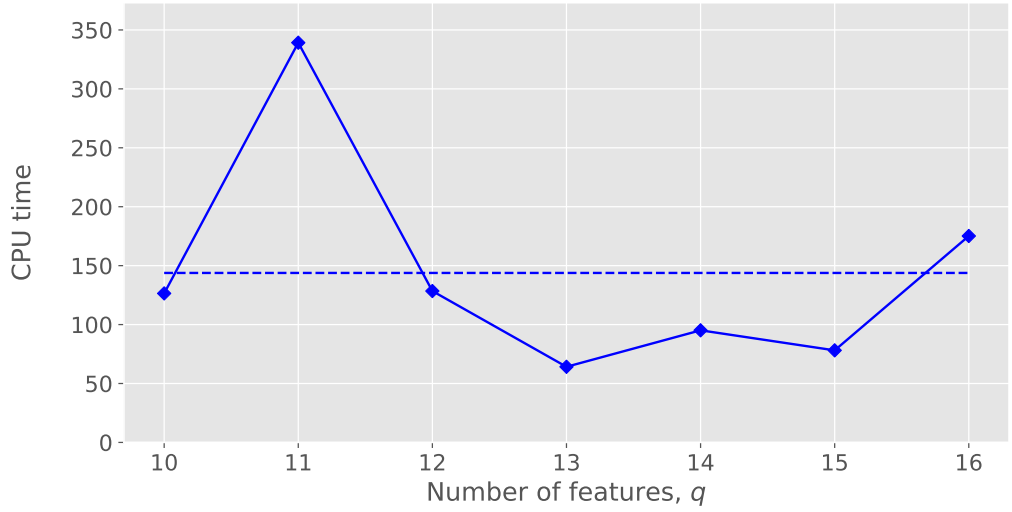
Because the exact Shapley values are often unavailable, one can use the loss function (36) to rank the models on a test set. Then, the choice of the hyperparameters of the FastSHAP model is similar to in a supervised machine learning task since the model's predictions can be evaluated using the loss function (36). The model's hyperparameters can be tuned by e.g. randomized grid searches and evaluated using e.g. cross-validation. However, this requires the user to have in-depth knowledge of machine learning since it is not included in the FastSHAP libraries in PyTorch and TensorFlow [4], meaning that the user would have to implement it themselves. In addition, such hyperparameter tuning is computationally costly.

4.5.6 Evaluation in Terms of Computational Cost

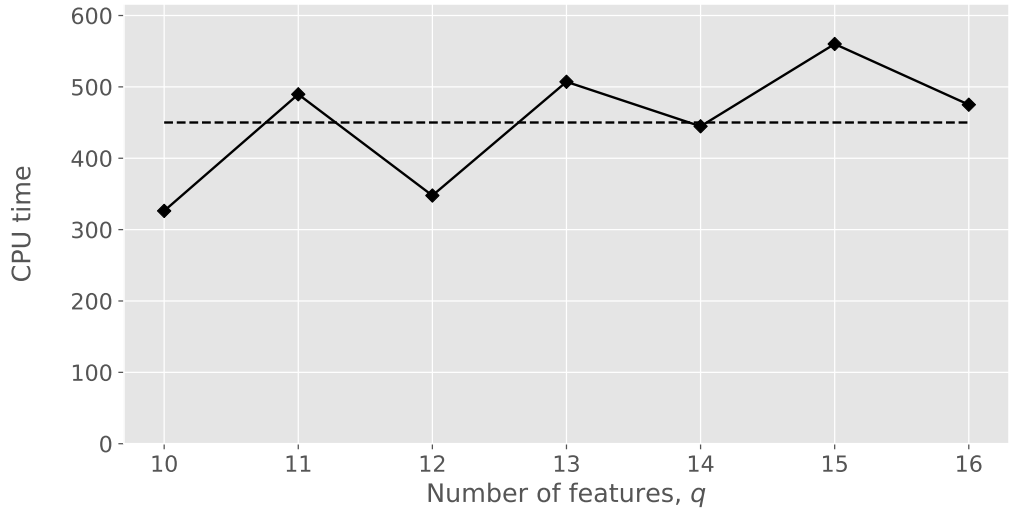
In this section, the computational cost of the KernelSHAP and FastSHAP estimation methods will be estimated by the CPU time. CPU time was briefly described in Section 3.7. This section discusses the computational cost of the Shapley value estimators, assuming that the contribution function, or an estimate thereof, is *given* for all test observations. In Section 3.7, we also outlined how the computation of the methods is performed and how it can be divided into an initialization step that only has to be performed once per black box model and an estimation step that must be repeated for every new instance that the black box model's prediction is to be explained.

It is assumed that the vector $\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*, y^*}$ containing the contribution function estimate for all coalitions $\mathcal{S} \in \mathcal{D}$ as defined in the KernelSHAP approximation (34) is given. Thus, to obtain the KernelSHAP estimate, only the matrix product $\boldsymbol{R}_{\mathcal{D}}\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*, y^*}$ in (34) must be computed for each instance of interest. Recall that in this product, only $\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*, y^*}$ depends on the instance of instance. The matrix $\boldsymbol{R}_{\mathcal{D}}$ can be precomputed and stored, and the cost of computing it is amortized over all instances to be explained. Also the computational cost of FastSHAP is divided into two steps. Firstly, the FastSHAP model must be trained according to Algorithm 3. The cost of training the FastSHAP model is amortized across the instances to be explained. After the initial training, the only additional cost related to providing an explanation for an instance is to evaluate the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ at the instance. In practice, it is important to distinguish the cost of initializing/training each estimation method from the cost of providing a new explanation, given that the method has been initialized/trained. This is because, in many practical situations, it is feasible to use a method with a high initial computational cost if the method can be evaluated fast after the initialization phase. In this section, all results are with multivariate normally distributed features with some correlation determined by the covariance matrix $\boldsymbol{\Sigma}_{\text{some\_corr}}$ defined in (49). The results for the other simulation models are, however, analogous.

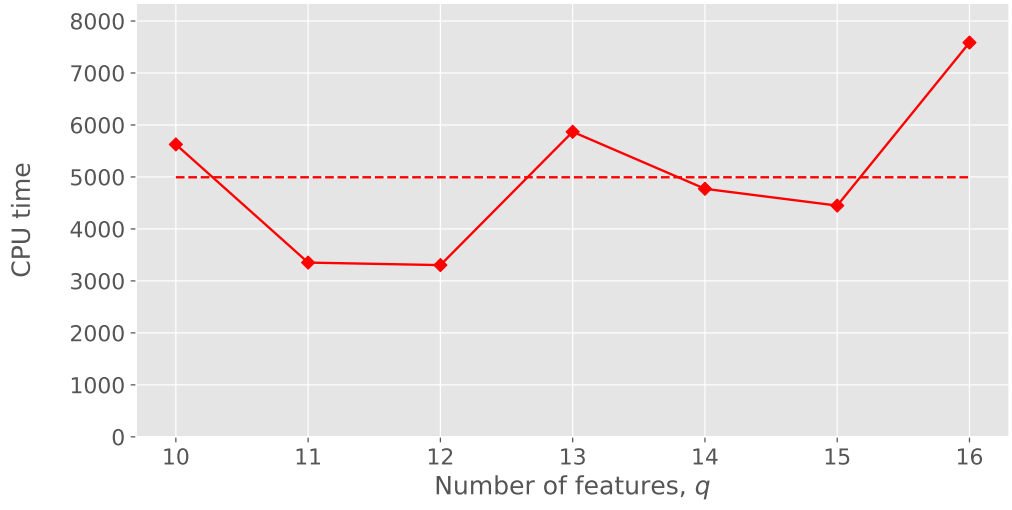
We start by discussing the initial computational cost of FastSHAP, which corresponds to training the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$ according to Algorithm 3. This cost is amortized over the instances and can be performed in a development phase. In Figure 22, the CPU time of training the FastSHAP32 model is plotted as a function of the number of features q in the model for n_{train} equal to 1,280, 6,400 and 64,000. The figure shows that the mean CPU time over the number of features increases as a function of n_{train}. The mean CPU time for the example with $n_{\text{train}} = 6{,}400$ is roughly three times larger than when n_{train} equals 1,280, corresponding to a fifth of the number of training observations. Comparing the cases with n_{train} equal to 6,400 and 64,000, we see that the



(a) $n_{\text{train}} = 1{,}280$.



(b) $n_{\text{train}} = 6{,}400$.



(c) $n_{\text{train}} = 64{,}000$.

Figure 22: **Multivariate Normal, $\boldsymbol{\Sigma}_{\text{some\_corr}}$, $n_{\text{coals}} = 32$.**
The CPU time of training the FastSHAP32 model as a function of the number of features q in the data set. The mean CPU time taken over the different values of q is shown in dashed lines in the plots.

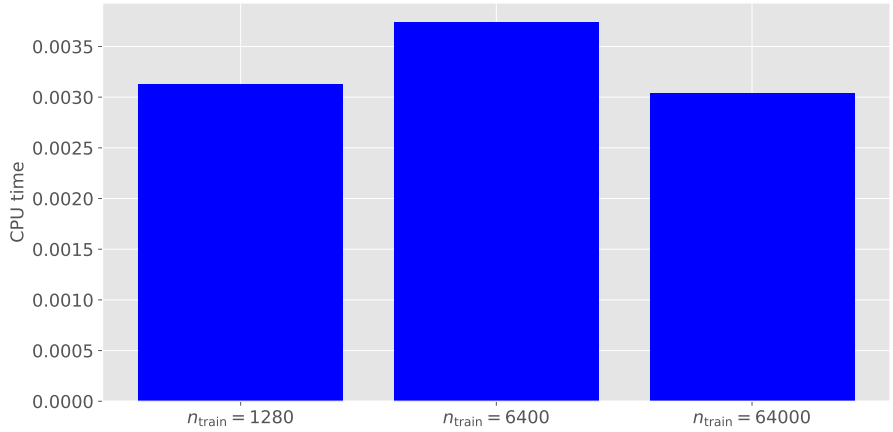


Figure 23: **Multivariate Normal, $\boldsymbol{\Sigma}_{\text{some\_corr}}$, $n_{\text{coals}} = 32$.**
The CPU time of providing a single explanation using FastSHAP32.

mean CPU time of training the FastSHAP32 models is approximately 11 times higher for the 10 times larger data set. Hence, the increase in CPU time is roughly linear in the number of training observations. We cannot expect a fully linear increase in CPU time caused by increasing n_{train} since there is some initialization etc. in the network that is unaffected by n_{train}. In addition, since we use early stopping and a learning rate reduction schedule in the training of the FastSHAP32 model, some variation must be expected in relation to the effect of the size of the training data set on the computational cost of training the model.

In order to compare the computational cost of providing a single explanation using FastSHAP and KernelSHAP, we include the evaluation time of the FastSHAP model. The evaluation time is computed as the average time the model takes to provide a prediction over 100 test observations. To stabilize the results, we redo this 1000 times and report the average value as the CPU time of the prediction for a single observation. The resulting CPU time is shown in Figure 23 for the simulation model with somewhat correlated normally distributed features with a total of $q = 16$ features for $n_{\text{train}} = 1{,}280$, 6,400 and 64,000. The CPU time corresponds to one model evaluation of the FastSHAP model with the hyperparameter n_{coals} equal to 32. Since the FastSHAP model is trained on different data sets when n_{train} differs, the resulting FastSHAP models will differ. Therefore, the time one model evaluation takes will differ somewhat for each case, as shown in the figure. However, the CPU times are very small compared with e.g. the time it takes to compute the off-manifold estimate for an instance of interest, as shown in Figure 6. In addition, compared to the CPU time of training the FastSHAP model, the time it takes to provide an additional explanation is very short. This demonstrates that FastSHAP can be used to provide explanations almost instantaneously after the training phase has been performed.

As previously mentioned, assuming that the vector containing the estimate of the contribution function $\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$ is given for all observations, the KernelSHAP estimate corresponds to computing the matrix-vector product $\boldsymbol{R}_{\mathcal{D}}\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$ in (34), where the matrix $\boldsymbol{R}_{\mathcal{D}}$ can be computed once and used for all the instances that will be explained. Therefore, as for FastSHAP, we divide the computational time into two phases. The first is to generate the matrix $\boldsymbol{R}_{\mathcal{D}}$ that will be used for all explanations. The CPU of generating this matrix is shown as a function of the number of feature coalitions $|\mathcal{D}| = 50, 60, 70, \ldots, 2{,}000$ in Figure 24. The matrix $\boldsymbol{R}_{\mathcal{D}}$ only depends on the number of features in the model q and the number of feature coalitions $|\mathcal{D}|$, not the number of training observations. Therefore, the initialization time is irrespective of the simulation model. To stabilize the result, we redo the computation 500 times and present the CPU time as the average of these. The figure shows that the CPU time of generating $\boldsymbol{R}_{\mathcal{D}}$ increases roughly linearly as a function of $|\mathcal{D}|$. In the figure, there are some deviations from this at $|\mathcal{D}|$ less than circa 250. Because the dimensions of the matrices and vectors in (34) increase as $|\mathcal{D}|$ increased, one should expect the computation of $\boldsymbol{R}_{\mathcal{D}}$ to be slower for increasing $|\mathcal{D}|$, since e.g. matrix inversion and multiplication generally is slower in higher dimensions. Therefore, we believe the deviations at $|\mathcal{D}|$ less than 250 are random fluctuations. Note that the initialization step is much faster than

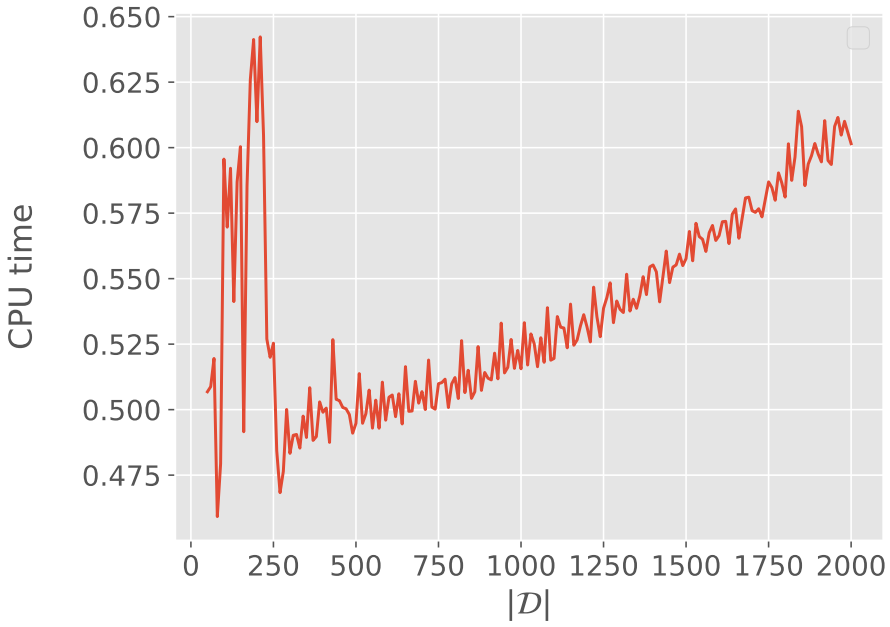


Figure 24: **Irrespective of the simulation model,** $q = 16$**.**
The CPU time of initializing the KernelSHAP method.

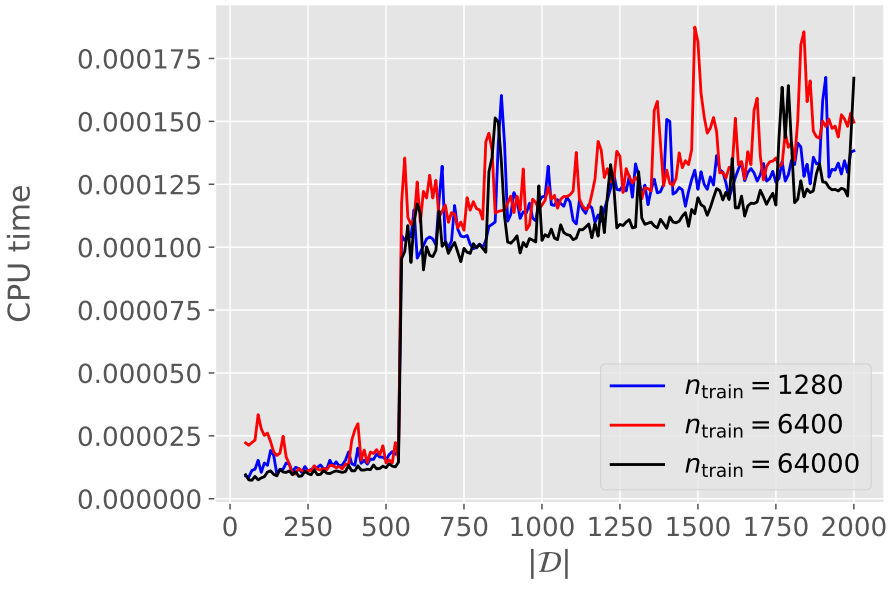


Figure 25: **Multivariate Normal,** $\boldsymbol{\Sigma}_{\text{some\_corr}}, q = 16$**.**
The CPU time of providing a single explanation using KernelSHAP as a function of the number of feature coalitions $|\mathcal{D}|$ in the KernelSHAP estimate (34).

training the FastSHAP model, and only amounts to less than a second of CPU time at worst.

After the matrix $\boldsymbol{R}_{\mathcal{D}}$ has been computed, all that remains to provide the KernelSHAP estimate is to compute the matrix-vector product $\boldsymbol{R}_{\mathcal{D}}\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$. To stabilize the CPU time, we repeat the computation of $\boldsymbol{R}_{\mathcal{D}}\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$ 1000 times for $|\mathcal{D}| = 50, 60, 70, \ldots, 2000$. This is done for 100 test observations. The resulting CPU time is shown in Figure 25. Because the KernelSHAP estimate corresponds to the matrix-vector product $\boldsymbol{R}_{\mathcal{D}}\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$, the computational time of the evaluation phase of the estimate will be determined by the speed of the library used to compute the matrix-vector product. In our experiments, we use the Python library NumPy [26]. As shown in the figure, the CPU time increases as a function of $|\mathcal{D}|$. Noticeably, at around $|\mathcal{D}| = 550$, there is a jump in the CPU time. Further investigations of why this occurs are omitted because it is outside the scope of this thesis since it is determined by the implementation of the matrix-vector product in the NumPy [26] library. For us, it is important to note that regardless of $|\mathcal{D}|$, the computation is fast. The figure shows that the CPU time of computing the KernelSHAP estimates (34) are around a tenth of the CPU time of evaluating the FastSHAP models, as shown in Figure 23. This is sensible since the FastSHAP model is a neural network, and its evaluation will correspond to a forward pass. In contrast, the KernelSHAP estimate is even simpler since it is only a matrix-vector product. In the figure, there is some difference in the CPU time for the different values of n_{train}. The computational time of matrix-vector inversions and products can generally be bounded analytically by the dimension of the matrices and vectors. Details are omitted because it is outside the scope of this thesis. In practice, it will depend e.g. on the number of zeros in the matrices and vectors. Therefore, since $\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$ depends on the simulation model and the matrix $\boldsymbol{R}_{\mathcal{D}}$ has been randomly generated for each simulation model[3], there may be a difference in the CPU time even though the dimensions of all the vectors and matrices are the same for all values of n_{train}.

Figures 23 and 25 show that the computational time of estimating the Shapley values after the initialization phase can be achieved very fast with both methods. Compared to the computational time of estimating the contribution function, it is clear that the off-manifold estimate has a much higher computational cost, as shown in Figure 6, than the estimation of the Shapley values when disregarding the initialization phase. In the full estimation procedure, including the estimation of both the contribution function and the Shapley values, the computational time will differ more based on which of KernelSHAP and FastSHAP is used in the last step. The computation time of providing an explanation with the FastSHAP-Off-Manifold method will be distributed in a similar way as the computation time of FastSHAP-Surrogate shown in Figure 23. This is because, in the full procedure, when using FastSHAP, the contribution function estimation is "built into" the FastSHAP model, and its computational complexity is only relevant during the training of the model. Providing a new explanation will always correspond to a single evaluation of the FastSHAP model $\boldsymbol{\phi}^{\text{fast}}(\boldsymbol{x}, y; \boldsymbol{\theta})$, regardless of the estimator of the contribution function. The training time, however, will be severely longer if the off-manifold estimate is used because one off-manifold estimate (39) must be computed per coalition $\mathcal{S}$ per batch. Meaning that in the duration of an epoch, $n_{\text{train}} \cdot n_{\text{coals}}$ off-manifold estimates must be computed. Although we have not included results demonstrating this, we have verified it using the current FastSHAP PyTorch library [4], where the FastSHAP-Off-Manifold method is implemented.

In the full estimation procedure, the computational time of calculating the full KernelSHAP estimates will correspond to the time it takes to estimate the contribution function for all $|\mathcal{D}|$ coalitions in (34), in addition to computing the matrix-vector product $\boldsymbol{R}_{\mathcal{D}}\boldsymbol{v}^{\mathcal{D}}_{\boldsymbol{x}^*,y^*}$. The computation of the contribution function must be repeated for every instance to be explained. Since the computation of the off-manifold estimate of the contribution function is very high compared to the surrogate model and the methods in the second estimation step, the computation time of KernelSHAP-Off-Manifold method is significantly higher than the other methods, when disregarding the initialization/training phase. In order to facilitate a faster KernelSHAP approximation than the original version [2], one might replace the off-manifold estimate with the surrogate model. This would exploit that the accuracy of KernelSHAP is significantly higher than FastSHAP's when $|\mathcal{D}| \gtrapprox 500$ in KernelSHAP, as shown in almost all examples in Figures 12 to 16. However, in some cases, the

[3] We are using a global seed in the random generation to ensure that the results are reproducible. In the simulation we are basing the CPU time on, we first simulate the training observations. Therefore, a different number of random numbers have been generated before the matrix $\boldsymbol{R}_{\mathcal{D}}$ is generated, which means that $\boldsymbol{R}_{\mathcal{D}}$ will generally differ in the three simulations. However, we could have used the exact same matrix in the three cases.

off-manifold estimate is more accurate than the surrogate model's estimate, which one would have to take into account.

4.5.7 Other Results

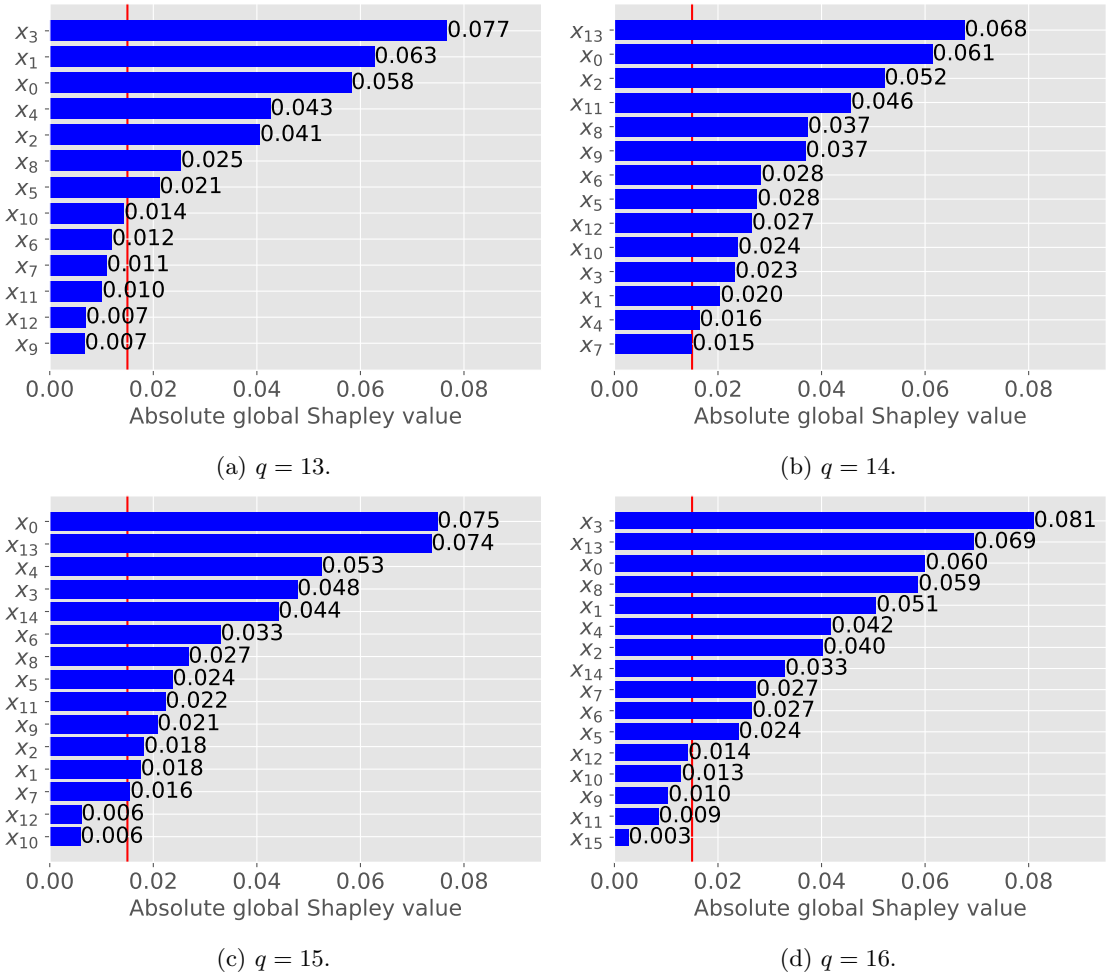
Recall that in the examples with highly correlated normally distributed features shown in Figure 14, the error of the KernelSHAP estimate is significantly higher when the number of features q equals 14 than for all other values of q, for all values of the number of feature coalitions $|\mathcal{D}|$ and all values of the number of training observations n_{train}. Also compared to the examples with independent and somewhat correlated normally distributed features, shown in Figures 12 and 13, and the examples with Burr distributed features, shown in Figures 15 and 16, the example with highly correlated features with $q = 14$ stands out. The KernelSHAP estimate has a higher error in this case, and it is interesting to determine the cause. The error of the FastSHAP estimate is similar to that in the other cases, and the method does not seem to be performing differently in this case.

When comparing the examples in Figure 14, it is important to recall that for each combination of n_{train} and q, the training data set differs. Thus, the black box and surrogate models are different in each case. This can lead to differences in the true exact Shapley values and errors that propagate differently because the surrogate models are different. However, the features are simulated according to the same simulation model, as described in Section 4.3.1, and the response variables are determined by (47) in all cases. Therefore, it is reasonable to expect some similarity between the black box models and surrogate models when the number of features is the same, even when the number of training observations differs. In the Shapley value estimation, for every instance of interest, we are trying to determine how each feature in the model affects the prediction of the black box model. Thus, not only the empirical properties of the data set are relevant to the Shapley value estimation problem, but also the properties of the black box model.

Doumard et al. [11] find that the error of KernelSHAP does not generally increase with the number of features in the model, which "is probably due to the fact that usually, the more features there are, the less influence amplitude each individual feature has in the prediction". Seemingly, it is easier for KernelSHAP to pick up when a feature has almost no effect on the model's predictions. Recall that KernelSHAP only considers a subset of size $|\mathcal{D}|$ of the total number of feature coalitions 2^q. If $|\mathcal{D}|$ is fixed, one would intuitively expect the KernelSHAP approximation to have a higher error as q increases because the proportion of feature coalitions considered decreases. However, keeping the results of Doumard et al. [11] in mind, it is perhaps more correct to expect the approximation error to increase when the number of *significant* features in the model increases.

To investigate whether this is the case, it is most interesting to consider the cases where the number of feature coalitions $|\mathcal{D}|$ in KernelSHAP is much lower than the total number of coalitions 2^q because these are the cases where the reduction in the computational cost of using KernelSHAP (34) compared to the exact solution (33) is most significant. Therefore, we compare the "outlier" examples with $q = 14$ to the examples with $q = 13, 15,$ and 16, shown in Figure 14. To provide an explanation of the overall workings of the black box model, we consider the global Shapley values, described in Section 3.8. We estimate the global Shapley values by taking the average of the absolute ground truth Shapley values (33), with the surrogate model as contribution function estimator, over 100 test observations. The surrogate model must be used to replicate the experiment in Figure 14 since this was the estimator we used there.

The global Shapley values of the black box models with $q = 13, 14, 15$ and 16 features are shown in Figures 26, 27 and 28 for n_{train} equal to 1,280, 6,400 and 64,000, respectively. The global Shapley values are sorted in decreasing order in all the plots. For each value of n_{train}, we indicate the smallest absolute global Shapley value in the case with $q = 14$ by a red line. This red line is also shown in the plots for $q = 13, 15,$ and 16. Although it is difficult to determine a threshold value for when a feature is significant to the model, we use the smallest global Shapley values for the cases with $q = 14$ as a threshold. Since all the black box models are classification models and the predictions are the predicted probabilities of belonging to a class, and therefore, in the range (0,1), the Shapley values of each model are on the same scale, and hence, comparable. Noticeably, for



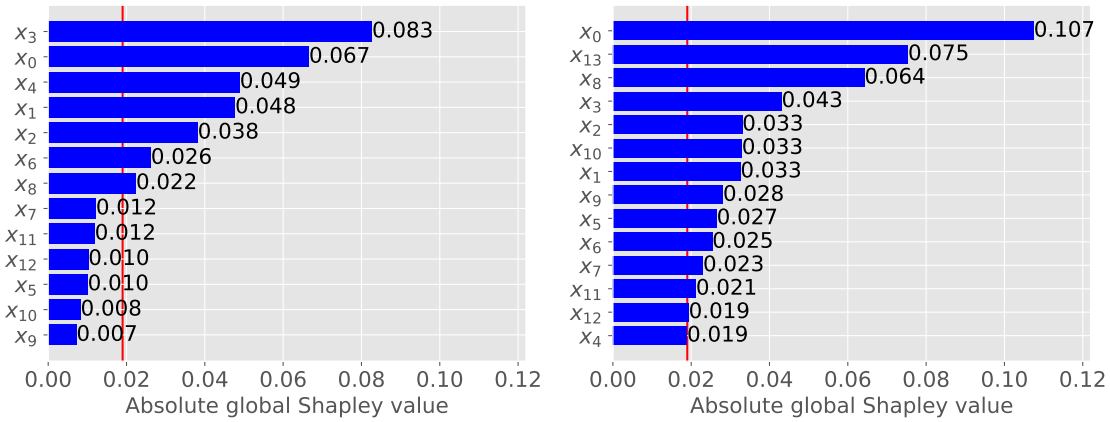
(a) $q = 13$.

(b) $q = 14$.

(c) $q = 15$.

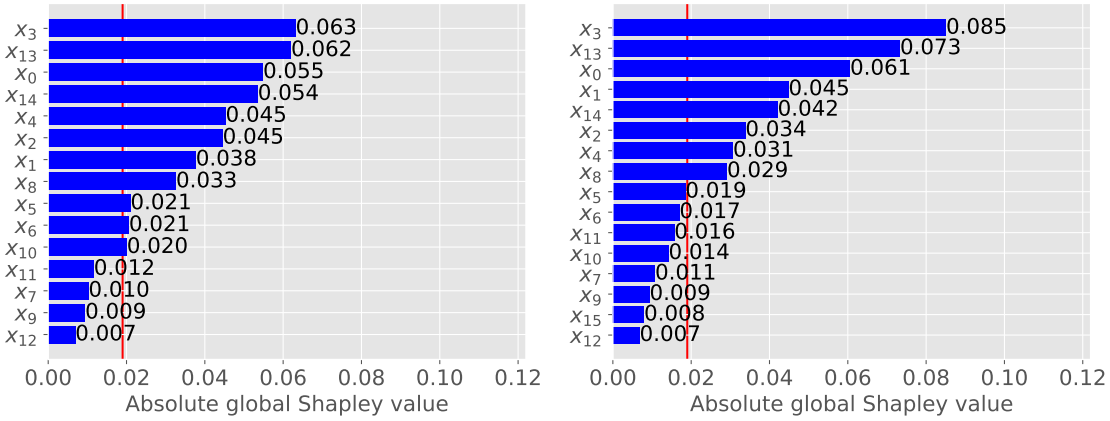
(d) $q = 16$.

Figure 26: **Multivariate Normal, $\boldsymbol{\Sigma}_{\text{high\_corr}}, n_{\text{train}} = 1{,}280$.**
The global absolute ground truth Shapley values of the black box model used to get the results shown in Figure 14 for $q = 13, 14, 15$ and 16.



(a) $q = 13$.

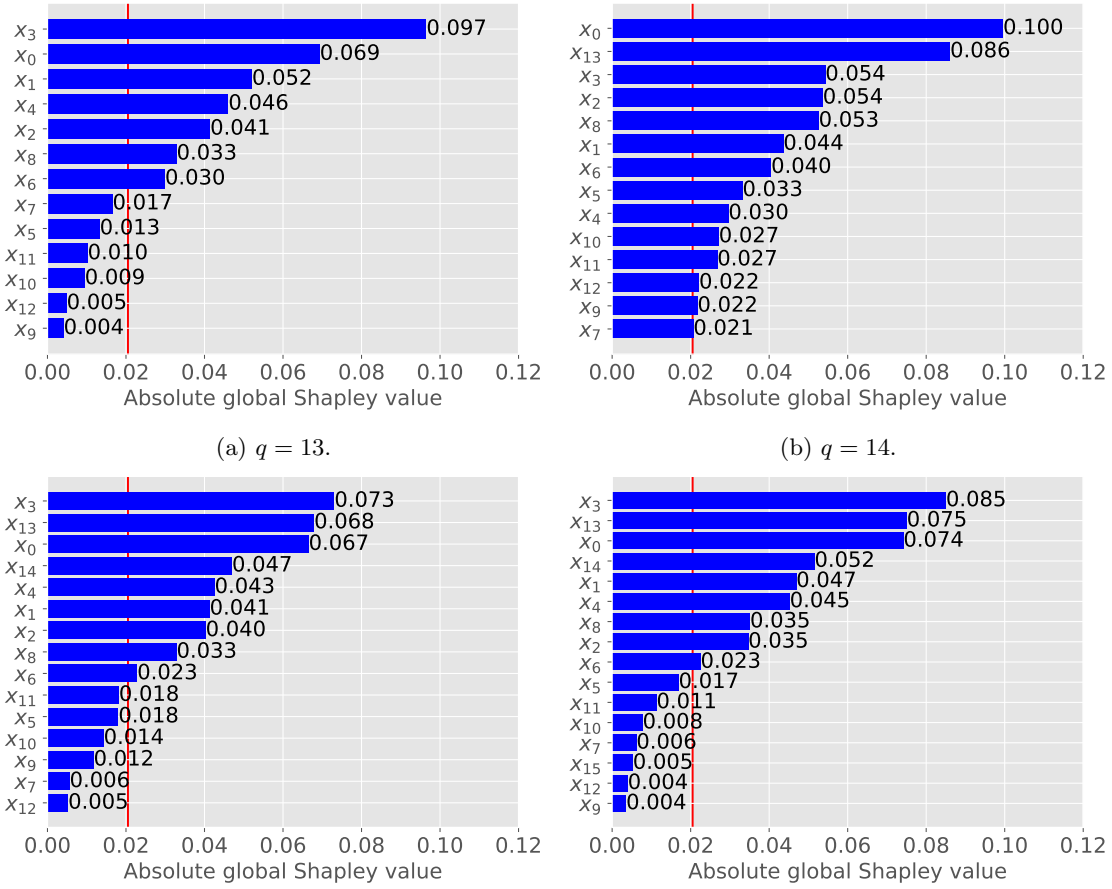
(b) $q = 14$.



(c) $q = 15$.

(d) $q = 16$.

Figure 27: **Multivariate Normal, $\boldsymbol{\Sigma}_{\text{high\_corr}}, n_{\text{train}} = 6{,}400$.**
The global absolute ground truth Shapley values with $n_{\text{train}} = 6{,}400$ for $q = 13, 14, 15$ and 16.



(a) $q = 13$.

(b) $q = 14$.

(c) $q = 15$.

(d) $q = 16$.

Figure 28: **Multivariate Normal, $\boldsymbol{\Sigma}_{\text{high\_corr}}, n_{\text{train}} = 64{,}000$.**
The global absolute ground truth Shapley values of the black box model with highly correlated multivariate normally distributed features for $q = 13, 14, 15$ and 16.

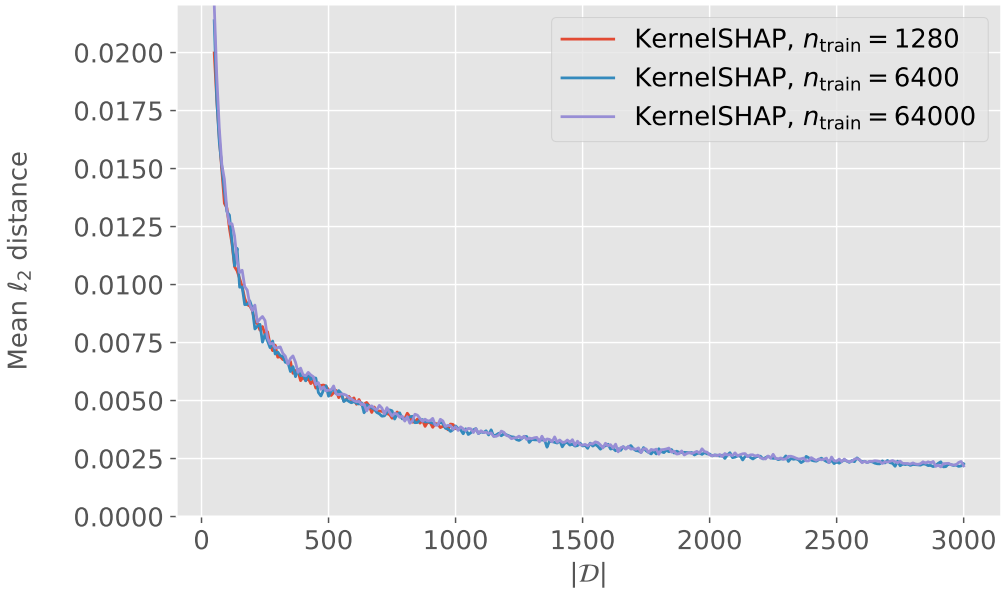


Figure 29: **Multivariate Normal,** $\Sigma_{\text{high\_corr}}$**,** $q = 14$ **.**
The error in the KernelSHAP estimate is plotted as a function of the number of coalitions $|\mathcal{D}|$ for the "outlier examples" in Figure 14.

all values of n_{train} and $q = 13, 15$, and 16, several of the smallest global absolute Shapley values are below the threshold value indicated by the red line. Thus, several features are assigned less importance, by a significant margin, than the least important feature in the case with $q = 14$. Hence, although the case with $q = 14$ is not the case with most features in the model, it can be argued that it is the case with the highest number of significant features. This suggests that to provide high accuracy estimates, KernelSHAP needs to consider a higher number of feature coalitions for black box models with a higher number of significant features. FastSHAP does not seem to be affected by this, at least not in our experiments.

The observant reader may have noticed that the highest value of the number of feature coalitions $|\mathcal{D}|$ used in the examples in Figure 14 is $|\mathcal{D}|$ equal to 1000. For the training data sets with n_{train} equal to 6,400 and 64,000 in the case with $q = 14$ in Figure 14, the best FastSHAP model is competitive to KernelSHAP1000. As previously shown in the example in Figure 18, the error of the KernelSHAP estimate has not fully converged for this value. However, as Figure 29 shows, when considering $|\mathcal{D}| > 1000$, KernelSHAP outperforms FastSHAP also in the cases with $q = 14$. The figure shows that the estimate of KernelSHAP reaches an error of slightly below 0.0025 when $|\mathcal{D}| = 3000$ for all values of n_{train}. On the other hand, the error of the best FastSHAP model is around 0.006, 0.004, and 0.003 when n_{train} equals 1,280, 6,400 and 64,000, respectively. Recall that the computational cost of KernelSHAP increases as a function of $|\mathcal{D}|$. In the full estimation procedure, if KernelSHAP-Off-Manifold is used, the cost increases drastically when $|\mathcal{D}|$ increases because it is necessary to compute the values of the contribution function for all $|\mathcal{D}|$ feature coalitions. Therefore, the computation of the KernelSHAP estimate with e.g. 3000 feature coalitions will be expensive and must be weighed against the increase in accuracy. In the example with $q = 14$, since the cost of FastSHAP, disregarding the model training, is much lower and the accuracy is similar to that of KernelSHAP1000, one might prefer FastSHAP in this case. This may indicate that FastSHAP is more competitive compared to KernelSHAP when the number of *significant* features in the model is higher.

5 A Real-World Data Experiment

In addition to the experiments in the simulation study presented in the previous chapter, we have performed an evaluation of the estimators for a real-world data set. Firstly, we will describe the data set in Section 5.1 and the black box model in Section 5.2. The contribution function (29) is unknown in this case, and unlike in the simulation study, we cannot provide an accurate, unbiased estimate in a reasonable amount of time. Therefore, in Section 5.3, an evaluation metric that does not require access to the true values of the contribution function is introduced. It can be used to rank the estimates of the contribution function. Based on this metric and the CPU time of the contribution function estimators, the results of evaluating the contribution function estimators for the real-world data set are presented in Section 5.4. Then, in Section 5.5, the Shapley value estimators are evaluated. The majority of Sections 5.1 and 5.2 are from my specialization project [12].

5.1 The "Adult Data" Set

To illustrate the use of the explanation methods on a real-world data set, the data set "Adult Data", which can be found here [38], will be used. The data set is also known as the "Census Income" data set. The data set contains 12 features[4] and a response variable. The data set is split into a training set and a test set consisting of 32,561 and 16,281 instances, respectively. There are some missing values in the data set. After removing rows, corresponding to instances, containing missing values, there are 30,162 instances left in the training set and 15,060 in the test set. Only a small proportion of the data is removed when deleting the instances with missing values. Therefore, it is assumed that the model's accuracy will not be significantly worse because of the use of fewer data points. To facilitate the learning of the neural networks in the surrogate and FastSHAP models, we further split the training data set into a training and a validation data set. Then there are 24,129 observations in the training data set and 6,033 observations in the validation data set.

The response variable is categorical with two categories. Hence, this is a classification problem. The classes are whether the adult earned more than 50,000 $ per year or less than or equal to 50,000 $ per year. These groups are referred to as high-earners and low-earners, respectively. The data is from the US census in 1994. The data set is unbalanced, with circa 25 % of the observations in the high-earner group (after deleting the rows with missing values). According to common practice, the majority class is encoded '0', which is the group of low-earners, and the minority class is encoded '1', which is the group of high-earners. There are five numerical and seven categorical features in the data set. The numerical features are `age`, `education-num`, `capital-gain`, `capital-loss` and `hours-per-week`. The categorical features are `workclass`, `marital-status`, `occupation`, `relationship`, `race`, `sex` and `native-country`.

5.2 The Black Box Model

In the experiments, the black box model is an XGBoost classifier. The model classifies instances to belong to class 1 if the predicted probability of belonging to that class exceeds 0.5. If the predicted probability is predicted to be below 0.5, the instance is classified as class 0. The model obtains an AUC score of circa 0.91 and an AP/AUPCR [39] score of around 0.80 on the test data set. The focus of this thesis is to evaluate the contribution function and Shapley value estimators. Therefore, it has not been a focus to tune the hyperparameters in the XGBoost model. To illustrate the use of the explanation methods, it is sufficient to have a black box model that performs relatively well. Therefore, in the XGBoost model, the default parameter choices have been used. XGBoost is known to be quite robust against overfitting, and it is, therefore, not essential to tune its hyperparameters to get a model that performs relatively well. The AUC and

[4] In the original "Adult Data" set [38], there are 14 features, but according to common practice, two of the features have been removed, this is also done in e.g. [4]. It is outside the scope of this thesis to find the best possible model for the "Adult Data" set. Therefore, we omit results related to how the removal of these features affects the performance of the model.

AP/AUPCR scores we obtained on the test data set also indicate that the model performs quite well.

The XGBoost package in Python [16] has many hyperparameters and we will not go into detail on all of them here. In the theoretical description of XGBoost in Section 2.2.1, an outline of the method was given. The full XGBoost method is more flexible and has more functionality. Thus, not all the hyperparameters in XGBoost were introduced in Section 2.2.1.1. Again, since the aim of this thesis is to investigate the contribution function and Shapley value estimators, not the XGBoost model, we refer to the original paper on XGBoost [16] and the documentation of the XGBoost package for more details. For reproducibility, we still list the most important hyperparameters in the XGBoost model we use. In the default settings, the most important hyperparameters in the model are as follows, the number of boosted trees `n_estimators` is set to 100, the learning rate `eta` is set to 0.3, the maximum depth `max_depth` of each tree is set to 6, the L2-regularization penalty parameter λ is set to 1, the minimum loss reduction required to proceed with a new partition on a leaf of the tree `gamma` is set to 0, i.e. the model will keep building trees even if there was no improvement in the last step, and `min_child_weight` determining the minimum weight needed inside a child to keep partitioning is set to 1, this parameter corresponds to the minimum number of instances needed in each node if linear regression mode is used. To handle categorical variables, some of the hyperparameters in the model must be specified. Therefore, in the XGBoost model, the hyperparameter `tree_method` set to `hist` and the hyperparameter `enable_categorical` set to `True`. How XGBoost handles categorical variables was briefly outlined in Section 2.2.1.1. For the interested reader, we refer to the documentation of the XGBoost Python package [16] for more details on the hyperparameters.

5.3 Evaluation Metric

For the "Adult Data" set, the values of the contribution function (29) are generally unknown. Therefore, the evaluation metric (45) cannot be used because it requires access to the true value of the contribution function or an unbiased, accurate estimate thereof. However, in accordance with Frye et al. [3] and Olsen et al. [40], we introduce a metric that can be used to rank two estimates of the contribution function, that only requires access to the predictions of the black box model $\hat{f}(\boldsymbol{x}_i^*)$ and the contribution function estimates $\hat{\boldsymbol{v}}_{\boldsymbol{x}_i^*, y_i^*}$. Here, $(\boldsymbol{x}^*, y^*)$ is the instance of interest with $i \in \{1, 2, \ldots, n\}$, where n is the number of test observations for which the contribution function is estimated. The metric originates from the contribution function being the minimizer of the loss (41), which is equivalent to minimizing (43), with minimizer (44). Therefore, to get an estimate of the relative performance of the estimators, we can approximate the expectation $\mathbb{E}_{p(\boldsymbol{x})}$ with considering n observations from the test set drawn according to the empirical distribution in the test set. Moreover, we approximate the expectation $\mathbb{E}_{p(\mathcal{S})}$ by considering all 2^q possible feature coalitions. Therefore, Frye et al. [3] and Olsen et al. [40] propose to use the mean squared error (MSE) over n test observations and 2^q coalitions as the evaluation metric. As argued in Section 3.5.2, the minimizer of both (41) and the expectation of the MSE over $p(\boldsymbol{x})$ and $p(\mathcal{S})$ is (44). Therefore, both these metrics are valid choices. We choose to use the MSE, as in [3, 40].[5] Correspondingly, the evaluation metric we introduce is

$$\text{EM}_3 = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2^q} \sum_{\mathcal{S} \in \mathcal{Q}} \left[\hat{f}(\boldsymbol{x}_i^*) - \hat{\boldsymbol{v}}_{\boldsymbol{x}_i^*, y_i^*}(\mathcal{S}) \right]^2, \tag{50}$$

where $\mathcal{Q}$ denotes the set of all 2^q coalitions of the q features.

For the Shapley values, since there are only $q = 12$ features in the "Adult Data" set, the exact Shapley values (33) can be computed if the contribution function is given. Therefore, the evaluation metric (46) can also be used for the "Adult Data" set. Note that by "exact Shapley values", we refer to the exact minimizer (33) given an estimate of the contribution function (29). Since we cannot provide a ground truth value of the contribution function for the "Adult Data" set, the exact Shapley values are not truly exact because (29) is unknown. However, as previously mentioned,

[5] Note that in our experiments, the ranking of the methods is found to be the same irrespective of the choice of metric.

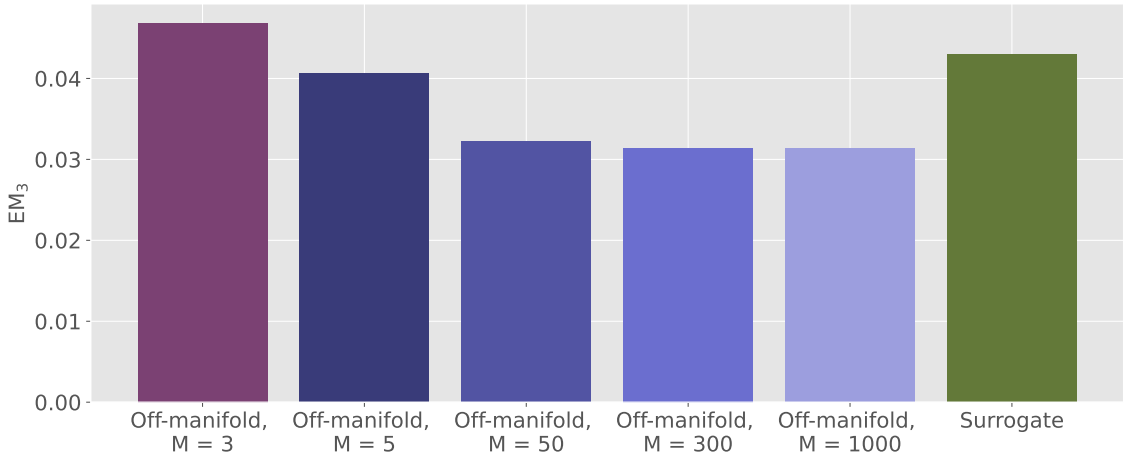


Figure 30: The error of the contribution function estimates for the "Adult Data" set.

Jethani et al. [4] find that the relative performance of the Shapley value estimators is similar for different choices of contribution function estimators when the same estimator is used in all Shapley value estimates and the ground truth. Therefore, we stick to this convention and evaluate the Shapley values for a given, fixed estimate of the contribution function as in the simulation study.

5.4 Evaluation of the Contribution Function Estimators

The error according to the evaluation metric (50) of the off-manifold estimate (39) with $M = 3, 5, 50, 300$, and 1,000 Monte Carlo samples, and the error of the estimate of the surrogate model, trained according to Algorithm 4, are shown in Figure 30. The figure shows that the off-manifold estimate outperforms the surrogate model when $M \geq 5$ per feature coalition $\mathcal{S}$, which is a quite low number of Monte Carlo samples M. The numerical features in the "Adult Data" set have an average absolute empirical linear correlation of around 0.08, with absolute correlations between 0.03 and 0.15. It is not obvious how to compute correlations between pairs of categorical variables and between a numerical and a categorical variable. Hence, we do not compute such correlations. Based on the low values of the correlation of the numerical features, we compare the results with the simulation examples with some linear correlation between the features.

In the simulation study, for the case with the normal distribution with somewhat correlated features, the surrogate model outperformed the off-manifold estimate for all values of n_{train} and all values of the number of Monte Carlo samples M. The results of this example are shown in Figure 7. In addition, we found that with Burr distributed features with an average empirical correlation of between 0.11 and 0.12, shown in Figure 10, the surrogate model only outperformed the off-manifold estimate in one case, where n_{train} was equal to 128,000 and the off-manifold estimate used 100 samples. In the "Adult Data" set, there are 24,129 training observations after splitting into a training, validation, and test set. Therefore, when comparing with the example with the Burr distribution in Figure 10, it seems reasonable that the off-manifold method outperforms the surrogate model when $M \geq 5$ because of the relatively small number of training observations in the "Adult Data" set. If we assume that the data in the "Adult Data" set arises from a distribution that is more difficult to learn than data generated from the normal distribution, which will often be the case for real-world data, it seems reasonable that the surrogate model performs worse for the "Adult Data" set than the data set with somewhat correlated multivariate normal features, which is shown in Figure 7.

In summary, we believe that these results align with the results for the simulated data and that for somewhat correlated features, the biased off-manifold estimate can provide more accurate results unless the training data set is sufficiently large. If, in addition, the data arises from a distribution that is more difficult to learn, the accuracy of the off-manifold estimate is better relative to the

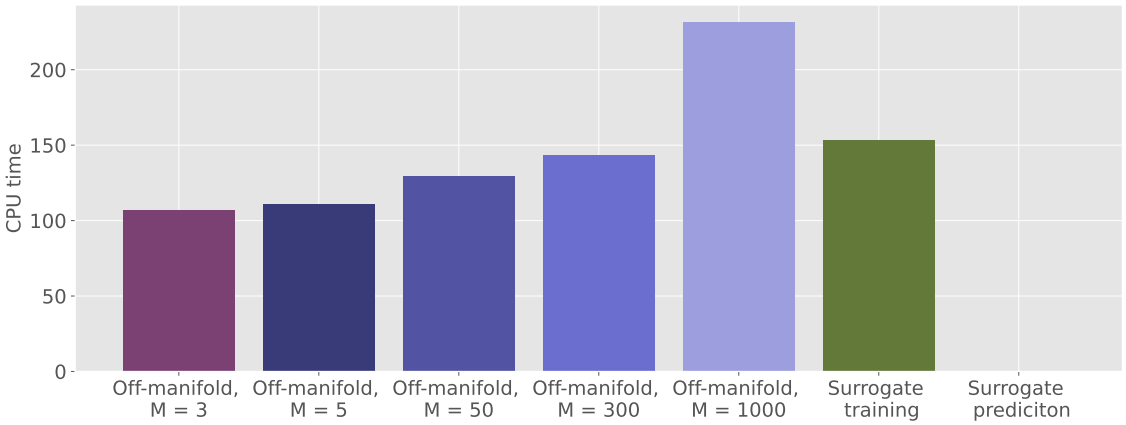


Figure 31: The CPU time of computing the contribution function estimates for the "Adult Data" set.

surrogate model. However, the surrogate model's hyperparameters have not been tuned. By tuning the hyperparameters and architecture of the surrogate model, it may be possible to improve the performance of the surrogate model. However, this is not implemented in the current versions of the FastSHAP library [4]. Therefore, incorporating this requires the user to have in-depth domain knowledge. We have used the same hyperparameter values in our surrogate model as Jethani et al. [4] to replicate their results.

In the results presented in the original paper describing the surrogate model [3], the surrogate model outperforms the off-manifold estimate for the "Adult Data" set. In our experiments, the relative performance is the opposite when the number of Monte Carlo samples M in the off-manifold estimate is greater than or equal to 5, as shown in Figure 30. However, it is unclear how many Monte Carlo samples are used to compute the off-manifold estimate for the "Adult Data" set in [3]. Therefore, if Frye et al. [3] use fewer Monte Carlo samples, this can explain why our results differ in terms of the relative performance of the methods. As shown in Figure 30, the surrogate model is more accurate than the off-manifold estimate with $M = 3$ Monte Carlo samples per feature coalition. In addition, in the surrogate model, we use the same hyperparameters and architecture as Jethani et al. [4], which differs from the surrogate model in [3]. This means that our surrogate model is not equivalent, which can explain why the accuracy differs.

Lastly, the CPU time of estimating the contribution function is shown in Figure 31 for the different estimators. In the figure, the off-manifold estimates' CPU time is the average time of calculating one estimate of the off-manifold estimate (39) for all coalitions $\mathcal{S} \in \mathcal{Q}$, where $\mathcal{Q}$ denotes the set of all 2^q coalitions of the q features. The average is taken over 100 test observations. This CPU time measures the cost related to the computation that has to be repeated for every instance of interest to be explained. For the surrogate model, the CPU time of the model's training is shown. The cost of training the surrogate model is an initial cost that does not have to be repeated and is amortized over the instances to be explained. The cost corresponding to estimating the contribution function for all coalitions $\mathcal{S} \in \mathcal{Q}$ corresponds to evaluating the surrogate model $|\mathcal{Q}|$ times. The CPU time of evaluating the surrogate model $|\mathcal{Q}|$ time is calculated for 100 test observations, and the average CPU time is shown in Figure 31, where it is denoted "surrogate prediction".

The figure shows that the surrogate prediction CPU time, which is so small it is not visible in the bar chart, is negligible compared to the CPU time of computing the off-manifold estimates for all values of the number of Monte Carlo samples M. Moreover, the training time of the surrogate model is of a similar order of magnitude as computing the off-manifold estimate for a single instance of interest when $M \approx 300$. Therefore, the surrogate model will provide explanations faster even if only a few instances are to be explained and much faster if many instances are to be explained. Thus, if computational speed is of the essence, one might prefer the surrogate model, although it has a higher error than the off-manifold estimates with $M \geq 5$. If the hyperparameters of the surrogate model are tuned, and we include the full computation time of this in the initial training phase of the surrogate model, this initial computation time will significantly increase. However,

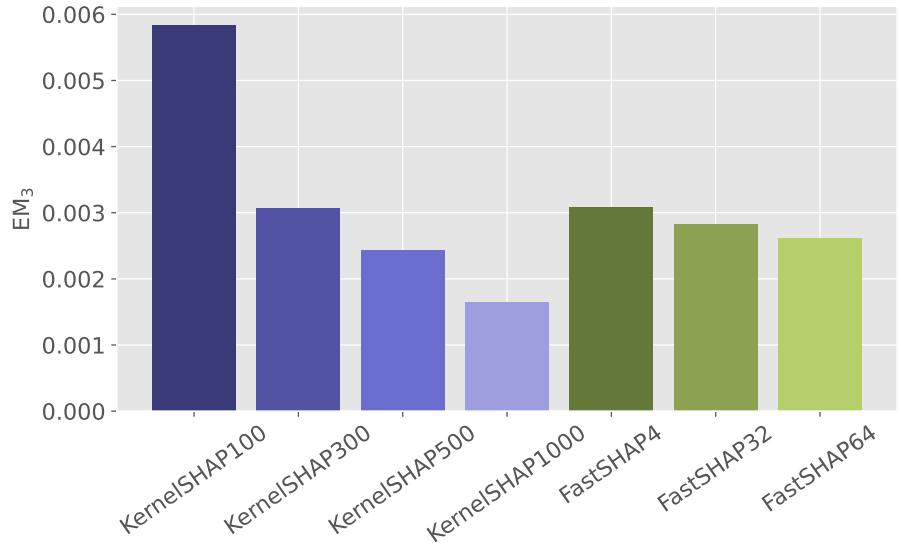


Figure 32: The error of the Shapley value estimates for the "Adult Data" set.

the cost amortizes over the instances that are explained and can be performed in an initial phase, which can be feasible in practice.

5.5 Evaluation of the Shapley Values Estimators

In this section, the results of evaluating the Shapley value estimators for the "Adult Data" set are presented. As previously mentioned, Jethani et al. [4] find that the relative accuracy of the Shapley value estimators is the same, regardless of the choice of contribution function estimators, when the same estimator is used in the ground truth and Shapley value estimates. Accordingly, the estimates of the contribution function values used in the computation of the ground truth Shapley values (33) and all estimates are the predictions of the surrogate model. This also makes it easier to compare the results with the simulation study where the same choice was made. The KernelSHAP estimate (34) is computed with the number of feature coalitions $|\mathcal{D}|$ equal to 100, 300, 500, and 1,000[6], with the surrogate model's predictions as the contribution function estimates. As in the previous section, we refer to KernelSHAP with $|\mathcal{D}|$ coalitions as KernelSHAP$|\mathcal{D}|$, e.g. KernelSHAP100. Moreover, we consider FastSHAP with n_{coals} equal to 4, 32, and 64, which we refer to as FastSHAP4, FastSHAP32, and FastSHAP64, respectively. The FastSHAP models are also trained with the surrogate model as the contribution function estimator. Details are given in Algorithm 3.

The error of the estimates according to the metric (46) is shown in Figure 32. For this example, FastSHAP4 and FastSHAP32 perform similarly to KernelSHAP300, and the performance of FastSHAP64 is somewhere between KernelSHAP300 and KernelSHAP500. KernelSHAP1000 clearly outperforms all the FastSHAP models. In Figure 33, the accuracy of FastSHAP is plotted as a function of the hyperparameter n_{coals}. As the figure shows, the accuracy seems to have converged at n_{coals} equal to 32. The FastSHAP32 model is the "best possible" FastSHAP model in our experiments since it has the lowest possible error, and among the models that attain this error, it has the lowest computational cost. The KernelSHAP1000 estimate is significantly better than the best FastSHAP model and should therefore be preferred in terms of accuracy.

The CPU time of initializing and evaluating KernelSHAP1000 is shown in Figure 34. This is the average for 100 test observations. In addition, the CPU time of training the FastSHAP32 model and the mean CPU time of providing a prediction with it for 100 test observations are shown in the

[6]The convergence of KernelSHAP as a function of $|\mathcal{D}|$ is very similar for the simulated data and the "Adult Data" set. Therefore, we omit it from the report and refer to Figure 18 for an illustration of the effect of the number of feature coalitions $|\mathcal{D}|$ on the KernelSHAP estimate (34).

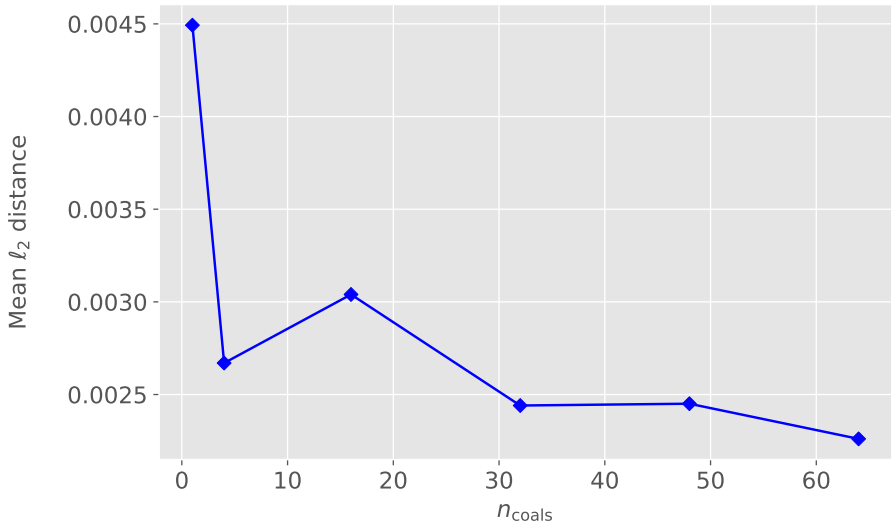


Figure 33: The accuracy of FastSHAP as a function of the hyperparameter n_{coals} for the "Adult Data" set.

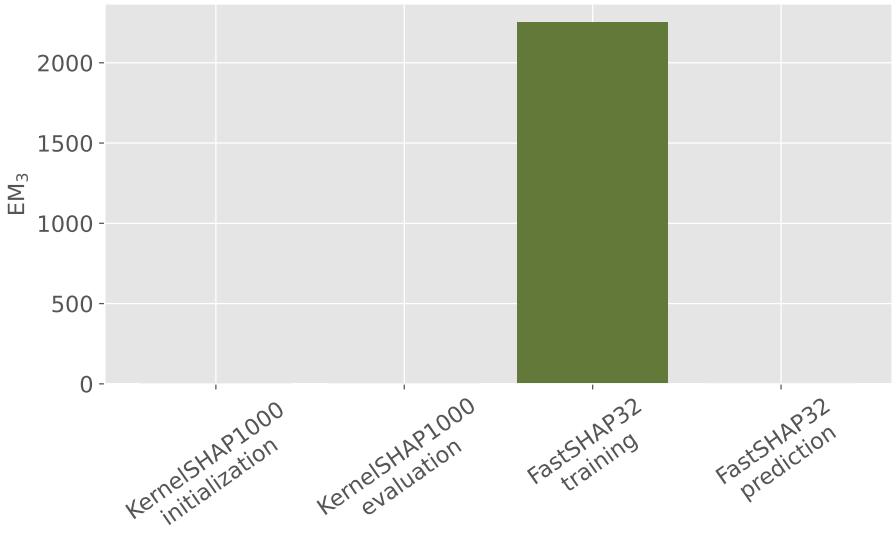


Figure 34: The CPU time of computing the Shapley value estimates for the "Adult Data" set.

figure. In the figure, the only visible bar is the FastSHAP32 training CPU time. In comparison, the other CPU times are negligible. Thus, the KernelSHAP is the fastest of the two methods when only the Shapley value estimation is considered. For more details on the CPU time of the methods, and a discussion on the computational time of the full estimation procedures, we refer to the results of the simulation study presented in Section 4.5.6.

6 Summary and Discussion of the Experiments

6.1 Contribution Function Estimation

We have evaluated the off-manifold and surrogate estimates of the contribution function on several simulated data sets. The data sets varied in terms of the number of training observations, the degree of correlation between the features, and the difficulty of the generating distribution. In the experiments, in terms of accuracy, we found that neither of the two methods outperforms the other in all situations. On the contrary, the examples illustrate that in some situations, the off-manifold method is clearly to be preferred, and in others, the surrogate model should be preferred.

In summary, the experiments show that the surrogate model outperforms the off-manifold method if the training data set is large, especially if the feature variables have a higher linear correlation. Since the off-manifold method assumes feature independence, the method fails more severely when the assumption is far from true, such as in these examples. On the other hand, the off-manifold method tends to outperform the surrogate model when the training data set is small, especially if the features are independent or "approximately independent", as illustrated with features simulated with only some linear correlation. In these situations, the error caused by the bias in the off-manifold estimate is relatively small. The poor performance of the surrogate model for smaller data sets can be explained by neural networks generally requiring access to much data to perform well. The off-manifold method, on the other hand, uses at most the number of Monte Carlo samples M from the training data set since the samples are drawn with replacements. In our experiments, we find that the variance of the Monte Carlo estimate has been empirically minimized when the number of Monte Carlo samples used per feature coalition is above circa 300. Generally, when the variance is minimized, the error of the off-manifold estimate attains its obtainable minimum since the bias term caused by falsely assuming feature independence will remain. Thus, the off-manifold method requires few training observations compared to the surrogate model. Lastly, fixing the number of training observations, we find that the off-manifold method performs better relative to the surrogate model in situations where the data distribution is more difficult to learn. Specifically, in our experiments, when the degree of correlation is similar, and the number of training observations is fixed, we find that the surrogate model performs worse in the case of Burr distributed features than in the case of normally distributed features. Since the surrogate model is a neural network, it needs many training observations, where "many" will vary depending on how hard the data generating distribution is to learn. The off-manifold estimate has a similar accuracy regardless of the difficulty of the data in our experiments. However, also in the case of the more difficult data, the surrogate model can outperform the off-manifold method if the training data set is sufficiently large and/or if the independence assumption is far from true.

Based on the accuracy of the methods, it seems that the methods are to be preferred in different situations. The previous paragraph roughly outlines when each method is appropriated and should be preferred. However, we cannot make any firm statements regarding e.g. how many training observations are enough for a data set to be large enough for the surrogate model to be preferred. Therefore, a priori, one cannot be certain which methods will perform best. However, if time allows, one can calculate the contribution function estimate for a given number of test observations and rank the methods to choose the best one analogously to Frye et al. [3] and Olsen et al. [40]. From the perspective of accuracy, the problem of estimating the contribution function is similar to the original problem of fitting a model to learn the data; one has to test out several methods and choose the best based on e.g. the performance on a test set.

Related to this, as pointed out by Chen et al. [10], it remains to develop robust techniques for optimizing the hyperparameters and architecture of the surrogate model. In our experiments, we used a default set of hyperparameters and architecture in the surrogate model for all the data sets. To fully determine the performance of the surrogate model in terms of accuracy, we should have performed hyperparameter tuning, e.g., by performing a randomized grid search over its hyperparameters. However, since this is not available in the current FastSHAP libraries [4], this would have to be implemented by the user and requires the user to have in-depth knowledge of the working mechanisms of the method and domain knowledge regarding machine learning in general. Therefore, our results represent what a non-expert can expect when applying the method. The

off-manifold does not require as much hyperparameter tuning. Generally, increasing the number of Monte Carlo samples will reduce the estimate's variance, and there is no downside in increasing the number from an accuracy point of view. Another possible improvement in the surrogate training algorithm is to consider several feature coalitions $\mathcal{S}$ per instance in a batch. In the version we are using, which corresponds to the version in [4], only one feature coalition is considered per instance in a batch. However, as shown in Figures 19, 20, 21 and 33, the FastSHAP model significantly improves when several coalitions are considered per instance in a batch, determined by using a value of the hyperparameter n_{coals} that is greater than 1. Therefore, allowing the use of several coalitions $\mathcal{S}$ per instance in a batch in the training of the surrogate model may lead to an increase in its accuracy.

From the experiments, it is clear that the surrogate model is faster than the off-manifold method when more than a few instances are to be explained, especially if the initial cost of training the surrogate model is disregarded. In some practical situations, the initial cost can be disregarded because it is possible to spend time developing the model, but once it has been put into production, it is required to make fast predictions, which is exactly what the surrogate model achieves. If the process of tuning the surrogate's hyperparameters is included, the initial cost will significantly increase. However, this might be necessary to ensure that the surrogate model performs well and can also be feasible because it is an initial cost that amortizes across the instances to be explained. The off-manifold estimate, on the other hand, has a relatively high computational cost that must be repeated for every instance that is to be explained.

In summary, when considering the accuracy and computational cost of the methods, it is not straightforward to conclude on which method to use. In some situations, there are very clear preferences towards one of the methods. For example, in the cases with higher correlations shown in Figures 8 and 11 for the normal and Burr distributions, respectively, the surrogate model should be preferred since it has a higher, or equally as high, accuracy, *and* a lower computational cost. However, with e.g. some correlation and a small training data set, one has to weigh accuracy against computation time when choosing a method. We also found that for the real-world "Adult Data" set, even though the correlation between the numerical features was low, the off-manifold estimate outperformed the surrogate model when sufficiently many Monte Carlo samples were used in the estimate. This reflects that for real-world data, using a biased estimator can still give better results because the data is too difficult for the surrogate model to learn with the available number of training observations.

6.2 Shapley Value Estimation

We now summarize the results of the Shapley value estimation procedure, where we have considered the KernelSHAP and FastSHAP methods. Based on the simulation study, it is clear that KernelSHAP should be preferred in terms of accuracy for smaller training data sets. Compared to the estimation of the contribution function, we do not find as clear trends on how the remaining properties of the data set affect the performance of the methods. An interesting observation of [11] is that the accuracy of KernelSHAP does not necessarily increase for an increasing number of features in the model. However, we show that KernelSHAP's accuracy is lower if there is a higher number of *significant* features in the model. For all the simulated data sets and the real-world "Adult Data" set, we find that when the number of coalitions $|\mathcal{D}|$ included in KernelSHAP is sufficiently high, e.g. $|\mathcal{D}| \geq 500$, KernelSHAP outperforms FastSHAP. Lastly, as for the surrogate model, to fully evaluate the accuracy of the FastSHAP model, we should have optimized all the hyperparameters, both those specific to the method and those that are general for neural networks. An argument against doing this, however, is that it requires the user to have domain knowledge. In the experiments, we find that increasing the value of the hyperparameter n_{coals} can improve the estimate of the FastSHAP model.

Isolating the cost of the Shapley value estimation step, both the KernelSHAP and FastSHAP methods can provide explanations very fast. The disadvantage of KernelSHAP, however, is that $|\mathcal{D}|$ estimates of the contribution function must be provided per instance to be explained. For the full KernelSHAP-Off-Manifold estimation procedure, this is severe in terms of the computational cost because the computation of the off-manifold estimate is high. However, for the KernelSHAP-

Surrogate method, it may be acceptable because the evaluation of the surrogate model is fast. FastSHAP, on the other hand, can provide explanations by a single evaluation of the FastSHAP model, regardless of the estimation of the contribution function, after the model has been trained. The speed of the estimation of the contribution function will only affect the initial training of the model. However, because of the high computational cost of the off-manifold estimate, FastSHAP-Off-Manifold will have a significantly higher initial training time than FastSHAP-Surrogate. If it is essential to provide explanations instantaneously, there seem to be three viable options. The options are KernelSHAP-Surrogate, FastSHAP-Off-Manifold, and FastSHAP-Surrogate.

6.3 The Usefulness of the Estimated Shapley Values

In this thesis, we have focused on evaluating the Shapley value estimation methods based on accuracy and computational cost. However, there is a larger context that these methods can be evaluated within, something we will briefly discuss in the following paragraphs. Through the four properties they are required to satisfy, one can argue that the exact Shapley values provide an objective standard for what an explanation of a black box machine learning model is. If so, the Shapley values can be seen as a universal explanation framework that can be used to increase trust in black box machine learning models. However, since, in practice, it is only feasible to estimate the Shapley values, the resulting explanations will be more subjective. An example of this is that the user must weigh the need to provide explanations fast against the need to provide accurate explanations. If the off-manifold method is used to estimate the contribution function, another problem related to the resulting explanation is that one must be very careful regarding the validity of the independence assumption.

In the surrogate and FastSHAP models, neural networks are used to provide explanations of the black box models. It raises the question of whether neural networks, which are typical examples of non-interpretable black box machine learning models, should also be used to explain these models. To some extent, it is paradoxical to use non-interpretable machine learning models to increase the interpretability of black box models. If the explanation method is non-interpretable, to which extent is it useful? Is it sufficient to use these methods based on empirical evidence demonstrating their accuracy?

We have briefly mentioned some aspects related to the ongoing discussion of what a good explanation method is. However, there are many other considerations that should be taken into account, e.g. whether an explanation is intuitively understandable, also for laypeople from a psychological point of view. It has not been a focus of this thesis to tackle the challenges that remain within XAI regarding developing a framework for what a good explanation is, how it should standardize etc. However, these are key questions that must be taken into account also for Shapley values. Therefore, we end this discussion by pointing out that this is an important part of the further development of Shapley value estimators as explanation methods.

7 Conclusion and Future Work

7.1 Conclusion

In this thesis, we have considered Shapley values as a model-agnostic post hoc explanation method within the field of XAI. The computation of the exact Shapley values involves determining exponentially many values of the contribution function, which corresponds to a potentially high dimensional integral of a complex expression that often is analytically intractable. Because numerical integration techniques suffer in high dimensions due to the grid size increasing exponentially [31], they cannot be used to approximate the integral for general real-world problems. The need for XAI is especially relevant for high dimensional data sets, since the Shapley value computation is exponential in the number of features q, this is a severe drawback. Moreover, since the black box models we are interested in explaining have become extremely complex, the integrand in the contribution function is very complicated when considering these models. However, the Shapley values have a solid theoretical foundation as an explanation method, something few other methods have. The explanations provided by the Shapley values can be seen as correct and fair, which is a huge advantage of the method.

Therefore, it is desirable to use Shapley values to explain black box machine learning models. Since the exact Shapley values are unknown in many real-world problems, several estimators have been introduced. Separating the estimation of the contribution function from the estimation of the Shapley values, we have evaluated various estimators based on accuracy and computational cost. We find that the off-manifold method and surrogate model are highly accurate in different situations. Regarding the Shapley value estimators, we find that KernelSHAP generally can provide estimates with significantly higher accuracy than FastSHAP. Based on various simulated data sets and a real-world data set, we have studied the relative performance of the estimation methods in different settings. In terms of accuracy, we find some empirical properties of the data set that, a priori, can indicate how the estimation methods are expected to perform, especially for the contribution function estimators. In practical applications, this is useful since one can favor one method over another based on the properties of the data set. However, we cannot provide any strict rules for when a method should be preferred. Therefore, in practice, it may be necessary to calculate estimates with all methods and use evaluation metrics to rank their performance on a test set to determine which method has the highest accuracy.

In many real-world applications, the computational cost of the methods is also important. In our results, we find that the off-manifold estimate of the contribution function has a much higher computation time than the surrogate estimate of the contribution function and both Shapley value estimators. In addition, how the computation time is distributed varies for the two Shapley value estimators. FastSHAP has a high initial computation time but can provide real-time explanations for a new instance of interest because this corresponds to a single evaluation of a neural network, even if the slow off-manifold estimate is used in combination with it. KernelSHAP, on the other hand, has a low initial computation time, but to explain a new instance, the contribution function must be computed for $|\mathcal{D}|$ feature coalitions. If the contribution function estimator is the off-manifold estimate, the computation time of the full estimation procedure is high, which means that it is infeasible to use the method in many real-world problems.

In practice, the user must decide which property is most important, accuracy or computation time, which makes the explanation method suffer from the user's subjectivity. This is a drawback since one of the reasons for using Shapley values as an explanation method is the objective theoretical foundation that ensures the method's correctness and fairness. To further evaluate the estimation methods, it is necessary to develop a stricter definition and consensus of what a good explanation is, e.g., one must agree on whether non-interpretable models can be used to provide explanations. Overall, we cannot say that one estimation procedure is preferred based on all evaluation criteria. Therefore, it may be necessary for these methods to coexist and perhaps be combined when providing an explanation. This may reduce the objectivity of the explanations but increase the accuracy and improve the computation time.

7.2 Future Work

In our experiments, we have performed an empirical study of how the FastSHAP and KernelSHAP methods perform in various situations depending on the empirical properties of the data set. However, when considering e.g. the number of features q in the black box model, we do not find that it has a clear effect on the accuracy of the Shapley values estimates. As pointed out by Doumard et al. [11], "this is probably due to the fact that usually, the more features there are, the less influence amplitude each individual feature has in the prediction". This reflects that the performance of the estimators does not only depend on the data set but also on the properties of the black box models. Therefore, it could be interesting to investigate if the KernelSHAP estimate, which considers a subset of size $|\mathcal{D}|$ of the 2^q feature coalitions, breaks down when the number of significant features is higher. A strategy for considering this is to use convergence detection techniques [32] to approximate the ground truth Shapley values in higher dimensional settings, e.g. when the number of features q is around 30. This could lead to models with a higher number of significant features, which perhaps alters the relative performance of the estimation methods.

Although we have studied the performance of the estimation methods on different data sets, it would be interesting to replicate the study of Doumard et al. [11]. In the study, they consider the accuracy of various additive local explanation methods for 304 OpenML data sets. However, they do not consider FastSHAP. In addition, the estimation of the contribution function is not separated from the estimation of the Shapley values. It would be interesting to replicate the study for the full estimation procedures, KernelSHAP-Off-Manifold, KernelSHAP-Surrogate, FastSHAP-Off-Manifold, and FastSHAP-Surrogate. Separating the two estimation steps, the contribution function estimators can be ranked. Moreover, fixing the estimate of the contribution function, the estimate of the Shapley values can be evaluated compared to the exact Shapley values. This would give further insight into the estimation methods' performance for real-world data sets. In [11], TreeSHAP [2] is also considered. It is a model-specific method that can be used for tree-based models and has a lower computational cost than e.g. KernelSHAP. However, in the SHAP library [2], the off-manifold estimate is used in TreeSHAP as well. Frye et al. [3] find that the problems related to using the off-manifold estimate are also present for TreeSHAP. Therefore, it is interesting to investigate how the TreeSHAP model performs when it is combined with the surrogate model. Moreover, since we found evidence that may suggest that KernelSHAP performs worse if there are more *significant* features in the model, it would be interesting to replicate the experiments in [11], also for data sets with more than 13 features, which is the highest number of features considered in the study.

In their experiments, Jethani et al. [4] find that regardless of which estimate of the contribution function is used, FastSHAP can provide accurate estimates of the Shapley values. To be clear, they fix an estimate of the contribution function and compare different Shapley value estimators. Therefore, the result is regarding the Shapley values estimators given an estimate of the contribution function. With this in mind, it seems sensible to evaluate the performance of the estimators by separating the two estimation procedures, such as we have done in our experiments. However, it is not given that the most accurate estimator in the full procedure is the combination of the most accurate estimator in each step. For real-world data sets, it is not generally possible to evaluate the full estimation procedure by using the exact Shapley value. However, when considering data simulated from e.g. the Burr and normal distributions, the contribution function can be estimated accurately and used in the computation of the exact Shapley values. This will provide a ground truth estimate that the full estimation procedures can be evaluated against.

As future work within the field of XAI, it is necessary to develop a more rigorous framework for evaluating the explanation methods and a consensus on what a good explanation is. If more precise requirements of what an explanation consists of, e.g., within a legal framework, one can develop evaluation metrics that take these into account. This will enable us to develop better explanation methods. A better framework may reduce matters of subjectivity, e.g., that the user must prioritize accuracy versus computation time, related to the explanation methods, leading to better and more trustworthy explanation methods.

References

[1] Štrumbelj, E. and Kononenko, I. 'Explaining Prediction Models and Individual Predictions with Feature Contributions'. In: *Knowledge and information systems* 41.3 (2014), pp. 647–665.

[2] Lundberg, S. M. and Lee, S. 'A Unified Approach to Interpreting Model Predictions'. In: *Advances in Neural Information Processing Systems*. Long Beach, CA, USA: Curran Associates, Inc., 2017, pp. 4765–4774.

[3] Frye, C. et al. 'Shapley Explainability on the Data Manifold'. In: *International Conference on Learning Representations (ICLR)*. 2021.

[4] Jethani, N. et al. 'FastSHAP: Real-Time Shapley Value Estimation'. In: *International Conference on Learning Representations (ICLR)*. 2022.

[5] Molnar, C. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. 2nd ed. 2022. URL: https://christophm.github.io/interpretable-ml-book.

[6] Lervik, F. and Hansen, E. M. 'Brytningstid - En studie av kunstig intelligens i norske banker og forsikringsselskap'. A report by PA Consulting ordered by Finansforbundet investigating the use of AI in the Norwegian finance industry. Cited Oct. 10th 2022. Full pdf avaliable at https://www.finansforbundet.no/folk-og-fag/forbundsnytt/slik-bruker-norsk-finans-kunstig-intelligens/. 2022.

[7] Shapley, L.S. 'A value for n-person games'. In: *Contributions to the Theory of Games* 2 (1953), pp. 307–317.

[8] Aas, K., Jullum, M. and Løland, A. 'Explaining Individual Predictions when Features are Dependent: More Accurate Approximations to Shapley Values'. In: *Artificial Intelligence* 298 (2021), p. 103502.

[9] Jethani, N. et al. 'Have We Learned to Explain?: How Interpretability Methods Can Learn to Encode Predictions in their Interpretations.' In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2021, pp. 1459–1467.

[10] Chen, H. et al. 'Algorithms to Estimate Shapley Value Feature Attributions'. In: *arXiv preprint arXiv:2207.07605* (2022).

[11] Doumard, E. et al. 'A Comparative Study of Additive Local Explanation Methods Based on Feature Influences'. In: vol. 3130. paper 4. 2022, pp. 31–40.

[12] Aase, F. S. 'An Exploration of LIME and Shapley Values as Local Model-Agnostic Explanation Methods'. The report from my Specialization project from the course TMA4500 - Industrial Mathematics, Specialization Project at the Norwegian University of Science and Technology (NTNU). Dec. 2022.

[13] Hastie, T., Tibshirani, R. and Friedman, J. H. *The Elements of Statistical Learning. Data Mining, Inference, and Prediction*. 2nd ed. New York: Springer, 2009.

[14] Lewis, R. J. 'An Introduction to Classification and Regression Tree (CART) Analysis'. In: *Annual Meeting of the Society for Academic Emergency Medicine*. Vol. 14. San Francisco, California, 2000.

[15] Freund, Y., Schapire, R. and Abe, N. 'A Short Introduction to Boosting'. In: *Japanese Society For Artificial Intelligence* 14.5 (1999), pp. 771–780.

[16] Chen, T. and Guestrin, C. 'XGBoost: A Scalable Tree Boosting System'. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794.

[17] Friedman, J. H. 'Greedy Function Approximation: a Gradient Boosting Machine'. In: *Annals of Statistics* 29.5 (2001), pp. 1189–1232.

[18] Fisher, Walter D. 'On grouping for maximum homogeneity'. In: *Journal of the American statistical Association* 53.284 (1958), pp. 789–798.

[19] Sagi, O. and Rokach, L. 'Approximating XGBoost with an Interpretable Decision Tree'. In: *Information Sciences* 572 (2021), pp. 522–542.

[20] Neutelings, I. *Neural Networks [Internet]*. Note: Licensed according to: Creative Commons Attribution-ShareAlike 4.0 International License. Cited on February 3rd, 2023. URL: https://tikz.net/neural_networks/.

[21] Guilhoto, L. F. 'An Overview of Artificial Neural Networks for Mathematicians'. In: *Univ. Chicago* (2018).

[22] Robbins, H. and Monro, S. 'A stochastic approximation method'. In: *The Annals of Mathematical Statistics* 22.3 (1951), pp. 400–407.

[23] Kingma, P. K. and Ba, J. L. 'Adam: A Method for Stochastic Optimization'. In: *International Conference for Learning Representations (ICLR)*. 2015.

[24] Hornik, K., Stinchcombe, M. and White, H. 'Multilayer Feedforward Networks are Universal Approximators'. In: *Neural networks* 2.5 (1989), pp. 359–366.

[25] Csáji, B. C. 'Approximation with Artificial Neural Networks'. In: *Faculty of Sciences, Etvs Lornd University, Hungary* 24.48 (2001), p. 7.

[26] Harris, C. R. et al. 'Array programming with NumPy'. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362.

[27] Takahasi, K. 'Note on the Multivariate Burr's Distribution'. In: *Annals of the Institute of Statistical Mathematics* 17.1 (1965), pp. 257–260.

[28] Yari, G. H. and Jafari, A. M. 'Information and Covariance Matrices for Multivariate Pareto (IV), Burr, and Related Distributions'. In: *IUST International Journal of Engineering Science* 17.3-4 (2006), pp. 61–69.

[29] Aas, K. et al. 'Explaining Predictive Models using Shapley Values and Non-Parametric Vine Copulas'. In: *Dependence Modeling* 9.1 (2021), pp. 62–81.

[30] Lun, Z. and Khattree, R. *NonNorMvtDist: Multivariate Lomax (Pareto Type II) and Its Related Distributions*. R package version 1.0.2. 2020. URL: https://CRAN.R-project.org/package=NonNorMvtDist.

[31] Givens, G. H. and Hoeting, J. A. *Computational Statistics*. 2nd ed. Hoboken, New Jersey: John Wiley & Sons, Inc., 2013.

[32] Covert, I. and Lee, S. 'Improving KernelSHAP: Practical Shapley Value Estimation via Linear Regression'. In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. San Diego, California, USA: PMLR, 2021, pp. 3457–3465.

[33] Ruiz, L. M., Valenciano, F. and Zarzuelo, J. M. 'The Family of Least Square Values for Transferable Utility Games'. In: *Games and Economic Behavior* 24.1-2 (1998), pp. 109–130.

[34] LeCun, Y., Cortes, C. and Burges, C. J. C. *THE MNIST DATABASE of handwritten digits [Internet]*. 2010. URL: http://yann.lecun.com/exdb/mnist/ (visited on 27/02/2023).

[35] Covert, I. C., Lundberg, S. and Lee, S. 'Explaining by Removing: A Unified Framework for Model Explanation.' In: *Journal of Machine Learning Research (JMLR)* 22.209 (2021), pp. 1–90.

[36] Python Software Foundation, ©Copyright 2001-2023. *Time Access and Conversions [Internet]*. Cited on May 24th, 2023. Python version 3.8.5 is used. URL: https://docs.python.org/3.8/library/time.html#time.process_time.

[37] Cansiz, S. *Multivariate Outlier Detection in Python [Internet]*. Towards Data Science, March 20th, 2021. Cited on May 23th, 2023. URL: https://towardsdatascience.com/multivariate-outlier-detection-in-python-e946cfc843b3.

[38] Dua, D. and Graff, C. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[39] Rosenberg, D. *Unbalanced Data? Stop Using ROC-AUC and Use AUPRC Instead [Internet]*. Towards Data Science, June 7th, 2022. Cited on May 5th, 2023. URL: https://towardsdatascience.com/imbalanced-data-stop-using-roc-auc-and-use-auprc-instead-46af4910a494.

[40] Olsen, L.-H. B. et al. 'Using Shapley Values and Variational Autoencoders to Explain Predictive Models with Dependent Mixed Features'. In: *Journal of machine learning research* 23.213 (2022), pp. 1–51.

[41] A., Martín et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

Appendix

A Implementation Details.

The implementation of all the estimation methods that we have used in our experiments can be found on GitHub. For KernelSHAP and the off-manifold method, all the relevant hyperparameters are stated in the description of the experiments in Chapters 4 and 5. Therefore, they are not listed here. For the surrogate and FastSHAP models, the hyperparameters we have chosen are based on the examples in the FastSHAP implementation in TensorFlow [4]. In the experiments, we have explicitly stated some of the hyperparameters. These are the hyperparameters that are varied between the experiments. The remaining hyperparameters and architecture of the models will be given in what follows.

The Surrogate Model

In the implementation of the surrogate model, we use a TensorFlow [41] neural network. The hyperparameters and the architecture of the model are the same in all the experiments. The model has two hidden dense layers with 128 nodes. The layers are fully connected. Between the input and hidden layers, we use the ReLU (15) activation function. In the output layer, we use a softmax (16) activation function. The number of nodes in the output layer is set to two in all our experiments, since in both the simulation study and real-data experiment there are only two classes the response variable can belong to.

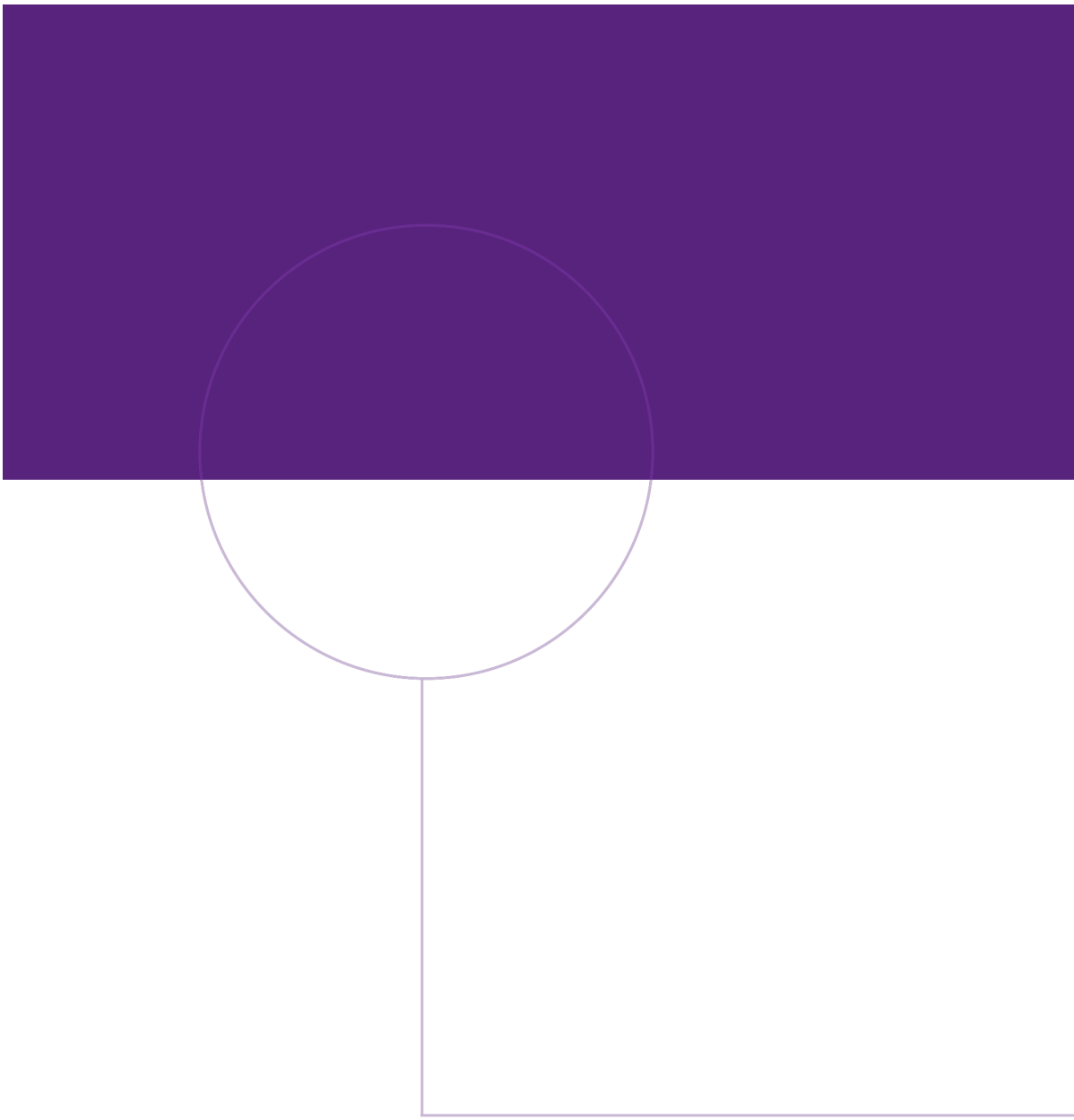
The batch size is set to 64, and the maximum number of epochs is set to 100. The loss function in the model is the categorical cross-entropy (18). In the training, early stopping and a learning rate schedule that reduces the learning date are used. The initial learning rate is set to 10^{-3} and it is reduced by a factor of 0.9 if there is no improvement for two epochs, determined by setting the `patience` to two. The minimum learning rate is set to 10^{-5}, the training will stop if the learning rate attains this value. Moreover, we use early stopping with a `lookback` of 10, meaning that if no improvement occurs for 10 epochs, the training will stop. A validation batch is used, and we set the validation batch size to 10,000. We use the Adam optimizer [23], according to Algorithm 1, in the surrogate model. We do not use paired sampling or append the masking net to the training data.

The FastSHAP Model

In the implementation of the FastSHAP model, we also use a TensorFlow [41] neural network. The hyperparameters and the architecture of the model are the same in all the experiments. The model has two hidden dense layers with 128 nodes. The layers are fully connected. Between the input and hidden layers, we use the ReLU (15) activation function. In the output layer, we use a linear activation function.

The batch size is set to 32, and the maximum number of epochs is set to 200. In the training, we use early stopping and a learning rate schedule that reduces the learning date. The initial learning rate is set to 10^{-3}, and it is reduced by a factor of 0.9 if there is no improvement for two epochs, determined by setting the `patience` to two. The minimum learning rate is set to 10^{-5}. The training will stop if the learning rate attains this value. Moreover, we use early stopping with a `lookback` of 10, meaning that if no improvement occurs for 10 epochs, the training stops. A validation batch is used, and we set the validation batch size to 64. We use the Adam optimizer [23], according to Algorithm 1, in the FastSHAP model. We do not use paired sampling or append the masking net to the training data.

In the FastSHAP model, several hyperparameters can be used to enforce the efficiency constraint of the Shapley values. These were described in Section 3.4.1. Jethani et al. [4] find that setting the hyperparameter $\gamma > 0$ in the regularized loss function (38) leads to less accurate estimates than using $\gamma = 0$. Therefore, we set $\gamma = 0$. Moreover, they demonstrate that using additive efficiency normalization (37) during both training and inference increases the accuracy of the estimates. In accordance with this, we also use additive efficiency normalization in both steps.



NTNU

Norwegian University of
Science and Technology