High-throughput Machine Learning Algorithms

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

Rory Mitchell

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2021

Abstract

The field of machine learning has become strongly compute driven, such that emerging research and applications require larger amounts of specialised hardware or smarter algorithms to advance beyond the state-of-the-art. This thesis develops specialised techniques and algorithms for a subset of computationally difficult machine learning problems. The applications under investigation are quantile approximation in the limited-memory data streaming setting, interpretability of decision tree ensembles, efficient sampling methods in the space of permutations, and the generation of large numbers of pseudorandom permutations. These specific applications are investigated as they represent significant bottlenecks in real-world machine learning pipelines, where improvements to throughput have significant impact on the outcomes of machine learning projects in both industry and research. To address these bottlenecks, we discuss both theoretical improvements, such as improved convergence rates, and hardware/software related improvements, such as optimised algorithm design for high throughput hardware accelerators.

Some contributions include: the evaluation of bin-packing methods for efficiently scheduling small batches of dependent computations to GPU hardware execution units, numerically stable reduction operators for higher-order statistical moments, and memory bandwidth optimisation for GPU shuffling. Additionally, we apply theory of the symmetric group of permutations in reproducing kernel Hilbert spaces, resulting in improved analysis of Monte Carlo methods for Shapley value estimation and new, computationally more efficient algorithms based on kernel herding and Bayesian quadrature. We also utilise reproducing kernels over permutations to develop a novel, linear-time, statistical test for the hypothesis that a sample of permutations is drawn from a uniform distribution.

The techniques discussed lie at the intersection of machine learning, high-performance computing, and applied mathematics. Much of the above work resulted in open source software used in real applications, including the GPUTreeShap library [38], shuffling primitives for the Thrust parallel computing library [2], extensions to the Shap package [31], and extensions to the XGBoost library [6].

Acknowledgements

I thank my excellent supervisors, Dr. Eibe Frank and Dr. Geoffrey Holmes, for consistent guidance and encouragement throughout this process. Thanks to Dr. Joshua Cooper, for lending his considerable mathematical expertise, and Daniel Stokes for his ideas and contributions to shuffling algorithms. Thanks to H2O.ai and Nvidia for funding. Thanks to my colleagues and collaborators Jiaming Yuan, Philip Cho and John Zedlewski. Thanks to Dr. Tianqi Chen for supporting and enabling my work on XGBoost, Dr. Scott Lundberg for guidance around Shapley values, and finally, thanks to those other numerous developers of open source software, which much of this work is built upon.

Contents

Chapter 1

Introduction

In 2018, the ACM Turing award was jointly awarded to Yoshua Bengio, Geoffrey Hinton, and Yann LeCun for "conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing" [1]. The benefits and application of these machine learning breakthroughs were widely acknowledged in 2018, yet many of the foundational publications of key authors occurred long before. See, for example, "Learning Representations by Back-Propagating Errors" [49], published in 1988, "Long Short-Term Memory"[22], published in 1997, "Gradient-based learning applied to document recognition"[29], published in 1998. This delay can be partly attributed to the computational demands of the algorithms, requiring increasingly large amounts of data and more sophisticated optimisation techniques. The availability of large datasets along with powerful hardware and software frameworks catalysed the development of new algorithms, software libraries, and hardware advances, to fully exploit the foundational theoretical work and enable further breakthroughs. Thus, the development of the field occurs hand-in-hand with advances in optimisation, hardware architecture, open-source software ecosystems, and the application of mathematical tools for further analysis and improvements of algorithms. In many machine learning applications, better results can be achieved trivially by increasing the amount of training data [17]. This fact can be taken advantage of, conditional on the availability of cost-

effective computing infrastructure and software to collect, store, preprocess, analyse, train models, and perform inference on that data. This thesis focuses specifically on the aforementioned computational issues in a subset of machine learning applications, with relevance to problems encountered in industry and research. Some common techniques applied towards this goal include specialised algorithm design, use of graphics processing units as hardware accelerators and novel applications of reproducing kernel Hilbert spaces. The core research proposal of this thesis can be summarised as follows.

> Thesis statement: Graphics processing units (GPUs) and specialised algorithm design can significantly improve the throughput of machine learning pipelines.

Consider the end-to-end deployment of machine learning models in real-world systems, which may include some or all of the following phases [50, 7, 47]:

- Data collection/preparation.
- Exploratory analysis (e.g. summary statistics, data visualisation).
- Feature transformation.
- Model training.
- Hyper-parameter tuning.
- Model selection.
- Inference.
- Interpretability.

We apply optimisation methodologies from both a hardware and software perspective to specific sub-problems in this pipeline, with a view to increasing overall throughput. The sub-problems are addressed as thesis chapters as follows.

In Chapter 3, we develop methods for density estimation and quantile approximation, as applied in the exploratory analysis and feature transformation phases. In particular, the quantile approximation problem forms a key bottleneck in gradient boosted tree algorithms (introduced in Section 2.1). We survey existing solutions to the density estimation/quantile approximation problem based on the sample moments of the data, where approximation of the respective probability density function is performed via the method of maximum entropy or various orthogonal series in a polynomial or trigonometric basis. Previously identified shortcomings of these methods are addressed using specialised numerical methods for higher-order moment aggregation, and a high throughput version based on GPU tree-reduction is developed. The result is sketching methods with runtime orders of magnitude faster than existing solutions (see the KLL sketch [27]) and excellent empirical performance for low memory sketches on large data streams.

A significant part of this thesis (Chapters 4 and 5) deals with the emerging interpretability phase of the above-mentioned machine learning pipeline. Interpretability has received much attention in light of the so-called "right to explain" [26] and its impact on the deployment of machine learning systems in real-world environments. As models become highly complex "black boxes", it becomes difficult to explain why a particular result is obtained. One algorithmic solution to the interpretability problem created by black-box machine learning models is the Shapley value [51]. While the Shapley value has many desirable characteristics, it is NP-hard in general, and poses significant computational challenges. Chapter 4 discusses a polynomial-time algorithm for the Shapley values of decision tree models and its nontrivial adaptation to GPUs. The result is GPUTreeShap, a scalable software library achieving increases in throughput of between 15-340x as compared to existing CPU based algorithms for the Shapley value problem in decision trees. GPUTreeShap is natively integrated into XGBoost [6], one of the most popular machine learning libraries in the world [25]. In Chapter 5, we discuss Shapley value approximation in the

general case, using quasi Monte Carlo type methods. By applying kernels over permutations, we develop methods with improved analysis and convergence rates compared to previous literature.

Finally in Chapter 6, we address a gap in the literature for GPU shuffling algorithms, developing Bijective shuffle, a highly practical algorithm achieving orders of magnitude higher throughput than CPU based alternatives. We also reuse work from Chapter 5 regarding kernels on permutations, developing a statistical hypothesis test for the uniform distribution of permutations, and empirically verifying the correctness of our shuffling algorithm. Bijective shuffle is adopted by the Thrust library [2], the standard library for the CUDA language, and applied at various levels of the machine learning pipeline, including sampling without replacement during model training (feature sampling or bootstrapping), model selection (e.g. cross validation), as well as the Shapley value approximations discussed in Chapter 5, used in the interpretability phase.

In summary, this thesis contains four primary contributions:

- The development of lightweight, hardware efficient, algorithms for density estimation and quantile approximation.
- Specialised GPU algorithms for computing Shapley values of decision tree ensembles, and the accompanying software package, GPUTreeShap.
- The development and analysis of sampling algorithms for Shapley value estimation using reproducing kernels for permutations.
- A fast algorithm for shuffling data using GPUs, which is adopted by Thrust [2], the standard library for CUDA, and a kernel test for uniform distributions of permutations.

These contributions are connected by a common thread of software and algorithm optimisation techniques, and together account for significant improvements in total throughput for real-world machine learning pipelines. In particular, the above solutions are applied to research projects in computational

finance [42] and differential privacy [46], as well as use cases in industry such as customer retention and loan delinquency prediction (and interpretability of predictions).

We begin the thesis by discussing relevant background material in Chapter 2, including supervised learning and decision tree models, hardware acceleration via general-purpose GPU computing, Shapley values, and reproducing kernel Hilbert spaces. Chapters 3-6 include self-contained publications corresponding to the primary contributions of the thesis, and Chapter 7 the conclusion.

Chapter 2

Background

This chapter provides an introduction to foundational concepts for understanding the work presented in this thesis. Some material may be repeated as later chapters are self-contained publications. We begin by introducing supervised learning in Section 2.1, with a specific focus on decision trees, boosting, and the XGBoost algorithm. In Chapter 4, we develop fast interpretability methods for decision trees and apply them as an extension to the open source XGBoost library. These models are also used to generate an interpretability baseline in Chapter 5. Furthermore, the quantile sketching methods discussed in Chapter 3 can be applied as a preprocessing step in a quantised version of the XGBoost algorithm.

In Section 2.2, we introduce GPU computing. GPU acceleration techniques are used in Chapters 3, 4, and 6 to develop significantly higher throughput versions of existing algorithms.

In Section 2.3, we introduce Shapley values. This material is relevant to Chapters 4 and 5, where we specifically discuss the solutions of compute intensive Shapley value problems for interpretability in machine learning.

Section 2.4 introduces reproducing kernel Hilbert spaces, and specifically, reproducing kernels on the symmetric group. These kernels are used in the analysis and development of sampling strategies over permutations in Chapter 5, and to develop a statistical hypothesis test for the output of a shuffling

algorithm in Chapter 6.

2.1 Supervised Learning

The methods described in this thesis are applied to end-to-end machine learning workflows, focusing in particular on *supervised learning* applications. Informally, supervised learning refers to the task of forming a predictive model of the output of some system, given a number of training examples demonstrating the expected input/output behaviour of the system. In the standard setting considered in this thesis, the inputs are denoted by the matrix X, where each row constitutes an example and each column represents a unique input, or "feature". The outputs are represented by the vector y, containing one value for each row of X. Given the (X, y) input pair, the task is to find some model f that makes predictions

$$f(X_i) \approx y_i,$$

where the index i refers to a single training row. In practical applications of supervised learning, the hope is that f will continue to make accurate predictions when given new, unseen, data. Assuming the new data is i.i.d., and invoking some notion of complexity of the function f, there are theoretical results providing bounds on the expected accuracy of f a model given its accuracy on the training data and the amount of data it was trained on [54]. The notion of accuracy can be made precise by the definition of a loss function, such as mean squared error:

$$L(X, y, f) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(X_i))^2$$

Consider the simple linear model

$$f(X_i) = \sum_{j=1}^{m} \beta_j X_{ij} = \hat{y}$$

with parameter vector β. The supervised learning task in this case simplifies to finding the parameters β that minimise the loss function $L(X, y, f)$, for a

given (X, y) training pair. Substituting the linear model into the loss function L and using vector notation, we have

$$L = (y - X\beta)^T(y - X\beta).$$

Differentiating with respect to the parameters β and setting equal to zero yields

$$\frac{dL}{d\beta} = X^T(y - X\beta) = 0,$$

with optimal parameters (assuming X^TX is non-singular) given by

$$\beta = (X^TX)^{-1}X^Ty. \tag{2.1}$$

The above solution to a linear model with a mean squared error loss function is known analytically, but this is not the case in general, and more sophisticated models and loss functions require iterative optimisation methods to converge to a solution. Furthermore, the solution above happens to be a *global minimum* of the loss function, verifiable due to the convexity of the optimisation problem, which ensures any local minimum is also a global minimum [40]. More sophisticated models can be used in order to further reduce the loss function, but the optimisation of model parameters may no longer be convex.

2.1.1 Decision Trees

We now describe a decision tree model, a model commonly learned by attempting to recursively partition training instances, grouping those with similar labels (y), and making predictions for these groups as a whole. We show an example of decision tree learning using the Boston housing dataset [19]. This dataset contains 506 training examples and 13 features, where the features are characteristics of houses, and the labels to be predicted are the median house prices. Figure 2.1 shows a depth 2 decision tree model trained on this dataset using the scikit-learn [43] library. Prediction is performed by testing the appropriate feature values against a condition at each split node. When a leaf node is reached, the leaf value is returned. In this case, the learning algorithm

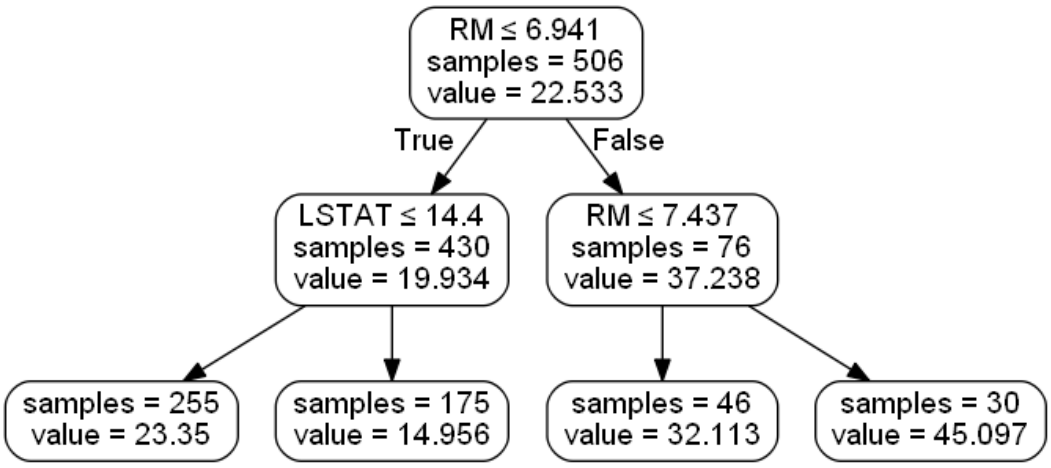


Figure 2.1: Decision tree model

has chosen to split on the RM (average number of rooms per dwelling) and LSTAT (% lower status of the population) features. The decision tree learning algorithm, trained with the mean squared error loss function, evaluates all possible splits along each feature, greedily subdividing the data at each level in order to minimise the loss function. In the case of mean squared error, this corresponds to finding the split that minimises the variance of house prices in each new partition.

Figure 2.2 plots the entire dataset in terms of the RM and LSTAT features, with points coloured according to house value. The dotted lines show the decision tree boundaries, separating the dataset into quadrants with similar house value characteristics. Given a new house with the RM and LSTAT attributes, its value can be estimated by the mean house value of the training data in the appropriate quadrant. From this diagram, we see that the decision tree algorithm has learned to group houses with approximately eight rooms or more in a high-value category. When encountering a new property with eight rooms, the model will predict a correspondingly high house value.

2.1.2 Boosting

While decision trees alone are useful models, multiple trees can be combined to form models with significantly higher predictive accuracy using a process

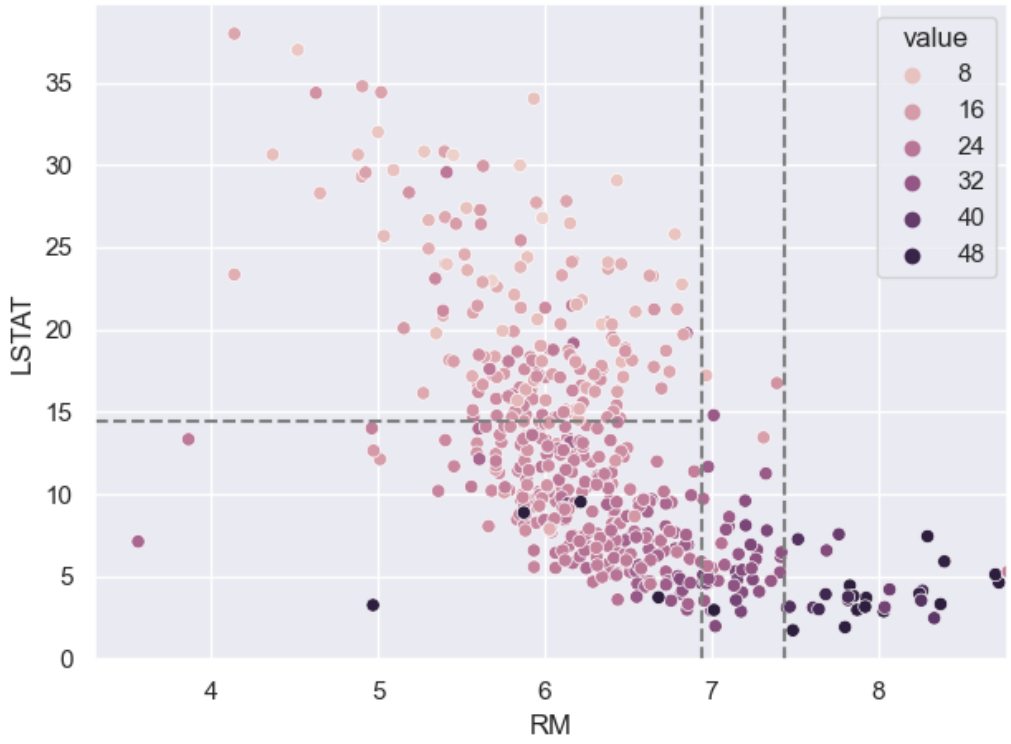


Figure 2.2: Boston housing data with partitions corresponding to decision tree in Figure 2.1

known as boosting. Boosting comes from the idea of combining individually weak models into a more powerful committee of models, where each additional model refines the predictions of the previous models. This is in contrast to bagging [4], another popular ensemble method that constructs weak models *simultaneously* on different subsamples of the data.

A significant early boosting algorithm is AdaBoost, due to Freund and Schapire [14]. AdaBoost is a binary classification algorithm that proceeds by fitting a model $f_t(x)$, producing values in $\{1, -1\}$, at each time step t on *weighted* training instances. At each step t, training instances incorrectly classified by the ensemble model $F_{t-1}(x) = \sum_{i=1}^{t-1} \alpha_i f_i(x)$, with suitable coefficients α_i, are given increased weight, and training instances that were correctly classified are given reduced weight. Thus, instances for which the current ensemble model performs poorly are given greater importance when inducing $f_t(x)$. Algorithm 1 describes the boosting process in greater detail.

Algorithm 1: AdaBoost.M1

Input: Training instances $X = \{x_1, x_2, \ldots, x_N\}$, binary class labels

$$y = \{y_1, y_2, \ldots, y_N\} \in \{1, -1\}$$

Output: Ensemble model $F(x)$

$F(x) \leftarrow \emptyset$

$w_i \leftarrow 1/N, i = 1, \ldots, N$

for $t \leftarrow 1$ ***to*** T **do**

  $f_t(x) \leftarrow learn(X, y, w)$ `// Train weak learner with weights`

  $\epsilon_t \leftarrow \frac{\sum_{i=1}^{n} w_i \mathbb{1}_{f_t(x_i) \neq y_i}}{\sum_{i=1}^{n} w_i}$ `// Calculate error`

  $\alpha_t \leftarrow log((1 - \epsilon_t)/\epsilon_t)$

  $w_i \leftarrow w_i \cdot exp(\alpha_t \cdot \mathbb{1}_{f_t(x_i) \neq y_i}), i = 1, \ldots, N$ `// Update weights`

  $F(x) \leftarrow F(x) + \alpha_t f_t(x)$ `// Add to ensemble`

end

return $F(x)$

AdaBoost is a powerful tool for classification, but can be generalised to a wider variety of objectives, such as real-valued regression, by re-framing the learning process as functional gradient descent. From this perspective, AdaBoost space can be seen to minimise an exponential loss function [20], which, for a single observation x, is defined as

$$L(y, F(x)) = exp(-yF(x)).$$

To illustrate this, we now derive an algorithm for minimising this loss function and then show its equivalence to Algorithm 1 (within some constant factor). In each iteration of the algorithm, the minimisation problem is

$$(\alpha_t, f_t(x)) = \underset{\alpha_t, f_t(x)}{\arg\min} \sum_{i=1}^{N} exp[-y_i(F_{t-1}(x_i) + \alpha_t f_t(x_i))]. \tag{2.2}$$

We proceed by first holding α_t constant, finding $f_t(x)$ that minimises the exponential loss, then substituting $f_t(x)$ to find α_t that further minimises this loss. Rearranging to extract terms that do not depend on α_t or $f_t(x_t)$ and

assuming α_t is fixed, we have

$$\arg\min_{f_t(x)} \sum_{i=1}^{N} exp(-y_i F_{t-1}(x_i)) \cdot exp(-y_i \alpha_t f_t(x_i)).$$

Define w_i as an instance weight for training instance i:

$$w_i = exp(-y_i F_{t-1}(x_i)). \tag{2.3}$$

The problem is then to find a new ensemble member $\alpha_t f(x)$ minimising the *weighted* exponential loss.

$$\arg\min_{f_t(x)} \sum_{i=1}^{N} w_i \cdot exp(-y_i \alpha_t f_t(x_i)).$$

For each training instance i, $f_t(x_i)$ outputs either 1 or -1, therefore the term $-y_i f_t(x_i)$ will be -1 if $y_i = f_t(x_i)$ and 1 if $y_i \neq f_t(x_i)$. Accordingly, we can separate the sum into two parts and obtain

$$\arg\min_{f_t(x)} \left\{ \sum_{i:y_i=f_t(x_i)}^{N} w_i \cdot e^{-\alpha_t} \right\} + \left\{ \sum_{i:y_i \neq f_t(x_i)}^{N} w_i \cdot e^{\alpha_t} \right\}. \tag{2.4}$$

Expressed only in terms of instances where $y_i \neq f_t(x_i)$, we have

$$\arg\min_{f_t(x)} \left\{ \sum_{i}^{N} w_i \cdot e^{-\alpha_t} \; - \sum_{i:y_i \neq f_t(x_i)}^{N} w_i \cdot e^{-\alpha_t} \right\} + \sum_{i:y_i \neq f_t(x_i)}^{N} w_i \cdot e^{\alpha_t}.$$

Rearranging, we get

$$\arg\min_{f_t(x)} \left\{ \sum_{i}^{N} w_i \cdot e^{-\alpha_t} \; + \sum_{i:y_i \neq f_t(x_i)}^{N} w_i (e^{\alpha_t} - e^{-\alpha_t}) \right\}.$$

The first sum and $(e^{\alpha_t} - e^{-\alpha_t})$ are constants, so the optimisation problem becomes

$$\arg\min_{f_t(x)} \sum_{i:y_i \neq f_t(x_i)}^{N} w_i. \tag{2.5}$$

(2.5) shows that as long as the new model $f_t(x)$ minimises the weighted misclassification error, subject to our definition of weights, the exponential loss is minimised.

The loss is further minimised by optimising α_t. Assuming $f_t(x)$ is fixed in (2.4), we have the following optimization problem:

$$\alpha_t = \arg\min_{\alpha_t} \left\{ \sum_{i:y_i=f_t(x_i)}^{N} w_i \right\} e^{-\alpha_t} + \left\{ \sum_{i:y_i \neq f_t(x_i)}^{N} w_i \right\} e^{\alpha_t}.$$

Taking derivatives with respect to α_t and solving, we have

$$\frac{\partial}{\partial \alpha_t} = -\left\{ \sum_{i:y_i = f_t(x_i)}^{N} w_i \right\} e^{-\alpha_t} + \left\{ \sum_{i:y_i \neq f_t(x_i)}^{N} w_i \right\} e^{\alpha_t} = 0$$

$$\left\{ \sum_{i:y_i \neq f_t(x_i)}^{N} w_i \right\} \frac{e^{\alpha_t}}{e^{-\alpha_t}} = \left\{ \sum_{i:y_i = f_t(x_i)}^{N} w_i \right\}$$

$$e^{2\alpha_t} = \frac{\sum_{i:y_i = f_t(x_i)}^{N} w_i}{\sum_{i:y_i \neq f_t(x_i)}^{N} w_i}$$

$$\alpha_t = \frac{1}{2} ln \left(\frac{\sum_{i:y_i = f_t(x_i)}^{N} w_i}{\sum_{i:y_i \neq f_t(x_i)}^{N} w_i} \right)$$

$$\alpha_t = \frac{1}{2} ln \left(\frac{1 - \frac{\sum_{i:y_i \neq f_t(x_i)}^{N} w_i}{\sum_{i}^{N} w_i}}{\frac{\sum_{i:y_i \neq f_t(x_i)}^{N} w_i}{\sum_{i}^{N} w_i}} \right)$$

$$\alpha_t = \frac{1}{2} ln \left(\frac{1 - \epsilon}{\epsilon} \right). \tag{2.6}$$

We now show that (2.6) is equivalent to Line 6 from Algorithm 1 up to a factor of $\frac{1}{2}$ by showing that the weight update from Equation 2.3 is equivalent to Line 7 from Algorithm 1. Based on our weight update in (2.3), the weight function for the next iteration is given by

$$w_{i,t+1} = exp(-yF_t(x_i)).$$

Expanding $F_t(x_i)$ and rearranging,

$$\begin{aligned} w_{i,t+1} &= exp(-y(\alpha_0 f_0(x_i) + \alpha_1 f_1(x_i) + \cdots + \alpha_t f_t(x_i))) \\ &= exp(-yF_{t-1}(x_i) + (-\alpha_t y f_t(x_i))) \\ &= exp(-yF_{t-1}(x_i)) \cdot exp(-\alpha_t y f_t(x_i)) \\ &= w_{i,t} \cdot exp(-\alpha_t y f_t(x_i)). \end{aligned}$$

Remembering $y, f_t(x) \in \{-1, 1\}$, see that

$$-y f_t(x_i) = |f_t(x_i) - y_i| - 1$$

So we can write

$$\begin{aligned} w_{i,t+1} &= w_{i,t} \cdot exp(\alpha_t(|f_t(x_i) - y_i| - 1)) \\ &= w_{i,t} \cdot exp(\alpha_t \cdot |f_t(x_i) - y_i|) \cdot exp(-\alpha_t) \end{aligned}$$

The term $exp(-\alpha_t)$ is the same for all training instances so it has no effect in terms of the relative change in weights between training instances. Hence the weight update derived from the exponential loss function has the same effect as Line 7 from Algorithm 1, and AdaBoost is minimising the exponential loss function.

Gradient boosting [15] is a variation on boosting which represents the learning problem as gradient descent on arbitrary differentiable loss functions. More specifically, the boosting algorithm executes T boosting iterations to learn a function $F(x)$ that outputs predictions $\hat{y} = F(x)$ minimising some loss function $L(y, \hat{y})$. As before, at each iteration, we add a new estimator $f(x)$ attempting to correct the predictions of the previous iteration:

$$F_{t+1}(x) = F_t(x) + f(x) = y.$$

In the case of the squared error loss function, the ensemble is updated by setting $f(x)$ to:

$$f(x) = y - F_t(x).$$

The $f(x)$ for the current boosting iteration is learned according to the *residuals* $y - F_t(x)$ of the previous iteration. In practice, we approximate $f(x)$, for example, by using a depth limited decision tree. This iterative process can be shown to be a gradient descent algorithm when the loss function is the squared error:

$$L(y, F(x)) = \frac{1}{2}(y - F(x))^2.$$

To see this, consider that the loss over all training instances can be written as

$$J = \sum_i L(y_i, F(x_i)).$$

J is minimised by adjusting $F(x_i)$. The gradient for a particular instance x_i is given by

$$\frac{dJ}{dF(x_i)} = \frac{d\sum_i L(y_i, F(x_i))}{dF(x_i)} = \frac{dL(y_i, F(x_i))}{dF(x_i)} = F_t(x_i) - y_i,$$

and so the residuals are the negative gradient of the squared error loss function:

$$f(x) = y - F_t(x) = -\frac{dL(y, F(x))}{dF(x)}.$$

Assuming an appropriate learning rate, adding a model approximating the negative gradient moves the ensemble closer to a local minimum of the loss function, thus implementing gradient descent.

2.1.3 XGBoost

The XGBoost algorithm by Chen and Guestrin [6] brought several improvements to the standard gradient boosting algorithm. Their algorithm and software implementation is used in several places in this thesis. The standard gradient boosting algorithm by Friedman specifies a model update of the form

$$F_t(x) = F_{m-1}(x) + \alpha_t f(x),$$

where the new model $f(x)$ is fit using first-order gradients and a line-search procedure is applied to find α_t. XGBoost differs in that it calculates second-order functional gradients for each training instance, finding decision tree leaf weights by exact minimisation of a second-order Taylor expansion. This removes the need for a line-search procedure and allows the magnitude of the individual leaf weights in the tree to vary independently. Additionally, XGBoost introduces the idea of a regularisation penalty to the learning objective to prevent individual decision trees in the ensemble from growing too large.. Given some learning objective, such as minimising mean squared error, we add an additional function $\Omega(f)$ that is large when the model is complicated and small when the model is simple. The regularisation term forces the learning algorithm to trade off model complexity for accuracy on the training data, and can help prevent the overfitting of highly complicated models to the training data.

We derive the XGBoost algorithm below, closely following [6]. An objective function with two parts is defined, a loss function over the training set and a regularisation term penalising model complexity:

$$Obj = \sum_i L(y_i, \hat{y_i}) + \sum_k \Omega(f_k)$$

$L(y_i, \hat{y}_i)$ can be any convex, twice differentiable, loss function measuring the difference between the prediction and true label for a given training instance. $\Omega(f_k)$ describes the complexity of tree f_k and is defined as

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda w^2 \tag{2.7}$$

where T is the number of leaves of tree f_k and the vector w contains the leaf weights (i.e., the predicted values stored at the leaf nodes). When $\Omega(f_k)$ is included in the objective function, the algorithm optimises for a less complex model that simultaneously minimizes $L(y_i, \hat{y}_i)$. γT provides a constant penalty for each additional tree leaf and λw^2 penalises extreme weights. γ and λ are user-configurable parameters.

Given that boosting proceeds in an iterative manner, the objective function for the current iteration m is stated in terms of the prediction of the previous iteration $\hat{y_i}^{(m-1)}$, adjusted by the newest tree f_k:

$$Obj^m = \sum_i L(y_i, \hat{y_i}^{(m-1)} + f_k(x_i)) + \sum_k \Omega(f_k).$$

The objective is then minimised via choice of f_k. Taking the second-order Taylor expansion of the objective function about the point $\hat{y}_i^{(m-1)}$, we have

$$Obj^m \simeq \sum_i [L(y_i, \hat{y_i}^{(m-1)}) + g_i f_k(x_i) + \frac{1}{2} h_i f_k(x_i)^2] + \sum_k \Omega(f_k) + constant,$$

where g_i and h_i are the first and second derivatives for instance i:

$$g_i = \frac{dL(y_i, \hat{y_i}^{(m-1)})}{d\hat{y_i}^{(m-1)}}$$
$$h_i = \frac{d^2 L(y_i, \hat{y_i}^{(m-1)})}{d(\hat{y_i}^{(m-1)})^2}.$$

Note that the model $\hat{y_i}^{(m-1)}$ is left unchanged during the optimisation process. The simplified objective function with constants removed is

$$Obj^m = \sum_i [g_i f_k(x) + \frac{1}{2} h_i f_k(x)^2] + \sum_k \Omega(f_k).$$

With the observation that a decision tree predicts constant values within a leaf, $f_k(x)$ is represented as $w_{q(x)}$, where w is the vector containing scores for

each leaf and $q(x)$ maps instance x to a leaf:

$$Obj^m = \sum_{j=1}^{T}[(\sum_{i\in I_j} g_i)w_{q(x)} + \frac{1}{2}(\sum_{i\in I_j} h_i)w_{q(x)}^2] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w^2.$$

Here, I_j refers to the set of training instances in leaf j. Define G_j, H_j as follows:

$$G_j = \sum_{i\in I_j} g_i$$

$$H_j = \sum_{i\in I_j} h_i.$$

As $w_{q(x)}$ is a constant within each leaf and can be represented as w_j, we have

$$Obj^m = \sum_{j=1}^{T}[G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2] + \gamma T. \tag{2.8}$$

The weight w_j for each leaf minimises the objective function at

$$\frac{\partial Obj^m}{\partial w_j} = G_j + (H_j + \lambda)w_j = 0,$$

and the best leaf weight w_j given the current tree structure is

$$w_j^* = -\frac{G_j}{H_j + \lambda}.$$

Substituting w_j^* into Equation 2.8, the objective function for the tree structure becomes

$$Obj^m = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T. \tag{2.9}$$

Equation 2.9 is used in XGBoost as a measure of the quality of a given tree. Decision tree splits are greedily selected to minimise this objective, and the tree structure can be expanded arbitrarily while additional nodes continue to reduce the objective.

2.2 GPU Computing

In this thesis, we make extensive use of graphics processing units (GPUs) as hardware accelerators, developing massively parallel, hardware-optimised algorithms to improve the performance of several machine learning sub-problems.

```
__global__ void example(float *d_a, float *d_b,
  float *d_output, int n){

  int global_tid = blockIdx.x * blockDim.x + threadIdx.x;

  if(global_tid < n){
    d_output[global_tid] = d_a[global_tid] + d_b[global_tid];
  }
}
```

Listing 1: Example CUDA Kernel

GPUs can be thought of as specialised processing units optimised for throughput instead of latency, where traditional CPUs optimise for latency over throughput. GPUs achieve high throughput by launching large numbers of independent operations simultaneously and without order guarantees. For GPUs, memory load operations commonly incur significantly higher latencies than subsequent control flow or arithmetic instructions [41]. The lack of ordering guarantees allows the processor to schedule computation so as to amortise expensive memory read operations. In effect, a single on-chip processing unit (called a streaming multiprocessor in CUDA) swaps between a number of active tasks while waiting for memory load instructions, minimising stalling and maximising total throughput by hiding memory latency [53].

Without loss of generality, implementations in this thesis, are described in terms of the CUDA programming model [41]. Simple programs (kernels) in CUDA can be expressed as operations performed by thousands of parallel threads. Listing 1 shows a most basic kernel that adds two arrays together, with one thread per data element.

More sophisticated programs can be constructed to take advantage of lower-latency memory types, communicate between threads, synchronise threads, or perform atomic memory updates. Figure 2.3 shows an overview of the CUDA

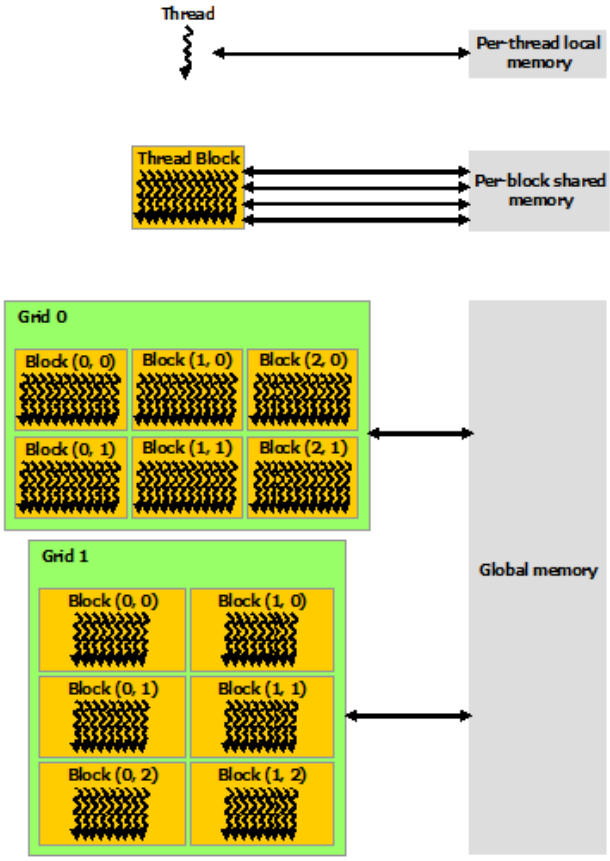
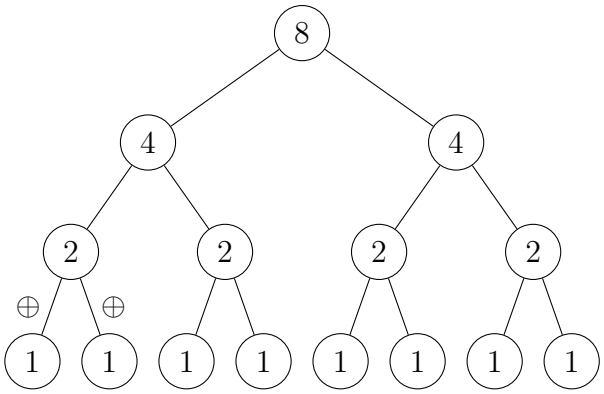


Figure 2.3: CUDA Memory Hierarchy (from CUDA Programminng Guide [41])

memory model, with the individual threads organised into blocks, and blocks organised into grids. Of the three tiers of memory, global memory is the most abundant, has the highest latency, and may be accessed by any thread. A limited amount[1] of shared memory is available to threads in a block at considerably reduced latency compared to global memory. Typically shared memory is used as working space for thread blocks, for example, loading an image tile from global to shared memory for further cooperative processing by a thread block. Threads cannot read or write shared memory for a thread block of which they are not a member. Furthermore, each thread has access to local registers with even lower latency than shared memory. Importantly, threads inside a so-called warp (group of 32 adjacent threads) may read/write each others registers via special intrinsics.

[1]Up to 164kB for GPUs at the time of writing.

Figure 2.4: Sum parallel reduction



2.2.1 Parallel primitives

Parallel primitives are versatile algorithms that are commonly used as building blocks in more sophisticated algorithms. Here we describe two primitives, reduction and prefix-sum, which are used in Chapters 3 and 6 respectively.

Parallel reduction applies a binary associative operator successively to elements of a vector, returning a single value. The associative property guarantees the same result regardless of the order the operator is applied. Given a binary associative operator $\oplus$ and an array of elements the reduction returns

$$(a_1 \oplus a_2 \oplus ... \oplus a_n). \tag{2.10}$$

The reduction operation is easily implemented in parallel by passing partial reductions up a tree, taking $O(\log n)$ iterations given n input items and n processors. This is illustrated in Figure 2.4.

In practice, optimised implementations of reduction do not launch one thread per input item, instead performing parallel reductions over *tiles* of input items, then summing the tiles together sequentially. The size of a tile varies according to the optimal granularity for a given hardware architecture. Note that floatingpoint addition is not strictly associative. This means any reordering of operations is likely to result in a slightly different answer (the same applies to the scan operation described below). Results for the same data can change, for example, when the GPU block size is modified.

CUDA implementations of reduction are typically tiered into three layers -

```
__device__
float warp_reduce(float x) {
  for (int d = 16; d > 0; d /= 2)
    x += __shfl_down(x, d);
  return x;
}
```

Listing 2: Warp Reduction

warp, block and kernel, with each level of the reduction taking advantage of fast instructions or shared memory as available. Individual warps can efficiently perform partial reductions over 32 items using shuffle instructions introduced from Nvidia's Kepler GPU architecture onwards. These partial reductions can then be combined at the block level to complete a title of input. Individual thread blocks can iterate over many input tiles sequentially, summing the reduction from each. When thread blocks finish processing all assigned tiles, each writes a single output to global memory, where a final kernel consisting of a single block sums these partial results into the final output. Reductions are very efficient operations on GPUs. An implementation is given in [34] that approaches the maximum bandwidth of the device tested.

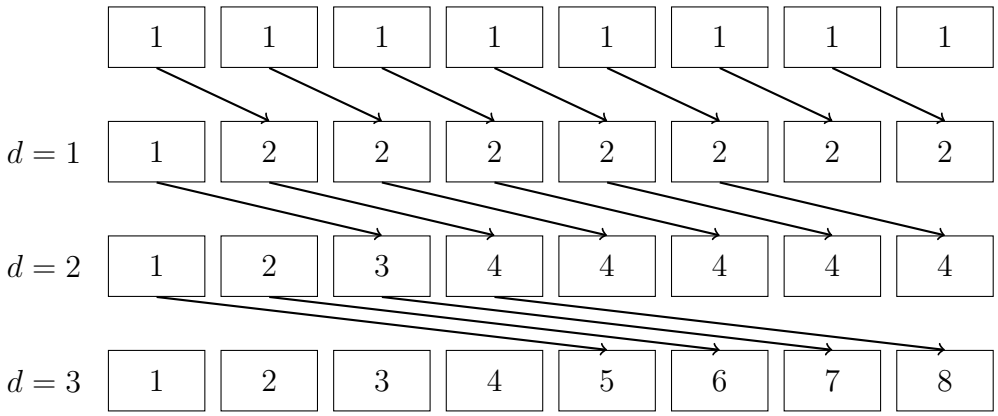
Listing 2 shows a possible implementation of warp level reduction, utilising shuffle intrinsics to communicate between threads in the same warp. The algorithm requires five iterations to sum over 32 items. Shuffle intrinsics are particularly useful for cooperatively solving problems requiring communication between small thread groups, as they consume significantly fewer cycles than shared memory or global memory operations, and do not require synchronisation between warps. Warp intrinsics are used extensively in Chapter 4 to solve batches of small dynamic programming problems with very high throughput.

In contrast to reduction, prefix-sum accepts the vector a and binary associative operator $\oplus$, returning the vector

Algorithm 2: Hillis and Steele scan

```
1 for d=1 to log2 n do
2     for k in parallel do
3         if k ≥ 2^(d−1) then
4             x[k] := x[k − 2^(d−1)] + x[k]
5         end
6     end
7 end
```

Figure 2.5: Simple Parallel Scan Example



$$[a_1, (a_1 \oplus a_2), ..., (a_1 \oplus a_2, ..., a_n)] \tag{2.11}$$

The simple parallel prefix-sum algorithm of Hillis and Steele [21] is given in Algorithm 2.

Figure 2.5 illustrates the process for a vector of length eight, containing all 1s. Assuming eight parallel threads, the algorithm takes $\log_2(8) = 3$ iterations to complete, applying the operator $\oplus$ $O(n \log(n))$ times. As a sequential prefix-sum operation using a single thread performs only $O(n)$ operations, this parallel prefix-sum implementation is not *work efficient*. Under the work-time framework for the analysis of parallel algorithms [23], the work performed by p parallel processors is the total number of primitive operations performed by those processors. A parallel algorithm is said to be work-efficient when it has

the same complexity of total operations as the best known sequential algorithm. A more complicated work-efficient prefix-sum is given by Blelloch in [3], with a GPU implementation described in [18].

Like reduction, prefix-sum may also be implemented at warp, block and then grid level to take advantage of faster localised operations. Listing 3 shows a warp level implementation of Algorithm 2 using shuffle intrinsics. State-of-the-art prefix-sum implementations combine multiple strategies, for example, using the work-inefficient Hillis and Steele scan at the warp level, then switching to a work efficient version at the block or grid level. An excellent summary of GPU prefix-sum strategies is given in [35].

```
__device__
float warp_prefix_sum(float x) {
        int lane_id = threadIdx.x % 32;
        for (int d = 1; d < 32; d *= 2){
                float tmp = __shfl_up(x, d);
                if (lane_id >= offset){
                        x += tmp;
                }
        }
        return x;
}
```

Listing 3: Warp Prefix-Sum

2.3 Shapley Values

In Chapters 4 and 5, we deal with computationally intensive interpretability algorithms for supervised learning models. In particular, we focus on the Shapley value, which is widely used to interpret supervised learning models.

Shapley values [51], named after Lloyd Shapley, are a solution to a cooperative game that attributes the 'winnings' of the game among the players by

measuring the marginal contributions of each player to the final outcome. It is based on considering all possible subgroups of players called "coalitions". Shapley values have become particularly relevant to machine learning in light of the so-called "right to explanation" [26], where consumers affected by automated decision making systems may have the right to have those decisions explained. Thus, the use of black-box machine learning models in domains such as credit score evaluation may be constrained. The Shapley value can provide a principled attribution of inputs with respect to the final classification or regression output of a machine learning model, without regard to its internal complexity or structure.

The Shapley value Sh_i for coalition member i is defined as

$$\mathrm{Sh}_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(|N| - |S| - 1)!}{|N|!}(v(S \cup \{i\}) - v(S)) \tag{2.12}$$

where S is a partial coalition, N is the grand coalition (consisting of all players), and v is the so-called "characteristic function" that is assumed to return the proceeds (i.e., value) obtained by any coalition.

The Shapley value function may also be conveniently expressed in terms of permutations

$$\mathrm{Sh}_i(v) = \frac{1}{|N|!} \sum_{\sigma \in \mathfrak{S}_{|N|}} \left[v([\sigma]_{i-1} \cup \{i\}) - v([\sigma]_{i-1})\right] \tag{2.13}$$

where $[\sigma]_{i-1}$ represents the set of players ranked lower than i in the ordering σ, and $\mathfrak{S}_{|N|}$ is the set of all permutations of length $|N|$. The Shapley value is unique and has the following desirable properties:

Efficiency: The sum of Shapley values for each coalition member is the value of the grand coalition N.

Proposition 2.1. *Assuming* $v(\{\}) = v([\sigma]_0) = 0$,

$$\sum_{i=1}^{|N|} \mathrm{Sh}_i(v) = v(N)$$

Proof.

$$\begin{aligned}
\sum_{i=1}^{n} \mathrm{Sh}_i(v) &= \sum_{i=1}^{|N|} \frac{1}{|N|!} \sum_{\sigma \in \mathfrak{S}_{|N|}} \big[v([\sigma]_{i-1} \cup \{i\}) - v([\sigma]_{i-1})\big] \\
&= \frac{1}{|N|!} \sum_{\sigma \in \mathfrak{S}_{|N|}} \sum_{i=1}^{|N|} \big[v([\sigma]_{i-1} \cup \{i\}) - v([\sigma]_{i-1})\big] \\
&= \frac{1}{|N|!} \sum_{\sigma \in \mathfrak{S}_{|N|}} \big[\sum_{i=1}^{|N|} v([\sigma]_i) - \sum_{i=0}^{|N|-1} v([\sigma]_i)\big] \\
&= \frac{1}{|N|!} \sum_{\sigma \in \mathfrak{S}_{|N|}} v([\sigma]_{|N|}) \\
&= v(N)
\end{aligned}$$

□

Symmetry: If two players have the same marginal effect on each coalition, their Shapley values are the same.

Proposition 2.2. *If,* $\forall S \subseteq N \setminus \{i, j\}, v(S \cup \{i\}) = v(S \cup \{j\})$*, then* $\mathrm{Sh}_i = \mathrm{Sh}_j$ *.*

Proof. Straightforward from the definition of (2.12) □

Linearity: The Shapley values of a sum of games is the sum of the Shapley values of the respective games.

Proposition 2.3.

$$\mathrm{Sh}_i(u + w) = \mathrm{Sh}_i(u) + \mathrm{Sh}_i(w)$$

Proof. Define characteristic function $v(S) = u(S) + w(S)$. Substituting into (2.12)

$$\begin{aligned}
\mathrm{Sh}_i(v) &= \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(|N| - |S| - 1)!}{|N|!} (u(S \cup \{i\}) + w(S \cup \{i\}) - u(S) - w(S)) \\
&= \mathrm{Sh}_i(u) + \mathrm{Sh}_i(w)
\end{aligned}$$

□

Dummy: The coalition member whose marginal impact is always zero has a Shapley value of zero.

Proposition 2.4. *If,* $\forall S \subseteq N \setminus \{i\}, v(S \cup \{i\}) = v(S)$*, then* $\mathrm{Sh}_i = 0$

Proof. Substituting $v(S \cup \{i\}) = v(S)$ into (2.12):

$$\mathrm{Sh}_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(|N| - |S| - 1)!}{|N|!}(v(S \cup \{i\}) - v(S))$$

$$\mathrm{Sh}_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(|N| - |S| - 1)!}{|N|!}(0)$$

$$\mathrm{Sh}_i(v) = 0$$

□

The original work of Shapley [51] dealt with *superadditive* games, meaning that for two disjoint subsets of N, S and T, $v(S \cup T) \geq v(S) + v(T)$. Effectively, adding more players to a coalition is at least as valuable as the separate coalitions. Under this condition, the Shapley value is the unique value satisfying the axioms of efficiency, symmetry, linearity and dummy.

In the machine learning context, Shapley values are used as an attribution of feature relevance to model outputs ([9, 52, 55, 32]). In the terminology of supervised learning, we have some learned model $f(x) = y$ that maps a vector of features x to a prediction y. The value of the characteristic function is assumed to be given by y, and the grand coalition is given by the full set of features. In a partial coalition, only some of the features are considered "active" and their values made available to the model to obtain a prediction. Applying the characteristic function for partial coalitions requires the definition of $f(x_S)$, where the input features x are perturbed in some way according to the active subset S. A taxonomy of possible approaches is given in [10], ranging from simply setting inactive features to zero, to filling inactive features with some conditional expectation based on a background distribution.

Shapley values used in machine learning have some slight differences from the formulation of [51]. The characteristic function evaluated on the empty set $v(\{\})$ is not necessarily zero, and may instead correspond to a global bias

term in the model. This leads to the modified efficiency axiom

$$v(N) = v(\{\}) + \sum_{i=1}^{|N|} \mathrm{Sh}_i(v).$$

Consider the simple linear model evaluated at training example x, with a bias term β_0, set to the average of the training labels y.

$$f(x) = \beta_0 + \sum_{i=1}^{n} \beta_i x_i.$$

Assuming inactive features are considered to have value zero, $f(x)$ can be represented as a game as follows

$$v(S) = \beta_0 + \sum_{i=1}^{n} \mathbb{1}_{i \in S} \beta_i x_i$$

where the indicator function $\mathbb{1}_{i \in S}$ is one if feature i is contained in S, or zero otherwise. In this example, there are no interactions between features, so $v(S \cup T) = v(S) + v(T)$ for disjoint subsets S and T, and the Shapley values are trivially given by

$$\mathrm{Sh}_i(v) = \beta_i x_i.$$

The decision tree model shown in Figure 2.1 is somewhat more complicated, containing interactions between features. Even more complicated models such as large decision tree ensembles or neural networks can be described as "black-boxes", difficult to manually interpret by machine learning practitioners. In these cases, Shapley values take on the important role of describing feature importance, taking into account complex interactions and providing an axiomatic summary coinciding with human intuition [30].

A downside of Shapley values is the exponential time complexity of directly evaluating (2.12). For specific classes of games (for example, the linear model described above), faster algorithms exist, but the problem is NP-hard in general [11]. Chapter 4 discusses optimised polynomial-time algorithms for ensembles of decision trees, and Chapter 5 discusses sampling algorithms for approximation of the Shapley value in the general case.

2.4 Reproducing Kernel Hilbert Spaces

In Chapter 5, we make use of reproducing kernel Hilbert spaces over the symmetric group (the group whose elements are the bijections from the set to itself) to formulate approximate algorithms for solving Shapley value problems, and to experimentally evaluate the quality of point sets in the space of permutations. We use these techniques again in Chapter 6 to formulate a kernel-based statistical tests for detecting uniform distributions of random permutations.

Consider a bivariate function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, whose Gram matrix associated with the arbitrary set $\mathcal{X}$ is positive semi-definite. This function, or *kernel*, induces a reproducing kernel Hilbert space (RKHS) $\mathcal{F}_K$, the set of functions defined as the closure of the span of $\{K(x, \cdot) \mid x \in \mathcal{X}\}$. For two functions f and g,

$$f = \sum_{i=1}^{n} a_i K(x_i, \cdot)$$
$$g = \sum_{j=1}^{m} b_j K(x_j, \cdot),$$

with $n, m \leq |\mathcal{X}|$, define the inner product

$$\langle f, g \rangle_K = \sum_i \sum_j a_i b_i K(x_i, x_j)$$

and norm

$$||f||_K = \sqrt{\langle f, f \rangle_K}.$$

The RKHS $\mathcal{F}_K$ may also be expressed in terms of a feature map $\varphi : \mathcal{X} \to \mathcal{F}_K$ where

$$K(x, x') = \langle \varphi(x), \varphi(x') \rangle_K.$$

Thus, the kernel implicitly represents an inner-product on feature transformations of elements of $\mathcal{X}$.

Aside from being positive semi-definite, certain kernels may also be *universal* and *characteristic*, two properties which become critical in later chapters. A continuous kernel on a compact metric space $\mathcal{X}$ is said to be universal if $\mathcal{F}_K$

is dense in $C(\mathcal{X})$, the space of continuous functions on $\mathcal{X}$ [36]. In other words, for any $g \in C(\mathcal{X})$ and any $\epsilon > 0$ there is a function in the RKHS $f \in \mathcal{F}_K$ such that

$$||f - g||_\infty \leq \epsilon.$$

A kernel is said to be characteristic if the kernel mean embedding of any distribution over $\mathcal{X}$ is unique to that distribution [16]. Define the kernel mean embedding $\mu_{K,P} \in \mathcal{F}_K$ for distribution P as

$$\mu_{K,P}(t) = \langle \mu_{K,P}, K(t, \cdot) \rangle = \mathbb{E}_{x \sim P}[K(t, x)] = \mathbb{E}_{x \sim P}[\varphi(x)],$$

and maximum mean discrepancy for two distributions P and Q as

$$\mathrm{MMD}_K(P, Q) = ||\mu_{K,P} - \mu_{K,Q}||_K.$$

Given a characteristic kernel, $P = Q$ if and only if $\mathrm{MMD}_K(P, Q) = 0$.

We now describe the Kendall and Mallows kernels over the symmetric group $\mathfrak{S}_d$ and summarise the result of [33], showing that the Mallows kernel is both characteristic and universal while the Kendall kernel is neither. We use the universal property in Chapter 5 to extend a convergence result for functions of permutations in a RKHS to the set of all continuous functions on permutations. The characteristic property is used in Chapter 6 to develop a statistical test for the uniform distribution of permutations, using the fact that the mean embedding of this distribution in the RKHS induced by the Mallows kernel is unique.

The Kendall and Mallows kernels were first introduced in [24]. Given two permutations σ and σ' of the same length, both kernels are based on the number of concordant and discordant pairs between the permutations:

$$n_{\mathrm{con}}(\sigma, \sigma') = \sum_{i<j} [\mathbb{1}_{\sigma(i)<\sigma(j)} \mathbb{1}_{\sigma'(i)<\sigma'(j)} + \mathbb{1}_{\sigma(i)>\sigma(j)} \mathbb{1}_{\sigma'(i)>\sigma'(j)}]$$

$$n_{\mathrm{dis}}(\sigma, \sigma') = \sum_{i<j} [\mathbb{1}_{\sigma(i)<\sigma(j)} \mathbb{1}_{\sigma'(i)>\sigma'(j)} + \mathbb{1}_{\sigma(i)>\sigma(j)} \mathbb{1}_{\sigma'(i)<\sigma'(j)}]$$

Assuming the length of the permutation is d, the Kendall kernel, corresponding to the well-known Kendall tau correlation coefficient [28], is

$$K_\tau(\sigma, \sigma') = \frac{n_{\mathrm{con}}(\sigma, \sigma') - n_{\mathrm{dis}}(\sigma, \sigma')}{\binom{d}{2}},$$

or equivalently

$$K_\tau(\sigma, \sigma') = 1 - \frac{2n_{\text{dis}}(\sigma, \sigma')}{\binom{d}{2}}.$$

The Mallows kernel, for $\lambda \geq 0$, is defined as

$$K_M^\lambda(\sigma, \sigma') = e^{-\lambda n_{\text{dis}}(\sigma,\sigma')/\binom{d}{2}}.$$

While the straightforward implementation of the Kendall and Mallows kernels has time complexity $O(d^2)$, an efficient $O(d \log d)$ version can be implemented using a Fenwick tree [13].

Note that K_τ can also be expressed in terms of a feature map of $\binom{d}{2}$ elements,

$$\varphi_\tau(\sigma) = \left(\frac{1}{\sqrt{\binom{d}{2}}} (2\mathbb{1}_{\sigma(i)<\sigma(j)} - 1) \right)_{1 \leq i < j \leq d} \tag{2.14}$$

so that

$$K_\tau(\sigma, \sigma') = \varphi(\sigma)^T \varphi(\sigma').$$

The Mallows kernel corresponds to a more complicated feature map, although still finite dimensional, given in [33].

Let us now consider why the Kendall kernel is not universal, and why the Mallows kernel is both universal and characteristic. Recall that functions in the RKHS induced by the kernel take the form

$$f(\cdot) = \sum_{i=1}^{d!} a_i K(\sigma_i, \cdot). \tag{2.15}$$

Define the $d! \times d!$ kernel Gram matrix M_K with elements $M_{K,ij} = K(\sigma_i, \sigma_j)$. See that the linear span of (2.15) is the span of M_K. Thus, the 'expressiveness' of the RKHS relates to the rank of M_K, and the kernel K is universal if M_K has full rank [33]. The Kendall tau kernel can be expressed as an inner product of its feature map (2.14), so its Gram matrix can be written

$$A^T A = M_\tau,$$

where A is a $\binom{d}{2} \times d!$ matrix with each column the feature map of a permutation. As $\text{rank}(A^T A) = \text{rank}(A)$, and $\text{rank}(A)$ is at most $\binom{d}{2}$, the Kendall tau kernel is not a universal kernel.

We can show the Mallows kernel *is* a universal kernel using Theorem 2.2 of [8], which states that for a separable Hilbert space $\mathcal{F}$, compact metric space $\mathcal{X}$, and an injective (one-to-many) function $\phi : \mathcal{X} \to \mathcal{F}_K$, the kernel

$$K(\sigma, \sigma') = \exp(-\lambda ||\phi(\sigma) - \phi(\sigma')||_{\mathcal{F}}^2)$$

is universal. Applying the Kendall tau kernel's feature map (2.14) as $\phi = \varphi$, we have

$$\begin{aligned} K(\sigma, \sigma') &= \exp(-\lambda ||\varphi(\sigma) - \varphi(\sigma')||_{\mathcal{F}}^2) \\ &= \exp(-\lambda \left[\varphi(\sigma)^T \varphi(\sigma) - 2\varphi(\sigma)^T \varphi(\sigma') + \varphi(\sigma')^T \varphi(\sigma')\right]) \\ &= \exp(-\lambda \left[1 - 2\varphi(\sigma)^T \varphi(\sigma') + 1\right]). \end{aligned}$$

Substituting $K_\tau = \varphi(\sigma)^T \varphi(\sigma') = 1 - \frac{2n_{\text{dis}}(\sigma, \sigma')}{\binom{d}{2}}$, we have

$$\begin{aligned} K(\sigma, \sigma') &= \exp(-\lambda \left[2 - 2(1 - \frac{2n_{\text{dis}}(\sigma, \sigma')}{\binom{d}{2}})\right]) \\ &= \exp(-\lambda 4 \frac{n_{\text{dis}}(\sigma, \sigma')}{\binom{d}{2}}) \end{aligned}$$

which is equivalent to the Mallows kernel up to a constant, and so the Mallows kernel is universal.

From the universal property we can also prove the Mallows kernel is characteristic [16]. The following lemma from [12] (where d represents a metric, not to be confused with d in $\mathfrak{S}_d$) states the uniqueness of a distribution in terms of its mean in the space of bounded continuous functions.

Lemma 2.5. *For a metric space $(\mathcal{X}, d)$ with two Borel probability measures P, Q defined on $\mathcal{X}$, $P = Q$ if and only if $\mathbb{E}_{x \sim P}[f(x)] = \mathbb{E}_{y \sim Q}[f(y)]$ for all bounded continuous functions $f \in C(\mathcal{X})$.*

While Lemma 2.5 offers a means of discriminating between distributions, it is not practical to work with $C(\mathcal{X})$ [16]. Recall that a kernel is characteristic when

$$|\mu_{K,P} - \mu_{K,Q}| = 0, \tag{2.16}$$

if and only if $P = Q$. Consider the universal RKHS $\mathcal{F}_K$, where for any $\epsilon > 0$, $f \in C(\mathcal{X})$, there exists $g \in \mathcal{F}_K$ such that

$$||f(x) - g(x)||_\infty \leq \epsilon.$$

Therefore, we can write

$$|\mathbb{E}[g(x)] - \mathbb{E}[g(y)]| \leq |\mathbb{E}[f(x)] - \mathbb{E}[f(y)]| + |\mathbb{E}[f(x)] - \mathbb{E}[g(x)]| + |\mathbb{E}[g(y)] - \mathbb{E}[f(y)]|$$
$$|\mathbb{E}[g(x)] - \mathbb{E}[g(y)]| \leq |\mathbb{E}[f(x)] - \mathbb{E}[f(y)]| + 2\epsilon$$

$$|\mathbb{E}[f(x)] - \mathbb{E}[f(y)]| - |\mathbb{E}[g(x)] - \mathbb{E}[g(y)]| \leq 2\epsilon.$$

As the function g is a member of our RKHS, then

$$\begin{aligned}\mathbb{E}[g(x)] - \mathbb{E}[g(y)] &= \langle \mu_{P,K}, g\rangle - \langle \mu_{Q,K}, g\rangle \\ &= \langle \mu_{P,K} - \mu_{Q,K}, g\rangle.\end{aligned}$$

Thus, if $||\mu_{P,K} - \mu_{Q,K}||_K = 0$, then $|\mathbb{E}[g(x)] - \mathbb{E}[g(y)]| = 0$ and

$$|\mathbb{E}[f(x)] - \mathbb{E}[f(y)]| \leq 2\epsilon,$$

for any $\epsilon > 0$, and by Lemma 2.5, $P = Q$ if and only if $||\mu_{P,K} - \mu_{Q,K}||_K = 0$.

Therefore the Mallows kernel is both a universal and characteristic kernel on the symmetric group $\mathfrak{S}_d$, properties which become useful in later chapters.

Chapter 3

An Empirical Study of Moment Estimators for Quantile Approximation

An Empirical Study of Moment Estimators for Quantile Approximation

RORY MITCHELL, Nvidia and University of Waikato, New Zealand
EIBE FRANK, University of Waikato, New Zealand
GEOFFREY HOLMES, University of Waikato, New Zealand

We empirically evaluate lightweight moment estimators for the single-pass quantile approximation problem, including maximum entropy methods [13] and orthogonal series with Fourier, Cosine, Legendre, Chebyshev and Hermite basis functions. We show how to apply stable summation formulas to offset numerical precision issues for higher-order moments, leading to reliable single-pass moment estimators up to order 15. Additionally we provide an algorithm for GPU-accelerated quantile approximation based on parallel tree reduction. Experiments evaluate the accuracy and runtime of moment estimators against the state-of-the-art KLL [17] quantile estimator on 14,072 real-world datasets drawn from the OpenML [2] database. Our analysis highlights the effectiveness of variants of moment-based quantile approximation for highly space efficient summaries: their average performance using as few as five sample moments can approach the performance of a KLL sketch containing 500 elements. Experiments also illustrate the difficulty of applying the method reliably and showcases which moment-based approximations can be expected to fail or perform poorly.

CCS Concepts: • **Mathematics of computing** → *Distribution functions*; *Statistical software*; • **Information systems** → *Data mining*; • **Computing methodologies** → *Machine learning algorithms*.

Additional Key Words and Phrases: density estimation, quantiles, data streams

ACM Reference Format:
Rory Mitchell, Eibe Frank, and Geoffrey Holmes. 2020. An Empirical Study of Moment Estimators for Quantile Approximation. *ACM Trans. Datab. Syst.* 1, 1, Article 1 (January 2020), 21 pages. https://doi.org/10.1145/3442337

1 INTRODUCTION

Quantile estimators are fundamental for characterising and manipulating data in a wide range of applications, either as building blocks for other algorithms or as data inspection and visualization tools in their own right. The moments sketch algorithm [13] is a method for extracting approximate quantile information from a data stream using a data structure based on the sample moments of the input stream. This approach is unique and appealing for a number of reasons. The sketch uses a fixed size data structure with a constant memory requirement, typically less than 200 bytes, making it suitable for tracking many data streams concurrently using cache memory only, and rendering it a good candidate for implementation on devices like graphics processing units (GPUs). Incremental updates and merge operations to the moments sketch are trivial and require only basic arithmetic. Across a distributed system, merge operations can be implemented using a single call

Authors' addresses: Rory Mitchell, ramitchellnz@gmail.com, Nvidia, University of Waikato, Department of Computer Science, New Zealand; Eibe Frank, eibe.frank@waikato.ac.nz, University of Waikato, Department of Computer Science, New Zealand; Geoffrey Holmes, geoffrey.holmes@waikato.ac.nz, University of Waikato, Department of Computer Science, New Zealand.

0362-5915/2020/1-ART1 $15.00
https://doi.org/10.1145/3442337

to AllReduce [12], a widely available primitive for efficient communication. Moreover, quantile estimates derived from the sketch typically produce estimates of less than 1% error (as we show in this paper), making it a useful tool in applications where this level of error is acceptable.

However, the basic algorithm presented in [13] suffers from drawbacks that make it difficult to use in practice. The primary challenge is numerical imprecision caused by maintaining higher-order moments using finite-precision arithmetic across a large range of real number inputs. In exact math, the algorithm works in a single pass, but in practice this is only true for well conditioned data. Our empirical evaluation shows that estimates for sample moments on many real-world inputs have unacceptable loss of precision.

In this paper, we investigate the behaviour of moments sketching on 14,072 datasets extracted from the OpenML database [2], evaluating accuracy, speed, and reliability. These datasets vary greatly in size, number of duplicate values, and proportion and scale of floating point values, providing a challenging testbed for an algorithm relying on numerical optimisation and numerical integration for accurate approximations. Considering the presence of duplicate values in real-world data, it is important to note that the theory of some of the quantile approximation methods we consider is based on the assumption of a density function. Our experiments with real-world data containing duplicates give an indication of how well these methods perform when this assumption is violated.

We consider several modifications of the basic moments sketch method: using stable higher order moment summation to improve accuracy on poorly conditioned inputs, using massively parallel GPUs to accumulate sample moments, and solving for the output distribution using Legendre polynomials instead of the more complicated maximum entropy method. Additionally, we compare the accuracy of the moments sketch to a related family of two-pass orthogonal series estimators and the state-of-the-art KLL [17] quantile sketch under a set of space constraints.

2 QUANTILE APPROXIMATION BACKGROUND

The ϕ-quantile q_ϕ is defined as $q_\phi = F^{-1}(\phi)$, where $F(x) = P(X \leq x)$ is the cumulative distribution function of random variable X, and F^{-1} is the generalised inverse, where $F^{-1}(u) = \min\{x : F(x) \geq u\}$. Given a finite sample dataset S of size n that is sorted in ascending order, an estimate of q_ϕ is the element in S whose rank is $\lfloor n\phi \rfloor$, where $\text{rank}(x)$ is defined as the number of elements in S smaller than x. The median is equivalent to the $\phi = 0.5$ quantile.

The naïve algorithm for computing quantiles is to sort the input data and extract the element at index $\lfloor n\phi \rfloor$. One of the earliest efficient approaches is the selection algorithm [3], which can find any given quantile in linear time. The problem of approximating quantiles using sublinear memory and a single pass is well studied. In [23], it was shown that computing the median using p passes over the data requires $\Omega(n^{1/p})$ space. As a result, any algorithm that computes quantiles in sublinear space, and in a single pass, must be an approximation. Approximate algorithms become necessary in the streaming setting, where only a partial view of the dataset can be observed at any given point, or in a distributed setting, where it becomes computationally infeasible to provide access to the entire dataset at each node.

The performance of quantile approximation algorithms delivering a quantile approximation $\hat{q}$ is typically described in terms of space required to achieve ϵ accurate quantile queries, where the true rank is bounded by $(\phi - \epsilon)n$ and $(\phi + \epsilon)n$. The algorithm of Greenwald and Khanna (GK sketch) [14] accumulates samples from an input stream into a buffer and applies a pruning scheme to the active set that maintains error bounds. The algorithm uses $O(\frac{1}{\epsilon}\log(n))$ space and is widely considered to be state-of-the-art. The Q-Digest algorithm of Shrivastava et. al. [29] operates on a fixed size universe $[u]$ (for example, the representable range of a floating-point variable). The

algorithm constructs a sparse tree representation of incoming data over the input universe using dyadic ranges, achieving space complexity of $O(\frac{1}{\epsilon}\log(u))$.

The count-sketch [6] and related count-min-sketch [8] algorithm were designed for tracking frequent elements in data streams. Input elements are accumulated into a set of constant-size hash tables, allowing well-defined probabilistic queries for summary statistics. Both algorithms may be adapted to return approximate quantile queries by partitioning the input universe into dyadic ranges and maintaining hash tables for the corresponding ranges (see [20]). The approximation guarantee of these algorithms relative to size is inferior to the GK sketch or Q-Digest, but they have the advantage of allowing deletions as well as insertions.

We recommend [20] for an in-depth and accessible summary of the above methods. More recent methods such as T-Digest [10] and DDSketch [22] provide improved performance for quantile queries on heavily skewed data and around the tails of the distribution. Of particular relevance to this paper is the KLL sketch [17], which is a randomized sketch providing ϵ accurate quantiles with probability $1-\delta$ using space $O(\frac{1}{\epsilon}\log\log(1/\delta))$.

The above quantile approximation algorithms are sample based — they store a sub-linear number of input elements as a sketch, approximating the empirical quantile function $F^{-1}(\phi)$ as a discontinuous step function via queries to these stored elements. Another approach is to define some parameterised uniformly continuous function closely approximating $F^{-1}(\phi)$. The moments sketch [13], which we discuss in detail in the next section, uses this approach. Another example is [30], in which an orthogonal Hermite series is used to generate quantile estimates. We discuss approaches based on orthogonal series in Section 4 and consider estimation using Legenrde series in particular in Section 5.

3 MOMENTS SKETCH

The strength of the moments sketch [13] lies in providing a low-memory, constant-size data structure that can be trivially updated and merged. It maintains a working set of power sums from orders 0 to l, where the sum at order k is

$$S_k = \sum_{i=0}^{n} x_i^k. \tag{1}$$

The sample moments $\mu_k = E[x^k]$ are obtained from the sketch as

$$\mu_k = \frac{S_k}{S_0}. \tag{2}$$

The moments sketch also maintains the values *min* and *max* bounding the range of the input data so that moments can be scaled and centered into the range $-1 \le x \le 1$. This rescaling and centering is necessary so that operations to obtain a density estimate from sample moments will be sufficiently well conditioned.

We rescale the moments using $s = \frac{2}{max-min}$ to obtain

$$\hat{\mu}_k = s^k \mu_k \tag{3}$$

and then shift by $-c = -s \cdot \frac{min+max}{2}$ using the binomial formula:

$$\tilde{\mu}_k = \sum_{i=0}^{k} \binom{k}{i} \hat{\mu}_i (-c)^{k-i}. \tag{4}$$

Given sample moments in the range $[-1, 1]$, a matching distribution based on a density function $f(x)$ is constructed. Many distributions may exist that match a finite set of sample moments [1].

One method of obtaining a unique distribution makes use of the principle of maximum entropy [19]. The entropy of a distribution's density function $f(x)$ is

$$H = -\int_{-\infty}^{\infty} f(x) \log f(x) dx.$$

Based on the maximum entropy principle, we search for the density function $f(x)$ that maximises H subject to the moment matching constraint

$$\mu_k = \int_{-\infty}^{\infty} x^k f(x) dx.$$

This maximum entropy density represents the least informative (i.e., "simplest") distribution that matches the moments. It is unique and has the form

$$f(x) = \exp(\sum_{k=0}^{l} \theta_k x^k). \tag{5}$$

The parameters θ_k are determined by minimising the loss function

$$L(\theta) = \int_{x_{min}}^{x_{max}} \exp(\sum_{k=0}^{l} \theta_k x^k) dx - \sum_{k=0}^{l} \theta_k \mu_k. \tag{6}$$

Minimising this convex loss function yields the distribution of maximum entropy among those that match the sample moments [16]. This minimisation can be performed by applying Newton's method with gradient

$$\frac{\partial L(\theta)}{\partial \theta_a} = \int_{x_{min}}^{x_{max}} x^a \exp(\sum_{k=0}^{l} \theta_k x^k) dx - \mu_a \tag{7}$$

and Hessian

$$\frac{\partial^2 L(\theta)}{\partial \theta_a \partial \theta_b} = \int_{x_{min}}^{x_{max}} x^a x^b \exp(\sum_{k=0}^{l} \theta_k x^k) dx. \tag{8}$$

Note that $x^a x^b = x^{(a+b)}$, leading to computational efficiencies because integrals from Equation 7 can be reused.

Figure 1 depicts the order $l = 10$ maximum entropy distribution fit to a range of simple datasets, comparing it to the fit obtained using other moment-based methods discussed in Section 4. In particular, we can see the maximum entropy distribution avoids the oscillations of approximations based on Legendre polynomials or the Fourier series.

It was noted in [13] that Chebyshev polynomials of the first kind improve the conditioning of the optimisation problem. This can be achieved by converting the moments $E[x^k]$ to 'Chebyshev moments' $E[T_k(x)]$, using expressions derived from the recurrence relations of Chebyshev polynomials [21] and replacing occurrences of x^k with $T_k(x)$. Our implementation uses this variant. See [9] for a more in-depth discussion of solving maximum entropy problems using Chebyshev polynomials instead of monomials. Additionally, in [13], the authors define a variant of the moments sketch utilising log moments $T_k(\log(x))$, although this requires strictly positive inputs. In Section 8, we evaluate versions of the moments sketch using log Chebyshev moments.

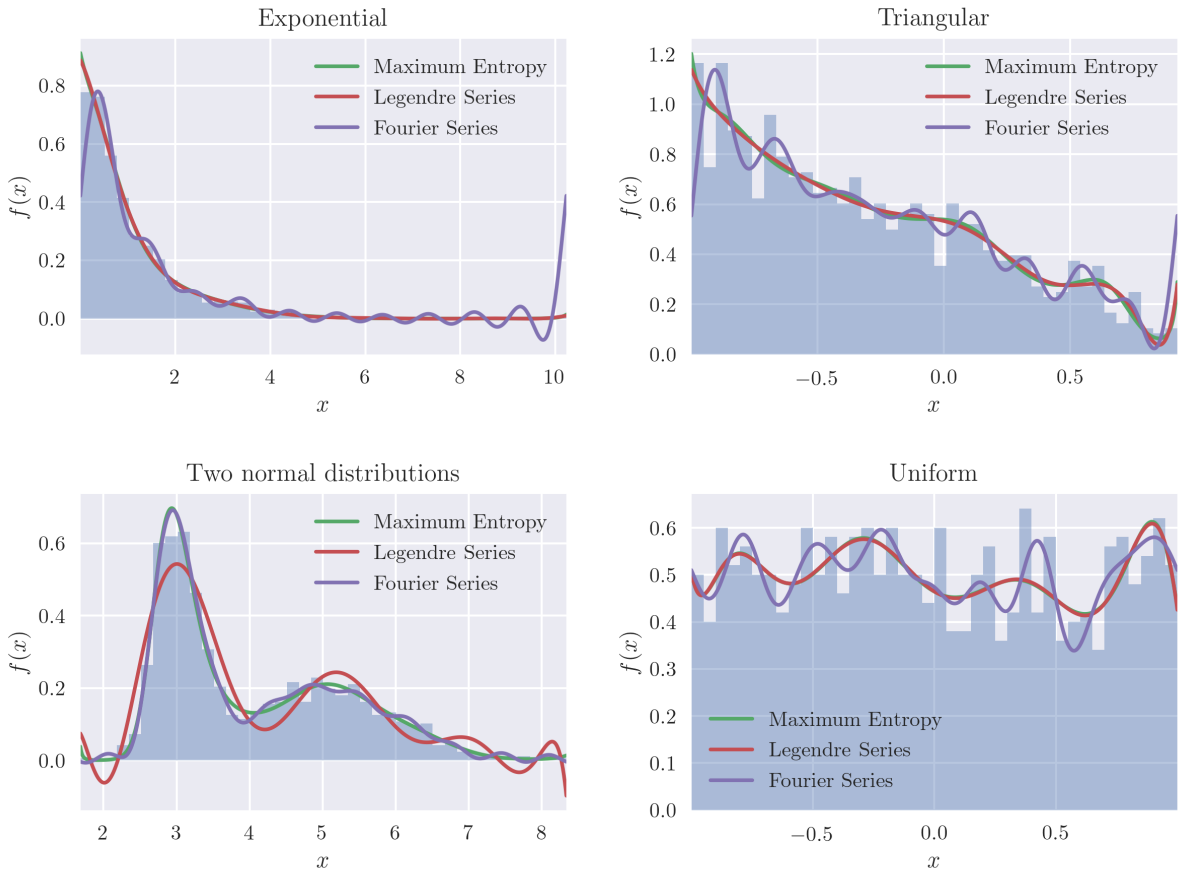


Fig. 1. Various density estimates of order $l = 10$

3.1 Solving for q_ϕ from a density function

Given the moment-matching distribution of maximum entropy, numerical integration techniques can be used to find q_ϕ satisfying $\phi = F(q_\phi) = \int_{-\infty}^{q_\phi} f(x)dx$, for example, integrating $f(x)$ with the trapezoid rule to approximate $F(x)$ and applying binary search to find q_ϕ. Alternatively, the problem can be posed as a system of ordinary differential equations (ODEs) so that widely available software packages will output q_ϕ for given ϕ. Consider the first-order differential equation

$$y'(t) = g(t, y)$$

$$y(t_0) = y_0$$

where we seek the value of function $y(t)$ at time points $t_1, t_2, \cdots$ and have access to the function g and y_0. We substitute t for ϕ, $y(t)$ for the inverse $F^{-1}(\phi)$, and y_0 for $F^{-1}(0) = q_0 = x_{min}$. To apply an ODE solver, we need $g(t) = (F^{-1})'(\phi)$. We know that

$$F(F^{-1}(\phi)) = \phi$$

and taking derivatives with respect to ϕ using the chain rule yields

$$f(F^{-1}(\phi))(F^{-1})'(\phi) = 1$$

which means

$$(F^{-1})'(\phi) = \frac{1}{f(F^{-1}(\phi))}$$

Algorithm 1 Moments sketch algorithm

Input: Datastream $(x_0, x_1, \cdots, x_n)$, maximum order l, $(\phi_0, \phi_1, \cdots, \phi_m)$
Output: Quantiles $(q_{\phi_0}, q_{\phi_1}, \cdots, q_{\phi_m})$

1: Compute power sums $(S_0, S_1, \cdots, S_l)$ from the input stream (Eq. 1)
2: Obtain sample moments $(\mu_0, \mu_1, \cdots, \mu_l)$ (Eq. 2)
3: Rescale sample moments to get $(\hat{\mu}_0, \hat{\mu}_1, \cdots, \hat{\mu}_l)$ (Eq. 3)
4: Shift sample moments to get $(\tilde{\mu}_0, \tilde{\mu}_1, \cdots, \tilde{\mu}_l)$ (Eq. 4)
5: Solve $\theta^* = \arg\min_\theta \int_{x_{min}}^{x_{max}} \exp(\sum_{k=0}^{l} \theta_k x^k) dx - \sum_{k=0}^{l} \theta_k \tilde{\mu}_k$ (Eq. 6)
6: Solve ODE with $y'(t, y) = \frac{1}{\exp(\sum_{k=0}^{l} \theta_k^* y^k)}$, $y_0 = x_{min}$ and $(t_0, t_1, \cdots) = (\phi_0, \phi_1, \cdots)$, yielding $(\tilde{q}_{\phi_0}, \tilde{q}_{\phi_1}, \cdots, \tilde{q}_{\phi_m})$
7: Shift and scale $(\tilde{q}_{\phi_0}, \tilde{q}_{\phi_1}, \cdots, \tilde{q}_{\phi_m})$ back to original domain, output $(q_{\phi_0}, q_{\phi_1}, \cdots, q_{\phi_m})$

where f is the density function from Equation 5. Now, when applying the ODE solver, $F^{-1}(\phi)$ corresponds to the state variable y provided by the solver on each iteration.

The end-to-end moments sketch algorithm for quantile approximation based on maximum entropy density estimation is summarised in Algorithm 1.

4 ORTHOGONAL FUNCTION DENSITY ESTIMATION

An alternative method of approximating a density function for the purposes of quantile approximation is possible via orthogonal functions. A set of functions $g_0(x)$, $g_1(x) \ldots g_k(x)$ is defined to be orthogonal over the interval $a < x < b$ when

$$\int_a^b g_i(x) g_j(x) w(x) dx = \delta_{ij} c_i$$

where $w(x)$ is a weight function, c_i is some scaling constant, and δ_{ij} is the Kronecker delta yielding 0 when $i \neq j$ and 1 when $i = j$.

A function $f(x)$ may be represented by an infinite series in a basis of orthogonal functions

$$f(x) = \sum_{i=0}^{\infty} a_i g_i(x) \tag{9}$$

with coefficients

$$a_i = \frac{1}{c_i} \int_a^b f(x) g_i(x) w(x) dx.$$

This is of practical relevance because the truncation of this series gives a useful approximation to $f(x)$. Some examples of orthogonal basis functions are listed in Table 1. As the table shows, the first three expansions are only applicable to restricted domains, but it is possible to map arbitrary data to the corresponding ranges. Note that the Fourier series is also an example of an orthogonal expansion, but it generates two sets of coefficients instead of one.

The series expansion based on orthogonal functions can naturally be used to estimate a probability density $f(x)$ from samples in the domain $[a, b]$ by realising that

$$\int_a^b f(x) g_i(x) w(x) dx = E[g_i(x) w(x)] \approx \frac{1}{n} \sum_{j=0}^{n} g_i(x_j) w(x_j) \tag{10}$$

In Equation 10, the series coefficients a_i from Equation 9 are estimated by a weighted average of the orthogonal basis functions applied to sample data, yielding a simple and computationally efficient density estimate.

Table 1. Orthogonal expansions

Expansion	Weight	Constant	Domain
Cosine	1	$c_0 = \pi, c_i = \frac{\pi}{2}$	$[0, \pi]$
Chebyshev	$\frac{1}{\sqrt{1-x^2}}$	$c_0 = \pi, c_i = \frac{\pi}{2}$	$[-1, 1]$
Legendre	1	$c_i = \frac{2}{2i+1}$	$[-1, 1]$
Hermite	$e^{-\frac{x^2}{2}}$	$c_i = \sqrt{2\pi} \cdot i!$	$[-\infty, \infty]$

This method of density estimation goes back to Cencov [4]. For a more recent summary, see [11]. Note that most work on orthogonal density estimation is concerned with the estimation of the unobserved density from sample data and truncates the series to obtain a smoother estimate. In contrast, for the quantile approximation problem, we aim to approximate the sample quantiles as closely as possible, and will not truncate the series except out of computational necessity.

A drawback of density estimation based on orthogonal functions is that the estimate may be negative in places. Correspondingly, the cumulative distribution function may not be monotonically increasing, as it would for a true probability density function. Simple corrections can be made by shifting the density function upwards and rescaling such that $\int f(x)dx = 1$. However, these corrections are not required for obtaining valid quantile estimates (see Section 5) and consistently have a negative effect on accuracy, so we do not consider them in this paper.

In Section 8, we provide an evaluation for the methods in Table 1 implemented as two-pass estimators, using a first pass to compute the range of the data, and a second pass to compute the orthogonal series coefficients (Equation 10) on the data mapped into the relevant domain. The Legendre series is an exception, and can be implemented in one pass directly from the moments sketch as explained in the next section. Note that, while the domain of the Hermite series is technically unbounded, its weight function $e^{-\frac{x^2}{2}}$ is poorly conditioned away from 0 and requires scaling to use in practice, so we implement it as a two-pass estimator.

5 LEGENDRE POLYNOMIAL SERIES

The Legendre series can be implemented as a one-pass estimator directly from the moments sketch as an alternative to the method of maximum entropy, providing the second moment-based quantile approximation method we consider in this paper. Given the Legendre polynomials defined by the recurrence relation

$$L_0(x) = 1$$
$$L_1(x) = x$$
$$(n+1)L_{n+1}(x) = (2n+1)xL_n(x) - nL_{n-1}(x),$$

the truncated Legendre series of order k is

$$f(x) \approx \sum_{n=0}^{k} a_n L_n(x)$$

where the coefficients a_n are

$$a_n = \frac{2n+1}{2} \int_{-1}^{1} f(x)L_n(x)w(x)dx$$

and the weight function is $w(x) = 1$. We can form the coefficients a_n directly from the scaled and shifted sample moments of the moments sketch, using the fact that these sample moments represent

data in the domain $[-1, 1]$, the weight function $w(x) = 1$ cancels out, and there are expressions directly converting monomials to Legendre polynomials.

To obtain the coefficients from sample moments, we use the following formula relating monomials to the Legendre polynomials:

$$L_0(x) = 1$$
$$L_1(x) = x$$
$$L_n(x) = \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} \frac{(-1)^k}{2^n} \binom{n}{k} \binom{2n-2k}{n} x^{n-2k}. \tag{11}$$

Applying the above formula with x^{n-2k} replaced by the sample moments μ_{n-2k} gives us the Legendre moments $\mu_n^{Leg} = \int_{-1}^{1} f(x) L_n(x) dx$, and we have the series coefficients by

$$a_n = \frac{2n+1}{2} \mu_n^{Leg}.$$

To get the cumulative distribution function $F(x) \approx \sum_{n=0}^{k} a_n \int_{-1}^{x} L_n(y) dy$, we use the integral of the Legendre polynomials

$$\int L_n(x) dx = \frac{L_{n+1}(x) - L_{n-1}(x)}{2n+1}.$$

As discussed in the context of the other orthogonal series estimators considered in the previous section, the density estimate can be negative in places, the cumulative distribution function $F(x)$ may not be monotonic and so its direct inverse—the quantile function $F^{-1}(y)$—may not exist. We can still find solutions to the inverse, with the caveat that they may not be unique. A standard root finding algorithm such as the bracketed Newton-Raphson approach of [27] is effective. In our implementation, we take the first root found by the algorithm. In practice, if multiple roots occur, they occur close together and all represent valid solutions, so any deterministic method for selecting roots could be used.

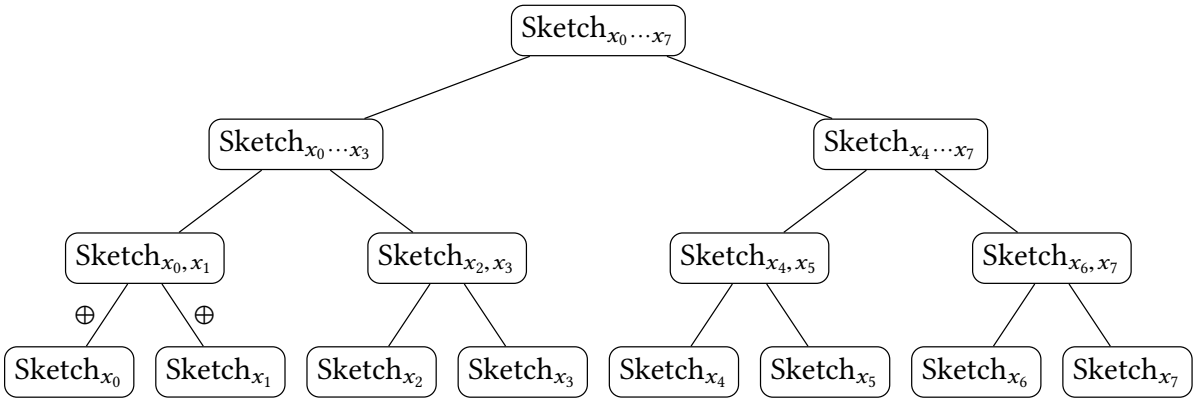
Considering the two moment-based density estimation approaches for quantile approximation we have now discussed, the Legendre series has some advantages over the maximum entropy distribution. It is comparatively simple to compute, avoiding the need to solve an optimisation problem with numerical integrals. The series is a polynomial and therefore has an analytic cumulative distribution function, useful for computing quantiles. Its disadvantage is also that it is a polynomial. The series will be effective for functions that are sufficiently smooth to be well approximated by a polynomial, otherwise it is common to encounter ringing artefacts as in Runge's phenomenon [28]. We include experiments comparing it to the maximum entropy approach in Section 8, concluding that it is a less accurate but considerably simpler one-pass method than the moments sketch with maximum entropy.

6 NUMERICALLY STABLE POWER SUMS

Methods based on the collection of sample moments suffer from limited available numerical precision when implemented on floating-point hardware. Early works such as [31] and [5] discuss the accumulation of rounding errors in the sample variance calculation and propose single-pass update formulas with significantly improved conditioning over naïve methods. More recently, Pébay *et al.* [26] generalise these formulas to higher-order central moments, providing both an incremental update formula for processing elements one at a time and a parallel update formula for merging partial sample moment estimates. The latter enables us to compute the power sum

$$M_k = \sum_{i=0}^{N} (x_i - \mu)^k,$$

Fig. 2. Parallel reduction



where μ is the sample mean, by splitting the full dataset into two multisets A and B of sizes N^A and N^B and combining the central moment estimates for those two multisets M_k^A and M_k^B in a numerically stable manner. Define $\delta_{B,A}$ as

$$\delta_{B,A} = \mu^B - \mu^A.$$

where μ^A, μ^B are the means of the respective multisets, then, for any integer $k \geq 2$,

$$M_k = N^A\left(\frac{-N^B}{N}\delta_{B,A}\right)^k + N^B\left(\frac{N^A}{N}\delta_{B,A}\right)^k + \sum_{i=0}^{k-2}\binom{k}{i}\left[M_{k-i}^A\left(\frac{-N^B}{N}\delta_{B,A}\right)^i + M_{k-i}^B\left(\frac{N^A}{N}\delta_{B,A}\right)^i\right] \quad (12)$$

Equation 12 facilitates numerically stable merging of power sums in parallel or distributed environments. In the case of incrementally summing elements, where $N^B = 1$ and $M_k^B = 0$ for $k > 0$, the formula simplifies to

$$M_k = \left[\frac{N-1}{(-N)^k} + \left(\frac{N-1}{N}\right)^k\right]\delta_{B,A}^k + \sum_{i=0}^{k-2}\binom{k}{i}M_{k-i}^A\left(\frac{-\delta_{B,A}}{N}\right)^i. \quad (13)$$

Given centred power sums from the above formulas, the moment estimates of Section 3 are trivially obtained, replacing S_k with M_k and shifting by $(\mu - c)$ instead of $(-c)$ in Equation 4.

In Section 8, we perform experiments comparing the naïve power sums of Equation 1 to the above formulas, showing that they significantly extend the usable order of sample moments at the cost of some speed due to extra computation.

7 GPU IMPLEMENTATION

Moment-based sketching is an attractive option for GPU accelerated quantile approximation. Computation of power sums by naïve summation or by the pairwise method of Section 6 can be performed in a data-parallel manner, taking advantage of the small fixed-size data structure and simple merge operations of the sketch. Other quantile sketch algorithms such as GK [14] or KLL [17] enable merging of sketches, but effective implementation of these algorithms on GPUs is nontrivial, due to dynamic resizing of the sketch and limited register and shared memory resources available to each GPU thread. In contrast, moment-based sketches can be stored entirely in registers on a per thread basis.

Implementing moment-based sketching in parallel using tree reduction on a GPU provides fast sketching at large input sizes as well as some numerical advantages over incremental sketching. The standard GPU tree reduction algorithm (see [7, 24] for reference) illustrated in Figure 2 takes an array $[x_0, x_1, x_2, \cdots, x_{n-1}]$ and a binary associative operator $\oplus$, returning as a single output value the result of $(x_0 \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_{n-1})$. In the context of moment-based sketching, we can perform a massively parallel reduction by initialising n sketches in n threads, each with a single data point, implementing an operator $\oplus$ for merging two sketches, and applying reduction from the pycuda library [18]. When performing moment-based sketching by maintaining power sums (Equation 1) as in the original moments sketch implementation of [13], two sketches are merged trivially by adding together the pair of power sums for each moment and establishing the minimum and maximum data point over both sketches. Using the numerically stable method from Section 6 instead, the two centered power sums for each moment are merged via Equation 12, and the minimum and maximum element are updated as before.

As well as providing the opportunity for faster sketching, parallel merging of sketches has the advantage of reducing round-off error in floating-point summation. [15] shows that the round-off error from tree-based summation grows proportionally to $O(\epsilon \log_2(n))$, whereas the error from naïve summation grows proportionally to $O(\epsilon n)$. Reduced round-off error is a useful property that arises naturally from parallel reduction on GPUs. As we will see from the experiments in Section 8, the number of usable sample moments is heavily constrained by available numerical precision.

8 EVALUATION

We perform numerical experiments evaluating the accuracy, speed, and numerical stability of moment-based sketching methods. We also include the KLL algorithm (specifically the first of two algorithms described in [17]) as a baseline state-of-the-art one-pass quantile estimator. Additionally, we compare against two-pass algorithms based on orthogonal density estimation. Algorithms are evaluated against the OpenML-CC18 benchmark suite [2], containing 72 machine learning datasets. Each dataset consists of tabular data with varying numbers of rows and columns. Sketches are evaluated for all columns of each dataset, for a combined total of 14,072 unique sketch inputs. We differentiate between an OpenML 'dataset' and 'sketch input' as each dataset contains multiple columns used independently in our experiments. Figure 3 shows a histogram of the sizes of the OpenML sketch inputs, and Figure 4 shows a histogram of the proportion of unique values contained in each sketch input. It is noteworthy that many sketch inputs contains duplicate values, showing the importance of considering each method's ability to deal with such inputs even if their theoretical derivation requires the existence of a probability density.

8.1 Measuring the accuracy of the quantile approximations

To evaluate the accuracy of the quantile approximations provided by the methods considered in this paper, we use the normalised integrated absolute error (NIAE):

$$NIAE = \frac{1}{b-a} \int_{0.0}^{1.0} |\hat{F}^{-1}(\phi) - F^{-1}(\phi)| d\phi \tag{14}$$

where $\hat{F}^{-1}(\phi)$ is the approximation of the quantile function, $F^{-1}(\phi)$ is the empirical quantile function, and (a, b) is the range of the data.

Note that the NIAE error metric has an equivalent formulation in terms of the cumulative distribution function $F(x)$, i.e.,

$$NIAE = \frac{1}{b-a} \int_{a}^{b} |\hat{F}(x) - F(x)| dx.$$

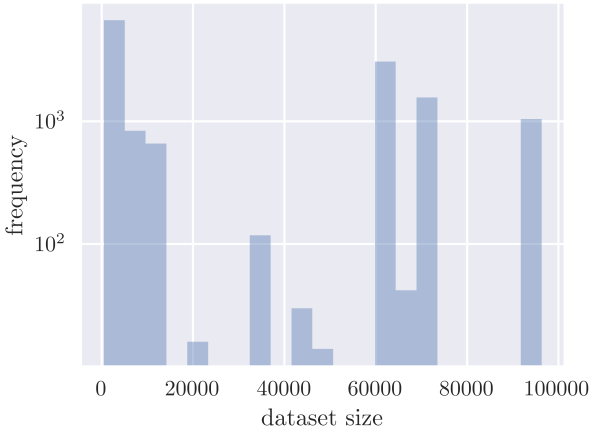


Fig. 3. Histogram of OpenML sketch input size

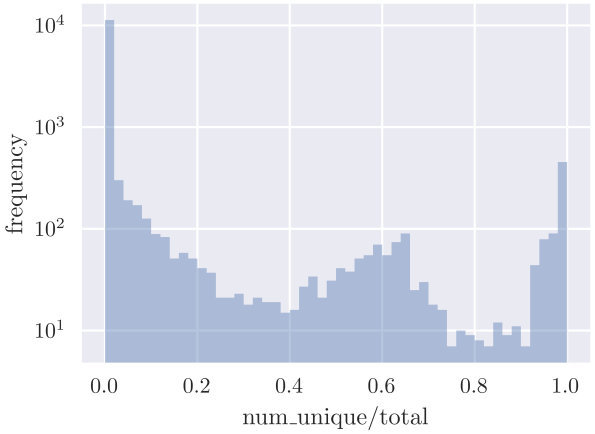
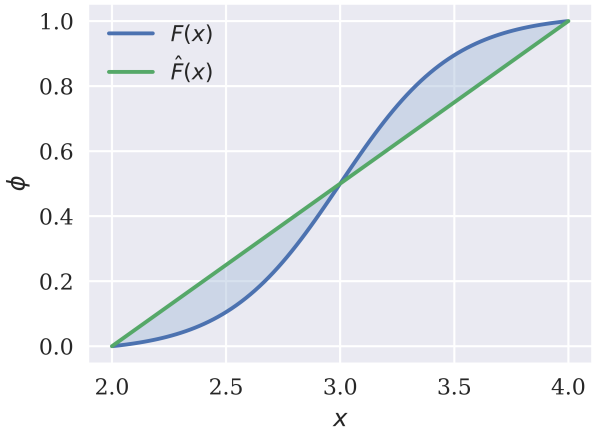
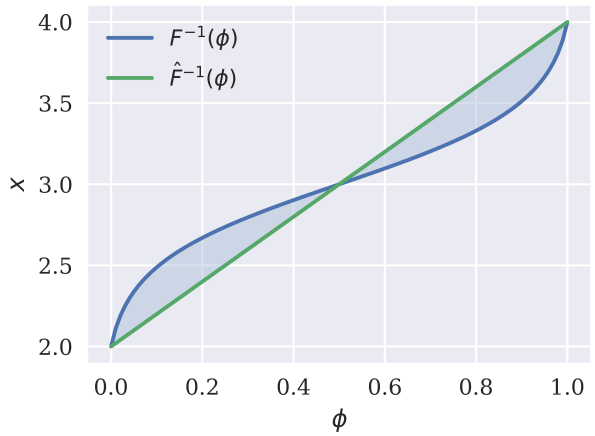


Fig. 4. Histogram of OpenML unique value ratio



(a) $\int_a^b |\hat{F}(x) - F(x)|dx$



(b) $\int_{0.0}^{1.0} |\hat{F}^{-1}(\phi) - F^{-1}(\phi)|d\phi$

Fig. 5. Integrated absolute error visualised

It can be seen from Figure 5 that the two formulations are equivalent. This gives another interpretation of NIAE as the expected value of $|\hat{F}(x) - F(X)|$ assuming x is drawn uniformly from the range (a, b):

$$X \sim U(a, b)$$

$$NIAE = E[|\hat{F}(X) - F(X)|].$$

In our experiments, we evaluate the integral from Equation 14 using the trapezoid rule with 1000 function evaluations. For each of the 14,072 sketch inputs obtained from OpenML, we evaluate the NIAE for our variants of the two moment-based sketching methods based on maximum entropy and Legendre polynomials respectively, and the KLL sketch as one-pass estimators, and various orthogonal series as two-pass estimators. We report overall performance of each estimator using violin plots of the distribution of the NIAE across the 14,072 sketch inputs and also provide the mean NIAE, its standard deviation, the median NIAE, and the minimum and maximum NIAE. Additionally, we report the number of failures due to numerical error, and exclude failures from final statistics (mean, median, std, min, max).

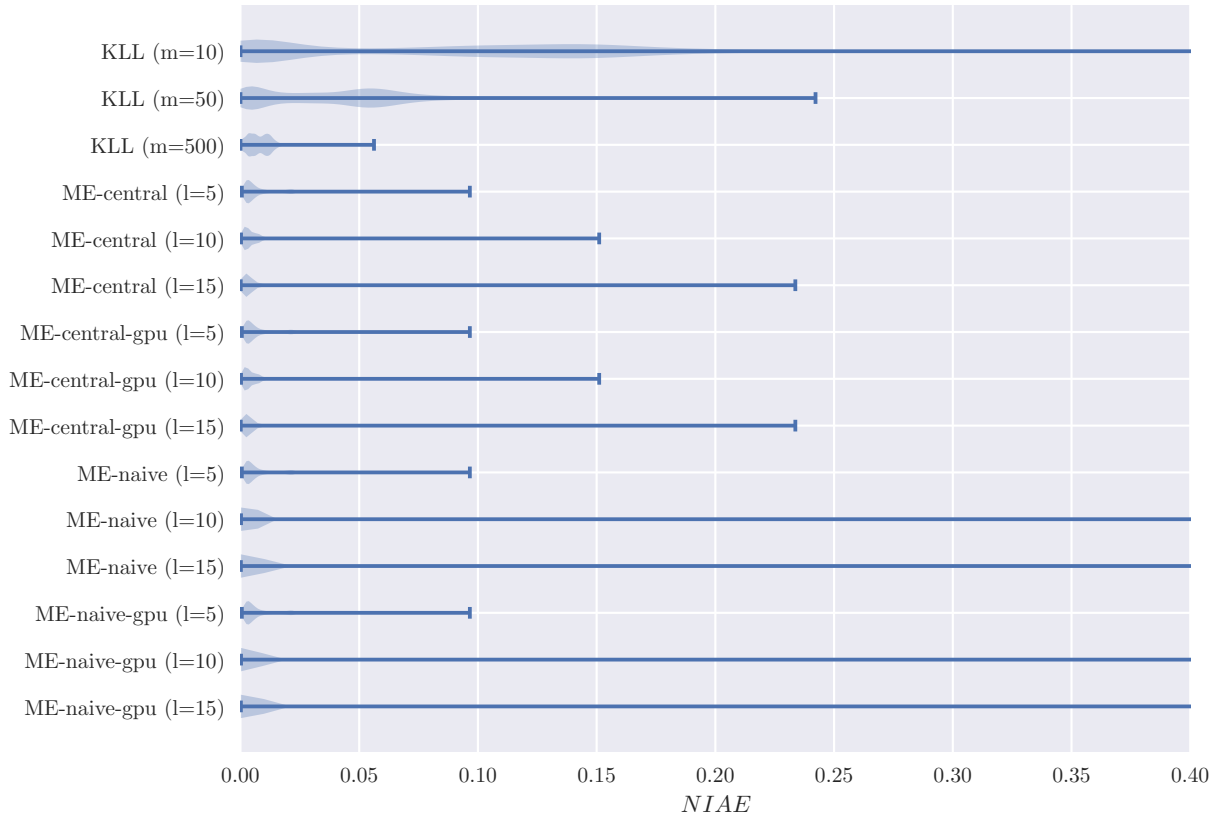


Fig. 6. Violin plot of NIAE measured on OpenML datasets - KLL and ME

8.1.1 One-pass Estimators. Figure 6, Figure 7, and Table 2 summarise the accuracy of the one-pass estimators. We evaluate several variants of the moment-based quantile approximation methods, differentiating on the number of moments l, the method of fitting the distribution (maximum entropy or Legendre polynomials), the algorithm used to accumulate moments (central moment formulas (Eq. 12) or naïve summation (Eq. 1)), and whether the moments were computed using tree reduction on the GPU, or in the standard manner on the CPU. The KLL estimator provides a state-of-the-art baseline. The m in KLL(m=10), etc. indicates the maximum size of the quantile summary.

Results verify the effectiveness of moment-based quantile estimators for answering queries with low memory requirements. Maximum entropy moment-based estimators (ME) with $l = 5$ store only 8 elements in memory ($l + 1$ moments, min and max) while providing a mean error of approximately 0.0084—very close to the mean error of 0.0081 for the KLL (m=500) estimator, which requires storing 500 elements in memory. The Legendre polynomial moment-based estimators (see Figure 7) are all outperformed by the equivalent maximum entropy estimators in terms of mean, median, and maximum error, but they compare favourably to KLL estimators as a low memory summary. Assuming some degree of error can be tolerated, they provide a simpler alternative to the method of maximum entropy, not requiring numerical integrals or the solution of a nonlinear optimisation problem.

Of the two strategies for sample moment summation, the naïve method quickly becomes unstable at $l > 5$, resulting in a number of failures and considerably reduced accuracy when solving for the output distribution using either the method of maximum entropy or the Legendre series. Using the central moment formulas of Pébay *et al.* [26] improves numerical stability dramatically, allowing the

Table 2. Error statistics on OpenML datasets

Estimator	mean	std	median	min	max	failures
KLL (m=10)	0.107872	0.096799	0.103475	0.000014	0.673525	0
KLL (m=50)	0.046058	0.037219	0.045693	0.000000	0.242182	0
KLL (m=500)	0.008104	0.004924	0.007611	0.000000	0.056235	0
Legendre-central (l=5)	0.081809	0.067712	0.083347	0.000062	0.163962	0
Legendre-central (l=10)	0.057979	0.050471	0.055742	0.000063	0.120809	0
Legendre-central (l=15)	0.048106	0.042234	0.046309	0.000052	0.100216	0
Legendre-central-gpu (l=5)	0.081809	0.067712	0.083347	0.000062	0.163962	0
Legendre-central-gpu (l=10)	0.057979	0.050471	0.055742	0.000063	0.120809	0
Legendre-central-gpu (l=15)	0.048106	0.042234	0.046309	0.000052	0.100216	0
Legendre-naive (l=5)	0.081811	0.067714	0.083355	0.000062	0.163962	1
Legendre-naive (l=10)	0.059386	0.054452	0.056886	0.000063	0.938904	131
Legendre-naive (l=15)	0.051749	0.051907	0.049966	0.000052	0.992505	467
Legendre-naive-gpu (l=5)	0.081811	0.067712	0.083347	0.000062	0.163962	0
Legendre-naive-gpu (l=10)	0.058923	0.052333	0.056635	0.000063	0.918821	87
Legendre-naive-gpu (l=15)	0.052065	0.054244	0.049645	0.000052	0.964786	355
ME-central (l=5)	0.008444	0.008461	0.005030	0.000570	0.096600	0
ME-central (l=10)	0.006171	0.008792	0.003628	0.000232	0.151073	0
ME-central (l=15)	0.005955	0.010254	0.003081	0.000119	0.233678	0
ME-central-gpu (l=5)	0.008444	0.008461	0.005030	0.000570	0.096600	0
ME-central-gpu (l=10)	0.006178	0.008791	0.003632	0.000232	0.151073	0
ME-central-gpu (l=15)	0.005962	0.010467	0.003048	0.000120	0.233682	0
ME-naive (l=5)	0.008447	0.008479	0.005030	0.000570	0.096600	0
ME-naive (l=10)	0.007116	0.015522	0.003754	0.000232	0.707496	20
ME-naive (l=15)	0.009518	0.032835	0.003232	0.000170	0.982960	155
ME-naive-gpu (l=5)	0.008447	0.008479	0.005030	0.000570	0.096600	0
ME-naive-gpu (l=10)	0.007112	0.019299	0.003701	0.000232	0.891338	15
ME-naive-gpu (l=15)	0.009329	0.033306	0.003212	0.000170	0.955920	112

Table 3. Two Pass Orthogonal Estimators: Error statistics on OpenML datasets

Estimator	mean	std	median	min	max	failures
Chebyshev (l=5)	0.063520	0.052235	0.067856	0.002066	0.975388	30
Chebyshev (l=10)	0.073640	0.067511	0.062730	0.001530	0.970769	30
Chebyshev (l=15)	0.032477	0.036878	0.037495	0.001335	0.969974	30
Cosine (l=5)	0.054256	0.043132	0.062123	0.001364	0.103496	0
Cosine (l=10)	0.031929	0.025494	0.033809	0.001363	0.062232	0
Cosine (l=15)	0.023803	0.018731	0.025451	0.001363	0.045502	0
Fourier (l=5)	0.220277	0.207780	0.166646	0.000053	0.499986	0
Fourier (l=10)	0.216825	0.208501	0.156493	0.000052	0.499986	0
Fourier (l=15)	0.215401	0.208858	0.151379	0.000051	0.499986	0
Hermite (l=5)	0.227591	0.195979	0.253604	0.001332	0.518415	0
Hermite (l=10)	0.255721	0.237913	0.229341	0.001083	0.593443	0
Hermite (l=15)	0.241525	0.223023	0.227826	0.001115	0.568650	0

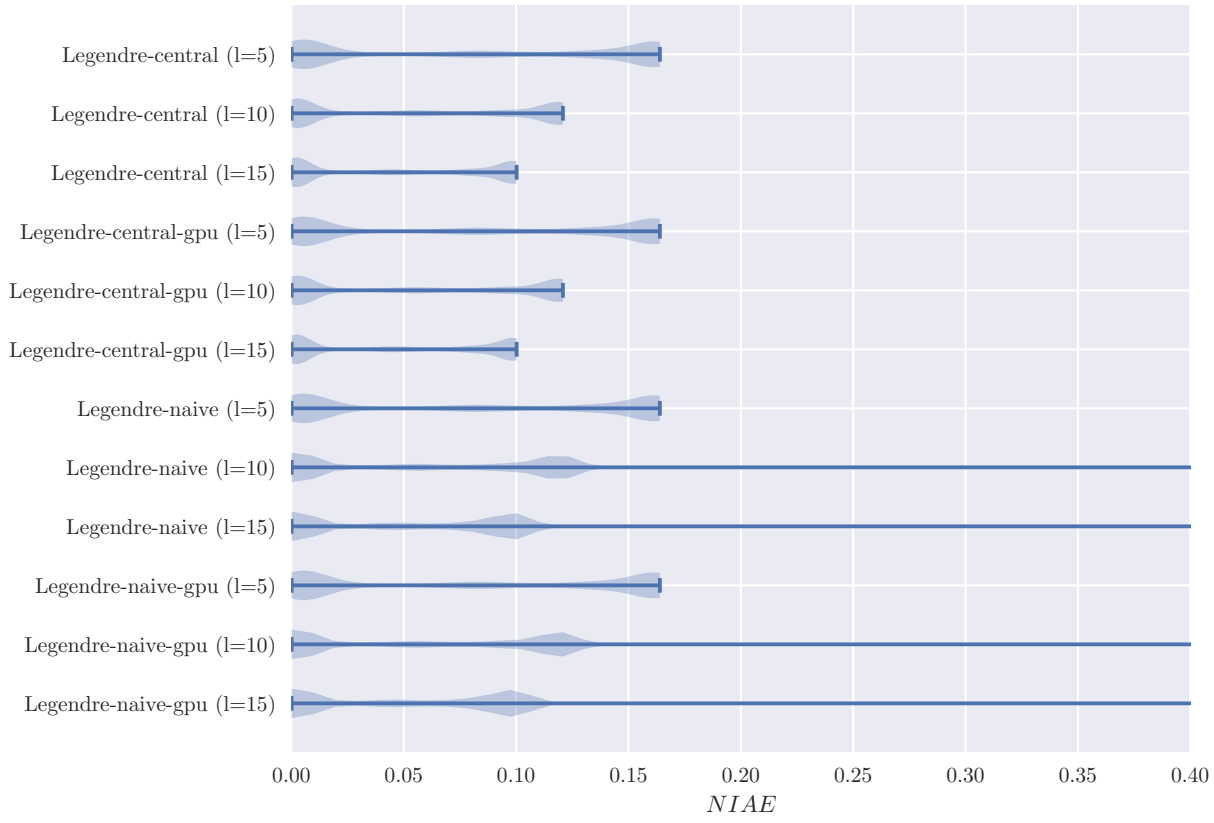


Fig. 7. Violin plot of NIAE measured on OpenML datasets - Legendre series

use of up to $l = 15$ moments without failures. Use of higher-order moments with stable summation allows a moderate reduction in mean error for both the maximum entropy and Legendre series estimators.

Parallel summation of moments on the GPU has a moderate effect on accuracy when naïve summation is used: for example, the mean error of ME-naïve (l=15) reduces from 0.009518 to 0.009329 with ME-naïve-gpu (l=15), and the number of failures reduces from 155 to 112. Overall, it is less effective than the use of stable central moment formulas; when parallel summation is applied in addition to stable moment summation, its effect is not significant.

We also evaluate a variant of the ME estimator using 'log Chebyshev moments' $E[T_k(\log(x))]$ instead of Chebyshev moments $E[T_k(x)]$, with the goal of improving stability for large-valued positive inputs. We test three estimators on the subset of 1,421 OpenML datasets where $x_{min} > 0$. ME-central acts as a baseline (although differing from the above charts as only positive datasets are considered), ME-log-naive computes moments using standard summation on log-transformed inputs, and ME-log-central computes moments using stable summation formulas on log-transformed inputs. The results are summarised by Figure 9 and Table 4.

In general, the use of log moments instead of conventional moments results in significantly reduced accuracy, and requires a priori knowledge that data is positive, limiting its usefulness for general database queries. Stable summation formulas provide considerable benefits even on log transformed inputs, showing that log transformation is not a solution to precision loss during summation.

Additionally, to test the robustness of one pass methods on large inputs, we draw 10^9 samples from the normal distribution $X \sim \mathcal{N}(1000, 1)$, numerically challenging due to its offset from zero.

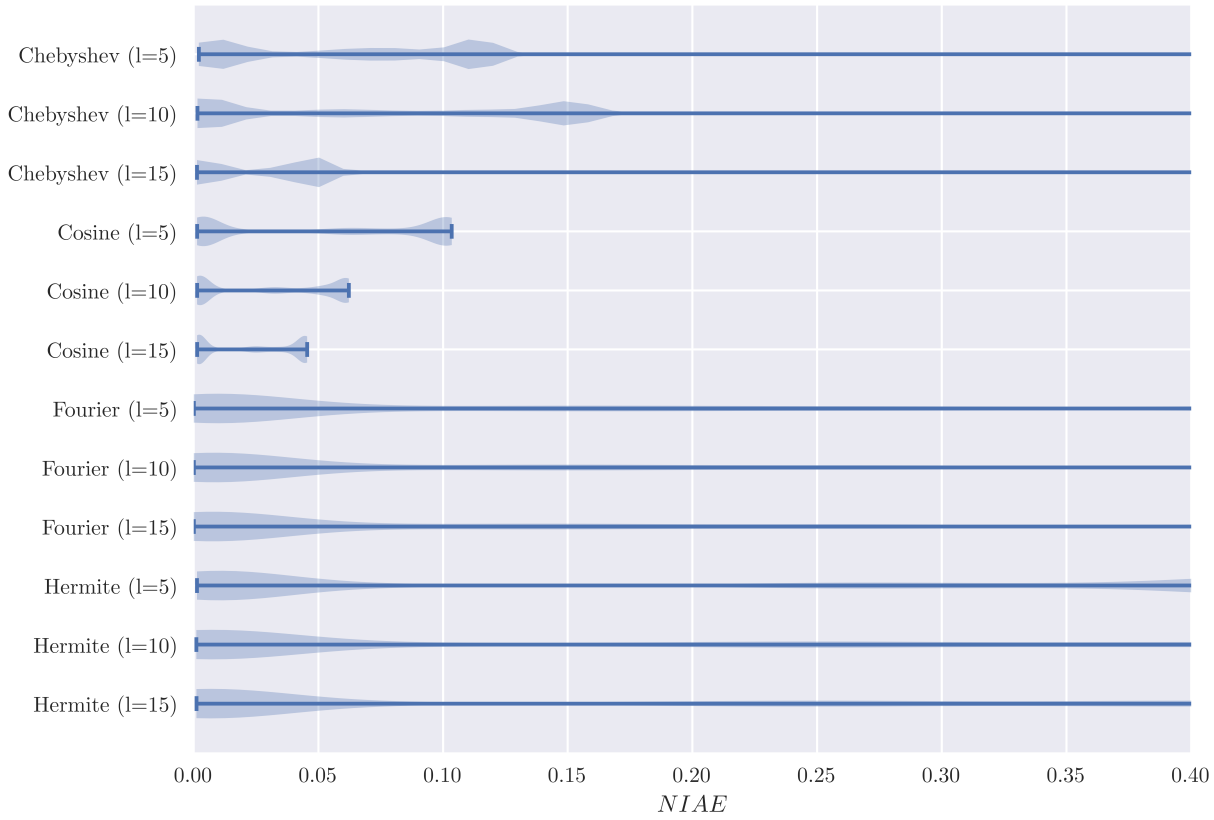


Fig. 8. Two Pass Orthogonal Estimators: Violin plot of NIAE measured on OpenML datasets

Table 4. ME Estimators: Error statistics on positive OpenML datasets

Estimator	mean	std	median	min	max	failures
ME-central (l=5)	0.009323	0.009896	0.004281	0.000913	0.063738	0
ME-central (l=10)	0.005444	0.005474	0.002916	0.000381	0.077327	0
ME-central (l=15)	0.005803	0.006204	0.002591	0.000330	0.080504	0
ME-log-central (l=5)	0.087116	0.107243	0.035912	0.002847	0.714728	1
ME-log-central (l=10)	0.085741	0.104820	0.033300	0.002477	0.779287	1
ME-log-central (l=15)	0.088461	0.109431	0.032717	0.002098	0.788754	1
ME-log-naive (l=5)	0.089040	0.106900	0.042254	0.002849	0.714728	1
ME-log-naive (l=10)	0.149801	0.127953	0.094661	0.003077	0.780095	2
ME-log-naive (l=15)	0.253283	0.161318	0.252858	0.002282	0.756054	35

Table 5 shows the resulting NIAE for one-pass moment estimators. Central moment summation formulas provide robust results, even on poorly conditioned inputs and at large sizes.

8.1.2 Two-pass Estimators. Figure 8 and Table 3 summarise the accuracy of the two-pass estimators based on orthogonal series. The estimators perform a first pass to establish the range of the data and then compute an orthogonal series per Equation 10 after the data has been scaled into the applicable domain. The Chebyshev estimator encounters some numerical failures, attributable to the weight function $w(x) = \frac{1}{\sqrt{1-x^2}}$, which approaches infinity at the endpoints of the domain

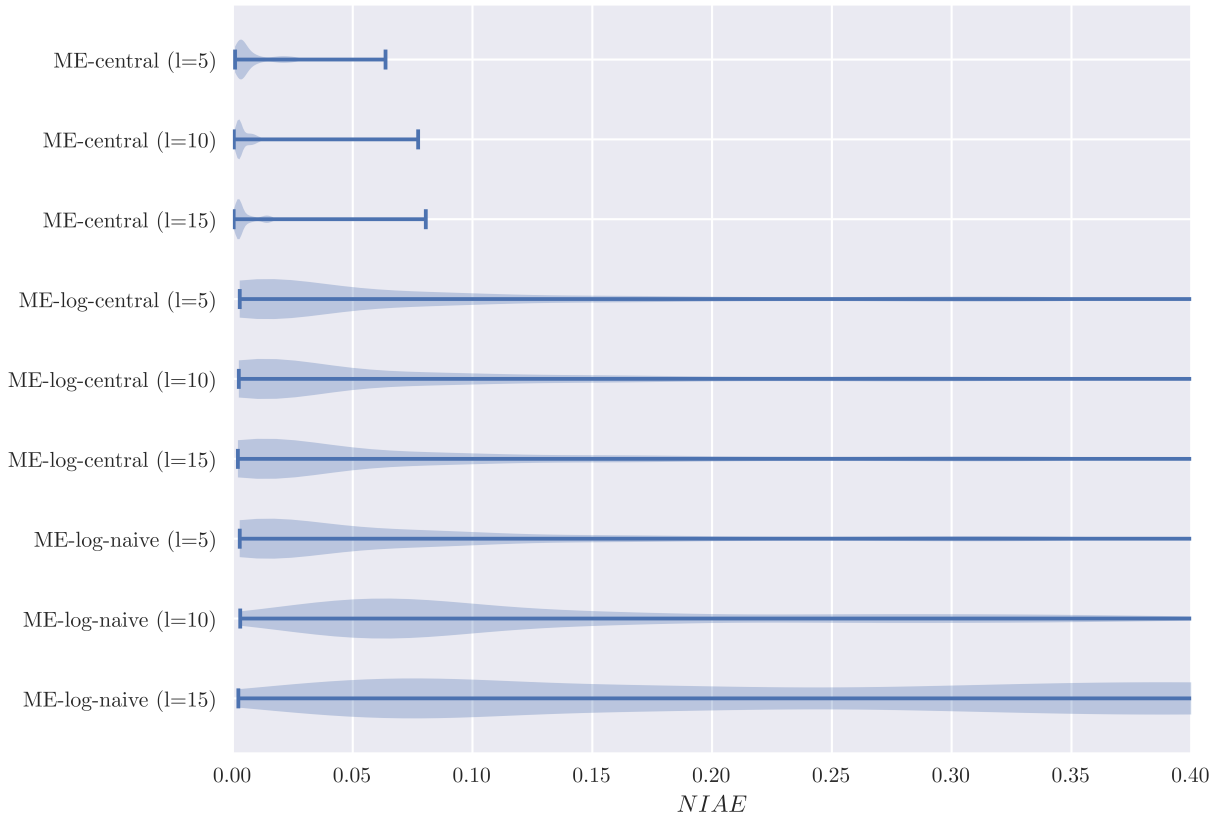


Fig. 9. Violin plot of NIAE measured on positive OpenML datasets

Table 5. NIAE for 10^9 samples from $X \sim \mathcal{N}(1000, 1)$

Estimator	NIAE
Legendre-central (l=5)	0.042877
Legendre-central (l=10)	0.005369
Legendre-central (l=15)	0.000997
Legendre-central-gpu (l=5)	0.042877
Legendre-central-gpu (l=10)	0.005369
Legendre-central-gpu (l=15)	0.000997
Legendre-naive (l=5)	1.000000
Legendre-naive (l=10)	1.000000
Legendre-naive (l=15)	1.000000
ME-central (l=5)	0.001777
ME-central (l=10)	0.001777
ME-central (l=15)	0.001808
ME-central-gpu (l=5)	0.001777
ME-central-gpu (l=10)	0.001777
ME-central-gpu (l=15)	0.001924
ME-naive (l=5)	0.225655
ME-naive (l=10)	1.000000
ME-naive (l=15)	1.000000

Table 6. Summary: Error statistics on OpenML datasets for the top performers

Estimator	mean	std	median	min	max	failures
Cosine (l=15)	0.023803	0.018731	0.025451	0.001363	0.045502	0
KLL (m=500)	0.008104	0.004924	0.007611	0.000000	0.056235	0
Legendre-central-gpu (l=15)	0.048106	0.042234	0.046309	0.000052	0.100216	0
ME-central-gpu (l=5)	0.008444	0.008461	0.005030	0.000570	0.096600	0
ME-central-gpu (l=15)	0.005962	0.010467	0.003048	0.000120	0.233682	0

$(-1.0, 1.0)$. The standout performer of the two-pass orthogonal estimators is Cosine ($l = 15$), with a mean error of 0.023803, being outperformed only by the maximum entropy estimators and KLL at $m = 500$ when comparing to the results in Table 2. It also has the lowest maximum error of any estimator tested, at 0.045502. This, as well as the simpler implementation of the cosine series, make it a compelling quantile estimator in settings where the range of the data is known or two passes are acceptable.

8.1.3 Comparing All Estimators. Table 6 summarises estimators with the lowest mean error using either one or two passes. For the moment-based estimators we show the most accurate summation method, using central moment update formulas and GPU tree reduction. ME-central-gpu (l=15) is the most accurate of all estimators in terms of mean and median NIAE, but with a significantly higher maximum error. KLL (m=500) follows closely behind in mean and median NIAE, but uses more than an order of magnitude more space. ME-central-gpu (l=5) is slightly less accurate on average than its (l=15) version, but benefits from a much lower maximum error and smaller space footprint, while being almost as accurate as KLL (m=500). Legendre-central-gpu (l=15) has more than 5 times the mean or median error of the KLL or maximum entropy methods, but its maximum error is significantly smaller than the maximum entropy estimator and its implementation is simpler. The two-pass Cosine (l=15) estimator sits between KLL and Legendre polynomials in terms of mean and median error, but notably, it has the lowest maximum error of any estimator and a simple implementation.

8.2 Sketch Time

We measure the runtime of sketches with respect to data size, evaluated on the standard normal distribution at varying sizes. Runtime is measured as time taken to accumulate the data stream into the sketch (excluding time to return quantile queries). All algorithms are implemented in native code and run on an AMD Ryzen 7 2700 @3.2GHz CPU and Nvidia 1080Ti GPU.

8.2.1 One-pass. Figure 10 shows the runtime of moment-based sketches and the KLL sketch implemented on the CPU. There is a considerable gap in performance between the KLL sketch and moment based sketches. This is in part due to the simplicity of accumulating power sums with basic arithmetic operations on a static data structure compared to maintaining a more complicated dynamic data structure in memory. Moment-based sketching using the stable central moment formulas is noticeably slower than naïve summation as the algorithmic complexity of Equations 12 and 13 is quadratic in the number of moments. GPU versions of the moments sketch are shown in Figure 11. The GPU versions computing the central moment formulas are considerably faster than CPU versions at sizes $> 10^5$ and GPU versions using naïve summation are moderately faster than their CPU alternatives at sizes $> 10^7$. This is an expected result as GPU architecture is comparatively optimised for throughput over latency whereas CPU architecture is optimised for latency over

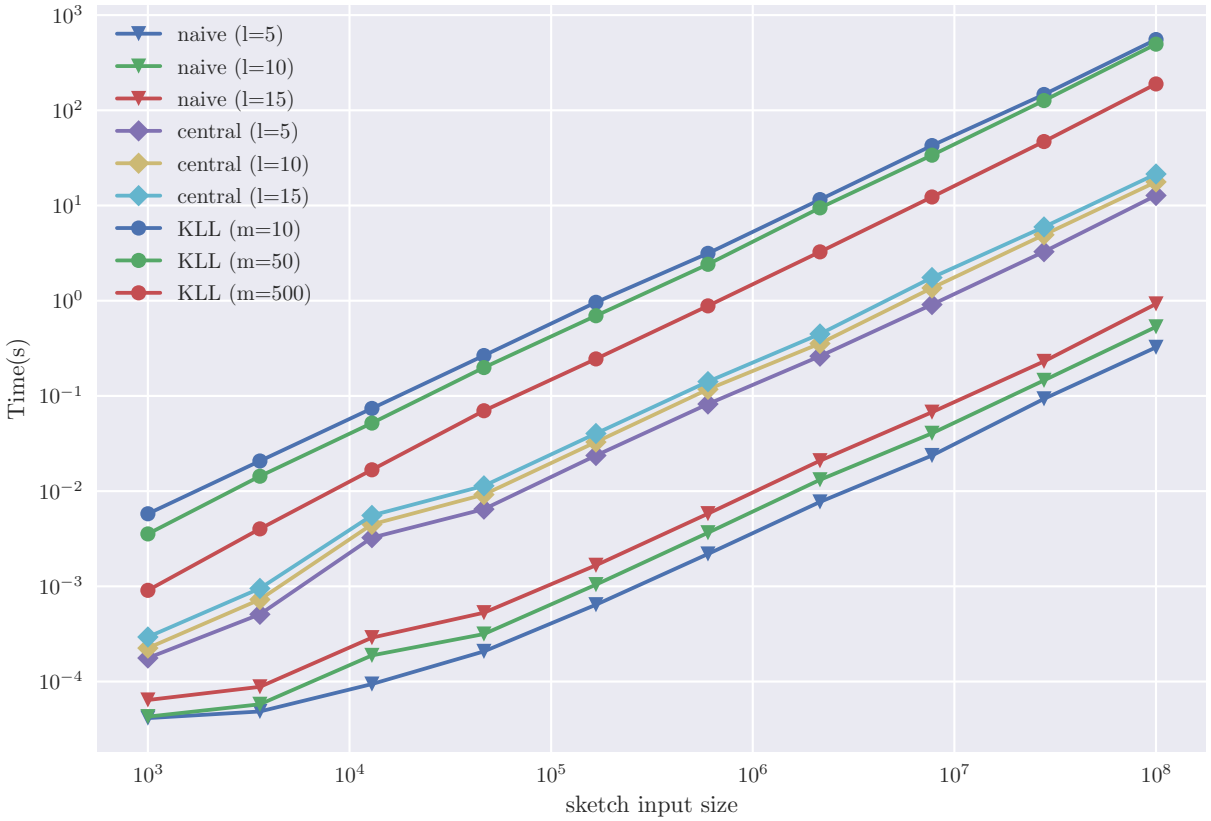


Fig. 10. Runtime of one-pass sketches on CPU

throughput [25]. Naïve summation implemented on the CPU remains the fastest method at smaller input sizes, but the GPU central moments sketch provides a compelling way to compute stable sample moments at higher orders while still retaining good performance.

8.2.2 Two-pass. Figure 12 shows the runtime of orthogonal estimators implemented on the CPU, including a first pass to establish the bounds of the data. This represents the cost of evaluating Equation 10 with different basis functions and weight functions. According to Figure 12, the effective speed of the orthogonal series estimators varies by a constant factor depending on the selection of basis and number of coefficients. Comparing to Figure 10, performance is similar to the moments sketch (in scenarios where the domain of the data is known).

9 CONCLUSION

We perform a study of moment-based sketching methods for the quantile estimation problem on 14,072 real-world datasets, comparing the state-of-the-art KLL estimator, a moment-based maximum entropy method, and orthogonal series estimation based on Legendre polynomials. We verify the result of [13] that moment-based sketching is effective at approximating quantiles with low memory requirements, but we show that its numerical stability deteriorates rapidly at order $l > 5$. We propose the use of stable higher-order moment summation formulas to address issues of numerical stability, and show that a reliable sketching based on higher-order moments is possible at the cost of some extra computation. We show that GPUs can be used to efficiently aggregate stable moment-based sketches at higher orders, allowing moment-based sketches that are both accurate and fast.

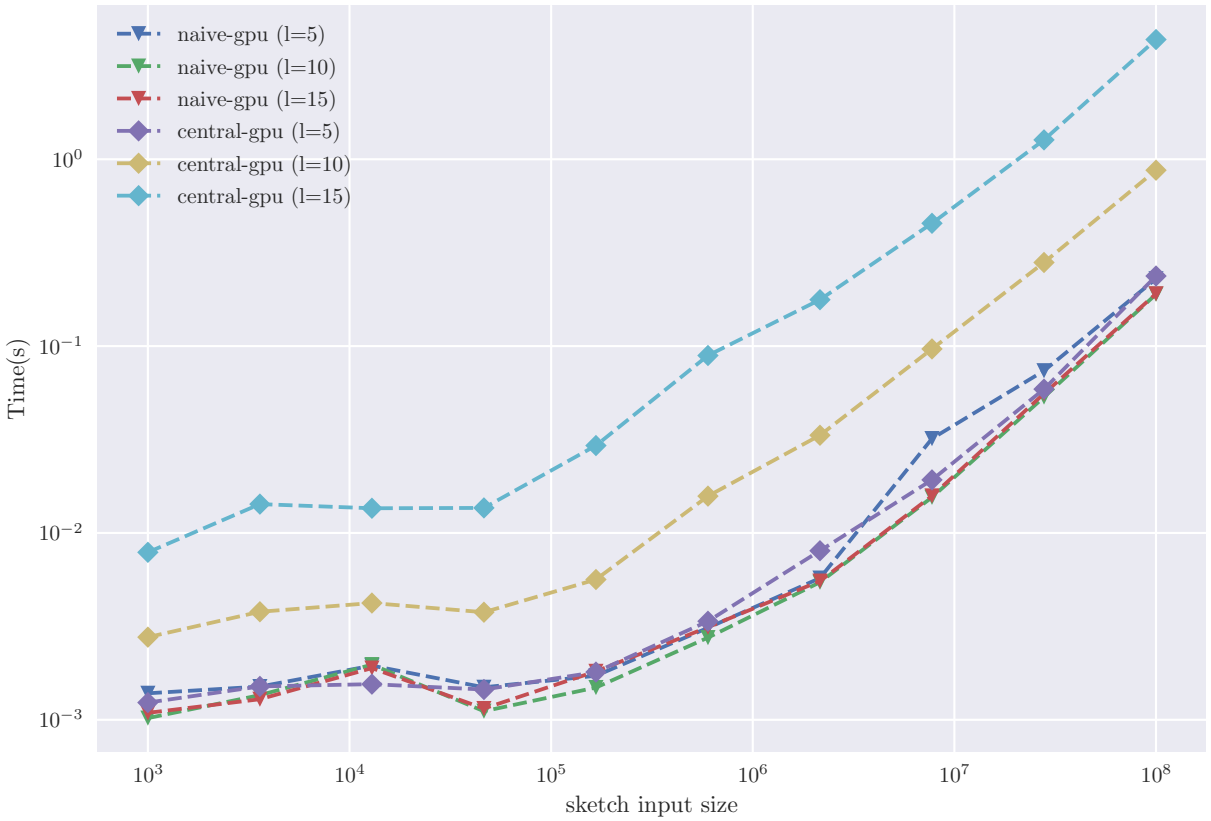


Fig. 11. Runtime of one-pass sketches on GPU

Additionally, we compare single-pass estimators against a related set of two-pass orthogonal series estimators, concluding that the cosine series is also a competitive, space-efficient estimator where two passes over the dataset are acceptable or the domain of the data is known.

Our primary conclusions for practitioners are the following:

- Moments sketch is accurate and fast in space-constrained settings compared to the state-of-the-art sample-based estimator, KLL.
- Moment-based sketching with naïve summation of sample moments is unstable at order $l > 5$.
- More sophisticated moment summation formulas can be used to improve reliability at some computational cost.
- Implementation of moment-based sketching for GPUs is practical and has benefits for speed and accuracy.
- The method of maximum entropy can be substituted with Legendre polynomials for a simpler implementation with reduced accuracy.
- If the domain of the input data is known or two passes are acceptable, we recommend the cosine orthogonal series as a simple, accurate estimator.

REFERENCES

[1] Naum IlÊźich Akhiezer. 1965. *The classical moment problem: and some related questions in analysis.* Vol. 5. Oliver & Boyd, Edinburgh.

[2] Bernd Bischl, Giuseppe Casalicchio, Matthias Feurer, Frank Hutter, Michel Lang, Rafael G. Mantovani, Jan N. van Rijn, and Joaquin Vanschoren. 2017. OpenML Benchmarking Suites. arXiv:stat.ML/1708.03731

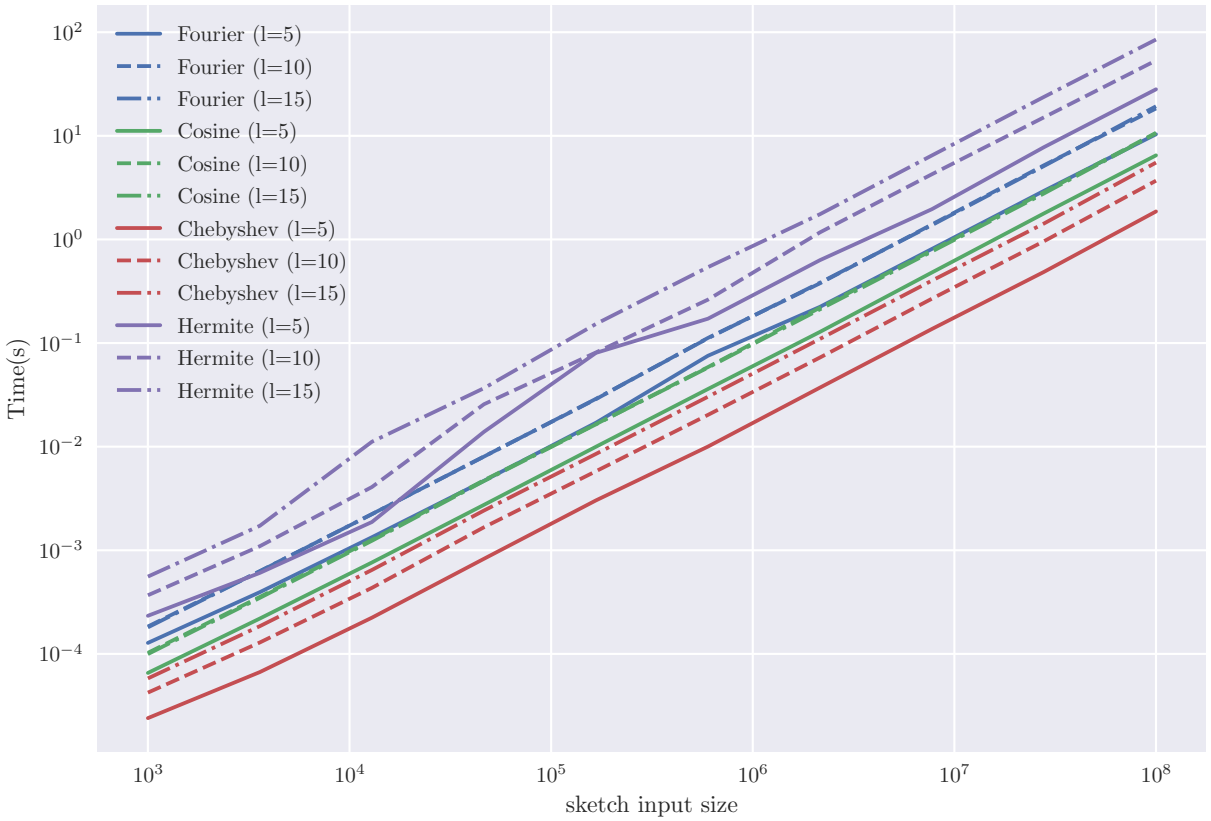


Fig. 12. Runtime of two-pass sketches

[3] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. 1973. Time Bounds for Selection. *J. Comput. Syst. Sci.* 7, 4 (Aug. 1973), 448–461. https://doi.org/10.1016/S0022-0000(73)80033-9

[4] Nikolai N Cencov. 1962. Estimation of an unknown distribution density from observations. *Soviet Math.* 3 (1962), 1559–1566.

[5] Tony F. Chan, Gene H. Golub, and Randall J. LeVeque. 1983. Algorithms for Computing the Sample Variance: Analysis and Recommendations. *The American Statistician* 37, 3 (1983), 242–247. http://www.jstor.org/stable/2683386

[6] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding Frequent Items in Data Streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP '02)*. Springer-Verlag, Berlin, Heidelberg, 693–703. http://dl.acm.org/citation.cfm?id=646255.684566

[7] John Cheng, Max Grossman, and Ty McKercher. 2014. *Professional CUDA C Programming* (1st ed.). Wrox Press Ltd., GBR.

[8] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[9] Jiu Ding, Noah H. Rhee, and Chenhua Zhang. 2016. On Polynomial Maximum Entropy Method for Classical Moment Problem. *Advances in Applied Mathematics and Mechanics* 8, 1 (2016), 117–127. https://doi.org/10.4208/aamm.2014.m504

[10] Ted Dunning and Otmar Ertl. 2019. Computing Extremely Accurate Quantiles Using t-Digests. arXiv:stat.CO/1902.04023

[11] Sam Efromovich. 2010. Orthogonal series density estimation. *Wiley Interdisciplinary Reviews: Computational Statistics* 2, 4 (2010), 467–476.

[12] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. MPI Forum, Knoxville, TN, USA.

[13] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-based quantile sketches for efficient high cardinality aggregation queries. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1647–1660.

[14] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient Online Computation of Quantile Summaries. *SIGMOD Rec.* 30, 2 (May 2001), 58–66. https://doi.org/10.1145/376284.375670

[15] Nicholas J. Higham. 1993. The Accuracy Of Floating Point Summation. *SIAM J. Sci. Comput* 14 (1993), 783–799.

[16] Edwin T Jaynes. 1957. Information theory and statistical mechanics. *Physical review* 106, 4 (1957), 620.

[17] Z. Karnin, K. Lang, and E. Liberty. 2016. Optimal Quantile Approximation in Streams. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, New Brunswick, NJ, USA, 71–78.
[18] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.* 38, 3 (2012), 157–174.
[19] Solomon Kullback. 1997. *Information theory and statistics*. Dover Publications, Massachusetts.
[20] Ge Luo, Lu Wang, Ke Yi, and Graham Cormode. 2016. Quantiles over Data Streams: Experimental Comparisons, New Analyses, and Further Improvements. *The VLDB Journal* 25, 4 (Aug. 2016), 449–472. https://doi.org/10.1007/s00778-016-0424-7
[21] John C Mason and David C Handscomb. 2002. *Chebyshev polynomials*. Chapman and Hall/CRC.
[22] Charles Masson, Jee E. Rim, and Homin K. Lee. 2019. DDSketch. *Proceedings of the VLDB Endowment* 12, 12 (Aug 2019), 2195–2205. https://doi.org/10.14778/3352063.3352135
[23] J Ian Munro and Mike S Paterson. 1980. Selection and sorting with limited storage. *Theoretical computer science* 12, 3 (1980), 315–323.
[24] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53. https://doi.org/10.1145/1365490.1365500
[25] CUDA Nvidia. 2011. Nvidia cuda c programming guide. *Nvidia Corporation* 120, 18 (2011), 8.
[26] Philippe Pébay, Timothy B Terriberry, Hemanth Kolla, and Janine Bennett. 2016. Numerically stable, scalable formulas for parallel and online computation of higher-order multivariate central moments with arbitrary weights. *Computational Statistics* 31, 4 (2016), 1305–1325.
[27] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. 2007. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, Cambridge.
[28] Carl Runge. 1901. Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten. *Zeitschrift für Mathematik und Physik* 46, 224-243 (1901), 20.
[29] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. 2004. Medians and Beyond: New Aggregation Techniques for Sensor Networks. In *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*. ACM, New York, NY, USA, 239–249. https://doi.org/10.1145/1031495.1031524
[30] Michael Stephanou, Melvin Varughese, Iain Macdonald, et al. 2017. Sequential quantiles via Hermite series density estimation. *Electronic Journal of Statistics* 11, 1 (2017), 570–607.
[31] Edward A Youngs and Elliot M Cramer. 1971. Some results relevant to choice of sum and sum-of-product algorithms. *Technometrics* 13, 3 (1971), 657–665.

Chapter 4

GPUTreeShap: Massively Parallel Exact Calculation of SHAP Scores for Tree Ensembles

GPUTreeShap: Massively Parallel Exact Calculation of SHAP Scores for Tree Ensembles

Rory Mitchell[1], **Eibe Frank**[2], **and Geoffrey Holmes**[2]

[1]**Nvidia Corporation**
[2]**Department of Computer Science, University of Waikato, New Zealand**

Corresponding author:
Rory Mitchell[1]

Email address: ramitchellnz@gmail.com

ABSTRACT

SHAP (SHapley Additive exPlanation) values (Lundberg and Lee, 2017) provide a game theoretic interpretation of the predictions of machine learning models based on Shapley values (Shapley, 1953). While exact calculation of SHAP values is computationally intractable in general, a recursive polynomial-time algorithm called TreeShap (Lundberg et al., 2020) is available for decision tree models. However, despite its polynomial time complexity, TreeShap can become a significant bottleneck in practical machine learning pipelines when applied to large decision tree ensembles. Unfortunately, the complicated TreeShap algorithm is difficult to map to hardware accelerators such as GPUs. In this work, we present GPUTreeShap, a reformulated TreeShap algorithm suitable for massively parallel computation on graphics processing units. Our approach first preprocesses each decision tree to isolate variable sized sub-problems from the original recursive algorithm, then solves a bin packing problem, and finally maps sub-problems to single-instruction, multiple-thread (SIMT) tasks for parallel execution with specialised hardware instructions. With a single NVIDIA Tesla V100-32 GPU, we achieve speedups of up to 19x for SHAP values, and speedups of up to 340x for SHAP interaction values, over a state-of-the-art multi-core CPU implementation executed on two 20-core Xeon E5-2698 v4 2.2 GHz CPUs. We also experiment with multi-GPU computing using eight V100 GPUs, demonstrating throughput of 1.2M rows per second—equivalent CPU-based performance is estimated to require 6850 CPU cores.

1 INTRODUCTION

Explainability and accountability are important for practical applications of machine learning, but the interpretation of complex models with state-of-the-art accuracy such as neural networks or decision tree ensembles obtained using gradient boosting is challenging. Recent literature (Ribeiro et al., 2016; Selvaraju et al., 2017; Guidotti et al., 2018) describes methods for "local interpretability" of these models, enabling the attribution of predictions for individual examples to component features. One such method calculates so-called SHAP (SHapley Additive exPlanation) values quantifying the contribution of each feature to a prediction. In contrast to other methods, SHAP values exhibit several unique properties that appear to agree with human intuition (Lundberg et al., 2020). Although exact calculation of SHAP values generally takes exponential time, the special structure of decision trees admits a polynomial-time algorithm. This algorithm, implemented alongside state-of-the-art gradient boosting libraries such as XGBoost (Chen and Guestrin, 2016) and LightGBM (Ke et al., 2017), enables complex decision tree ensembles with state-of-the-art performance to also output interpretable predictions.

However, despite improvements to algorithmic complexity and software implementation, computing SHAP values from tree ensembles remains a computational concern for practitioners, particularly as model size or size of test data increases: generating SHAP values can be more time-consuming than training the model itself. We address this problem by reformulating the recursive TreeShap algorithm, taking advantage of parallelism and increased computational throughput available on modern GPUs. We provide an open source module named GPUTreeShap implementing a high throughput variant of this algorithm

using NVIDIA's CUDA platform. GPUTreeShap is integrated as a backend to the XGBoost library, providing significant improvements to runtime over its existing multicore CPU-based implementation.

2 BACKGROUND

In this section, we briefly review the definition of SHAP values for individual features and the TreeShap algorithm for computing these values from decision tree models. We also review an extension of SHAP values to second-order interaction effects between features.

2.1 SHAP Values

SHAP values are defined as the coefficients of the following additive surrogate explanation model g, a linear function of binary variables

$$g(z') = \phi_0 + \sum_{i=1}^{M} \phi_i z'_i \tag{1}$$

where M is the number of features, $z' \in \{0,1\}^M$, and $\phi_i \in \mathbb{R}$. z'_i indicates the presence of a given feature and ϕ_i its relative contribution to the model output. The surrogate model $g(z')$ is a *local* explanation of a prediction $f(x)$ generated by the model for a feature vector x, meaning that a unique explanatory model may be generated for any given x. SHAP values are defined by the following expression:

$$\phi_i = \sum_{S \subseteq M \setminus \{i\}} \frac{|S|!(|M| - |S| - 1)!}{|M|!} [f_{S \cup \{i\}}(x) - f_S(x)] \tag{2}$$

where M is the set of all features and $f_S(x)$ describes the model output restricted to feature subset S. Equation 2 considers all possible subsets, and so has runtime exponential in the number of features.

We consider models that are decision trees with binary splits. Given a trained decision tree model f and data instance x, it is not necessarily clear how to restrict model output $f(x)$ to feature subset S—when feature j is not present in subset S along a given branch of the tree, and a split condition testing j is encountered, then how do we choose which path to follow to obtain a prediction for x? Lundberg et al. (2020) define a conditional expectation for the decision tree model $E[f(x)|x_S]$, where the split condition on feature j is represented by a Bernoulli random variable with distribution estimated from the training set used to build the model. In effect, when a decision tree branch is encountered, and the feature to be tested is not in the active subset S, we take the output of *both* the left and right branch. More specifically, we use the proportion of weighted instances that flow down the left or right branch during model training as the estimated probabilities for the Bernoulli variable. This process is also how the C4.5 decision tree learner deals with missing values (Quinlan, 1993). It is referred to as "cover weighting" in what follows.

Given this interpretation of missing features, Lundberg et al. (2020) give a polynomial-time algorithm for efficiently solving Equation 2, named TreeShap. The algorithm exploits the specific structure of decision trees: the model is additive in the contribution of each leaf. Equation 2 can thus be independently evaluated for each unique path from root to leaf node. These unique paths are then processed using a quadratic-time dynamic programming algorithm. The intuition of the algorithm is to keep track of the proportion of all feature subsets that flow down each branch of the tree, weighted according to the length of each subset $|S|$, as well as the proportion that flow down the left and right branches when the feature is missing.

We reproduce the recursive polynomial-time TreeSHAP algorithm as presented in Lundberg et al. (2020) in Algorithm 1, where m is a list representing the path of unique features split on so far. Each list element has four attributes: d is the feature index, z is the fraction of paths that flow through the current branch when the feature is not present, o is the corresponding fraction when the feature is present, and w is the proportion of feature subsets of a given cardinality that are present. The decision tree is represented by the set of lists $\{v, a, b, t, r, d\}$, where each list element corresponds to a given tree node, with v containing leaf values, a pointers to the left children, b pointers to the right children, t the split condition, r the weights of training instances, and d the feature indices. The FINDFIRST function returns the index of the first occurrence of a feature in the list m, or a null value if the feature does not occur.

At a high level, the algorithm proceeds by stepping through a path in the decision tree of depth D from root to leaf. According to Equation 2, we have a different weighting for the size of each feature subset,

Algorithm 1 TreeShap

function TS(x, tree)
  ϕ = array of $len(x)$ zeroes
  function RECURSE(j, m, p_z, p_o, p_i)
    m = EXTEND(m, p_z, p_o, p_i)
    if $v_j == leaf$ **then**
      for $i \leftarrow 2$ to $len(m)$ **do**
        $w = sum(\text{UNWIND}(m, i).w)$
        $\phi_{m_i.d} = \phi_{m_i.d} + w(m_i.o - m_i.z)v_j$
    else
      $h, c = x_{d_j} \leq t_j ? (a_j, b_j) : (b_j, a_j)$
      $i_z = i_o = 1$
      k = FINDFIRST($m.d, d_j$)
      if $k \neq nothing$ **then**
        $i_z, i_o = (m_k.z, m_k.o)$
        m = UNWIND(m, k)
      RECURSE($h, m, i_z r_h / r_j, i_o, d_j$)
      RECURSE($c, m, i_z r_c / r_j, 0, d_j$)
  function EXTEND(m, p_z, p_o, p_i)
    $l = len(m)$
    $m = copy(m)$ {m is copied so recursions down other branches are not affected.}
    $m_{l+1}.(d, z, o, w) = (p_i, p_z, p_o, l = 0 \ ? \ 1 : 0)$
    for $i \leftarrow l$ to 1 **do**
      $m_{i+1}.w = m_{i+1}.w + p_o \cdot m_i.w \cdot i/(l+1)$
      $m_i.w = p_z \cdot m_i.w \cdot (l+1-i)/(l+1)$
    return m
  function UNWIND(m, i)
    $l = len(m)$
    $n = m_l.w$
    $m = copy(m_{1 \cdots l-1})$
    for $j \leftarrow l-1$ to 1 **do**
      if $m_i.o \neq 0$ **then**
        $t = m_j.w$
        $m_j.w = n \cdot l/(j \cdot m_i.o)$
        $n = t - m_j.w \cdot m_i.z \cdot (l-j)/l$
      else
        $m_j.w = (m_j.w \cdot l)/(m_i.z \cdot (l-j))$
    for $j \leftarrow i$ to $l-1$ **do**
      $m_j.(d, z, o) = m_{j+1}.(d, z, o)$
    return m
  RECURSE($1, [], 1, 1, 0$)
  return ϕ

although we can accumulate feature subsets of the same size together. As the algorithm advances down the tree, it calls the method EXTEND, taking a new feature split and accumulating its effect on all possible feature subsets of length $1, 2, \ldots$ up to the current depth. The UNWIND method is used to undo addition of a feature that has been added to the path via EXTEND. UNWIND and EXTEND are commutative and can be called in any order. UNWIND may be used to remove duplicate feature occurrences from the path and to compute the final SHAP values. When the recursion reaches a leaf, the SHAP values ϕ_i for each feature present in the path are computed by calling UNWIND on feature i (line 7), temporarily removing it from the path; then, the overall effect of switching that feature on or off is adjusted by adding the appropriate term to ϕ_i.

Given an ensemble of T decision trees, Algorithm 1 has time complexity $O(TLD^2)$, using memory $O(D^2+M)$, where L is the maximum number of leaves for each tree, D is the maximum tree depth, and M the number of features (Lundberg et al., 2020). In this paper, we reformulate Algorithm 1 for massively parallel GPUs.

2.2 SHAP Interaction Values

In addition to the first-order feature relevance metric defined above, Lundberg et al. (2020) also provide an extension of SHAP values to second-order relationships between features, termed SHAP Interaction Values. This method applies the game-theoretic SHAP interaction index (Fujimoto et al., 2006), defining a matrix of interactions as

$$\phi_{i,j} = \sum_{S \subseteq M \setminus \{i,j\}} \frac{|S|!(M-|S|-2)!}{2(M-1)!} \nabla_{ij}(S) \tag{3}$$

for $i \neq j$, where

$$\nabla_{ij}(S) = f_{S\cup\{i,j\}}(x) - f_{S\cup\{i\}}(x) - f_{S\cup\{j\}}(x) + f_S(x) \tag{4}$$

$$= f_{S\cup\{i,j\}}(x) - f_{S\cup\{j\}}(x) - [f_{S\cup\{i\}}(x) - f_S(x)] \tag{5}$$

with diagonals

$$\phi_{i,i} = \phi_i - \sum_{j\neq i} \phi_{i,j}. \tag{6}$$

Interaction values can be efficiently computed by connecting Eq. 5 to Eq. 2, for which we have the polynomial time TreeShap algorithm. To compute $\phi_{i,j}$, TreeShap should be evaluated twice for ϕ_i, where feature j is alternately considered fixed to present and not present in the model. To evaluate TreeShap for a unique path conditioning on j, the path is extended as normal, but if feature j is encountered, it is *not included* in the path (the dynamic programming solution is not extended with this feature, instead skipping to the next feature). If j is considered not present, the resulting ϕ_i is weighted according to the probability of taking the left or right branch (cover weighting) at a split on feature j. If j is considered present, we evaluate the decision tree split condition $x_j < t_j$ and discard ϕ_i from the path not taken.

To compute interaction values for all pairs of features, TreeShap can be evaluated M times, leading to time complexity of $O(TLD^2M)$. Interaction values are challenging to compute in practice, with runtimes and memory requirements significantly larger than decision tree induction itself. In Section 3.5, we show how to reformulate this algorithm to the GPU and how to improve runtime to $O(TLD^3)$ (tree depth D is normally much smaller than the number of features M present in the data).

2.3 GPU Computing

GPUs are massively parallel processors optimised for throughput, in contrast to conventional CPUs, which optimise for latency. GPUs in use today consist of many processing units with single-instruction, multiple-thread (SIMT) lanes that very efficiently execute a group of threads operating in lockstep. In modern NVIDIA GPUs such as the ones we use for the experiments in this paper, these processing units, called "streaming multiprocessors" (SMs), have 32 SIMT lanes, and the corresponding group of 32 threads is called a "warp". Warps are generally executed on SMs without order guarantees, enabling latency in warp execution (e.g., from global memory loads) to be hidden by switching to other warps that are ready for execution (NVIDIA Corporation, 2020).[1]

[1] AMD GPUs have similar basic processing units, called "compute units"; the corresponding term for a warp is "wavefront".

Large speed-ups in the domain of GPU computing commonly occur when the problem can be expressed as a balanced set of vector operations with minimal control flow. Notable examples are matrix multiplication (Fatahalian et al., 2004; Hall et al., 2003; Changhao Jiang and Snir, 2005), image processing (Bo Fang et al., 2005; Moreland and Angel, 2003), deep neural networks (Perry et al., 2014; Coates et al., 2013; Chetlur et al., 2014), and sorting (Green et al., 2012; Satish et al., 2010). Prior work exists on decision tree induction (Sharp, 2008; Mitchell and Frank, 2017) and inference (Sharp, 2012) on GPUs, but we know of no prior work on tree interpretability specifically tailored to GPUs. Related work also exists on solving dynamic programming type problems (Liu et al., 2006; Steffen et al., 2010; Boyer et al., 2012), but dynamic programming is a broad term, and the referenced works discuss significantly different problem sizes and applications (e.g., Smith-Waterman for sequence alignment).

In Section 3, we discuss a unique approach to exploiting GPU parallelism, different from the above-mentioned works due to the unique characteristics of the TreeShap algorithm. In particular, our approach efficiently deals with large amounts of branching and load imbalance that normally inhibits performance on GPUs, leading to substantial improvements over a state-of-the-art multicore CPU implementation.

3 GPUTREESHAP

Algorithm 1 has properties that make it unsuitable for direct implementation on GPUs in a performant way. Conventional multi-threaded CPU implementations of Algorithm 1 achieve parallel work distribution by instances (Chen and Guestrin, 2016; Ke et al., 2017). For example, interpretability results for input matrix X are computed by launching one parallel CPU thread per row (i.e., data instance being evaluated). While this approach is embarrassingly parallel, CPU threads are different from GPU threads. If GPU threads in a warp take divergent branches, performance is reduced, as all threads must execute identical instructions when they are active (Harris and Buck, 2005). Moreover, GPUs can suffer from per-thread load balancing problems—if work is unevenly distributed between threads in a warp, finished threads stall until all threads in the warp are finished. Additionally, GPU threads are more resource-constrained than their CPU counterparts, having a smaller number of available registers due to limited per-SM resources. Excessive register usage results in reduced SM occupancy by limiting the number of concurrent warps. It also results in register spills to global memory, causing memory loads at significantly higher latency.

To mitigate these issues, we segment the TreeShap algorithm to obtain fine-grained parallelism, observing that each unique path from root to leaf in Algorithm 1 can be constructed independently because the ϕ_i obtained at each leaf are additive and depend only on features encountered on that unique path from root to leaf. Instead of allocating one thread per tree, we allocate a *group* of threads to cooperatively compute SHAP values for a given unique path through the tree. We launch one such group of threads for each (unique path, evaluation instance) pair, computing all SHAP values for this pair in a single GPU kernel. This method requires preprocessing to arrange the tree ensemble into a suitable form, avoid some less GPU-friendly operations of the original algorithm, and partition work efficiently across GPU threads. Our GPUTreeShap algorithm can be summarised by the following high-level steps:

1. Preprocess the ensemble to extract all unique decision tree paths.
2. Combine duplicate features along each path.
3. Partition path subproblems to GPU warps by solving a bin packing problem.
4. Launch a GPU kernel solving the dynamic programming subproblems in batch.

These steps are described in more detail below.

3.1 Extract Paths

Figure 1 shows a decision tree model, highlighting a unique path from root to leaf. The SHAP values computed by Algorithm 1 are simply the sum of the SHAP values from every unique path in the tree. Note that the decision tree model holds information about the weight of training instances that flow down paths in the *cover* variable. To apply GPU computing, we first preprocess the decision tree ensemble into lists of path elements representing all possible unique paths in the ensemble. Path elements are represented as per Listing 1.

As paths share information that is represented in a redundant manner in the collection of lists representing a tree, reformulating trees increases memory consumption: assuming balanced trees, it

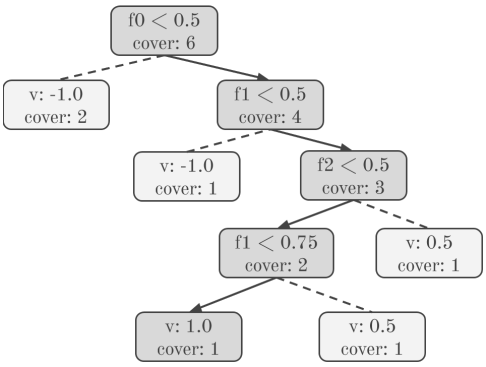


Figure 1. Unique decision tree path

```
struct PathElement {
  // Unique path index
  size_t path_idx;
  // Feature of this split, -1 is root
  int64_t feature_idx;
  // Range of feature value
  float feature_lower_bound;
  float feature_upper_bound;
  // Probability of following this path
  // when feature_idx is missing
  float zero_fraction;
  // Leaf weight at the end of path
  float v;
};
```

Listing 1. Path element structure

increases space complexity from $O(TL)$ to $O(TDL)$ in the worst case. However, this additional memory consumption is not significant in practice.

Considering each path element, we use a lower and an upper bound to represent the range of feature values that can flow through a particular branch of the tree when the corresponding feature is present. For example, the root node in Figure 1 has split condition $f_0 < 0.5$. Therefore, if the feature is present, the left branch from this node contains instances where $-\infty \leq f_0 < 0.5$, and the right branch contains instances where $0.5 \leq f_0 < \infty$. This representation is useful for the next preprocessing step, where we combine duplicate feature occurrences along a decision tree path.

Figure 2 shows two unique paths extracted from Figure 1. An entire tree ensemble can be represented in this form. Crucially, this representation contains sufficient information to compute the ensemble's SHAP values.

3.2 Remove Duplicate Features

Part of the complexity of Algorithm 1 comes from a need to detect and handle multiple occurrences of a feature in a single unique path. In Line 12, the candidate feature of the current recursion step is checked against existing features in the path. If a previous occurrence is detected, it is removed from the path using the UNWIND function. The p_z and p_o values for the old and new occurrences of the feature are multiplied, and the path extended with these new values.

Unwinding previous features to deal with multiple feature occurrences in this manner is problematic for GPU implementation because it requires threads to cooperatively evaluate FINDFIRST and then UNWIND, introducing branching as well as extra computation. Instead, we take advantage of our representation of a tree ensemble in path element form, combining duplicate features into a single occurrence. To do this, recognise that a path through a decision tree from root to leaf represents a single hyperrectangle in the M dimensional feature space, with boundaries defined according to split conditions along the path. The boundaries of the hyperrectangle may alternatively be represented with a lower and upper bound on each feature. Therefore, any number of decision tree splits over a single feature can be reduced to a single range, represented by these bounds. Moreover, note that the ordering of features within a path is irrelevant to the final SHAP values. As noted in (Lundberg et al., 2020), the EXTEND

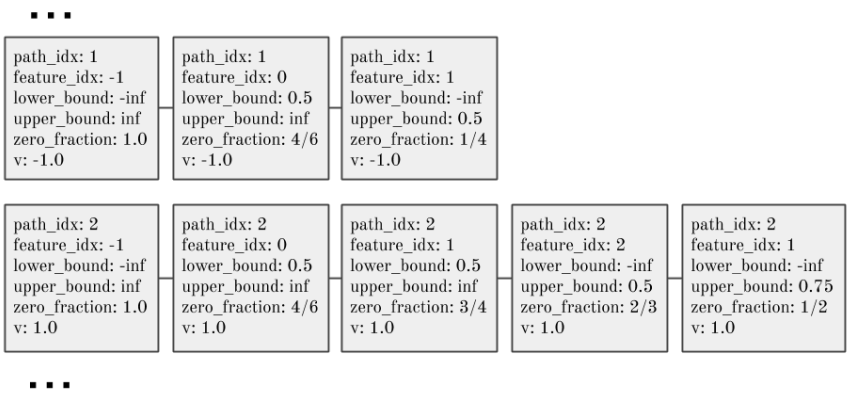


Figure 2. Two unique paths from the decision tree in Figure 1.

and UNWIND functions defined in Algorithm 1 are commutative; therefore, features may be added to or removed from a path in any order, and we can sort unique path representations by feature index, combining consecutive occurrences of the same feature into a single path element.

3.3 Bin Packing For Work Allocation

Each unique path sub-problem identified above is mapped to GPU warps for hardware execution. A decision tree ensemble contains L unique paths, where L is the number of leaves, and each path has length between 1 and maximum tree depth D. To maximise throughput on the GPU, it is important to maximise utilisation of the processing units by saturating them with threads to execute. In particular, given a 32-thread warp, multiple paths may be resident and executed concurrently on a single warp. It is also important to assign all threads processing the same decision tree path to the same warp as we wish to use fast warp hardware intrinsics for communication between these threads and avoid synchronisation cost. Consequently, in our GPU algorithm, sub-problems are constrained to not overlap across warps. This implies that the maximum depth of a decision tree processed by our algorithm must be less than or equal to the GPU warp size of 32. Given that the number of nodes in a balanced decision tree increases exponentially with depth, and real-world experience showing $D \leq 16$ in high-performance boosted decision tree ensembles almost always, we believe this to be a reasonable constraint.

To achieve the highest device utilisation, path sub-problems should be mapped to warps such that the total number of warps is minimised. Given the above constraint, this requires solving a bin packing problem. Given a finite set of items I, with sizes $s(i) \in Z^+$, for each $i \in I$, and maximum bin capacity B, I must be partitioned into the disjoint sets $I_0, I_1, \ldots, I_K$ such that the sum of sizes in each set is less than B. The partitioning minimising K is the optimal bin packing solution. In our case, the bin capacity, $B = 32$, is the number of threads per warp, and our item sizes, $s(i)$, are given by the unique path lengths from the tree ensemble. In general, finding the optimal packing is strongly NP-complete (Garey and Johnson, 1979), although there are heuristics that can achieve close to optimal performance in many cases. In Section 4.1, we evaluate three standard heuristics for the off-line bin packing problem, *Next-Fit* (NF), *First-Fit-Decreasing* (FF), and *Best-Fit-Decreasing* (BFD), as well as a baseline where each item is placed in its own bin. We briefly describe these algorithms and refer the reader to Martello and Toth (1990) or Coffman et al. (1997) for a more in-depth survey.

Next-Fit is a simple algorithm, where only one bin is open at a time. Items are added to the bin as they arrive. If bin capacity is exceeded, the bin is closed and a new bin is opened. In contrast, *First-Fit-Decreasing* sorts the list of items by non-increasing size. Then, beginning with the largest item, it searches for the first bin with sufficient capacity and adds it to the bin. Similarly, *Best-Fit-Decreasing* also sorts items by non-increasing size, but assigns items to the feasible bin with the smallest residual capacity. FFD and BFD may be implemented in $O(n \log n)$ time using a tree data structure allowing bin updates and insertions in $O(\log n)$ operations (see Johnson (1974) for specifics).

Existing literature gives worst-case approximation ratios for the above heuristics. For a given set of items I, let $A(I)$ denote the number of bins used by algorithm A, and $OPT(I)$ be the number of bins for

Table 1. Bin packing time complexities and worst-case approximation ratios

ALGORITHM	TIME	R_A
NF	$O(n)$	2.0
FFD	$O(n \log n)$	1.222
BFD	$O(n \log n)$	1.222

the optimal solution. The approximation ratio $R_A \leq \frac{A(I)}{OPT(I)}$ describes the worst-case performance ratio for any possible I. Time complexities and approximation ratios for each of the three above bin packing heuristic are listed in Table 1, as per Coffman et al. (1997).

As this paper concerns the implementation of GPU algorithms, we would ideally formulate the above heuristics in parallel. Unfortunately, the bin packing problem is known to be hard to parallelise. In particular, FFD and BFD are P-complete, indicating that it is unlikely that these algorithms may be sped up significantly via parallelism (Anderson and Mayr, 1984). An efficient parallel algorithm with the same approximation ratio as FFD/BFD is given in Anderson et al. (1989), but the adaptation of this algorithm to GPU architectures is nontrivial and beyond the scope of this paper. Fortunately, as shown by our evaluation in Section 3.3, CPU-based implementations of the bin packing heuristics give acceptable performance for our task, and the main burden of computation still falls on the GPU kernels computing SHAP values in the subsequent step. We perform experiments comparing the three bin packing heuristics in terms of runtime and impact on efficiency for GPU kernels in Section 4.1.

3.4 The GPU Kernel for Computing SHAP Values

Given a unique decision tree path extracted from a decision tree in an ensemble predictor, with duplicates removed, we allocate one path element per GPU thread and cooperatively evaluate SHAP values for each row in a test dataset X. The dataset X is assumed to be queryable from the device. Listing 2 provides a simplified overview of the GPU kernel that is the basis of GPUTreeShap, further details can be found at `https://github.com/rapidsai/gputreeshap`. A single kernel is launched, parallelising computation of Shapley values across GPU threads in three dimensions:

1. Dataset rows.
2. Unique paths in tree model from root to leaf.
3. Elements in each unique path.

GPU threads are launched according to the solution of the bin-packing problem described in Section 3.3, which allocates threads efficiently to this unevenly sized, three-dimensional problem space. A contiguous thread group of size ≤ 32 is launched and assigned to each dataset row and model path sub-problem.

To enable non-recursive GPU-based implementation of Algorithm 1, it remains to describe how to compute permutation weights for each possible feature subset with the EXTEND function (Line 4), as well as how to UNWIND each feature in the path and calculate the sum of permutation weights (Line 7). The EXTEND function represents a single step in a dynamic programming problem. In the GPU version of the algorithm, it processes a single path in a decision tree, represented as a list of path elements. As discussed above, all threads processing the same path are assigned to the same warp to enable efficient processing. Data dependencies between threads occur when each thread processes a single path element. Figure 3 shows the data dependency of each call to EXTEND on previous iterations when using GPU threads for the implementation. Each thread depends on its own previous result and the previous result of the thread to its "left".

This dependency pattern leads to a natural implementation using warp shuffle instructions, where threads directly access registers of other threads in the warp at considerably lower cost than shared or global memory. Algorithm 2 shows pseudo-code for a single step of a parallel EXTEND function on the device. In pseudocode, we define a shuffle function analogous to the corresponding function in NVIDIA's CUDA language, where the first argument is the register to be communicated, and the second argument is the thread to fetch the register from—if this thread does not exist, the function returns 0, else it returns

```
__device__ float GetOneFraction(
    const PathElement& e, DatasetT X, size_t row_idx) {
  // First element in path (bias term) is always zero
  if (e.feature_idx == -1) return 0.0;
  // Test the split
  // Does the training instance continue down this
  // path if the feature is present?
  float val = X.GetElement(row_idx, e.feature_idx);
  return val >= e.feature_lower_bound &&
         val < e.feature_upper_bound;
}

template <typename DatasetT>
__device__ float ComputePhi(
    const PathElement& e, size_t row_idx,
    const DatasetT& X,
    const ContiguousGroup& group,
    float zero_fraction) {
  float one_fraction = GetOneFraction(e, X, row_idx);
  GroupPath path(group, zero_fraction, one_fraction);
  size_t unique_path_length = group.size();
  // Extend the path
  for (auto unique_depth = 1ull;
       unique_depth < unique_path_length;
       unique_depth++) {
    path.Extend();
  }
  float sum = path.UnwoundPathSum();
  return sum * (one_fraction - zero_fraction) * e.v;
}

template <typename DatasetT, size_t kBlockSize,
          size_t kRowsPerWarp>
__global__ void ShapKernel(
    DatasetT X, size_t bins_per_row,
    const PathElement* path_elements,
    const size_t* bin_segments, size_t num_groups,
    float* phis) {
  __shared__ PathElement s_elements[kBlockSize];
  PathElement& e = s_elements[threadIdx.x];
  // Allocate some portion of rows to this warp
  // Fetch the path element assigned to this
  // thread
  size_t start_row, end_row;
  bool thread_active;
  ConfigureThread<DatasetT, kBlockSize, kRowsPerWarp>(
      X, bins_per_row, path_elements,
      bin_segments, &start_row, &end_row, &e,
      &thread_active);
  if (!thread_active) return;
  float zero_fraction = e.zero_fraction;
  auto labelled_group =
      active_labeled_partition(e.path_idx);
  for (int64_t row_idx = start_row;
       row_idx < end_row; row_idx++) {
    float phi =
        ComputePhi(e, row_idx, X, labelled_group,
                   zero_fraction);
    // Write results
    if (!e.IsRoot()) {
      atomicAdd(&phis[IndexPhi(
                    row_idx, num_groups, e.group,
                    X.NumCols(), e.feature_idx)],
                phi);
    }
  }
}
```

Listing 2. GPU kernel overview — Threads are mapped to elements of a path sub-problem, then groups of threads are formed. These small thread groups cooperatively solve dynamic programming problems, accumulating the final SHAP values using global atomics.

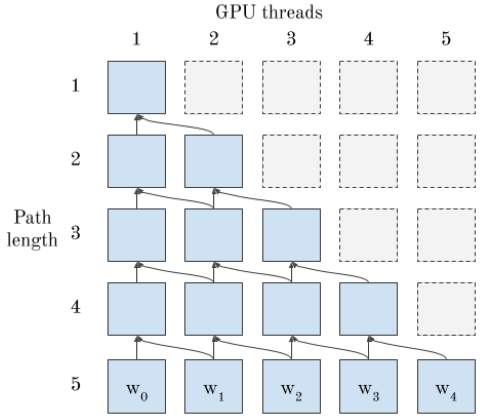


Figure 3. Data dependencies of EXTEND — 5 GPU threads communicate using warp shuffle intrinsics to solve a dynamic programming problem instance.

Algorithm 2 Parallel EXTEND

1: **function** PARALLEL_EXTEND(m, p_z, p_o, p_i)
2: $l = len(m)$
3: $m_{l+1}.(d,z,o,w) = (p_i, p_z, p_o, l = 0 \ ? \ 1 : 0)$
4: **for** $i \leftarrow 2$ to $l+1$ **in parallel, do**
5: $left\_w = \text{shuffle}(m_i.w, i-1)$
6: $m_i.w = m_i.w \cdot p_z \cdot (l+1-i)/(l+1)$
7: $m_i.w = m_i.w + p_o \cdot left\_w \cdot i/(l+1)$
8: **return** m

the register value at the specified thread index. In Algorithm 2, the shuffle function is used to fetch the element $m_i.w$ from the current thread's left neighbour if this neighbour exists, and otherwise returns 0.

Given the permutation weights for the entire path, it is also necessary to establish how to UNWIND the effect of each individual feature from the path to evaluate its relative contribution (Algorithm 1, Line 7). We distribute this task among threads, with each thread "unwinding" a unique feature. Pseudo-code for UNWOUNDSUM is given in Algorithm 3, where each thread i is effectively undoing the EXTEND function for a given feature and returning the sum along the path. Shuffle instructions are used to fetch weights w_j from other threads in the group. The result of UNWOUNDSUM is used to compute the final SHAP value as per Algorithm 1, Line 8.

3.5 Computing SHAP Interaction Values

Computation of SHAP interaction values makes use of the same preprocessing steps as above, and the same basic kernel building blocks, except that the thread group associated with each row/path pair evaluates SHAP values multiple times, iterating over each unique feature and conditioning on that feature being fixed to present or not present respectively. There are some difficulties in conditioning on features with our algorithm formulation so far—conditioning on a feature j requires ignoring it and not adding it to the active path. This introduces complexity when neighbouring threads are communicating via shuffle instructions (see Figure 3). Each thread must adjust its indexing to skip over a path element being conditioned on. We found a more elegant solution is to swap a path element used for conditioning to the end of the path, then simply stop before adding it to the path (taking advantage of the fact that the ordering of path elements is irrelevant). Thus, to evaluate SHAP interaction values, we use a GPU kernel similar to the one used for computing per-feature SHAP values, except that we loop over each unique feature, conditioning on that feature as on or off.

One major difference that arises between our GPU algorithm and the CPU algorithm of Lundberg et al. (2020), is that we can easily avoid conditioning on features that are not present in a given path. It is clear from Equation 5 that $f_{S\cup\{i,j\}}(x) = f_{S\cup\{i\}}(x)$, $f_{S\cup\{j\}}(x) = f_S(x)$ and $\nabla_{ij}(S) = 0$ if we condition on

Algorithm 3 Parallel UNWOUNDSUM

```
1:  function PARALLEL_UNWOUNDSUM(m, p_z, p_o, p_i)
2:      l = len(m)
3:      sum = [] array of l zeroes
4:      for i ← 1 to l + 1 in parallel, do
5:          next = shuffle(m_i.w, l)
6:          for j ← l to 1 do
7:              w_j = shuffle(m_i.w, j)
8:              tmp = (next · (l − 1) + 1)/j
9:              sum_i = sum_i + tmp · p_o
10:             next = w_j − tmp · (l − j) · p_z/l
11:             sum_i = sum_i + (1 − p_o) · w_j · l/((l − j) · p_z)
12:     return sum
```

Table 2. Datasets used to train XGBoost models for Shapley value evaluation.

NAME	ROWS	COLS	TASK	CLASSES	REF
COVTYPE	581012	54	CLASS	8	BLACKARD (1998)
CAL_HOUSING	20640	8	REGR	-	PACE AND BARRY (1997)
FASHION_MNIST	70000	785	CLASS	10	XIAO ET AL. (2017)
ADULT	48842	14	CLASS	2	KOHAVI (1996)

feature j that is not present in the path. Therefore, our approach has runtime proportional to $O(TLD^3)$ instead of $O(TLD^2M)$ by exploiting the limited subset of possible feature interactions in a tree branch. This modification has a significant impact on runtime in practice (because, normally, $M >> D$).

4 EVALUATION

We train a range of decision tree ensembles using the XGBoost algorithm on the datasets listed in Table 2. Our goal is to evaluate a wide range of models representative of different real-world settings, from simple exploratory models to large ensembles of thousands of trees. For each dataset, we train a small, medium, and large variant by adjusting the number of boosting rounds (10, 100, 1000) and maximum tree depth (3, 8, 16). The learning rate is set to 0.01 to prevent XGBoost learning the model in the first few trees and producing only stumps in subsequent iterations. Using a low learning rate is also common in practice to minimise generalisation error. Other hyperparameters are left as default. Summary statistics for each model variant are listed in Table 3, and our testing hardware is listed in Table 4.

4.1 Evaluating Bin Packing Performance

We first evaluate the performance of the NF, FFD, and BFD bin packing algorithms from Section 3.3. We also include "none" as a baseline, where no packing occurs and each unique path is allocated to a single warp. All bin packing heuristics are single-threaded and run on the CPU. We report the execution time (in seconds), utilisation, and number of bins used (K). Utilisation is defined as $\frac{\sum_{i \in I} s(i)}{32K}$, the total weight of all items divided by the bin space allocated, or for our purposes, the fraction of GPU threads that are active for a given bin packing. Poor bin packings waste space on each warp and underutilise the GPU.

Results are summarised in Table 5. "None" is clearly a poor choice, with utilisation between 0.1 and 0.3, with worse utilisation for smaller tree depths–for example, small models with maximum depth three allocate items of size three to warps of size 32. The simple NF algorithm often provides competitive results with fast runtimes, but it can lag behind FFD and BFD when item sizes are larger, exhibiting utilisation as low as 0.79 for *fashion_mnist-large*. FFD and BFD achieve better utilisation than NF in all cases, reflecting their superior approximation guarantees. Interestingly, FFD and BFD achieve the same efficiency on every example tested. We have verified that they can produce different packings on contrived examples, but there is no discernible difference for our application. FFD and BFD have longer runtimes than NF due to their $O(n \log n)$ time complexity. FFD is slightly faster than BFD because it

Table 3. XGBoost models used for evaluation. Small, medium and large variants are created for each dataset.

MODEL	TREES	LEAVES	MAX_DEPTH
COVTYPE-SMALL	80	560	3
COVTYPE-MED	800	113888	8
COVTYPE-LARGE	8000	6636440	16
CAL_HOUSING-SMALL	10	80	3
CAL_HOUSING-MED	100	21643	8
CAL_HOUSING-LARGE	1000	3317209	16
FASHION_MNIST-SMALL	100	800	3
FASHION_MNIST-MED	1000	144154	8
FASHION_MNIST-LARGE	10000	2929521	16
ADULT-SMALL	10	80	3
ADULT-MED	100	13074	8
ADULT-LARGE	1000	642035	16

Table 4. Details of Nvidia DGX-1 used for benchmarking.

PROCESSOR	DETAILS
CPU	2X 20-CORE XEON E5-2698 V4 2.2 GHZ
GPU	8X TESLA V100-32

uses a binary tree packed into an array, yielding greater cache efficiency, but its implementation is more complicated. In contrast, BFD is implemented easily using *std::set*.

Based on these results, we recommend BFD for its strong approximation guarantee, simple implementation, and acceptable runtime when packing jobs into batches for GPU execution. Its runtime is at most 1.6s in our experiments, for our largest model (*covtype-large*) with 6.7M items, and is constant with respect to the number of test rows because the bin packing occurs once per ensemble and is reused for each additional data point, allowing us to amortise its cost over improvements in end-to-end runtime from improved kernel efficiency. The gains in GPU thread utilisation from using BFD over NF directly translate into performance improvements, as fewer bins used means fewer GPU warps are launched. On our large size models, we see improvements in utilisation of 10.1%, 3.2%, 16.7% and 9.6% from BFD over NF. We use BFD in all subsequent experiments.

4.2 Evaluating SHAP Value Throughput

We evaluate the performance of GPUTreeShap as a backend to the XGBoost library (Chen and Guestrin, 2016), comparing its execution time against the existing CPU implementation of Algorithm 1[2]. The baseline CPU algorithm is efficiently multithreaded using OpenMP, with a parallel for loop over all test instances. See `https://github.com/dmlc/xgboost` for exact implementation details for the baseline and `https://github.com/rapidsai/gputreeshap` for GPUTreeShap implementation details.

Table 6 reports the runtime of GPUTreeShap on a single V100 GPU compared to TreeShap on 40 CPU cores. Results are averaged over five runs and standard deviations are also shown. We observe speedups between 13-18x for medium and large models evaluated on 10,000 test rows. We observe little to no speedup for the small models as insufficient computation is performed to offset the latency of launching GPU kernels.

Figure 4 plots the time to evaluate varying numbers of test rows for the *cal_housing-med* model. We plot the average of five runs; the shaded area indicates the 95% confidence interval. This illustrates the throughput vs. latency trade-off for this particular model size. The CPU is more effective for < 180 test rows due to lower latency, but the throughput of the GPU is significantly higher at larger batch sizes.

[2]We do not benchmark against TreeShap implementations in the Python SHAP package or LightGBM because they are written by the same author, also in C++, and are functionally equivalent to XGBoost's implementation.

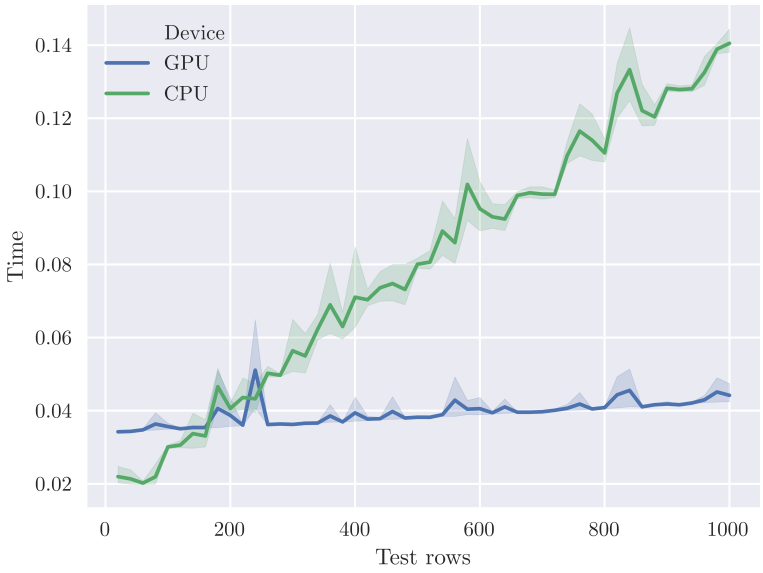


Figure 4. The crossover point where the V100 GPU outperforms 40 CPU cores occurs at around 200 test rows for the *cal_housing-med* model.

SHAP value computation is embarrassingly parallel over dataset rows, so we expect to see linear scaling of performance with respect to the number of GPUs or CPUs, given sufficient data. We set the number of rows to 1 million and evaluate the effect of additional processors for the *cal_housing-med* model, measuring throughput in rows per second. Figure 5 reports throughput up to the eight GPUs available on the DGX-1 system, showing the expected close to linear scaling and reaching a maximum throughput of 1.2M rows per second. Reported throughputs are from the average of five runs—error bars are too small to see due to relatively low variance. Figure 6 shows linear scaling with respect to CPU cores up to a maximum throughput of 7000 rows per second. The shaded area indicates the 95% confidence interval from 5 runs. We speculate that the dip at 40 cores is due to contention with the operating system requiring threads for other system functions, and so ignore it for this scaling analysis. We can reasonably approximate from Figure 6, using a throughput of 7000 rows/s per 40 cores, that it would require 6850 Xeon E5-2698 v4 CPU cores, or 343 sockets, to achieve the same throughput as eight V100 GPUs for this particular model.

4.3 SHAP Interaction Values

Table 7 compares single GPU vs. 40 core CPU runtime for SHAP interaction values. For this experiment, we lower the number of test rows to 200 due to the significantly increased computation time. Computing interaction values is challenging for datasets with larger numbers of features, in particular for *fashion_mnist* (785 features). Our GPU implementation achieves moderate speedups on *cal_housing* and *adult* due to the relatively low number of features; these speedups are roughly comparable to those obtained for standard SHAP values (Table 6). In contrast, for *covtype-large* and *fashion_mnist-large*, we see speedups of 114x and 340x, in the most extreme case reducing runtime from six hours to one minute. This speedup comes from both the increased throughput of the GPU over the CPU and the improvements to algorithmic complexity due to omission of irrelevant features described in Section 3.5. Note that it may be possible to reformulate the CPU algorithm to take advantage of the improved complexity with similar preprocessing steps, but investigating this is beyond the scope of this paper.

5 CONCLUSION

SHAP values have proven to be a useful tool for interpreting the predictions of decision tree ensembles. We have presented GPUTreeShap, an algorithm obtained by reformulating the TreeShap algorithm to enable efficient computation on GPUs. We exploit warp-level parallelism by cooperatively evaluating dynamic programming problems for each path in a decision tree ensemble, thus providing massive

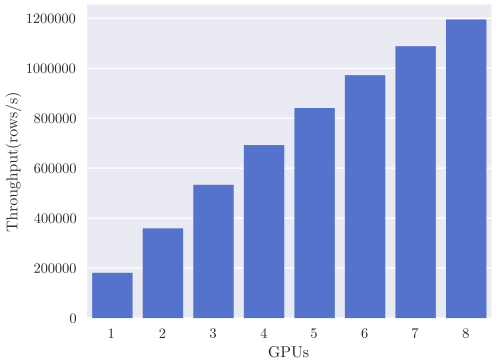


Figure 5. GPUTreeShap scales linearly with 8 V100 GPUs for the *cal_housing-med* model.

Figure 6. TreeShap scales linearly with 40 CPU cores, but at significantly lower throughput than GPUTreeShap.

parallelism for large ensemble predictors. We have shown how standard bin packing heuristics can be used to effectively schedule problems at the warp level, maximising GPU utilisation. Additionally, our rearrangement leads to improvement in the algorithmic complexity when computing SHAP interaction values, from $O(TLD^2M)$ to $O(TLD^3)$. Our library GPUTreeShap provides significant improvement to SHAP value computation over currently available software, allowing scaling onto one or more GPUs, and reducing runtime by one to two orders of magnitude.

REFERENCES

Anderson, R. and Mayr, E. (1984). Parallelism and greedy algorithms. Technical Report 1003, Computer Science Department, Stanford University.

Anderson, R. J., Mayr, E. W., and Warmuth, M. K. (1989). Parallel approximation algorithms for bin packing. *Inf. Comput.*, 82(3):262–277.

Blackard, J. A. (1998). *Comparison of Neural Networks and Discriminant Analysis in Predicting Forest Cover Types*. PhD thesis, Colorado State University.

Bo Fang, Guobin Shen, Shipeng Li, and Huifang Chen (2005). Techniques for efficient dct/idct implementation on generic gpu. In *2005 IEEE International Symposium on Circuits and Systems*, pages 1126–1129 Vol. 2.

Boyer, V., El Baz, D., and Elkihel, M. (2012). Solving knapsack problems on gpu. *Computers & Operations Research*, 39(1):42–47. Special Issue on Knapsack Problems and Applications.

Changhao Jiang and Snir, M. (2005). Automatic tuning matrix multiplication performance on graphics hardware. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 185–194.

Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *KDD*, pages 785–794. ACM.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning.

Coates, A., Huval, B., Wang, T., Wu, D. J., Ng, A. Y., and Catanzaro, B. (2013). Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page III–1337–III–1345. JMLR.org.

Coffman, E. G., Garey, M. R., and Johnson, D. S. (1997). Approximation algorithms for bin packing: A survey. In Hochbaum, D. S., editor, *Approximation Algorithms for NP-Hard Problems*, pages 46–93. PWS Publishing.

Fatahalian, K., Sugerman, J., and Hanrahan, P. (2004). Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, page 133–137, New York, NY, USA. Association for Computing Machinery.

Fujimoto, K., Kojadinovic, I., and Marichal, J.-L. (2006). Axiomatic characterizations of probabilistic and cardinal-probabilistic interaction indices. *Games and Economic Behavior*, 55:72–99.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman &; Co.

Green, O., McColl, R., and Bader, D. A. (2012). Gpu merge path: A gpu merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 331–340, New York, NY, USA. Association for Computing Machinery.

Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., and Pedreschi, D. (2018). A survey of methods for explaining black box models. *ACM Comput. Surv.*, 51(5).

Hall, J. D., Carr, N. A., and Hart, J. C. (2003). Cache and bandwidth aware matrix multiplication on the gpu.

Harris, M. and Buck, I. (2005). GPU flow-control idioms. In Pharr, M., editor, *GPU Gems 2*, pages 547–555. Addison-Wesley.

Johnson, D. S. (1974). Fast algorithms for bin packing. *J. Comput. Syst. Sci.*, 8(3):272–314.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. In *NIPS*, pages 3149–3157. Curran Associates.

Kohavi, R. (1996). Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *KDD*, pages 202–207. AAAI Press.

Liu, Y., Huang, W., Johnson, J., and Vaidya, S. (2006). Gpu accelerated smith-waterman. In Alexandrov, V. N., van Albada, G. D., Sloot, P. M. A., and Dongarra, J., editors, *Computational Science – ICCS 2006*, pages 188–195, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lundberg, S. M., Erion, G., Chen, H., DeGrave, A., Prutkin, J. M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N., and Lee, S.-I. (2020). From local explanations to global understanding with explainable ai for trees. *Nature Machine Intelligence*, 2(1):2522–5839.

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions. In *NIPS*, pages 4765–4774. Curran Associates.

Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons.

Mitchell, R. and Frank, E. (2017). Accelerating the xgboost algorithm using gpu computing. *PeerJ Computer Science*, 3:e127.

Moreland, K. and Angel, E. (2003). The fft on a gpu. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, page 112–119, Goslar, DEU. Eurographics Association.

NVIDIA Corporation (2020). CUDA C++ programming guide. Version 11.1.

Pace, R. K. and Barry, R. (1997). Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297.

Perry, M., Prosper, H. B., and Meyer-Baese, A. (2014). GPU implementation of bayesian neural network construction for data-intensive applications. *Journal of Physics: Conference Series*, 513(2):022027.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "Why Should I Trust You?": Explaining the predictions of any classifier. In *KDD*, pages 1135–1144. ACM.

Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D., and Dubey, P. (2010). Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 351–362, New York, NY, USA. Association for Computing Machinery.

Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., and Batra, D. (2017). Grad-CAM: Visual explanations from deep networks via gradient-based localization. In *ICCV*, pages 618–626. IEEE.

Shapley, L. S. (1953). A value for n-person games. *Contributions to the Theory of Games*, 2(28):307–317.

Sharp, T. (2008). Implementing decision trees and forests on a gpu. In Forsyth, D., Torr, P., and Zisserman, A., editors, *Computer Vision – ECCV 2008*, pages 595–608, Berlin, Heidelberg. Springer Berlin Heidelberg.

Sharp, T. (2012). Evaluating decision trees on a gpu. US Patent 8,290,882.

Steffen, P., Giegerich, R., and Giraud, M. (2010). Gpu parallelization of algebraic dynamic programming.

In Wyrzykowski, R., Dongarra, J., Karczewski, K., and Wasniewski, J., editors, *Parallel Processing and Applied Mathematics*, pages 290–299, Berlin, Heidelberg. Springer Berlin Heidelberg.

Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *arXiv*, abs/1708.07747.

Table 5. Bin packing performance

MODEL	ALG	TIME(S)	UTILISATION	BINS
COVTYPE-SMALL	NONE	0.0018	0.105246	560
COVTYPE-SMALL	NF	0.0041	0.982292	60
COVTYPE-SMALL	FFD	0.0064	0.998941	59
COVTYPE-SMALL	BFD	0.0086	0.998941	59
COVTYPE-MED	NONE	0.0450	0.211187	113533
COVTYPE-MED	NF	0.0007	0.913539	26246
COVTYPE-MED	FFD	0.0104	0.940338	25498
COVTYPE-MED	BFD	0.0212	0.940338	25498
COVTYPE-LARGE	NONE	0.0346	0.299913	6702132
COVTYPE-LARGE	NF	0.0413	0.851639	2360223
COVTYPE-LARGE	FFD	0.8105	0.952711	2109830
COVTYPE-LARGE	BFD	1.6702	0.952711	2109830
CAL_HOUSING-SMALL	NONE	0.0015	0.085938	80
CAL_HOUSING-SMALL	NF	0.0025	0.982143	7
CAL_HOUSING-SMALL	FFD	0.0103	0.982143	7
CAL_HOUSING-SMALL	BFD	0.0001	0.982143	7
CAL_HOUSING-MED	NONE	0.0246	0.181457	21641
CAL_HOUSING-MED	NF	0.0126	0.931429	4216
CAL_HOUSING-MED	FFD	0.0016	0.941704	4170
CAL_HOUSING-MED	BFD	0.0031	0.941704	4170
CAL_HOUSING-LARGE	NONE	0.0089	0.237979	3370373
CAL_HOUSING-LARGE	NF	0.0225	0.901060	890148
CAL_HOUSING-LARGE	FFD	0.3534	0.933114	859570
CAL_HOUSING-LARGE	BFD	0.8760	0.933114	859570
MODEL	ALG	TIME(S)	UTILISATION	BINS
FASHION_MNIST-SMALL	NONE	0.0022	0.123906	800
FASHION_MNIST-SMALL	NF	0.0082	0.991250	100
FASHION_MNIST-SMALL	FFD	0.0116	0.991250	100
FASHION_MNIST-SMALL	BFD	0.0139	0.991250	100
FASHION_MNIST-MED	NONE	0.0439	0.264387	144211
FASHION_MNIST-MED	NF	0.0008	0.867580	43947
FASHION_MNIST-MED	FFD	0.0130	0.880279	43313
FASHION_MNIST-MED	BFD	0.0219	0.880279	43313
FASHION_MNIST-LARGE	NONE	0.0140	0.385001	2929303
FASHION_MNIST-LARGE	NF	0.0132	0.791948	1424063
FASHION_MNIST-LARGE	FFD	0.3633	0.958855	1176178
FASHION_MNIST-LARGE	BFD	0.8518	0.958855	1176178
ADULT-SMALL	NONE	0.0016	0.125000	80
ADULT-SMALL	NF	0.0023	1.000000	10
ADULT-SMALL	FFD	0.0061	1.000000	10
ADULT-SMALL	BFD	0.0060	1.000000	10
ADULT-MED	NONE	0.0050	0.229014	13067
ADULT-MED	NF	0.0066	0.913192	3277
ADULT-MED	FFD	0.0575	0.950010	3150
ADULT-MED	BFD	0.1169	0.950010	3150
ADULT-LARGE	NONE	0.0033	0.297131	642883
ADULT-LARGE	NF	0.0035	0.858728	222446
ADULT-LARGE	FFD	0.0684	0.954377	200152
ADULT-LARGE	BFD	0.0954	0.954377	200152

Table 6. Speedups for V100 vs. 40 CPU cores on 10,000 test rows

MODEL	CPU(S)	STD	GPU(S)	STD	SPEEDUP
COVTYPE-SMALL	0.04	0.02	0.02	0.01	2.27
COVTYPE-MED	8.25	0.07	0.45	0.03	18.23
COVTYPE-LARGE	930.22	0.56	50.88	0.21	18.28
CAL_HOUSING-SMALL	0.01	0.01	0.01	0.01	0.96
CAL_HOUSING-MED	1.27	0.02	0.09	0.02	14.59
CAL_HOUSING-LARGE	315.21	0.30	16.91	0.34	18.64
FASHION_MNIST-SMALL	0.35	0.14	0.17	0.04	2.09
FASHION_MNIST-MED	15.10	0.07	1.13	0.08	13.36
FASHION_MNIST-LARGE	621.14	0.14	47.53	0.17	13.07
ADULT-SMALL	0.01	0.00	0.01	0.01	1.08
ADULT-MED	1.14	0.00	0.08	0.01	14.59
ADULT-LARGE	88.12	0.20	4.67	0.00	18.87

Table 7. Feature interactions — Speedups for V100 vs. 40 CPU cores on 10000 test rows

MODEL	CPU(S)	STD	GPU(S)	STD	SPEEDUP
COVTYPE-SMALL	0.14	0.01	0.02	0.01	8.32
COVTYPE-MED	21.50	0.32	0.19	0.02	114.41
COVTYPE-LARGE	2055.78	4.19	28.85	0.06	71.26
CAL_HOUSING-SMALL	0.01	0.00	0.01	0.00	1.44
CAL_HOUSING-MED	0.53	0.04	0.04	0.01	12.05
CAL_HOUSING-LARGE	93.67	0.28	8.55	0.04	10.96
FASHION_MNIST-SMALL	11.35	0.87	4.04	0.67	2.81
FASHION_MNIST-MED	578.90	1.23	4.91	0.71	117.97
FASHION_MNIST-LARGE	21603.53	622.60	63.53	0.78	340.07
ADULT-SMALL	0.06	0.09	0.01	0.00	11.25
ADULT-MED	1.74	0.30	0.04	0.01	39.38
ADULT-LARGE	67.29	6.22	2.76	0.00	24.38

Chapter 5

Sampling Permutations for Shapley Value Estimation

Journal of Machine Learning Research 1 (2000) 1-48 Submitted 4/00; Published 10/00

Sampling Permutations for Shapley Value Estimation

Rory Mitchell RAMITCHELLNZ@GMAIL.COM
Nvidia Corporation
Santa Clara
CA 95051, USA

Joshua Cooper COOPER@MATH.SC.EDU
Department of Mathematics
University of South Carolina
1523 Greene St.
Columbia, SC 29223, USA

Eibe Frank EIBE@CS.WAIKATO.AC.NZ
Department of Computer Science
University of Waikato
Hamilton, New Zealand

Geoffrey Holmes GEOFF@CS.WAIKATO.AC.NZ
Department of Computer Science
University of Waikato
Hamilton, New Zealand

Editor: Some editors

Abstract

Game-theoretic attribution techniques based on Shapley values are used to interpret black-box machine learning models, but their exact calculation is generally NP-hard, requiring approximation methods for non-trivial models. As the computation of Shapley values can be expressed as a summation over a set of permutations, a common approach is to sample a subset of these permutations for approximation. Unfortunately, standard Monte Carlo sampling methods can exhibit slow convergence, and more sophisticated quasi-Monte Carlo methods have not yet been applied to the space of permutations. To address this, we investigate new approaches based on two classes of approximation methods and compare them empirically. First, we demonstrate quadrature techniques in a RKHS containing functions of permutations, using the Mallows kernel in combination with kernel herding and sequential Bayesian quadrature. The RKHS perspective also leads to quasi-Monte Carlo type error bounds, with a tractable discrepancy measure defined on permutations. Second, we exploit connections between the hypersphere $\mathbb{S}^{d-2}$ and permutations to create practical algorithms for generating permutation samples with good properties. Experiments show the above techniques provide significant improvements for Shapley value estimates over existing methods, converging to a smaller RMSE in the same number of model evaluations.

Keywords: Interpretability, quasi-Monte Carlo, Shapley values

1. Introduction

The seminal work of Shapley (1953) introduces an axiomatic attribution of collaborative game outcomes among coalitions of participating players. Aside from their original appli-

cations in economics, Shapley values are popular in machine learning (Cohen et al. (2007); Strumbelj and Kononenko (2010); Štrumbelj and Kononenko (2014); Lundberg and Lee (2017)) because the assignment of feature relevance to model outputs is structured according to axioms consistent with human notions of attribution. In the machine learning context, each feature is treated as a player participating in the prediction provided by a machine learning model and the prediction is considered the outcome of the game. Feature attributions via Shapley values provide valuable insight into the output of complex models that are otherwise difficult to interpret.

Exact computation of Shapley values is known to be NP-hard in general (Deng and Papadimitriou (1994)) and approximations based on sampling have been proposed by several authors: Mann and Shapley (1960); Owen (1972); Castro et al. (2009); Maleki (2015); Castro et al. (2017). In particular, a simple Monte Carlo estimate for the Shapley value is obtained by sampling from a uniform distribution of permutations. The extensively developed quasi-Monte Carlo theory for integration on the unit cube shows that careful selection of samples can improve convergence significantly over random sampling, but these results do not extend to the space of permutations. Here, our goal is to better characterise 'good' sample sets for this unique approximation problem, and to develop tractable methods of obtaining these samples, reducing computation time for high-quality approximations of Shapley values. Crucially, we observe that sample evaluations, in this context corresponding to evaluations of machine learning models, dominate the execution time of approximations. Due to the high cost of each sample evaluation, considerable computational effort can be justified in finding such sample sets.

In Section 3, we define a reproducing kernel Hilbert space (RKHS) with several possible kernels over permutations by exploiting the direct connection between Shapley values and permutations. Using these kernels, we apply kernel herding, and sequential Bayesian quadrature algorithms to estimate Shapley values. In particular, we observe that kernel herding, in conjunction with the universal Mallows kernel, leads to an explicit convergence rate of $O(\frac{1}{n})$ as compared to $O(\frac{1}{\sqrt{n}})$ for ordinary Monte Carlo. An outcome of our investigation into kernels is a quasi-Monte Carlo type error bound, with a tractable discrepancy formula.

In Section 4, we describe another family of methods for efficiently sampling Shapley values, utilising a convenient isomorphism between the symmetric group $\mathfrak{S}_d$ and points on the hypersphere $\mathbb{S}^{d-2}$. These methods are motivated by the relative ease of selecting well-spaced points on the sphere, as compared to the discrete space of permutations. We develop two new sampling methods, termed orthogonal spherical codes and Sobol permutations, that select high-quality samples by choosing points well-distributed on $\mathbb{S}^{d-2}$.

Our empirical evaluation in Section 5 examines the performance of the above methods compared to existing methods on a range of practical machine learning models, tracking the reduction in mean squared error against exactly calculated Shapley values for boosted decision trees and considering empirical estimates of variance in the case of convolutional neural networks. Additionally, we evaluate explicit measures of discrepancy (in the quasi-Monte Carlo sense) for the sample sets generated by our algorithms. This evaluation of discrepancy for the generated samples of permutations may be of broader interest, as quasi-Monte Carlo error bounds based on discrepancy apply to any statistics of functions of permutations and not just Shapley values.

In summary, the contributions of this work are:

- The characterisation of the Shapley value approximation problem in terms of reproducing kernel Hilbert spaces.
- Connecting the Shapley value approximation problem to existing quasi-Monte Carlo approaches, using kernels and connections between the hypersphere and symmetric group.
- Experimental evaluation of these methods in terms of discrepancy, and the error of Shapley value approximations on tabular and image datasets.

2. Background and Related Work

We first introduce some common notation for permutations and provide the formal definition of Shapley values. Then, we briefly review the literature for existing techniques for approximating Shapley values.

2.1 Notation

We refer to the symmetric group of permutations of d elements as $\mathfrak{S}_d$. We reserve the use of n to refer to the number of samples. The permutation $\sigma \in \mathfrak{S}_d$ assigns rank j to element i by $\sigma(i) = j$. For example, given the permutation written in one-line notation

$$\sigma = \begin{pmatrix} 1 & 4 & 2 & 3 \end{pmatrix}$$

and the list of items

$$(x_1, x_2, x_3, x_4),$$

the items are reordered such that x_i occupies the $\sigma(i)$ coordinate

$$(x_1, x_3, x_4, x_2),$$

and the inverse $\sigma^{-1}(j) = i$ is

$$\sigma^{-1} = \begin{pmatrix} 1 & 3 & 4 & 2 \end{pmatrix}.$$

An *inversion* is a pair of elements in the permutation (σ_i, σ_j) such that $i < j$ and $\sigma(i) > \sigma(j)$. The identity permutation,

$$I = \begin{pmatrix} 1 & 2 & 3 & \cdots \end{pmatrix},$$

contains 0 inversions, and its reverse

$$\mathrm{Rev}(I) = \begin{pmatrix} \cdots & 3 & 2 & 1 \end{pmatrix},$$

contains the maximum number of inversions, $\binom{d}{2}$.

2.2 Shapley Values

Shapley values (Shapley (1953)) provide a mechanism to distribute the proceeds of a cooperative game among the members of the winning coalition by measuring marginal contribution to the final outcome. The Shapley value Sh_i for coalition member i is defined as

$$\mathrm{Sh}_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(|N| - |S| - 1)!}{|N|!} (v(S \cup \{i\}) - v(S)) \tag{1}$$

where S is a partial coalition, N is the grand coalition (consisting of all members), and v is the so-called "characteristic function" that is assumed to return the proceeds (i.e., value) obtained by any coalition.

The Shapley value function may also be conveniently expressed in terms of permutations

$$\mathrm{Sh}_i(v) = \frac{1}{|N|!} \sum_{\sigma \in \mathfrak{S}_d} \left[v([\sigma]_{i-1} \cup \{i\}) - v([\sigma]_{i-1}) \right] \tag{2}$$

where $[\sigma]_{i-1}$ represents the set of players ranked lower than i in the ordering σ. To see the equivalence between (1) and (2), consider that $|S|!$ is the number of unique orderings the members of S can join the coalition before i, and $(|N| - |S| - 1)!$ is the number of unique orderings the remaining members $N \setminus S \cup \{i\}$ can join the coalition after i. The Shapley value is unique and has the following desirable properties:

1. *Efficiency*: $\sum_{i=1}^{n} \mathrm{Sh}_i(v) = v(N)$. The sum of Shapley values for each coalition member is the value of the grand coalition N.

2. *Symmetry*: If, $\forall S \subseteq N \setminus \{i, j\}, v(S \cup \{i\}) = v(S \cup \{j\})$, then $\mathrm{Sh}_i = \mathrm{Sh}_j$. If two players have the same marginal effect on each coalition, their Shapley values are the same.

3. *Linearity*: $\mathrm{Sh}_i(v + w) = \mathrm{Sh}_i(v) + \mathrm{Sh}_i(w)$. The Shapley values of sums of games are the sum of the Shapley values of the respective games.

4. *Dummy*: If, $\forall S \subseteq N \setminus \{i\}, v(S \cup \{i\}) = v(S)$, then $\mathrm{Sh}_i = 0$. The coalition member whose marginal impact is always zero has a Shapley value of zero.

Evaluation of the Shapley value is known to be NP-hard in general (Deng and Papadimitriou (1994)) but may be approximated by sampling terms from the sum of either Equation 1 or Equation 2. This paper focuses on techniques for approximating Equation 2 via carefully chosen samples of permutations. We discuss characteristic functions v that arise in the context of machine learning models, with the goal of attributing predictions to input features.

Shapley values have been used as a feature attribution method for machine learning in many prior works (Cohen et al. (2007); Strumbelj and Kononenko (2010); Štrumbelj and Kononenko (2014); Lundberg and Lee (2017)). In the terminology of supervised learning, we have some learned model $f(x) = y$ that maps a vector of features x to a prediction y. In this context, the Shapley values will be used to evaluate the weighted marginal contribution of features to the output of the predictive model. The value of the characteristic function is assumed to be given by y, and the grand coalition is given by the full set of features. In

a partial coalition, only some of the features are considered "active" and their values made available to the model to obtain a prediction. Applying the characteristic function for partial coalitions requires the definition of $f(x_S)$, where the input features x are perturbed in some way according to the active subset S. A taxonomy of possible approaches is given in Covert et al. (2020).

2.3 Monte Carlo

An obvious Shapley value approximation is the simple Monte Carlo estimator,

$$\bar{\mathrm{Sh}}_i(v) = \frac{1}{n} \sum_{\sigma \in \Pi} \big[v([\sigma]_{i-1} \cup \{i\}) - v([\sigma]_{i-1}) \big], \tag{3}$$

for a uniform sample of permutations $\Pi \subset \mathfrak{S}_d$ of size n. Monte Carlo techniques were used to solve electoral college voting games in Mann and Shapley (1960), and a more general analysis is given in Castro et al. (2009). Equation 3 is an unbiased estimator that converges asymptotically at a rate of $O(1/\sqrt{n})$ according to the Central Limit Theorem.

From a practical implementation perspective, note that a single sample of permutations Π can be used to evaluate Sh_i for all features i. For each permutation $\sigma \in \Pi$ of length d, first evaluate the empty set $v(\{\})$, then walk through the permutation, incrementing i and evaluating $v([\sigma]_i)$, yielding $d + 1$ evaluations of v that are used to construct marginal contributions for each feature. $v([\sigma]_{i-1})$ is not evaluated, but reused from the previous function evaluation, providing a factor of two improvement over the naive approach.

2.4 Antithetic Sampling

Antithetic sampling is a variance reduction technique for Monte Carlo integration where samples are taken as correlated pairs instead of standard i.i.d. samples. The antithetic Monte Carlo estimate (see Rubinstein and Kroese (2016)) is

$$\hat{\mu}_{anti} = \frac{1}{n} \sum_{i=1}^{n/2} f(X_i) + f(Y_i)$$

with variance given by

$$\mathrm{Var}(\hat{\mu}_{anti}) = \frac{\sigma}{n}(1 + \mathrm{Corr}(f(X), f(Y)), \tag{4}$$

such that if $f(X)$ and $f(Y)$ are negatively correlated, the variance is reduced. A common choice for sampling on the unit cube is $X \sim U(0,1)^d$ with $Y_i = 1 - X_i$. Antithetic sampling for functions of permutations is discussed in Lomeli et al. (2019), with a simple strategy being to take permutations and their reverse. We implement this sampling strategy in our experiments with antithetic sampling.

2.5 Multilinear Extension

Another Shapley value approximation method is the multilinear extension of Owen (1972). The sum over feature subsets from (1) can be represented equivalently as an integral by

introducing a random variable for feature subsets. The Shapley value is calculated as

$$\mathrm{Sh}_i(v) = \int_0^1 e_i(q)dq \tag{5}$$

where

$$e_i(q) = \mathbb{E}[v(E_q \cup i) - v(E_q)]$$

and E_q is a random subset of features, excluding i, where each feature has probability q of being selected. $e_i(q)$ is estimated with samples. In our experiments, we implement a version of the multilinear extension algorithm using the trapezoid rule to sample q at fixed intervals. A form of this algorithm incorporating antithetic sampling is also presented in Okhrati and Lipani (2020), by rewriting Equation 5 as

$$\mathrm{Sh}_i(v) = \int_0^{\frac{1}{2}} e_i(q) + e_i(1-q)dq$$

where the sample set E_i is used to estimate $e_i(q)$ and the 'inverse set', $\{N \setminus \{E_i, i\}\}$, is used to estimate $e_i(1-q)$. In Section 5, we include experiments for the multilinear extension method both with and without antithetic sampling.

2.6 Stratified Sampling

Another common variance reduction technique is stratified sampling, where the domain of interest is divided into mutually exclusive subregions, an estimate is obtained for each subregion independently, and the estimates are combined to obtain the final estimate. For integral $\mu = \int_{\mathcal{D}} f(x)p(x)dx$ in domain $\mathcal{D}$, separable into J non-overlapping regions $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_J$ where $w_j = P(X \in \mathcal{D}_j)$ and $p_j(x) = w_j^{-1} p(x) \mathbb{1}_{x \in \mathcal{D}_j}$, the basic stratified sampling estimator is

$$\hat{\mu}_{strat} = \sum_{j=1}^{J} \frac{w_j}{n_j} \sum_{i=1}^{n_j} f(X_{ij}),$$

where $X_{ij} \sim p_j$ for $i = 1, \cdots, n_j$ and $j = 1, \cdots, J$ (see Owen (2003)). The stratum size n_j can be chosen with the Neyman allocation (Neyman (1934)) if estimates of the variance in each region are known. The stratified sampling method was first applied to Shapley value estimation by Maleki (2015), then improved by Castro et al. (2017). We implement the version in Castro et al. (2017), where strata $\mathcal{D}_i^\ell$ are considered for all $i = 1, \cdots, d$ and $\ell = 1, \cdots, d$, where $\mathcal{D}_i^\ell$ is the subset of marginal contributions with feature i at position ℓ.

This concludes discussion of existing work; the next sections introduce the primary contributions of this paper.

3. Kernel Methods

A majority of Monte Carlo integration work deals with continuous functions on $\mathbb{R}^d$, where the distribution of samples is well defined. In the space of permutations, distances between samples are not implicitly defined, so we impose a similarity metric via a kernel and select samples with good distributions relative to these kernels.

Given a positive definite kernel $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ over some input space $\mathcal{X}$, there is an embedding $\phi : \mathcal{X} \to \mathcal{F}$ of elements of $\mathcal{X}$ into a Hilbert space $\mathcal{F}$, where the kernel computes an inner product $K(x, y) = \langle \phi(x), \phi(y) \rangle_{\mathcal{K}}$ given $x, y \in \mathcal{X}$. Hilbert spaces associated with a kernel are known as reproducing kernel Hilbert spaces (RKHS). Kernels are used extensively in machine learning for learning relations between arbitrary structured data. In this paper, we use kernels over permutations to develop a notion of the quality of finite point sets for the Shapley value estimation problem, and for the optimisation of such point sets. For this task, we investigate three established kernels over permutations: the Kendall, Mallows, and Spearman kernels.

The Kendall and Mallows kernels are defined in Jiao and Vert (2015). Given two permutations σ and σ' of the same length, both kernels are based on the number of concordant and discordant pairs between the permutations:

$$n_{\text{con}}(\sigma, \sigma') = \sum_{i<j} [\mathbb{1}_{\sigma(i)<\sigma(j)} \mathbb{1}_{\sigma'(i)<\sigma'(j)} + \mathbb{1}_{\sigma(i)>\sigma(j)} \mathbb{1}_{\sigma'(i)>\sigma'(j)}]$$

$$n_{\text{dis}}(\sigma, \sigma') = \sum_{i<j} [\mathbb{1}_{\sigma(i)<\sigma(j)} \mathbb{1}_{\sigma'(i)>\sigma'(j)} + \mathbb{1}_{\sigma(i)>\sigma(j)} \mathbb{1}_{\sigma'(i)<\sigma'(j)}].$$

Assuming the length of the permutation is d, the Kendall kernel, corresponding to the well-known Kendall tau correlation coefficient (Kendall (1938)), is

$$K_\tau(\sigma, \sigma') = \frac{n_{\text{con}}(\sigma, \sigma') - n_{\text{dis}}(\sigma, \sigma')}{\binom{d}{2}}.$$

The Mallows kernel, for $\lambda \geq 0$, is defined as

$$K_M^\lambda(\sigma, \sigma') = e^{-\lambda n_{\text{dis}}(\sigma, \sigma') / \binom{d}{2}}.$$

Here, the Mallows kernel differs slightly from that of Jiao and Vert (2015). We normalise the $n_{dis(\sigma,\sigma')}$ term relative to d, allowing a consistent selection of the λ parameter across permutations of different length.

While the straightforward implementation of Kendall and Mallows kernels is of order $O(d^2)$, a $O(d \log d)$ variant based on merge-sort is given by Knight (1966).

Note that K_τ can also be expressed in terms of a feature map of $\binom{d}{2}$ elements,

$$\Phi_\tau(\sigma) = \left(\frac{1}{\sqrt{\binom{d}{2}}} (\mathbb{1}_{\sigma(i)>\sigma(j)} - \mathbb{1}_{\sigma(i)<\sigma(j)}) \right)_{1 \leq i < j \leq d}$$

so that

$$K_\tau(\sigma, \sigma') = \Phi(\sigma)^T \Phi(\sigma').$$

The Mallows kernel corresponds to a more complicated feature map, although still finite dimensional, given in Mania et al. (2018).

We also define a third kernel based on Spearman's ρ. The (unnormalised) Spearman rank distance

$$d_\rho(\sigma, \sigma') = \sum_{i=1}^{d} (\sigma(i) - \sigma'(i))^2 = ||\sigma - \sigma'||_2^2$$

is a semimetric of negative type (Diaconis (1988)), therefore we can exploit the relationship between semimetrics of negative type and kernels from Sejdinovic et al. (2013) to obtain a valid kernel. Writing $\sum_{i=0}^{d} \sigma(i)\sigma(i)'$ using vector notation as $\sigma^T\sigma'$, we have

$$\begin{aligned} d(\sigma, \sigma') &= K(\sigma, \sigma) + K(\sigma', \sigma') - 2K(\sigma, \sigma') \\ d_\rho(\sigma, \sigma') &= \sigma^T\sigma + \sigma'^T\sigma' - 2\sigma^T\sigma' \\ \implies K_\rho(\sigma, \sigma') &= \sigma^T\sigma' \end{aligned}$$

and the kernel's feature map is trivially

$$\Phi_\rho(\sigma) = \sigma.$$

Before introducing sampling algorithms, we derive an additional property for the above kernels: analytic formulas for their expected values at some fixed point σ and values drawn from a given probability distribution $\sigma' \sim p$. The distribution of interest for approximating (2) is the uniform distribution U. The expected value is straightforward to obtain for the Spearman and Kendall kernel:

$$\forall \sigma \in \Pi, \quad \mathbb{E}_{\sigma' \sim U}[K_\rho(\sigma, \sigma')] = \frac{d(d+1)^2}{4}$$

$$\forall \sigma \in \Pi, \quad \mathbb{E}_{\sigma' \sim U}[K_\tau(\sigma, \sigma')] = 0.$$

The Mallows kernel is more difficult. Let X be a random variable representing the number of inversions over all permutations of length d. Its distribution is studied in Muir (1898), with probability generating function given as

$$\phi_d(x) = \prod_{j=1}^{d} \frac{1 - x^j}{j(1-x)}.$$

There is no convenient form in terms of standard functions for its associated density function. From the probability generating function of X, we obtain the moment generating function:

$$\begin{aligned} M_d(t) &= \phi_d(e^t) \\ &= \prod_{j=1}^{d} \frac{1 - e^{tj}}{j(1-e^t)} \\ &= \mathbb{E}[e^{tX}]. \end{aligned}$$

The quantity $n_{\text{dis}}(I, \sigma)$, where I is the identity permutation, returns exactly the number of inversions in σ. Therefore, we have

$$\begin{aligned} M_d(-\lambda/\tbinom{d}{2}) &= \mathbb{E}[e^{-\lambda X/\binom{d}{2}}] \\ &= \mathbb{E}_{\sigma' \sim U}[K_M(I, \sigma')]. \end{aligned}$$

The quantity n_{dis} is right-invariant in the sense that $n_{\text{dis}}(\sigma, \sigma') = n_{\text{dis}}(\tau\sigma, \tau\sigma')$ for $\tau \in \mathfrak{S}_d$ (Diaconis (1988)), so

$$\begin{aligned}
\forall \tau \in \mathfrak{S}_d, \quad \mathbb{E}_{\sigma' \sim U}[K_M(I, \sigma')] &= \mathbb{E}_{\sigma' \sim U}[K_M(\tau I, \tau\sigma')] \\
&= \mathbb{E}_{\sigma' \sim U}[K_M(\tau I, \sigma')] \\
\forall \sigma \in \mathfrak{S}_d, \quad \mathbb{E}_{\sigma' \sim U}[K_M(I, \sigma')] &= \mathbb{E}_{\sigma' \sim U}[K_M(\sigma, \sigma')] \\
&= \prod_{j=1}^{d} \frac{1 - e^{-\lambda j / \binom{d}{2}}}{j(1 - e^{-\lambda / \binom{d}{2}})},
\end{aligned}$$

We now describe two greedy algorithms for generating point sets improving on simple Monte Carlo—kernel herding and sequential Bayesian quadrature.

3.1 Kernel Herding

A greedy process called "kernel herding" for selecting (unweighted) quadrature samples in a reproducing kernel Hilbert space is proposed in Chen et al. (2010). The sample $n+1$ in kernel herding is given by

$$x_{n+1} = \arg\max_x \left[\mathbb{E}_{x' \sim p}[K(x, x')] - \frac{1}{n+1} \sum_{i=1}^{n} K(x, x_i) \right] \tag{6}$$

which can be interpreted as a greedy optimisation process selecting points for maximum separation, while also converging on the expected distribution p. In the case of Shapley value estimation, the samples are permutations $\sigma \in \mathfrak{S}_d$ and p is a uniform distribution with $p(\sigma) = \frac{1}{\sigma!}, \forall \sigma \in \mathfrak{S}_d$.

Kernel herding has time complexity $O(n^2)$ for n samples, assuming the argmax can be computed in $O(1)$ time and $\mathbb{E}_{x' \sim p}[K(x, x')]$ is available. We have analytic formulas for $\mathbb{E}_{x' \sim p}[K(x, x')]$ from the previous section for the Spearman, Kendall, and Mallows kernels, and they give constant values depending only on the size of the permutation d. We compute an approximation to the argmax in constant time by taking a fixed number of random samples at each iteration and retaining the one yielding the maximum.

If certain conditions are met, kernel herding converges at the rate $O(\frac{1}{n})$, an improvement over $O(\frac{1}{\sqrt{n}})$ for standard Monte Carlo sampling. According to Chen et al. (2010), this improved convergence rate is achieved if the RKHS is universal, and mild assumptions are satisfied by the argmax (it need not be exact). Of the Spearman, Kendall and Mallows kernels, only the Mallows kernel has the universal property (Mania et al. (2018)).

Next, we describe a more sophisticated kernel-based algorithm generating weighted samples.

3.2 Sequential Bayesian Quadrature

Bayesian Quadrature (O'Hagan (1991); Rasmussen and Ghahramani (2003)) (BQ) formulates the integration problem

$$Z_{f,p} = \int f(x)p(x)dx$$

Algorithm 1: Sequential Bayesian Quadrature

Input: n, kernel K, sampling distribution p, integrand f

$X_0 \leftarrow RandomSample(p)$
$K^{-1} = I$ `// Inverse of covariance matrix`
$z_0 \leftarrow \mathbb{E}_{x' \sim p}[K(X_0, x')]$
for $i \leftarrow 2$ **to** n **do**
  $X_i \leftarrow \arg\min_x \mathbb{E}_{x,x' \sim p}[K(x, x')] - z^T K^{-1} z$
  $y \leftarrow \vec{0}$
  for $j \leftarrow 1$ **to** i **do**
    $y_j = K(X_i, X_j)$
  $K^{-1} \leftarrow CholeskyUpdate(K^{-1}, y)$
  $z_i \leftarrow \mathbb{E}_{x' \sim p}[K(X_i, x')]$
$w = z^T K^{-1}$
return $w^T f(X)$

as a Bayesian inference problem. Standard BQ imposes a Gaussian process prior on f with zero mean and kernel function K. A posterior distribution is inferred over f conditioned on a set of points $(x_0, x_1, \cdots, x_n)$. This implies a distribution on $Z_{f,p}$ with expected value

$$\mathbb{E}_{GP}[Z] = z^T K^{-1} f(X)$$

where $f(X)$ is the vector of function evaluations at points $(x_0, x_1, \cdots, x_n)$, K^{-1} is the inverse of the kernel covariance matrix, and $z_i = \mathbb{E}_{x' \sim p}[K(x_i, x')]$. Effectively, for an arbitrary set of points, Bayesian quadrature solves the linear system $Kw = z$ to obtain a reweighting of the sample evaluations, yielding the estimate

$$Z \simeq w^T f(X).$$

An advantage of the Bayesian approach is that uncertainty is propagated through to the final estimate. Its variance is given by

$$\mathbb{V}[Z_{f,p} | f(X)] = \mathbb{E}_{x,x' \sim p}[K(x, x')] - z^T K^{-1} z. \tag{7}$$

This variance estimate is used in Huszár and Duvenaud (2012) to develop sequential Bayesian quadrature (SBQ), a greedy algorithm selecting samples to minimise Equation 7. This procedure, summarised in Algorithm 1, is shown by Huszár and Duvenaud (2012) to be related to optimally weighted kernel herding. Note that the expectation term in (7) and Algorithm 1 is constant and closed-form for all kernels considered here.

SBQ has time complexity $O(n^3)$ for n samples if the argmin takes constant time, and an $O(n^2)$ Cholesky update algorithm is used to form K^{-1}, adding one sample at a time. In general, exact minimisation of Equation 7 is not tractable, so as with kernel herding, we approximate the argmin by drawing a fixed number of random samples and choosing the one yielding the minimum variance.

3.3 Error Analysis in RKHS

Canonical error analysis of quasi Monte-Carlo quadrature is performed using the Koksma-Hlawka inequality (Hlawka (1961); Niederreiter (1992)), decomposing error into a product

of function variation and discrepancy of the sample set. We derive a version of this inequality for Shapley value approximation in terms of reproducing kernel Hilbert spaces. Our derivation mostly follows Hickernell (2000), with modification of standard integrals to weighted sums of functions on $\mathfrak{S}_d$, allowing us to calculate discrepancies for point sets generated by kernel herding and SBQ with permutation kernels. The analysis is performed for the Mallows kernel, which is known to be a universal kernel (Mania et al. (2018)).

Given a symmetric, positive definite kernel K, we have a unique RKHS $\mathcal{F}$ with inner product $\langle \cdot, \cdot \rangle_K$ and norm $|| \cdot ||_K$, where the kernel reproduces functions $f \in \mathcal{F}$ by

$$f(\sigma) = \langle f, K(\cdot, \sigma) \rangle_K.$$

Define error functional

$$\text{Err}(f, \Pi, w) = \frac{1}{d!} \sum_{\sigma \in \mathfrak{S}_d} f(\sigma) - \sum_{\tau \in \Pi} w_\tau f(\tau),$$

where Π is a sample set of permutations and w_τ is the associated weight of sample τ. Because the Mallows kernel is a universal kernel, the bounded Shapley value component functions $f(\sigma)$ belong to $\mathcal{F}$. Given that $\text{Err}(f, \Pi, w)$ is a continuous linear functional on $\mathcal{F}$ and assuming that it is bounded, by the Riesz Representation Theorem, there is a function $\xi \in \mathcal{F}$ that is its representer: $\text{Err}(f, \Pi, w) = \langle \xi, f \rangle_K$. Using the Cauchy-Schwarz inequality, the quadrature error is bounded by

$$|\text{Err}(f, \Pi, w)| = |\langle \xi, f \rangle_K| \leq ||\xi||_K ||f||_K = D(\Pi, w) V(f)$$

where $D(\Pi, w) = ||\xi||_K$ is the discrepancy of point set Π with weights w and $V(f) = ||f||_K$ is the function variation. The quantity $D(\Pi, w)$ has an explicit formula. As the function ξ is reproduced by the kernel, we have:

$$\begin{aligned} \xi(\sigma') = \langle \xi, K(\cdot, \sigma') \rangle_K &= \text{Err}(K(\cdot, \sigma'), \Pi, w) \\ &= \frac{1}{d!} \sum_{\sigma \in \mathfrak{S}_d} K(\sigma, \sigma') - \sum_{\tau \in \Pi} w_\tau K(\tau, \sigma'). \end{aligned}$$

Then the discrepancy can be obtained, using the fact that $\text{Err}(f, \Pi, w) = \langle \xi, f \rangle_K$, by

$$\begin{aligned}
D(\Pi, w) &= ||\xi||_k = \sqrt{\langle \xi, \xi \rangle_K} = \sqrt{\text{Err}(\xi, \Pi, w)} \\
&= \left(\frac{1}{d!} \sum_{\sigma \in \mathfrak{S}_d} \xi(\sigma) - \sum_{\tau \in \Pi} w_\tau \xi(\tau) \right)^{\frac{1}{2}} \\
&= \left(\frac{1}{d!} \sum_{\sigma \in \mathfrak{S}_d} \left[\frac{1}{d!} \sum_{\sigma' \in \mathfrak{S}_d} K(\sigma, \sigma') - \sum_{\tau \in \Pi} w_\tau K(\tau, \sigma) \right] \right. \\
&\quad \left. - \sum_{\tau \in \Pi} w_\tau \left[\frac{1}{d!} \sum_{\sigma \in \mathfrak{S}_d} K(\sigma, \tau) - \sum_{\tau' \in \Pi} w_{\tau'} K(\tau, \tau') \right] \right)^{\frac{1}{2}} \\
&= \left(\frac{1}{(d!)^2} \sum_{\sigma, \sigma' \in \mathfrak{S}_d} K(\sigma, \sigma') - \frac{2}{d!} \sum_{\sigma \in \mathfrak{S}_d} \sum_{\tau \in \Pi} w_\tau K(\tau, \sigma) + \sum_{\tau, \tau' \in \Pi} w_\tau w_{\tau'} K(\tau, \tau') \right)^{\frac{1}{2}} \\
&= \left(\mathbb{E}_{\sigma, \sigma' \sim U}[K(\sigma, \sigma')] - 2 \sum_{\tau \in \Pi} w_\tau \mathbb{E}_{\sigma \sim U}[K(\tau, \sigma)] + \sum_{\tau, \tau' \in \Pi} w_\tau w_{\tau'} K(\tau, \tau') \right)^{\frac{1}{2}} \qquad (8)
\end{aligned}$$

It can be seen that kernel herding (Equation 6) greedily minimises $D(\Pi, w)^2$ with constant weights $\frac{1}{n}$, by examining the reduction in $D(\Pi, \frac{1}{n})^2$ obtained by the addition of a sample to Π. The kernel herding algorithm for sample $\sigma_{n+1} \in \Pi$ is

$$\sigma_{n+1} = \arg\max_{\sigma} \left[\mathbb{E}_{\sigma' \sim U}[K(\sigma, \sigma')] - \frac{1}{n+1} \sum_{i=1}^{n} K(\sigma, \sigma_i) \right]$$

Note that, since $K(\cdot, \cdot)$ is right-invariant, the quantity $\mathbb{E}_{\sigma' \sim U}[K(\sigma, \sigma')]$ does not depend on σ, so the argmax above is simply minimizing $\sum_{i=1}^{n} K(\sigma, \sigma_i)$. On the other hand, denoting the identity permutation by I, for a newly selected permutation sample π:

$$\begin{aligned}
D(\Pi, \tfrac{1}{n})^2 - D(\Pi \cup \{\pi\}, \tfrac{1}{n+1})^2 &= 2 \sum_{\tau \in \Pi \cup \{\pi\}} \frac{1}{n+1} \mathbb{E}_{\sigma \sim U}[K(\tau, \sigma)] - 2 \sum_{\tau \in \Pi} \frac{1}{n} \mathbb{E}_{\sigma \sim U}[K(\tau, \sigma)] \\
&\quad + \sum_{\tau, \tau' \in \Pi} \frac{1}{n^2} K(\tau, \tau') - \sum_{\tau, \tau' \in \Pi \cup \{\pi\}} \frac{1}{(n+1)^2} K(\tau, \tau') \\
&= 2 \frac{n+1}{n+1} \mathbb{E}_{\sigma \sim U}[K(I, \sigma)] - 2 \frac{n}{n} \mathbb{E}_{\sigma \sim U}[K(I, \sigma)] \\
&\quad + \sum_{\tau, \tau' \in \Pi} \frac{2n+1}{n^2 (n+1)^2} K(\tau, \tau') - 2 \sum_{\tau \in \Pi} \frac{1}{(n+1)^2} K(\tau, \pi) \\
&= \frac{K(I, I)}{(n+1)^2} + \sum_{\tau, \tau' \in \Pi} \frac{2n+1}{n^2 (n+1)^2} K(\tau, \tau') \\
&\quad - \frac{2}{(n+1)^2} \sum_{\tau \in \Pi} K(\tau, \pi),
\end{aligned}$$

where both equalities use right-invariance. Note that the first two summands in the last expression are constants (i.e., do not depend on the choice of π), so maximizing this quantity is the same as minimizing $\sum_{\tau\in\Pi} K(\tau,\pi)$, i.e., the same as the kernel herding optimization subproblem.

Furthermore, we can show that Bayesian quadrature minimises squared discrepancy via optimisation of weights. Writing $z_i = \mathbb{E}_{\sigma'\sim p}[K(\sigma_i,\sigma')]$ and switching to vector notation we have

$$D(\Pi, w)^2 = c - 2w^T z + w^T K w,$$

where the first term is a constant not depending on w. Taking the gradient with respect to w, setting it to 0, and solving for w, we obtain:

$$\begin{aligned} \nabla D(\Pi, w)^2 &= -2z + 2w^T K = 0 \\ w^* &= z^T K^{-1}, \end{aligned} \tag{9}$$

where (9) is exactly line 11 of Algorithm 1.

We use the discrepancy measure in (8) for numerical experiments in Section 5.4 to determine the quality of a set of sampled permutations in a way that is independent of the integrand f.

4. Sampling Permutations on $\mathbb{S}^{d-2}$

Kernel herding and sequential Bayesian quadrature directly reduce the discrepancy of the sampled permutations via greedy optimisation. We now describe two approaches to sampling permutations of length d based on a relaxation to the Euclidean sphere $\mathbb{S}^{d-2} = \left\{x \in \mathbf{R}^{d-1} : \|x\| = 1\right\}$, where the problem of selecting well-distributed samples is simplified. We describe a simple procedure for mapping points on the surface of this hypersphere to the nearest permutation, where the candidate nearest neighbours form the vertices of a Cayley graph inscribing the sphere. This representation provides a natural connection between distance metrics over permutations, such as Kendall's tau and Spearman's rho, and Euclidean space. We show that samples taken uniformly on the sphere result in a uniform distribution over permutations, and evaluate two unbiased sampling algorithms. Our approach is closely related to that of Plis et al. (2010), where an angular view of permutations is used to solve inference problems.

4.1 Spheres, Permutohedrons, and the Cayley Graph

Consider the projection of permutations $\sigma \in \mathfrak{S}_d$ as points in $\mathbf{R}^d$, where the i-th coordinate is given by $\sigma^{-1}(i)$. These points form the vertices of a polytope known as the permutohedron (Guilbaud and Rosenstiehl (1963)). The permutohedron is a $d-1$ dimensional object embedded in d dimensional space, lying on the hyperplane given by

$$\sum_{i=1}^{d} \sigma^{-1}(i) = \frac{d(d+1)}{2},$$

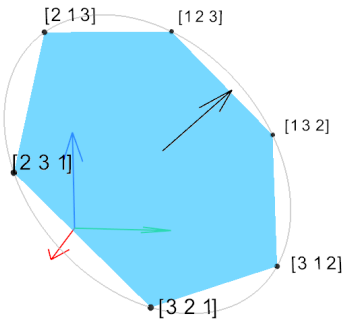


Figure 1: Cayley Graph of $d = 3$

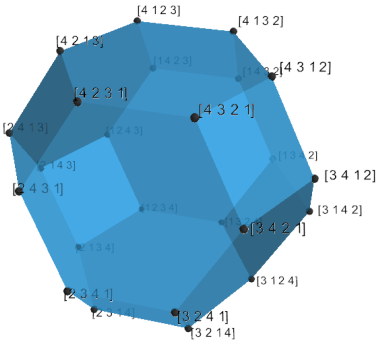


Figure 2: Cayley Graph of $d = 4$

with normal vector

$$\vec{n} = \begin{bmatrix} \frac{1}{\sqrt{d}} \\ \frac{1}{\sqrt{d}} \\ \vdots \\ \frac{1}{\sqrt{d}} \end{bmatrix}, \tag{10}$$

and inscribing the hypersphere $\mathbb{S}^{d-2}$ lying on the hyperplane, defined by

$$\sum_{i=1}^{d} \sigma^{-1}(i)^2 = \frac{d(d+1)(2d+1)}{6}.$$

Inverting the permutations at the vertices of the permutohedron gives a Cayley graph of the symmetric group with adjacent transpositions as the generating set. Figure 1 shows the Cayley graph for $\mathfrak{S}_3$, whose vertices form a hexagon inscribing a circle on a hyperplane, and Figure 2 shows the Cayley graph of $\mathfrak{S}_4$ projected into three dimensions (its vertices lie on a hyperplane in four dimensions). Each vertex σ^{-1} in the Cayley graph has $d-1$ neighbours, where each neighbour differs by exactly one adjacent transposition (one bubble-sort operation). Critically for our application, this graph has an interpretation in terms of distance metrics on permutations. The Kendall-tau distance is the graph distance in the vertices of this polytope, and Spearman distance is the squared Euclidean distance between two vertices (Thompson (1993)). Additionally, the antipode of a permutation is its reverse permutation. With this intuition, we use the hypersphere as a continuous relaxation of the space of permutations, where selecting samples far apart on the hypersphere corresponds to sampling permutations far apart in the distance metrics of interest.

We now describe a process for sampling from the set of permutations inscribing $\mathbb{S}^{d-2}$. First, shift and scale the permutohedron to lie around the origin with radius $r = 1$. The transformation on vertex σ^{-1} is given by

$$\hat{\sigma}^{-1} = \frac{\sigma^{-1} - \mu}{||\sigma^{-1}||}, \tag{11}$$

where $\mu = (\frac{d+1}{2}, \frac{d+1}{2}, \cdots)$ is the mean vector of all permutations, and $||\sigma^{-1}|| = \sqrt{\sum_{i=1}^{d} \sigma^{-1}(i)^2}$.

Now select some vector x of dimension $d-1$, say, uniformly at random from the surface of $\mathbb{S}^{d-2}$. Project x onto the hyperplane in $\mathbb{R}^d$ using the following $(d-1) \times d$ matrix:

$$U = \begin{bmatrix} 1 & -1 & 0 & \dots & 0 \\ 1 & 1 & -2 & \dots & 0 \\ & \vdots & & \ddots & \\ 1 & 1 & 1 & \dots & -(d-1). \end{bmatrix}$$

It is easily verifiable that this basis of row vectors is orthogonal to hyperplane normal $\vec{n}$. Normalising the row vectors of U gives a transformation matrix $\hat{U}$ used to project vector x to the hyperplane by

$$\tilde{x} = \hat{U}^T x,$$

so that

$$\tilde{x}^T \vec{n} = 0.$$

Given $\tilde{x}$, find the closest permutation $\hat{\sigma}^{-1}$ by maximising the inner product

$$\hat{y} = \arg\max_{\hat{\sigma}^{-1}} \tilde{x}^T \hat{\sigma}^{-1}. \tag{12}$$

This maximisation is simplified by noting that $\hat{\sigma}^{-1}$ is always a reordering of the same constants ($\hat{\sigma}^{-1}$ is a scaled and shifted permutation). The inner product is therefore maximised by matching the largest element in $\hat{\sigma}^{-1}$ against the largest element in $\tilde{x}$, then proceeding to the second-largest, and so on. Thus the argmax is performed by finding the permutation corresponding to the order type of $\tilde{x}$, which is order-isomorphic to the coordinates of $\tilde{x}$. The output $\hat{y}$ is a vertex on a scaled permutohedron — to get the corresponding point on the Cayley graph, undo the scale/shift of Eq. 11 to get a true permutation, then invert that permutation:

$$y = \text{inverse}(\hat{y}||\sigma^{-1}|| + \mu). \tag{13}$$

In fact, both Eq. 12 and 13 can be simplified via a routine *argsort*, defined by

$$\text{argsort}(a) = b$$

such that

$$a_{b_0} \leq a_{b_1} \leq \cdots \leq a_{b_n}.$$

In other words, b contains the indices of the elements of a in sorted position.

Algorithm 2 describes the end-to-end process of sampling. We use the algorithm of Knuth (1997) for generating points uniformly at random on $\mathbb{S}^{d-2}$: sample from $d-1$ independent Gaussian random variables and normalise the resulting vector to have unit length. We now make the claim that Algorithm 2 is unbiased.

Theorem 1 *Algorithm 2 generates permutations uniformly at random, i.e.,* $Pr(\sigma) = \frac{1}{d!}, \forall \sigma \in \mathfrak{S}_d$, *from a uniform random sample on* $\mathbb{S}^{d-2}$.

Algorithm 2: Sample permutation from $\mathbb{S}^{d-2}$

Output: σ, a permutation of length d

1 $x \leftarrow N(0,1)$ // x is a vector of $d-1$ i.i.d. normal samples

2 $x \leftarrow \frac{x}{||x||}$ // x lies uniformly on $\mathbb{S}^{d-2}$

3 $\tilde{x} = \hat{U}^T x$

4 $\sigma \leftarrow \text{argsort}(\tilde{x})$ // σ is a uniform random permutation

Proof The point $x \in \mathbb{S}^{d-2}$ from Algorithm 2, line 2, has multivariate normal distribution with mean 0 and covariance $\Sigma = aI$ for some scalar a and I as the identity matrix. $\tilde{x} = \hat{U}^T x$ is an affine transformation of a multivariate normal and so has covariance

$$\begin{aligned} \text{Cov}(\tilde{\text{x}}) &= \hat{U}^T \Sigma \hat{U} \\ &= a\hat{U}^T I \hat{U} \\ &= a\hat{U}^T \hat{U} \end{aligned}$$

The $d \times d$ matrix $\hat{U}^T\hat{U}$ has the form

$$\hat{U}^T\hat{U} = \begin{bmatrix} \frac{d-1}{d} & \frac{-1}{d} & \cdots & \frac{-1}{d} \\ \frac{-1}{d} & \frac{d-1}{d} & \cdots & \frac{-1}{d} \\ & \vdots & \ddots & \\ \frac{-1}{d} & \frac{-1}{d} & \cdots & \frac{d-1}{d} \end{bmatrix}$$

with all diagonal elements $\frac{d-1}{d}$ and off diagonal elements $\frac{-1}{d}$, and so $\tilde{x}$ is equicorrelated. Due to equicorrelation, $\tilde{x}$ has order type such that $\forall \tilde{x}_i, \tilde{x}_j \in x, i \neq j : Pr(\tilde{x}_i < \tilde{x}_j) = \frac{1}{2}$. In other words, all orderings of $\tilde{x}$ are equally likely. The function *argsort* implies an order-isomorphic bijection, that is, *argsort* returns a unique permutation for every unique ordering over its input. As every ordering of $\tilde{x}$ is equally likely, Algorithm 2 outputs permutations $\sigma \in \mathfrak{S}_d$ with $p(\sigma) = \frac{1}{d!}, \forall \sigma \in \mathfrak{S}_d$. ■

Furthermore, Equation 12 associates a point on the surface of $\mathbb{S}^{d-2}$ to the nearest permutation. This implies that there is a Voronoi cell on the same surface associated with each permutation σ_i, and a sample $\tilde{x}$ is associated with σ_i if it lands in its cell. Figure 3 shows the Voronoi cells on the hypersphere surface for $d = 4$, where the green points are equidistant from nearby permutations. A corollary of Theorem 1 is that these Voronoi cells must have equal measure, which is easily verified for $d = 4$.

4.2 Orthogonal Spherical Codes

Having established an order isomorphism $\mathbb{S}^{d-2} \rightarrow \mathfrak{S}_d$, we consider selecting well-distributed points on $\mathbb{S}^{d-2}$. Our first approach, described in Algorithm 3, is to select $2(d-1)$ dependent samples on $\mathbb{S}^{d-2}$ from a basis of orthogonal vectors. Algorithm 3 uses the Gram-Schmidt process to incrementally generate a random basis, then converts each component and its reverse into permutations by the same mechanism as Algorithm 2. The cost of each additional sample is proportional to $O(d^2)$. This sampling method is related to orthogonal Monte Carlo

Algorithm 3: Sample $k = 2(d-1)$ permutations from $\mathbb{S}^{d-2}$

$X \sim N(0,1)_{k/2,d}$ // iid. normal random Matrix
$Y \leftarrow 0_{k,d}$ // Matrix storing output permutations
for $i \leftarrow 1$ **to** $k/2$ **do**
  for $j \leftarrow 1$ **to** i **do**
    $X_i \leftarrow X_i - X_j X_i^T \cdot X_j$ // Gram-Schmidt process
  $X_i \leftarrow \frac{X_i}{||X_i||}$
  $Y_{2i} \leftarrow \text{argsort}(\hat{U}^T X_i)$
  $Y_{2i+1} \leftarrow \text{argsort}(\hat{U}^T(-X_i))$
return Y

techniques discussed in Choromanski et al. (2019). Writing $v([\sigma]_{i-1} \cup \{i\}) - v([\sigma]_{i-1}) = g_i(\sigma)$, the Shapley value estimate for samples given by Algorithm 3 is

$$\bar{\text{Sh}}_i^{\text{orth}}(v) = \frac{1}{n} \sum_{\ell=1}^{n/k} \sum_{j=1}^{k} g_i(\sigma_{\ell j}), \tag{14}$$

where $(\sigma_{\ell 1}, \sigma_{\ell 2}, \cdots, \sigma_{\ell k})$ are a set of correlated samples and n is a multiple of k.

Proposition 1 $\bar{\text{Sh}}_i^{orth}(v)$ *is an unbiased estimator of* $\text{Sh}_i(v)$.

Proof The Shapley value $\text{Sh}_i(v)$ is equivalently expressed as an expectation over uniformly distributed permutations:

$$\text{Sh}_i(v) = \frac{1}{|N|!} \sum_{\sigma \in \mathfrak{S}_d} \left[v([\sigma]_{i-1} \cup \{i\}) - v([\sigma]_{i-1}) \right]$$
$$\text{Sh}_i(v) = \mathbb{E}_{\sigma \sim U}[g_i(\sigma)]$$

The distribution of permutations drawn as orthogonal samples is clearly symmetric, so $p(\sigma_{\ell,j}) = p(\sigma_{\ell,m})$ for any two indices j, m in a set of k samples, and $\mathbb{E}[g_i(\sigma_{\ell,j})] = \mathbb{E}[g_i(\sigma_{\ell,m}))] = \mathbb{E}[g_i(\sigma^{ortho})]$. As the estimator (14) is a sum, by the linearity of expectation

$$\mathbb{E}[\bar{\text{Sh}}_i^{\text{orth}}(v)] = \frac{1}{n} \sum_{\ell=1}^{n/k} \sum_{j=1}^{k} \mathbb{E}[g_i(\sigma_{\ell j})] = \mathbb{E}[g_i(\sigma^{ortho})].$$

By Theorem 1, the random variable σ^{ortho} has a uniform distribution if its associated sample $x \in \mathbb{S}^{d-2}$ is drawn with uniform distribution. Let x be a component of a random orthogonal basis. If the random basis is drawn with equal probability from the set of orthogonal matrices of order $d-1$ (i.e. with Haar distribution for the orthogonal group), then it follows that $\mathbb{E}[g_i(\sigma^{ortho})] = \mathbb{E}_{\sigma \sim U}[g_i(\sigma)]$. The Gram-Schmidt process applied to a square matrix with elements as i.i.d. standard normal random variables yields a random orthogonal matrix with Haar distribution (Mezzadri (2006)). Therefore

$$\text{Sh}_i(v) = \mathbb{E}_{\sigma \sim U}[g_i(\sigma)] = \mathbb{E}_{\sigma \sim U}[g_i(\sigma)]$$
$$= \mathbb{E}[\bar{\text{Sh}}_i^{\text{orth}}(v)]$$

■

The variance of the estimator (14) can be analysed similarly to the antithetic sampling of Section 2.4, extended to k correlated random variables. By extension of the antithetic variance in Equation 4, we have

$$\mathrm{Var}(\bar{\mathrm{Sh}}_i^{\mathrm{orth}}(v)) = \frac{1}{n}\sum_{\ell=1}^{n/k}\sum_{j,m=1}^{k}\mathrm{Cov}(g(\sigma_{\ell j}), g(\sigma_{\ell m})).$$

The variance is therefore minimised by selecting k negatively correlated samples. Our experimental evaluation in Section 5 suggests that, for the domain of interest, orthogonal samples on the sphere are indeed strongly negatively correlated, and the resulting estimators are more accurate than standard Monte Carlo and antithetic sampling in all evaluations.

Samples from Algorithm 3 can also be considered as a type of spherical code. Spherical codes describe configurations of points on the unit sphere maximising the angle between any two points (see Conway et al. (1987)). A spherical code $A(n, \phi)$ gives the maximum number of points in dimension n with minimum angle ϕ. The orthonormal basis and its antipodes trivially yield the optimal code $A(d-1, \frac{\pi}{2}) = 2(d-1)$.

From their relative positions on the Cayley graph we obtain bounds on the Kendall tau kernel $K_\tau(\sigma, \sigma')$ from Section 3 for the samples of Algorithm 3. The angle between vertices of the Cayley graph is related to $K_\tau(\sigma, \sigma')$ in that the maximum kernel value of 1 occurs for two permutations at angle 0 and the minimum kernel value of -1 occurs for a permutation and its reverse, separated by angle π. As the angle between two points (x, x') on $\mathbb{S}_{d-2}$ increases from 0 to π, the kernel $K_\tau(\sigma, \sigma')$ for the nearest permutations (σ, σ') decreases monotonically and linearly with the angle, aside from quantisation error. If the angle between two distinct points (x, x') in our spherical codes is $\frac{\pi}{2}$, we obtain via the map, $\mathbb{S}^{d-2} \to \mathfrak{S}_d$, the permutations (σ, σ') such that

$$|K_\tau(\sigma, \sigma')| \leq 1/2 + \epsilon,$$

with some small constant quantisation error ϵ. Figure 4 shows $k = 6$ samples for the $d = 4$ case. This is made precise in the following result. Note that the statement and its proof are in terms of σ and σ' instead of their inverses (which label the vertices of the permutohedron in our convention), for simplicity; without this change, the meaning is the same, since $n_{\mathrm{dis}}(\sigma, \sigma') = n_{\mathrm{dis}}(\sigma^{-1}, \sigma'^{-1})$ and $A(\sigma)^T A(\sigma') = A(\sigma^{-1})^T A(\sigma'^{-1})$ for any permutations σ, σ'. First, let $\rho = \sqrt{d(d^2-1)/12}$, so that the map $A(y) = (y-\mu)/\rho$ maps the permutohedron to an isometric copy of $\mathbb{S}^{d-2}$ centered at the origin in $\mathbb{R}^d$, the intersection of the unit sphere $\mathbb{S}^{d-1}$ with the hyperplane orthogonal to $\vec{n}$.

Theorem 2 *Suppose $\sigma, \sigma' \in \mathfrak{S}_d$. Then*

$$-2+4\left(\frac{1-K_\tau(\sigma,\sigma')}{2}\right)^{3/2} \leq A(\sigma)^T A(\sigma') - 3K_\tau(\sigma,\sigma') + O(d^{-1}) \leq 2-4\left(\frac{1+K_\tau(\sigma,\sigma')}{2}\right)^{3/2}$$

and, if $A(\sigma)^T A(\sigma') = o(1)$, then

$$|K_\tau(\sigma, \sigma')| \leq 1/2 + o(1).$$

Proof of the above can be found in Appendix A. Theorem 2 is a kind of converse to the so-called Rearrangement Inequality, which states that the maximum dot product between a vector and a vector consisting of any permutation of its coordinates is maximized when the permutation is the identity and minimized when it is the reverse identity. Here, we show what happens in between: as one varies from the identity to its reverse one adjacent transposition at a time, the dot product smoothly transitions from maximal to minimal, with some variability across permutations having the same number of inversions. Interestingly, we do not know if the above bound is the best possible. A quick calculation shows that, letting $k \approx d2^{-1/3}$ be an integer, the permutation

$$\pi = (k, k-1, \ldots, 2, 1, k+1, k+2, \ldots, d-1, d)$$

has $\nu(\pi) = I^T\pi = d^3(1/4 + o(1))$, i.e, $A(I)^T A(\pi) \approx 0$. However, π admits $d^2(2^{-5/3} + o(1))$ inversions, whence $K_\tau(I, \pi) \approx 1 - 2^{-2/3} \approx 0.37 < 1/2$.

Figure 5 shows the distribution of pairs of unique samples taken from random vectors, versus unique samples from an orthogonal basis, at $d = 10$. Samples corresponding to orthogonal vectors are tightly distributed around $K_\tau(\sigma, \sigma') = 0$, and pairs corresponding to a vector and its antipodes are clustered at $K_\tau(\sigma, \sigma') = -1$. Figure 6 plots the bounds from Theorem 2 relating the dot product of vectors on $\mathbb{S}^{d-2}$ to the Kendall tau kernel at $d = 15$.

4.3 Sobol Sequences on the Sphere

We now describe another approach to sampling permutations via $\mathbb{S}^{d-2}$, based on standard quasi-Monte Carlo techniques. Low discrepancy point sets on the unit cube $[0, 1)^{d-2}$ may be projected to $\mathbb{S}^{d-2}$ via area preserving transformations. Such projections are discussed in depth in Brauchart and Dick (2012); Hardin et al. (2016), where they are observed to have good properties for numerical integration. Below we define transformations in terms of the inverse cumulative distribution of the generalised polar coordinate system and use transformed high-dimensional Sobol sequences to obtain well-distributed permutations.

In the generalised polar coordinate system of Blumenson (1960), a point on $\mathbb{S}^{d-2}$ is defined by radius r (here $r = 1$) and $d - 2$ angular coordinates $(r, \varphi_1, \varphi_2, \cdots, \varphi_{d-2})$, where $(\varphi_1, \cdots, \varphi_{d-3})$ range from $[0, \pi]$ and φ_{d-2} ranges from $[0, 2\pi]$.

The polar coordinates on the sphere are independent and have probability density functions

$$f(\varphi_{d-2}) = \frac{1}{2\pi},$$

and for $1 \leq j < d - 2$:

$$f(\varphi_j) = \frac{1}{B(\frac{d-j-1}{2}, \frac{1}{2})} \sin^{(d-j-2)}(\varphi_j),$$

where B is the beta function. The above density function is obtained by normalising the formula for the surface area element of a hypersphere to integrate to 1 (Blumenson (1960)). The cumulative distribution function for the polar coordinates is then

$$F_j(\varphi_j) = \int_0^{\varphi_j} f_j(u)du.$$

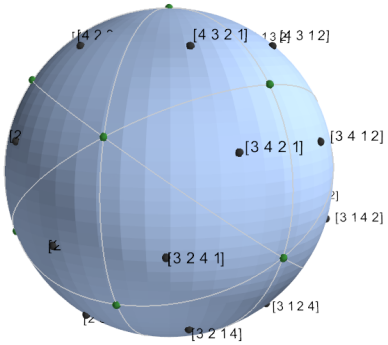


Figure 3: Voronoi cells for permutations on the n-sphere have equal measure. Uniform samples on the n-sphere mapped to these cells result in uniform samples of permutations.

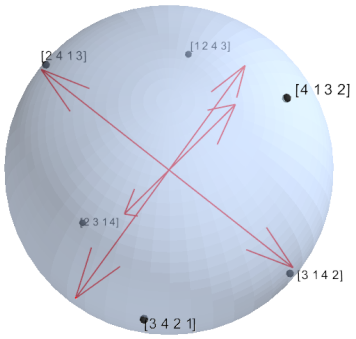


Figure 4: Orthogonal spherical codes: The permutations associated with each orthogonal vector on the n-sphere must be separated by a certain graph distance.

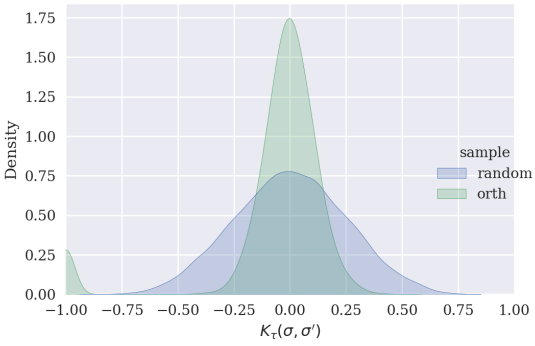


Figure 5: Kernel density estimate of the K_τ similarity of pairs of unique permutations drawn from orthogonal vectors or random vectors on the n-sphere. The leftmost peak for orth corresponds to the antipode samples. Orthogonal samples do not generate similar permutations.

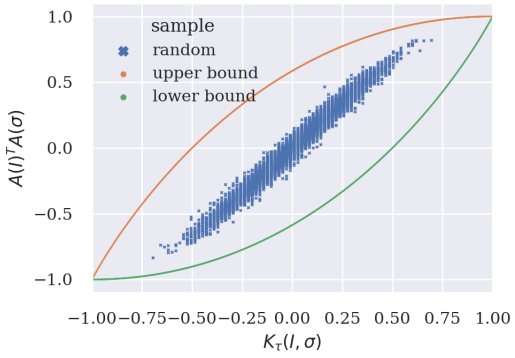


Figure 6: The dot product of two points on $\mathbb{S}^{d-2}$ is closely related to the graph distance $K_\tau(I, \sigma)$ between the associated permutations.

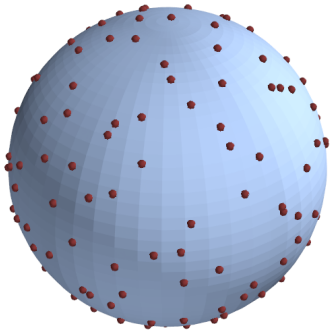


Figure 7: Sobol sphere

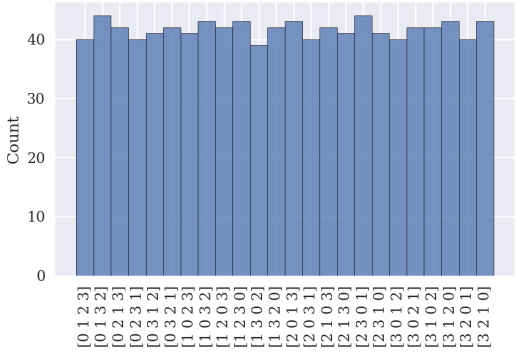


Figure 8: Sobol permutations

As per standard inverse transform sampling, we draw samples $x \in [0,1)^{d-2}$ uniformly from the unit cube and project them to polar coordinates uniformly distributed on the sphere as $\varphi_j = F_j^{-1}(x_j)$. F_j^{-1} can be obtained quickly via a root finding algorithm, such as the bracketing method described in Press et al. (2007).

The points $x \in [0,1]^{d-2}$ are generated using the Sobol sequence (Sobol' (1967)), also referred to as (t,s)-sequences in base 2. Analogously to our discrepancy for functions of permutations in Equation 8, derived with the Mallows kernel, Sobol points can be shown to minimise a discrepancy for the kernel

$$K(x,x') = \prod_{i=1}^{d} \min(1-x_j, 1-x'_j)$$

with $x, x' \in [0,1]^d$, where the discrepancy decreases at the rate $O(\frac{(\log n)^d}{n})$ (see Dick and Pillichshammer (2010)). Sobol points are relatively inexpensive to generate compared with other algorithms discussed in this paper, although explicit convergence rates for discrepancy on the cube do not translate to $\mathbb{S}^{d-2}$ or $\mathfrak{S}_d$.

Combining Sobol points with inverse transform sampling yields uniformly distributed points on $\mathbb{S}^{d-2}$. To map these points to permutations, we project from $[0,1)^{d-1}$ to the hyperplane in $\mathbb{R}^d$ containing the permutohedron (such that points are orthogonal to the normal in Eq. 10) using the matrix $\hat{U}$, and apply argsort to obtain permutations.

Combining all of the above, Algorithm 4 describes the process of generating permutation samples from a Sobol sequence. Figure 7 shows 200 Sobol points distributed on the surface of the sphere. As our Sobol sequence and inverse CDF sampling generate points uniformly distributed on the n-sphere, Theorem 1 applies, and Algorithm 4 samples permutations from a uniform distribution in an unbiased way. Figure 8 shows the distribution of 1000 permutations sampled with $d = 4$, which is clearly uniform.

In Section 3, we proposed sampling methods for the Shapley value approximation problem based on directly optimising discrepancy for the symmetric group. While these methods have some more explicit guarantees in terms of quadrature error they also suffer from expensive optimisation processes. The methods discussed in this section, based on the hypersphere, have the advantage of being linear-time in the number of samples n. Table 1

Algorithm 4: Sobol Permutations

Function `PolarToCartesian`$((r, \varphi_1, \varphi_2, \cdots, \varphi_{d-2}))$**:**
  Output: $\vec{x}$
  for $i \leftarrow 1$ **to** $d-1$ **do**
    $x_i \leftarrow r$
    for $j \leftarrow 1$ **to** $i-1$ **do**
      $x_i \leftarrow x_i \sin \varphi_j$
    if $i < d-2$ **then**
      $x_i \leftarrow x_i \cos \varphi_i$
  return x

Function `SobolPermutations`(n, d)**:**
  Output: Π
  for $i \leftarrow 1$ **to** n **do**
    $x \leftarrow \text{SobolPoint}(i, n, d)$ `// x has d − 2 elements`
    $\varphi \leftarrow \vec{0}$
    for $j \leftarrow 1$ **to** $d-2$ **do**
      $\varphi_j \leftarrow F_j^{-1}(x_j)$ `// Inverse CDF transformation`
    $y \leftarrow$`PolarToCartesian`$(1, \varphi)$ `// y has d − 1 elements`
    $z \leftarrow \hat{U}^T y$ `// z has d elements`
    $\Pi_i \leftarrow \text{argsort}(z)$
  return Π

Table 1: Complexity in n

Algorithm	Complexity
Herding	$O(n^2)$
SBQ	$O(n^3)$
Orthogonal	$O(n)$
Sobol	$O(n)$

summarises the complexity of the proposed algorithms. In the next section, we evaluate these algorithms in terms of quadrature error and runtime.

5. Evaluation

We evaluate the performance of permutation sampling strategies on tabular data, image data, and in terms of data-independent discrepancy scores. Table 2 describes a set of six tabular datasets. These datasets are chosen to provide a mixture of classification and regression problems, with varying dimensionality, and a mixture of problem domains. For this analysis, we avoid high-dimensional problems, such as natural language processing, due to the difficulty of solving for and interpreting Shapley values in these cases. For the image

Table 2: Tabular datasets

NAME	ROWS	COLS	TASK	REF
ADULT	48842	107	CLASS	Kohavi (1996)
BREAST_CANCER	699	30	CLASS	Mangasarian and Wolberg (1990)
BANK	45211	16	CLASS	Moro et al. (2014)
CAL_HOUSING	20640	8	REGR	Pace and Barry (1997)
MAKE_REGRESSION	1000	10	REGR	Pedregosa et al. (2011)
YEAR	515345	90	REGR	Bertin-Mahieux et al. (2011)

Table 3: Permutation sampling algorithms under evaluation

Sampling algorithm	Already proposed for Shapley values	Description and references
Monte-Carlo	Yes	Section 2.3
Monte-Carlo Antithetic	Yes	Section 2.4
Owen	Yes	Section 2.5
Owen-Halved	Yes	Section 2.5
Stratified	Yes	Section 2.6
Kernel herding	No	Section 3.1
SBQ	No	Section 3.2
Orthogonal Spherical Codes	No	Section 4.2
Sobol Sequences	No	Section 4.3

evaluation we use samples from the ImageNet 2012 dataset of Russakovsky et al. (2015), grouping pixels into tiles to reduce the dimensionality of the problem to 256.

Experiments make use of a parameterised Mallows kernel for the kernel herding and SBQ algorithms, as well as the discrepancy scores reported in Section 5.4. To limit the number of experiments, we fix the λ parameter for the Mallows kernel at $\lambda = 4$ and use 25 samples to approximate the argmax for the kernel herding and SBQ algorithms. These parameters are chosen to give reasonable performance in many different scenarios. Experiments showing the impact of these parameters and justification of this choice can be found in Appendix B.

To examine different types of machine learning models, we include experiments for gradient boosted decision trees (GBDT), a multilayer perceptron with a single hidden layer, and a deep convolutional neural network. All of these models are capable of representing non-linear relationships between features. We avoid simple models containing only linear relationships because their Shapley value solutions are trivial and can be obtained exactly in a single permutation sample. For the GBDT models, we are able to compute exact Shapley values as a reference, and for the other algorithms we use unbiased estimates of the Shapley values by averaging over many trials. More details are given in the respective subsections.

The sampling algorithms under investigation are listed in Table 3. The Monte-Carlo, antithetic Monte-Carlo, stratified sampling, Owen sampling, and Owen-halved methods have been proposed in existing literature for the Shapley value approximation problem. The kernel herding, SBQ, Orthogonal and Sobol methods are the newly proposed methods and form the main line of enquiry in this work.

The experimental evaluation proceeds as follows:

- Section 5.1 first evaluates existing algorithms on tabular data using GBDT models, reporting exact error scores. MC-Antithetic emerges as the clear winner, so we use this as a baseline in subsequent experiments against newly proposed algorithms.
- Section 5.2 examines Shapley values for newly proposed sampling algorithms as well as MC-Antithetic using GBDT models trained on tabular data, and reports exact error scores.
- Section 5.3 examines Shapley values for newly proposed sampling algorithms as well as MC-Antithetic using multilayer perceptron models trained on tabular data, and reports error estimates.
- Section 5.4 reports data-independent discrepancy and execution time for newly proposed sampling algorithms and MC-Antithetic.
- Section 5.5 evaluates Shapley values for newly proposed sampling algorithms and MC-Antithetic using a deep convolutional neural network trained on image data, reporting error estimates.

5.1 Existing algorithms - Tabular data and GBDT models

We train GBDT models on the tabular datasets listed in Table 2 using the XGBoost library of Chen and Guestrin (2016). Models are trained using the entire dataset (no test/train split) using the default parameters of the XGBoost library (100 boosting iterations, maximum depth 6, learning rate 0.3, mean squared error objective for regression, and binary logistic objective for classification). The exact Shapley values are computed for reference using the TreeShap Algorithm (Algorithm 3) of Lundberg et al. (2020), a polynomial-time algorithm specific to decision tree models.

Recall from Section 2.2, to define Shapley values for a machine learning model, features not present in the active subset must be marginalised out. To compare our results to the exact Shapley values, we use the same method as Lundberg et al. (2020). A small fixed set of 'background instances' is chosen for each dataset. These form a distribution with which to marginalise out the effect of features. To calculate Shapley values for a given row (a 'foreground' instance), features not part of the active subset are replaced with values from a background instance. The characteristic function evaluation $v(S)$ is then the mean of a set of model predictions, where each time, the foreground instance has features not in subset S replaced by a different background instance. For details, see Lundberg et al. (2020) or the SHAP software package. For classification models, we examine the log-odds output, as the polynomial-time exact Shapley Value algorithm only works when model outputs are additive, and because additive model outputs are consistent with the efficiency property of Shapley values.

For each dataset/algorithm combination, Shapley values are evaluated for all features of 10 randomly chosen instances, using a fixed background dataset of 100 instances to marginalise out features. Shapley values are expensive to compute, and are typically evaluated for a small number of test instances, not the entire dataset. The choice of 10 instances is a balance between computation time and representing the variation of Shapley values across the dataset. The approximate Shapley values for the 10 instances form a $10 \times d$

matrix, from which we calculate the elementwise mean squared error against the reference Shapley values. For $10 \times d$ matrix Z, the MSE for our approximation $\hat{Z}$ is defined as

$$\text{MSE}(Z, \hat{Z}) = \frac{1}{10d} \sum_{i}^{10} \sum_{j}^{d} (Z_{i,j} - \hat{Z}_{i,j})^2 \tag{15}$$

As the sampling algorithms are all randomised, we repeat the experiment 25 times (on the same foreground and background instances) to generate confidence intervals.

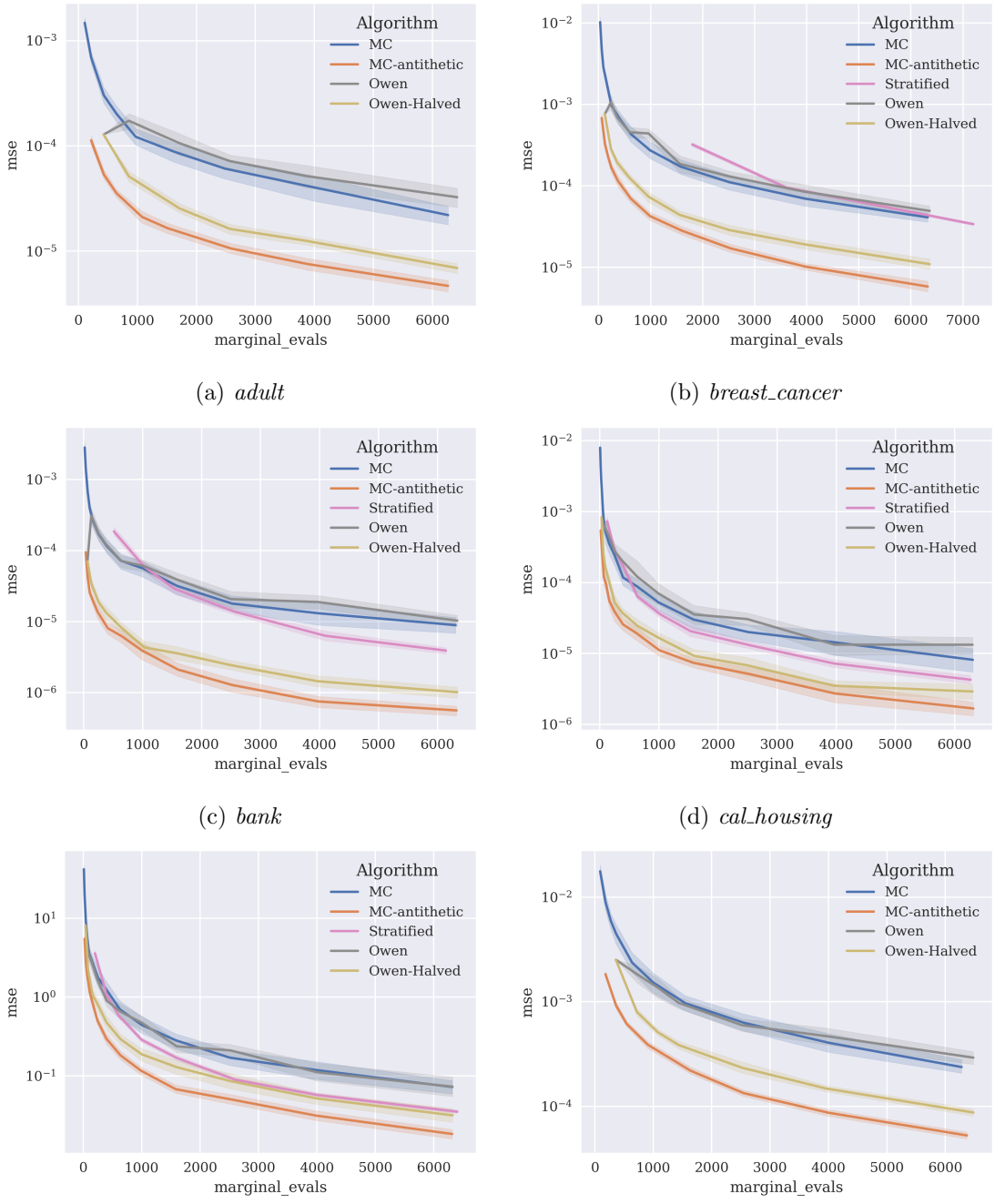
The results are shown in Figure 9. Algorithms are evaluated according to number of evaluations of $v(S \cup i) - v(S)$, written as 'marginal_evals' on the x-axis of figures. If the algorithm samples permutations, the number of marginal evaluations is proportional to nd, where n is the number of permutations sampled. The stratified sampling method is missing for the adult and year datasets because it requires at least $2d^2$ samples, which becomes intractable for the higher-dimensional datasets. The shaded areas show a 95% confidence interval for the mean squared error. Of the existing algorithms, MC-antithetic is the most effective in all experiments. For this reason, in the next sections, we use MC-Antithetic as the baseline when evaluating the kernel herding, SBQ, orthogonal and Sobol methods.

5.2 Proposed algorithms - Tabular data and GBDT models

Here, we perform experiments using the same methodology in the previous section, examining the mean squared error of the proposed algorithms kernel herding, SBQ, orthogonal and Sobol, against MC-antithetic as the baseline. Figure 10 plots the results. For the lower-dimensional *cal_housing* and *make_regression* datasets, we see good performance for the herding and SBQ methods. This good performance does not translate to the higher-dimensional datasets *adult* and *year*, where herding and SBQ are outperformed by the baseline MC-antithetic method. On the problems where herding and SBQ are effective, SBQ outperforms herding in terms of mean squared error, presumably due to its more aggressive optimisation of the discrepancy. The Sobol method is outperformed by the baseline MC-antithetic method in four of six cases. The orthogonal method shows similar performance to MC-antithetic for a small number of samples, but improves over MC-antithetic as the number of samples increases in all six problems. This is because the orthogonal method can be considered an extension of the antithetic sampling scheme — increasing the number of correlated samples from 2 to $2(d-1)$. The orthogonal method also appears preferable to the Sobol method on this collection of datasets: it loses on two of them (*cal_housing* and *make_regression*) but the difference in error is very small on these two datasets.

5.3 Proposed algorithms - Tabular data and MLP models

Now, we examine error estimates for the proposed algorithms on tabular data using a multi-layer perceptron (MLP) model, presenting the results in Figure 11. As for the GBDT models, we use the entire dataset for training. The model is trained using the scikit-learn library (Pedregosa et al. (2011)) with default parameters: a single hidden layer of 100 neurons, a relu activation function, and trained with the adam optimiser (Kingma and Ba (2014)) for 200 iterations with an initial learning rate of 0.001. MSE is optimised for regression data, and log-loss for classification data.



(a) *adult*

(b) *breast_cancer*

(c) *bank*

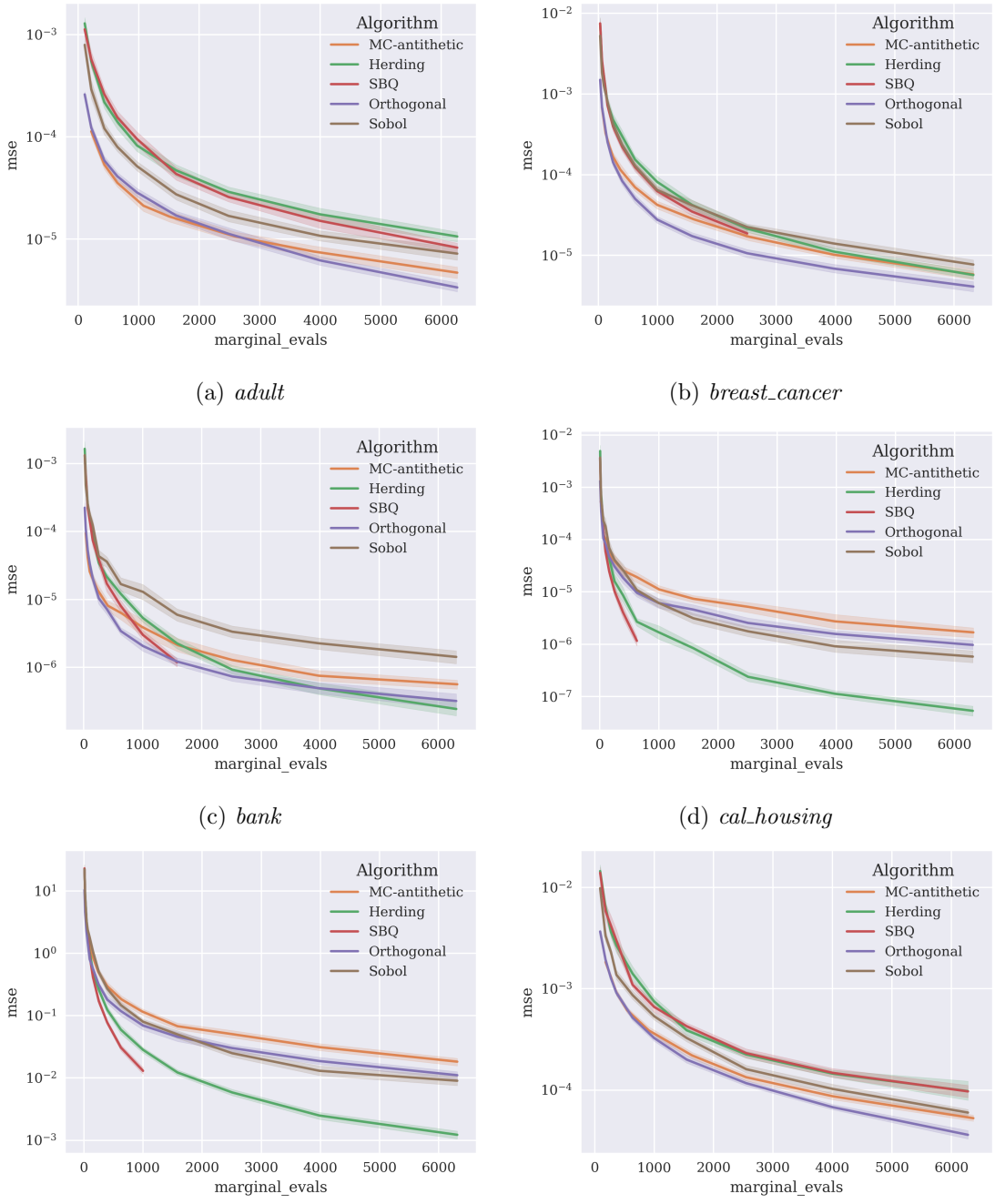
(d) *cal_housing*

(e) *make_regression*

(f) *year*

Figure 9: Existing algorithms - Tabular data, GBDT models

For Shapley value computation, features are marginalised out using background features in exactly the same way as for GBDT models. As we do not have access to exact Shapley



(a) *adult* (b) *breast_cancer*

(c) *bank* (d) *cal_housing*

(e) *make_regression* (f) *year*

Figure 10: Proposed algorithms - Tabular data, GBDT models

values, and all sampling algorithms are randomised, we use standard Monte Carlo error

estimates based on an unbiased sample estimate. The exact Shapley values Z are substituted with the elementwise mean of the estimates over 25 trials.

For the MLP models, we generally see similar results to the GBDT models: herding and SBQ converging quickly for the lower dimensional *cal_housing* and *make_regression* datasets, and the orthogonal method consistently outperforming MC-antithetic across datasets. The orthogonal method also again appears preferable overall to Sobol sampling. For some datasets, such as *adult*, results are more tightly clustered than for the GBDT model. This could indicate fewer higher-order feature interactions in the single layer MLP model, leading to lower variance in the Shapley value characteristic function with respect to the input subsets. In other words, the choice of permutation samples may matter less when strong features interactions are absent.

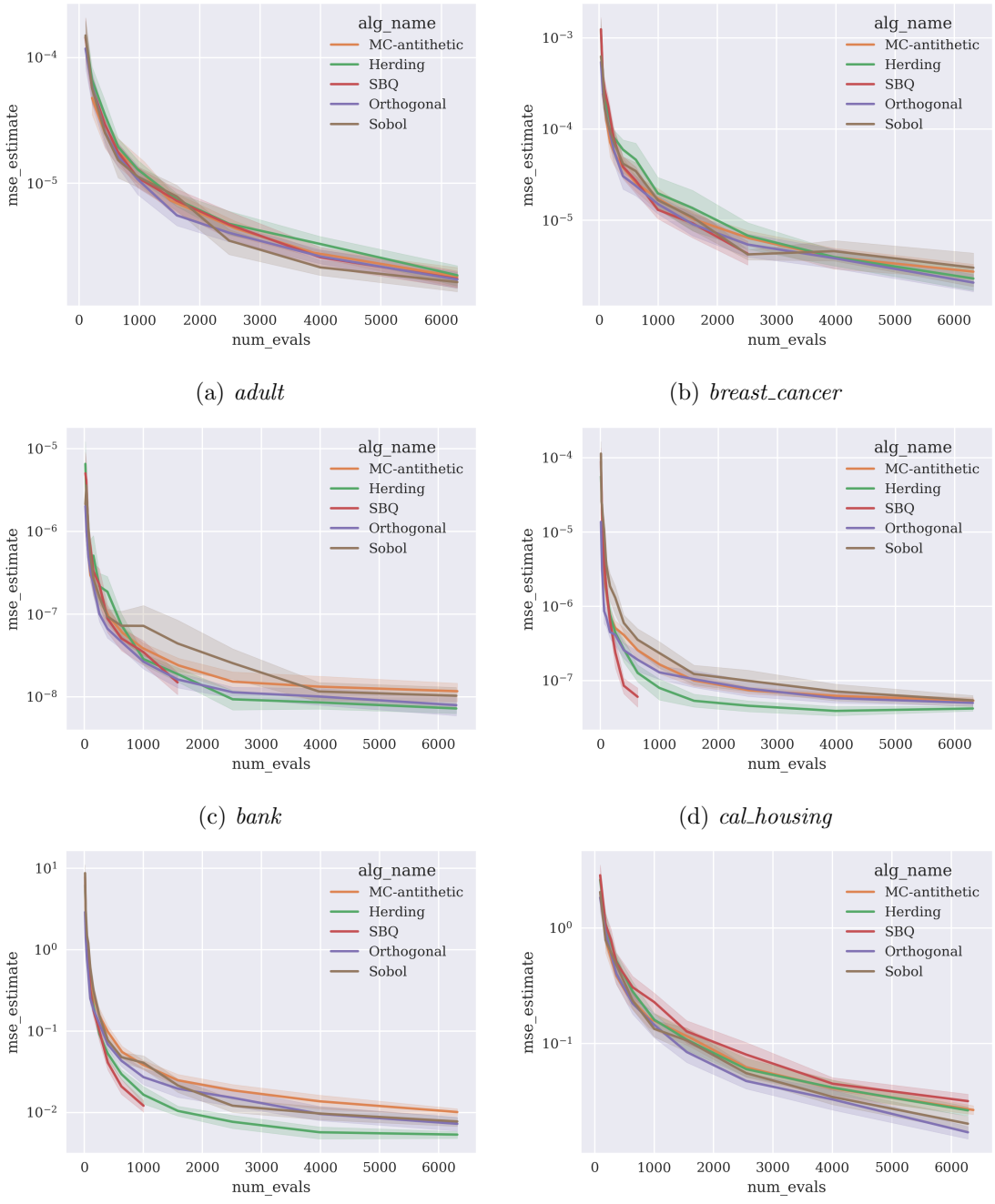
5.4 Proposed algorithms - Discrepancy scores

Table 4 shows mean discrepancies over 25 trials for the various permutation sampling algorithms, calculated as per Equation 8 using the Mallows kernel with $\lambda = 4$. Runtime (in seconds) is also reported, where permutation sets are generated using a single thread of a Xeon E5-2698 CPU. We omit results for SBQ at $n = 1000$ due to large runtime. At low dimension, the methods directly optimising discrepancy (herding and SBQ) achieve significantly lower discrepancies than the other methods. For $d = 10$, $n = 1000$, herding achieves almost a twofold reduction in discrepancy over antithetic sampling, directly corresponding to an almost twofold lower error bound under the Koksma-Hlawka inequality. Antithetic sampling has a higher discrepancy than all other methods here, except in one case ($d = 200$, $n = 10$) where it achieves lower discrepancy than herding and SBQ. In general, we see the orthogonal and Sobol methods are the most effective at higher dimensions, collectively accounting for the lowest discrepancies at $d = 200$. When n is large, the runtime of the herding and SBQ methods becomes impractical. Herding takes as long as 242s to generate $n = 1000$ permutations at $d = 200$. The Sobol and Orthogonal methods have more reasonable runtimes, the longest of which occurs with Sobol at $n = 1000, d = 200$, taking 2s. These results show that no single approach is best for all problems but significant improvements can be made over the baseline MC-antithetic method.

The discrepancies computed above are applicable beyond the particular machine learning problems discussed in this paper. Table 4 provides a reference for how to select samples of permutations at a given computational budget and dimension, not just for Shapley value approximation, but for any bounded function $f : \mathfrak{S}_d \to \mathbb{R}$.

5.5 Proposed algorithms - Image data and deep CNN models

We continue by evaluating the effectiveness of the proposed sampling algorithms for an image classification interpretability problem. Figure 12 depicts eight images randomly selected from the ImageNet 2012 dataset of Russakovsky et al. (2015). We use approximate Shapley values to examine the contribution of the different image tiles towards the output label predicted by a ResNet50 (He et al. (2016)) convolutional neural network. Images are preprocessed as per He et al. (2016), by cropping to a 1:1 aspect ratio, centering along the larger axis, resizing to 224x224, and subtracting the mean RGB values of the ImageNet training set. We examine the highest probability class output for each image. The predicted



(a) *adult* (b) *breast_cancer*

(c) *bank* (d) *cal_housing*

(e) *make_regression* (f) *year*

Figure 11: Proposed algorithms - Tabular data, MLP models

labels are displayed above each image in Figure 12. Note that labels may be incorrect (e.g. "vacuum"). To examine the Shapley values for each image, we group pixels into 14x14x3

Table 4: Discrepancy (lower is better) of permutation samples using Mallows kernel $\lambda = 4$

			Discrepancy		Time	
			mean	std	mean	std
d	n	Algorithm				
10	100	Herding	0.241113	0.002430	0.008067	0.001414
		MC-antithetic	0.263626	0.009881	0.000096	0.000083
		Orthogonal	0.244233	0.002904	0.000788	0.000100
		SBQ	0.240285	0.002483	0.111968	0.397417
		Sobol	0.258403	0.006546	0.003190	0.005711
		Herding	0.058985	0.000718	0.979663	0.602856
		MC-antithetic	0.083784	0.004493	0.000718	0.001173
		Orthogonal	0.070014	0.001722	0.011655	0.028917
		SBQ	0.055803	0.000410	41.546244	9.239474
		Sobol	0.068875	0.001716	0.047935	0.167501
		Herding	0.013353	0.000098	52.961361	4.024325
		MC-antithetic	0.026816	0.001522	0.018554	0.039945
		Orthogonal	0.021802	0.000622	0.110050	0.239145
		SBQ	-	-	-	-
		Sobol	0.017830	0.000405	0.049177	0.138812
50	100	Herding	0.270276	0.001269	0.022578	0.047269
		MC-antithetic	0.272373	0.001582	0.001219	0.003305
		Orthogonal	0.269126	0.000275	0.023522	0.045404
		SBQ	0.270045	0.001410	0.343958	0.879071
		Sobol	0.270824	0.000995	0.009038	0.007187
		Herding	0.079701	0.000242	1.129456	0.483339
		MC-antithetic	0.086087	0.000603	0.000522	0.000333
		Orthogonal	0.072479	0.000158	0.054071	0.170046
		SBQ	0.079182	0.000163	27.135371	7.967242
		Sobol	0.078587	0.000350	0.009147	0.006215
		Herding	0.022585	0.000033	85.038503	3.604017
		MC-antithetic	0.027253	0.000190	0.048581	0.201433
		Orthogonal	0.022904	0.000040	0.351684	1.165377
		SBQ	-	-	-	-
		Sobol	0.022337	0.000108	0.960075	0.712597
200	100	Herding	0.280202	0.000991	0.111566	0.400984
		MC-antithetic	0.273105	0.000257	0.000280	0.000425
		Orthogonal	0.272387	0.000042	0.196384	0.050607
		SBQ	0.280004	0.001052	0.098380	0.184844
		Sobol	0.272416	0.000259	0.795164	1.435970
		Herding	0.083784	0.000119	3.429252	1.765489
		MC-antithetic	0.086434	0.000157	0.043272	0.120758
		Orthogonal	0.083187	0.000019	0.463764	1.133650
		SBQ	0.083656	0.000115	39.163286	10.229544
		Sobol	0.083928	0.000075	0.691592	0.778084
		Herding	0.025508	0.000018	242.515501	6.934064
		MC-antithetic	0.027318	0.000042	0.007029	0.002159
		Orthogonal	0.023420	0.000010	0.560501	0.212319
		SBQ	-	-	-	-
		Sobol	0.023411	0.000040	1.995500	0.782391

tiles, considering each tile to be a single feature. This reduces the dimensionality of the interpretability problem from $224 \cdot 224 \cdot 3 = 150,528$ to a more tractable 256 dimensions.

Algorithms	Permutation time (s) mean	std	Other time (s) mean	std
Herding	3.050258	0.431358	40.791352	0.491466
MC	0.000986	0.000444	40.586155	0.537534
MC-antithetic	0.000701	0.000202	40.898156	0.553361
Orthogonal	0.230704	0.012352	40.665781	0.459624
SBQ	6.253297	1.125828	40.479887	0.437312
Sobol	0.050369	0.019457	40.621966	0.546090

Table 5: Time to generate Shapley values for a single image, separated into time to generate 100 permutations, and other (model evaluation and averaging of model evaluations).

When a tile is not part of the active feature set, its pixel values are set to (0,0,0) (black). For the purpose of computing Shapley values, we examine the log-odds output of the ResNet50 model, as the additivity of these outputs is consistent with the efficiency property of Shapley values. Sampling algorithms are applied to the Shapley value problem 25 times, each with a different seed. As computing an exact baseline is intractable, we estimate the mean squared error in the same manner as Section 5.3. Error estimates are presented as a bar chart in the third column of Figure 12. The second column displays a heat map of the estimated Shapley values for the first trial of the sampling algorithm with the lowest error estimate for the corresponding image. Yellow areas show image tiles that contribute positively to the predicted label, darker purple areas correspond to areas contributing negatively to the predicted label. From this analysis, we see that the Sobol method has the lowest error estimate in all cases. While the herding, orthogonal and SBQ methods generally show lower sample variance than plain Monte Carlo, they do not appear to generate significantly better solutions than the much simpler MC-antithetic method for this problem. This raises the question of whether the herding and SBQ methods could do better with a better choice of λ parameter. However, Figure 15 in Appendix B shows that alternative parameter values do not significantly improve the performance of herding and SBQ for this problem.

Table 5 shows the execution time of permutation generation compared compared to other computation needed to generate the Shapley values for a single image. This other computation consists of evaluating ResNet50 and performing weighted averages. Generating Shapley values for an image using 100 permutation samples and 256 features requires $100 \cdot (256 + 1) = 25700$ model evaluations, taking around 40s on an Nvidia V100 GPU. Permutations are generated using a single thread of a Xeon E5-2698 CPU. Of the permutation sampling algorithms, we see that the linear-time algorithms (MC, MC-antithetic, Orthogonal, Sobol) do not significantly affect total runtime, however the runtime of the Herding and SBQ algorithms is significant relative to the time required for obtaining predictions from the model.

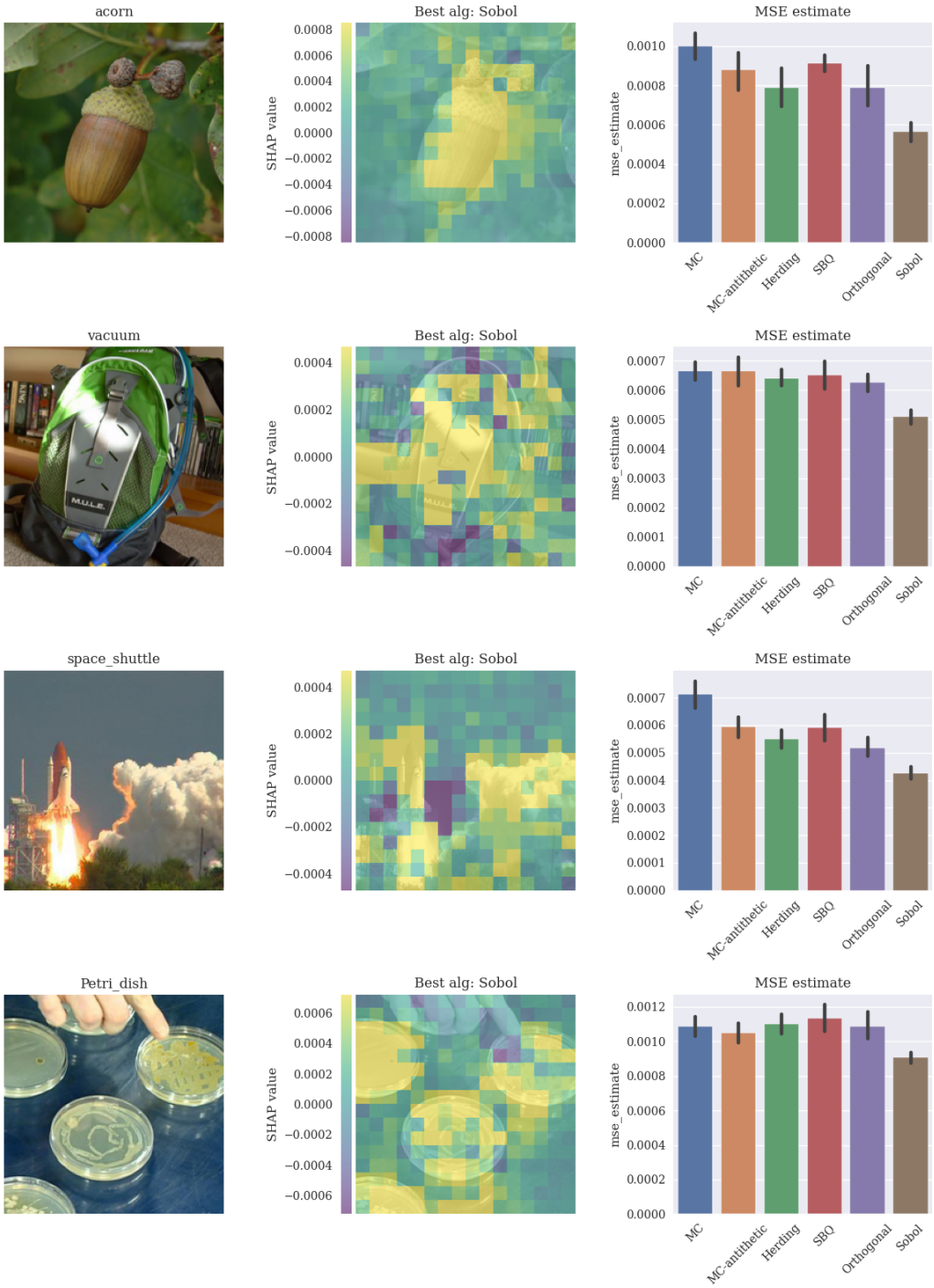


Figure 12: MSE estimates for 100 permutation samples applied to image classifications made by ResNet50

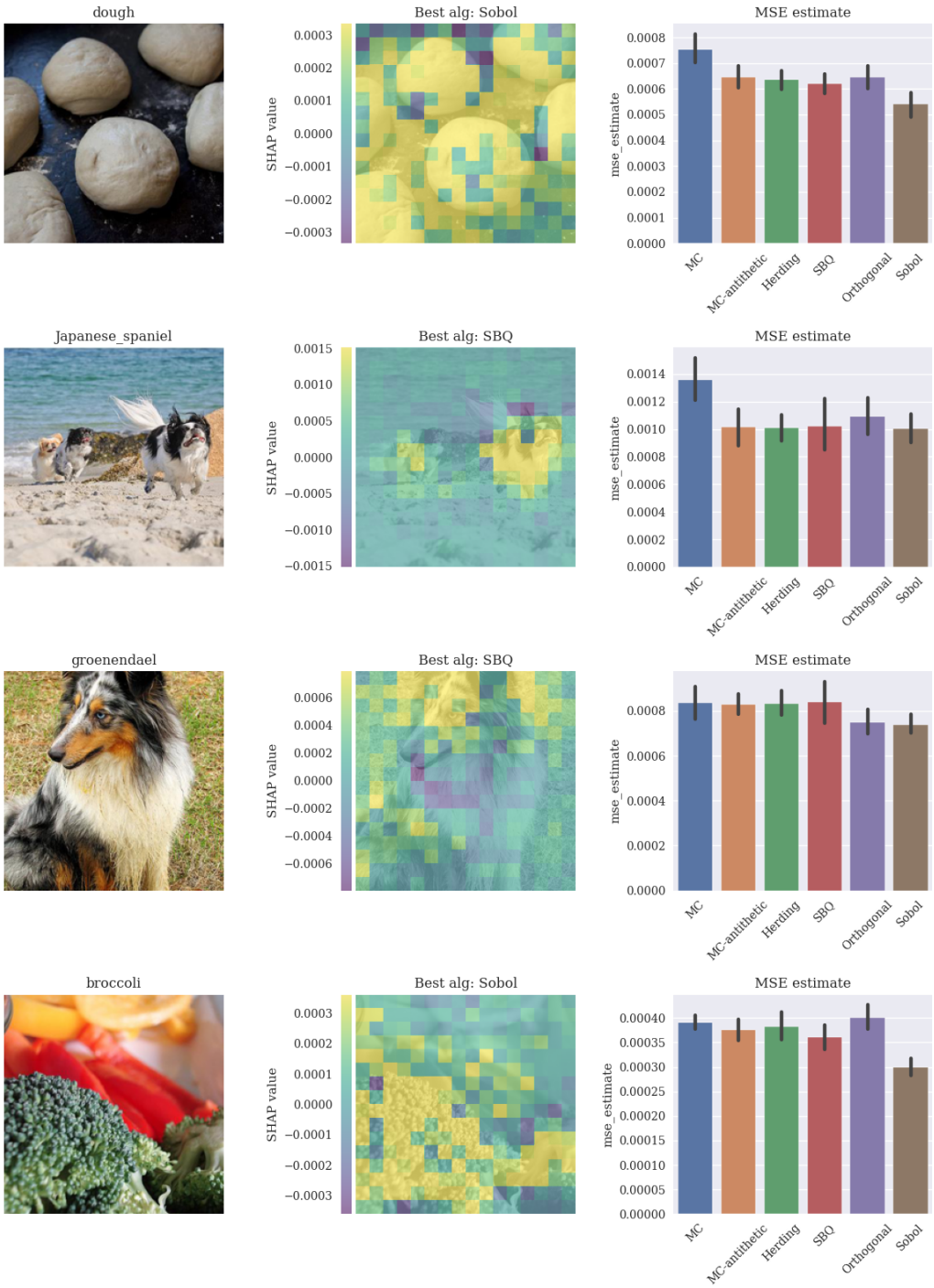


Figure 12 (Cont.): MSE estimates for 100 permutation samples applied to image classifications made by ResNet50

6. Conclusion

In this work, we propose new techniques for the approximation of Shapley values in machine learning applications based on careful selection of samples from the symmetric group $\mathfrak{S}_d$. One set of techniques draws on theory of reproducing kernel Hilbert spaces and the optimisation of discrepancies for functions of permutations, and another exploits connections between permutations and the hypersphere $\mathbb{S}^{d-2}$. We perform empirical analysis of approximation error for GBDT and neural network models trained on tabular data and image data. We also evaluate data-independent discrepancy scores for various sampling algorithms at different dimensionality and sample sizes. The introduced sampling methods show improved convergence over existing state-of-the-art methods in many cases. Our results show that kernel-based methods may be more effective for lower-dimensional problems, and methods sampling from $\mathbb{S}^{d-2}$ are more effective for higher-dimensional problems. Further work may be useful to identify the precise conditions under which optimising discrepancies based on a Mallows kernel is effective, and to clarify the impact of dimensionality on choice of sampling algorithm for Shapley value approximation.

References

Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

LE Blumenson. A derivation of n-dimensional spherical coordinates. *The American Mathematical Monthly*, 67(1):63–66, 1960.

Johann S Brauchart and Josef Dick. Quasi–Monte Carlo rules for numerical integration over the unit sphere $\mathbb{S}^2$. *Numerische Mathematik*, 121(3):473–502, 2012.

Javier Castro, Daniel Gómez, and Juan Tejada. Polynomial calculation of the shapley value based on sampling. *Computers & Operations Research*, 36(5):1726–1730, 2009. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2008.04.004. URL `https://www.sciencedirect.com/science/article/pii/S0305054808000804`. Selected papers presented at the Tenth International Symposium on Locational Decisions (ISOLDE X).

Javier Castro, Daniel Gómez, Elisenda Molina, and Juan Tejada. Improving polynomial estimation of the shapley value by stratified random sampling with optimum allocation. *Computers & Operations Research*, 82:180–188, 2017. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2017.01.019. URL `https://www.sciencedirect.com/science/article/pii/S030505481730028X`.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, pages 785–794. ACM, 2016.

Yutian Chen, Max Welling, and Alex Smola. Super-samples from kernel herding. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, page 109–116, Arlington, Virginia, USA, 2010. AUAI Press. ISBN 9780974903965.

Krzysztof Choromanski, Mark Rowland, Wenyu Chen, and Adrian Weller. Unifying orthogonal Monte Carlo methods. In *International Conference on Machine Learning*, pages 1203–1212. PMLR, 2019.

Shay Cohen, Gideon Dror, and Eytan Ruppin. Feature selection via coalitional game theory. *Neural Computation*, 19(7):1939–1961, 2007.

J. H. Conway, N. J. A. Sloane, and E. Bannai. *Sphere-Packings, Lattices, and Groups*. Springer-Verlag, Berlin, Heidelberg, 1987. ISBN 038796617X.

Ian Covert, Scott Lundberg, and Su-In Lee. Explaining by removing: A unified framework for model explanation. *arXiv preprint arXiv:2011.14878*, 2020.

Xiaotie Deng and Christos H Papadimitriou. On the complexity of cooperative solution concepts. *Mathematics of operations research*, 19(2):257–266, 1994.

Persi Diaconis. *Group representations in probability and statistics*. Institute of Mathematical Statistics Lecture Notes—Monograph Series, 11. Institute of Mathematical Statistics, Hayward, CA, 1988. ISBN 0-940600-14-5. URL `http://projecteuclid.org/euclid.lnms/1215467407`.

Josef Dick and Friedrich Pillichshammer. *Digital nets and sequences: discrepancy theory and quasi–Monte Carlo integration*. Cambridge University Press, 2010.

G. Th. Guilbaud and P. Rosenstiehl. Analyse algébrique d'un scrutin. *Mathématiques et Sciences humaines*, 4:9–33, 1963. URL `www.numdam.org/item/MSH_1963__4__9_0/`.

Doug P Hardin, TJ Michaels, and Edward B Saff. A comparison of popular point configurations on $\mathbb{S}^2$. *Dolomites Research Notes on Approximation*, 9:16–49, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Fred J. Hickernell. What affects the accuracy of quasi-Monte Carlo quadrature? In Harald Niederreiter and Jerome Spanier, editors, *Monte-Carlo and Quasi-Monte Carlo Methods 1998*, pages 16–55, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-642-59657-5.

Edmund Hlawka. Funktionen von beschränkter variatiou in der theorie der gleichverteilung. *Annali di Matematica Pura ed Applicata*, 54(1):325–333, 1961.

Ferenc Huszár and David Duvenaud. Optimally-weighted herding is bayesian quadrature. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, UAI'12, page 377–386, Arlington, Virginia, USA, 2012. AUAI Press. ISBN 9780974903989.

Yunlong Jiao and Jean-Philippe Vert. The kendall and mallows kernels for permutations. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1935–1944. JMLR.org, 2015.

Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

William R. Knight. A computer method for calculating kendall's tau with ungrouped data. *Journal of the American Statistical Association*, 61(314):436–439, 1966. ISSN 01621459. URL `http://www.jstor.org/stable/2282833`.

Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997. ISBN 0201896842.

Ron Kohavi. Scaling up the accuracy of naive-bayes classifiers: A decision-tree hybrid. In *KDD*, pages 202–207. AAAI Press, 1996.

Maria Lomeli, Mark Rowland, Arthur Gretton, and Zoubin Ghahramani. Antithetic and Monte Carlo kernel estimators for partial rankings. *Statistics and Computing*, 29(5): 1127–1147, 2019.

Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017. URL `http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf`.

Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. From local explanations to global understanding with explainable ai for trees. *Nature Machine Intelligence*, 2(1):2522–5839, 2020.

Sasan Maleki. *Addressing the computational issues of the Shapley value with applications in the smart grid*. PhD thesis, University of Southampton, 2015.

Olvi L Mangasarian and William H Wolberg. Cancer diagnosis via linear programming. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1990.

Horia Mania, Aaditya Ramdas, Martin J Wainwright, Michael I Jordan, and Benjamin Recht. On kernel methods for covariates that are rankings. *Electronic Journal of Statistics*, 12:2537–2577, 2018.

Irwin Mann and Lloyd S Shapley. *Values of large games, IV: Evaluating the electoral college by Montecarlo techniques*. Rand Corporation, 1960.

Francesco Mezzadri. How to generate random matrices from the classical compact groups. *arXiv preprint math-ph/0609050*, 2006.

Sérgio Moro, Paulo Cortez, and Paulo Rita. A data-driven approach to predict the success of bank telemarketing. *Decision Support Systems*, 62:22–31, 2014.

Thomas Muir. On a simple term of a determinant. In *Proc. Royal Society Edinburg*, volume 21, pages 441–477, 1898.

Jerzy Neyman. On the two different aspects of the representative method: The method of stratified sampling and the method of purposive selection. *Journal of the Royal Statistical Society*, 97(4):558–625, 1934. ISSN 09528385. URL http://www.jstor.org/stable/2342192.

Harald Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, USA, 1992. ISBN 0898712955.

A. O'Hagan. Bayes–hermite quadrature. *Journal of Statistical Planning and Inference*, 29 (3):245–260, 1991. ISSN 0378-3758. doi: https://doi.org/10.1016/0378-3758(91)90002-V. URL https://www.sciencedirect.com/science/article/pii/037837589190002V.

Ramin Okhrati and Aldo Lipani. A multilinear sampling algorithm to estimate shapley values. In *Proc. of ICPR*, ICPR, 2020.

Art B Owen. Quasi-Monte Carlo sampling. *Monte Carlo Ray Tracing: Siggraph*, 1:69–88, 2003.

Guillermo Owen. Multilinear extensions of games. *Management Science*, 18(5):P64–P79, 1972. ISSN 00251909, 15265501. URL http://www.jstor.org/stable/2661445.

R Kelley Pace and Ronald Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

S. M. Plis, T. Lane, and V. D. Calhoun. Permutations as angular data: Efficient inference in factorial spaces. In *2010 IEEE International Conference on Data Mining*, pages 403–410, 2010. doi: 10.1109/ICDM.2010.122.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, USA, 3 edition, 2007. ISBN 0521880688.

Carl Edward Rasmussen and Zoubin Ghahramani. Bayesian Monte Carlo. *Advances in neural information processing systems*, pages 505–512, 2003.

Reuven Y Rubinstein and Dirk P Kroese. *Simulation and the Monte Carlo method*, volume 10. John Wiley & Sons, 2016.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

Dino Sejdinovic, Bharath Sriperumbudur, Arthur Gretton, and Kenji Fukumizu. Equivalence of distance-based and rkhs-based statistics in hypothesis testing. *The Annals of Statistics*, pages 2263–2291, 2013.

Lloyd S Shapley. A value for n-person games. *Contributions to the Theory of Games*, 2(28): 307–317, 1953.

Il'ya Meerovich Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 7(4): 784–802, 1967.

Erik Strumbelj and Igor Kononenko. An efficient explanation of individual classifications using game theory. *J. Mach. Learn. Res.*, 11:1–18, March 2010. ISSN 1532-4435.

G. L. Thompson. Generalized permutation polytopes and exploratory graphical methods for ranked data. *The Annals of Statistics*, 21(3):1401–1430, 1993. ISSN 00905364. URL `http://www.jstor.org/stable/2242202`.

Erik Štrumbelj and Igor Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowl. Inf. Syst.*, 41(3):647–665, December 2014. ISSN 0219-1377. doi: 10.1007/s10115-013-0679-x. URL `https://doi.org/10.1007/s10115-013-0679-x`.

Appendix A. Proof of Theorem 2 (See page 18)

Theorem 2 *Suppose $\sigma, \sigma' \in \mathfrak{S}_d$. Then*

$$-2+4\left(\frac{1-K_\tau(\sigma,\sigma')}{2}\right)^{3/2} \leq A(\sigma)^T A(\sigma') - 3K_\tau(\sigma,\sigma') + O(d^{-1}) \leq 2-4\left(\frac{1+K_\tau(\sigma,\sigma')}{2}\right)^{3/2}$$

and, if $A(\sigma)^T A(\sigma') = o(1)$, then

$$|K_\tau(\sigma,\sigma')| \leq 1/2 + o(1).$$

Proof For $1 \leq a \leq d-1$, write $t_a \in \mathfrak{S}_d$ for the adjacent transposition of a and $a+1$, i.e., the permutation so that $t_a(j) = j$ for $j \neq a, a+1$, $t_a(a) = a+1$ and $t_a(a+1) = a$. We interpret a product of permutations to be their composition as functions. For a permutation $\pi \in \mathfrak{S}_d$, write $\nu(\pi)$ for the quantity $\sum_{j=1}^d j\pi(j)$, and note that $\nu(I) = \sum_{j=1}^d j^2 = d(d+1)(2d+1)/6$.

It is well-known that the number of inversions $n_{\text{dis}}(I,\pi) = |\{(i,j) : i < j \text{ and } \pi(i) > \pi(j)\}|$ in a permutation π equals the least k so that there exist $a_1, \ldots, a_k$ with

$$\pi = \prod_{i=1}^{k} t_{a_i}. \tag{16}$$

This quantity k is known as the "length" of π and is exactly the distance in the 1-skeleton of the permutohedron representation of $\mathfrak{S}_d$. Furthermore, the a_i can be obtained via bubble sort, i.e., the product (16) begins with

$$t_{\pi(1)-1} t_{\pi(1)-2} \cdots t_1$$

and proceeds recursively on $\pi|_{\{2,\dots,d\}}$. Write π_j for the product of the first j terms in (16) for $1 \le j \le k$, i.e., $\pi_j = \prod_{i=1}^{j} t_{a_i}$, with $\pi_0 = I$. Then the pairs $e_j = \{\pi_j(a_j), \pi_j(a_j+1)\}$ are all distinct, because entries of π in one-line notation switch places at most once when applying the adjacent transpositions, i.e., a larger value a, once it switches places with a smaller value b immediately to its left, never switches place with b again. Furthermore, note that

$$\begin{aligned}\nu(\pi_{j+1}) - \nu(\pi_j) &= (j\pi_{j+1}(a_j) + (j+1)\pi_{j+1}(a_j+1)) - (j\pi_j(a_j) + (j+1)\pi_j(a_j+1)) \\ &= (j\pi_j(a_j+1) + (j+1)\pi_j(a_j)) - (j\pi_j(a_j) + (j+1)\pi_j(a_j+1)) \\ &= \pi_j(a_j+1) - \pi_j(a_j),\end{aligned}$$

a quantity which is always negative because the sequence of transpositions obtained above only ever increases the number of inversions. Therefore, the collection $\{e_j\}_{j=1}^{k}$ consists of k distinct edges of a complete graph on $\{1,\dots,d\}$ and

$$\begin{aligned}\nu(\pi) = \nu(\pi_k) &= \nu(\pi_k) - \nu(I) + \frac{d(d+1)(2d+1)}{6} \\ &= \frac{d(d+1)(2d+1)}{6} + \sum_{j=1}^{k} \pi_j(a_j+1) - \pi_j(a_j) \\ &= \frac{d(d+1)(2d+1)}{6} - \sum_{j=1}^{k} \mathrm{wt}(e_j)\end{aligned}$$

where $\mathrm{wt}(\{a,b\}) = |b-a|$. By greedily selecting the highest-weight or lowest-weight edges of the complete graph K_d weighted by $\mathrm{wt}(\cdot)$, the quantity $\sum_{j=1}^{k} \mathrm{wt}(e_j)$ is always at least

$$1\cdot(d-1) + 2\cdot(d-2) + \cdots + (d-m)\cdot m = \frac{(d+2m-1)(d-m+1)(d-m)}{6}$$

where m is the smallest integer so that $\sum_{j=1}^{d-m}(d-j) = (d+m-1)(d-m)/2 \le k$, because the summands correspond to $d-1$ edges of weight 1, $d-2$ edges of weight 2, and so on up to m edges of weight $d-m$. Similarly, $\sum_{j=1}^{k} \mathrm{wt}(e_j)$ is at most

$$(d-1)\cdot 1 + (d-2)\cdot 2 + \cdots + M\cdot(d-M) = \frac{(d+2M-1)(d-M+1)(d-M)}{6}$$

where M is the largest integer so that $\sum_{j=1}^{d-M} j = (d-M)(d-M+1)/2 \ge k$, since in this case we bound the total edge weight via 1 edge of weight $d-1$, 2 edges of weight $d-2$, and so on up to $d-M$ edges of weight M. Then, letting $\alpha = k/\binom{d}{2}$ (so that $\alpha \in [0,1]$),

$$\begin{aligned}m &= \left\lfloor \frac{\sqrt{4d^2 - 4d - 8k + 1} + 1}{2} \right\rfloor = d\sqrt{1-\alpha} \pm 1 \\ M &= \left\lceil \frac{2d - \sqrt{8k+1} + 1}{2} \right\rceil = d(1-\sqrt{\alpha}) \pm 1\end{aligned}$$

It is straightforward to verify that, if $f(s) = (d+2s-1)(d-s+1)(d-s)/6$, then $s = O(d)$ implies $f(s \pm 1) = f(s) + O(d^2)$. So, letting $\alpha = k/\binom{d}{2}$ (so that $\alpha \in [0,1]$)

$$\begin{aligned}
\nu(\pi) &\leq \frac{d(d+1)(2d+1)}{6} - f(M) \\
&= \frac{d(d+1)(2d+1)}{6} - f(d\sqrt{1-\alpha}) + O(d^2) \\
&= \frac{d^3}{3} - \frac{d^3(1+2\sqrt{1-\alpha})(1-\sqrt{1-\alpha})^2}{6} + O(d^2) \\
&= d^3\left(\frac{2}{3} - \frac{\alpha}{2} - \frac{(1-\alpha)^{3/2}}{3}\right) + O(d^2)
\end{aligned}$$

and

$$\begin{aligned}
\nu(\pi) &\geq \frac{d(d+1)(2d+1)}{6} - f(m) \\
&= \frac{d^3}{3} - f(d(1-\sqrt{\alpha})) + O(d^2) \\
&= \frac{d^3}{3} - \frac{d^3(1+2(1-\sqrt{\alpha}))(1-(1-\sqrt{\alpha}))^2}{6} + O(d^2) \\
&= d^3\left(\frac{1}{3} - \frac{\alpha}{2} + \frac{\alpha^{3/2}}{3}\right) + O(d^2).
\end{aligned}$$

(Note that the functions in parentheses meet for $\alpha = 0, 1$.) Thus, applying the fact that $\nu(\sigma' \circ \sigma^{-1}) = I^T(\sigma' \circ \sigma^{-1}) = \sigma^T\sigma'$, where we regard permutations both as functions π of $\{1, \ldots, d\}$ and as vectors $(\pi(1), \ldots, \pi(d))$,

$$2 + 2\alpha^{3/2} \leq \frac{6\sigma^T\sigma'}{d^3} + O(d^{-1}) + 3\alpha \leq 4 - 2(1-\alpha)^{3/2}$$

Then, since

$$K_\tau(\sigma, \sigma') = 1 - \frac{2n_{\text{dis}}(I, \sigma'\sigma^{-1})}{\binom{d}{2}} = 1 - 2\alpha$$

we have

$$\frac{1}{4} + \left(\frac{1-K_\tau(\sigma,\sigma')}{2}\right)^{3/2} \leq \frac{3\sigma^T\sigma'}{d^3} + O(d^{-1}) - \frac{3K_\tau(\sigma,\sigma')}{4} \leq \frac{5}{4} - \left(\frac{1+K_\tau(\sigma,\sigma')}{2}\right)^{3/2}.$$

Writing $\sigma = \rho x + \mu$ and $\sigma' = \rho x' + \mu$ yields the first claim of the result, since then

$$\sigma^T\sigma' = \frac{d(d^2-1)}{12}A(\sigma)^T A(\sigma') + \frac{d(d+1)^2}{4}.$$

For the second claim, note that, if $\sigma^T\sigma' = d^3(1/4 + o(1))$ (the expected value for random permutations, corresponding to $A(\sigma)^T A(\sigma') \approx 0$),

$$-2 + 4\left(\frac{1-K_\tau(\sigma,\sigma')}{2}\right)^{3/2} \leq -3K_\tau(\sigma,\sigma') + O(d^{-1}) \leq 2 - 4\left(\frac{1+K_\tau(\sigma,\sigma')}{2}\right)^{3/2},$$

i.e.,

$$|K_\tau(\sigma, \sigma')| \leq 1/2 + o(1).$$

■

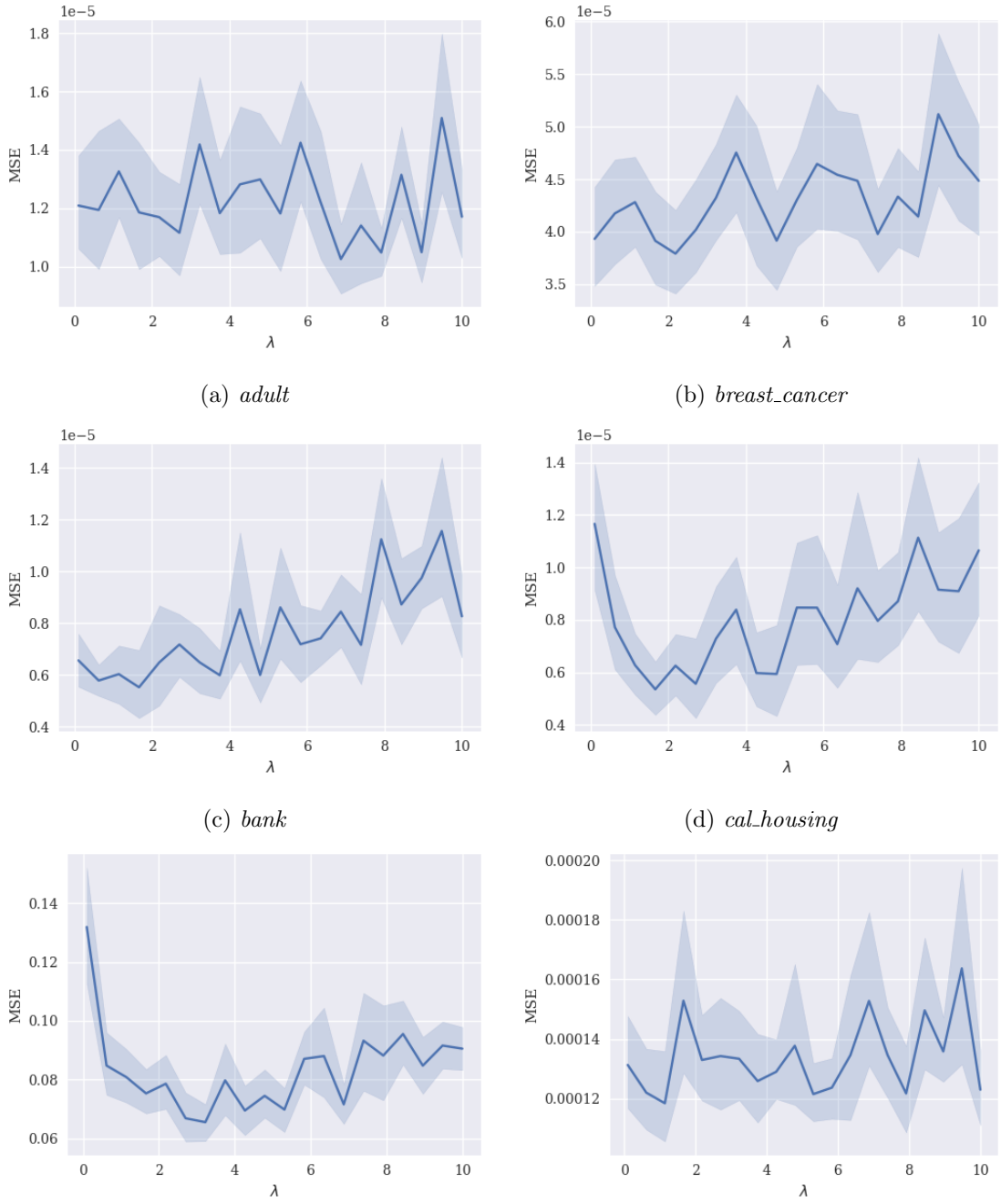
Appendix B. Selection of parameters for the Mallows kernel

The experimental analysis of Section 5 requires the selection of a Mallows kernel λ parameter for the kernel herding and SBQ algorithms, and for the calculation of discrepancies reported in Table 4. As a matter of practicality, we limit the comparisons to a single version of the Mallows kernel due to space constraints. In theory, this parameter could be tuned and the optimal performance reported for each dataset, however, we consider this an unfair reflection of the algorithms performance, as the total number of samples, including the tuning phase, would be considerably higher than for the other algorithms. For kernel-based methods to be effective in practice they should not require extensive parameter tuning. Therefore, we fix $\lambda = 4$, choosing this as an acceptable value based on experiments on different data sources presented below.

Figures 13, 14, and 15 show the error of the kernel herding algorithm using 100 permutation samples and various λ values. As usual, the shaded areas represent 95% confidence intervals. We perform these experiments for tabular datasets with GBDT models, tabular datasets with MLP models, and image data with a ResNet50 model, corresponding to the experiments of Section 5. For some dataset/model combinations a smaller λ value appears to be preferable, for others a larger value is preferable. In the case of image data, the impact of the parameter is small in terms of total MSE, and for tabular data, it is difficult to assign any particular trend due to the volatility of the results. In summary, we compromise with a selection of $\lambda = 4$, which appears to perform acceptably in a wide range of cases.

It is also necessary to choose the number of argmax samples for the herding and SBQ algorithms. Recall from Section 3.1 that we approximate the argmax in herding and SBQ, choosing a new permutation sample by selecting a set of uniform random permutations and selecting one to minimise the discrepancy. Figure 16 shows the effect of varying the number of argmax samples on mean squared error for tabular datasets and GBDT models. We find that 5 to 10 samples is too low for optimal performance, but there is little difference between 25 and 50 samples, so choose 25 samples as a compromise for good accuracy and reasonable runtime.

Given the parameters for the Mallows kernel above, we can also compare it to the Spearman and Kendall tau kernels introduced in Section 3 using the herding algorithm. Figure 17 compares the performance of these kernels on tabular data with GBDT models. The Mallows kernel is applied with $\lambda = 4$, and all kernels are using 25 argmax samples. The Spearman kernel is clearly outperformed by both other kernels. The Kendall Tau kernel is effective for 4 out of 6 datasets, but lags behind for *make_regression* and *cal_housing*. The Mallows kernel is either the most effective, or within a 95% confidence interval of the most effective kernel for all datasets. For this reason, as well as its universal property, we use the Mallows kernel exclusively in the experiments of Section 5.

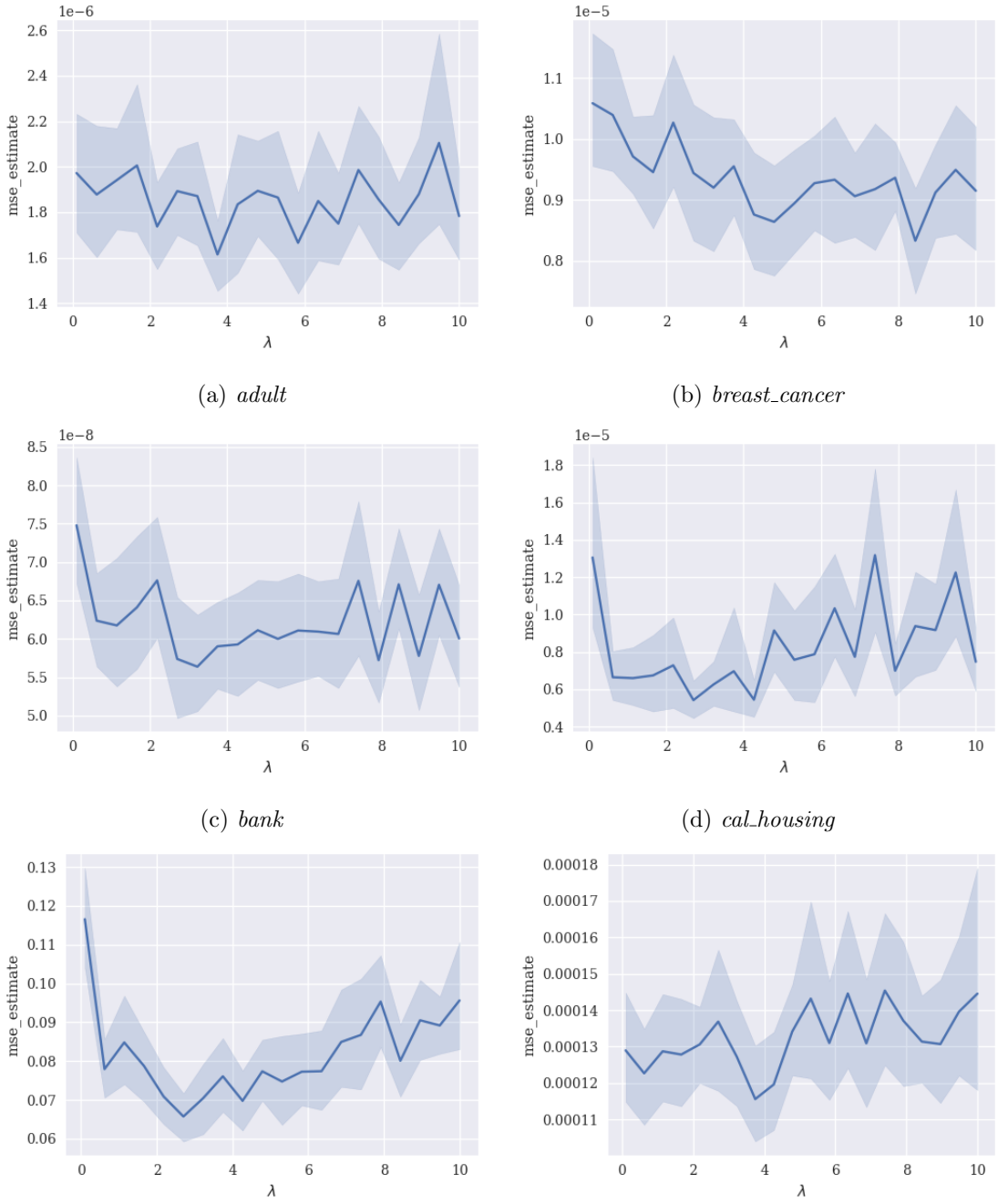


(a) *adult* (b) *breast_cancer*

(c) *bank* (d) *cal_housing*

(e) *make_regression* (f) *year*

Figure 13: Varying λ for 100 herding samples - Tabular data and GBDT models



(a) *adult* (b) *breast_cancer*

(c) *bank* (d) *cal_housing*

(e) *make_regression* (f) *year*

Figure 14: Varying λ for 100 herding samples - Tabular data and MLP models

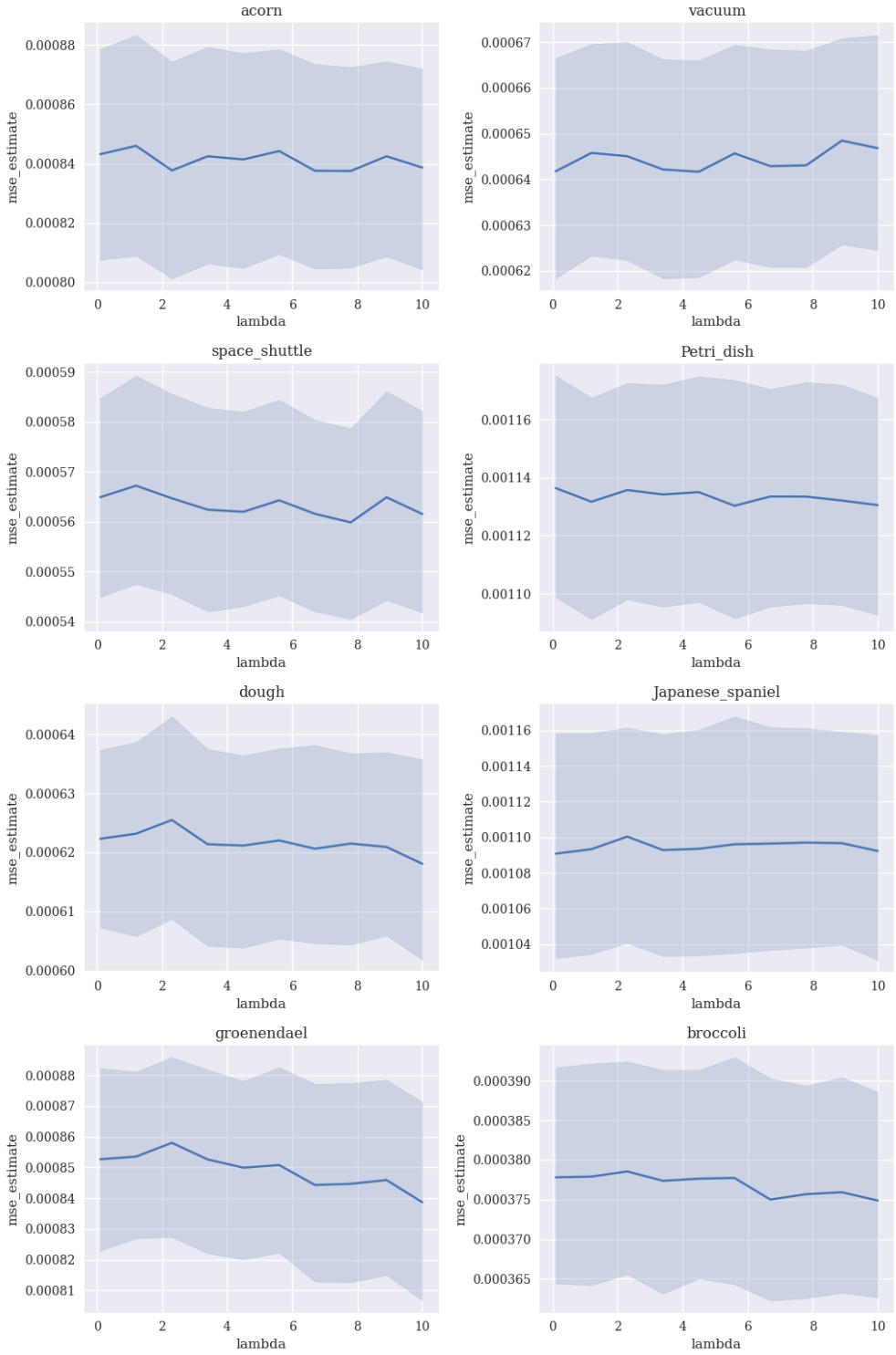
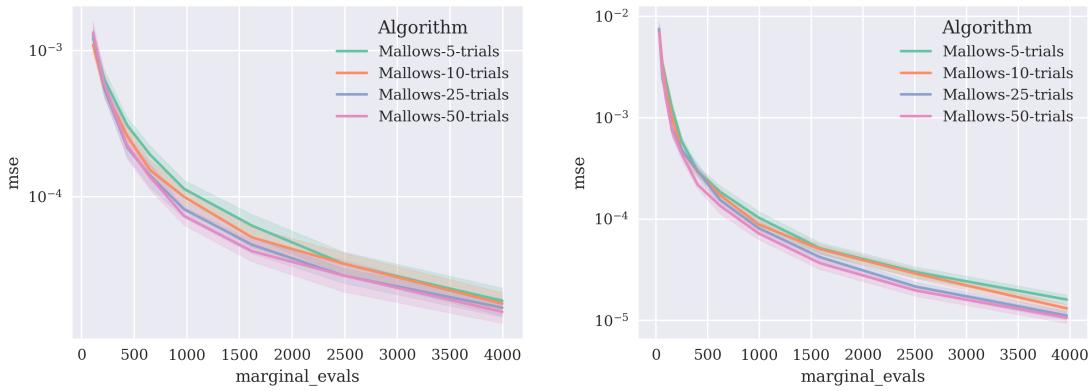
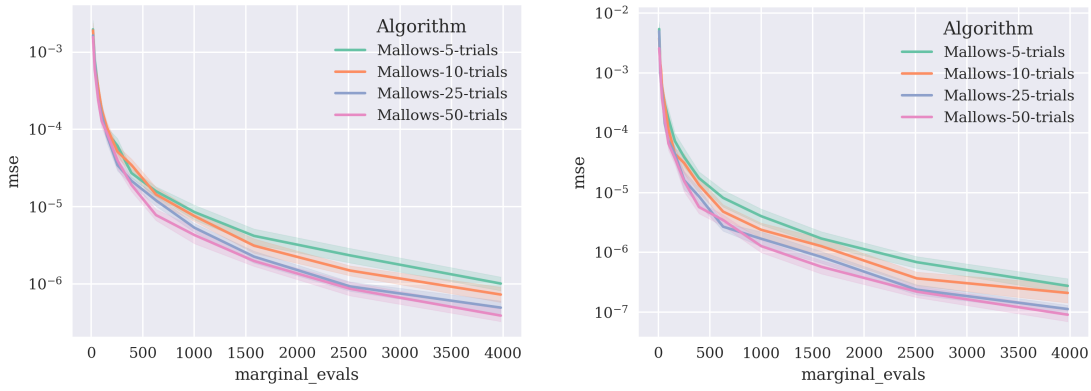


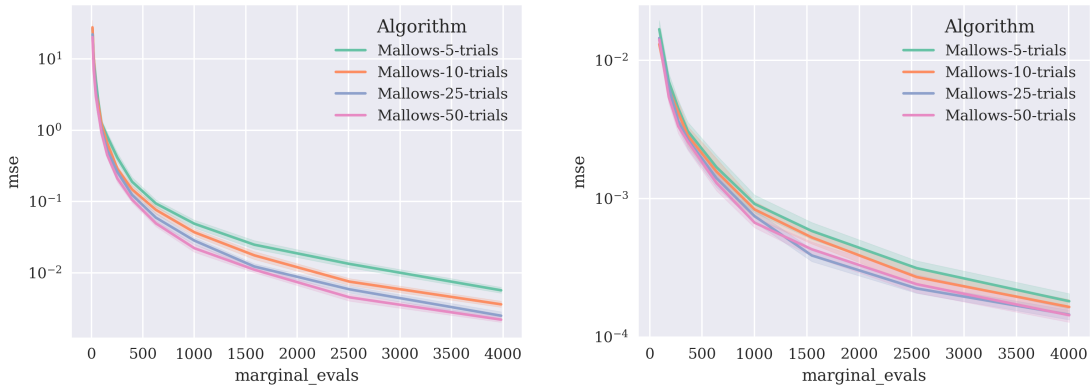
Figure 15: Varying λ for 100 herding samples - Image data and ResNet50 model



(a) *adult* (b) *breast_cancer*

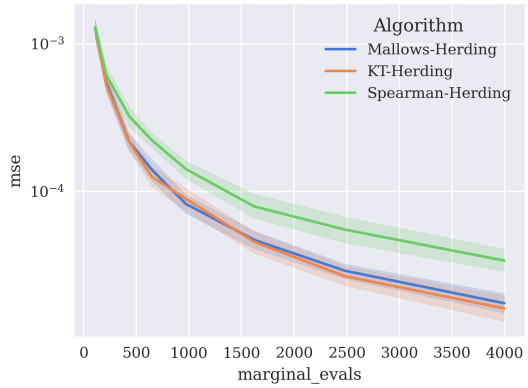


(c) *bank* (d) *cal_housing*

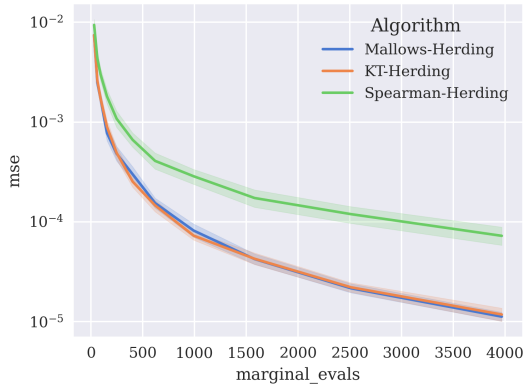


(e) *make_regression* (f) *year*

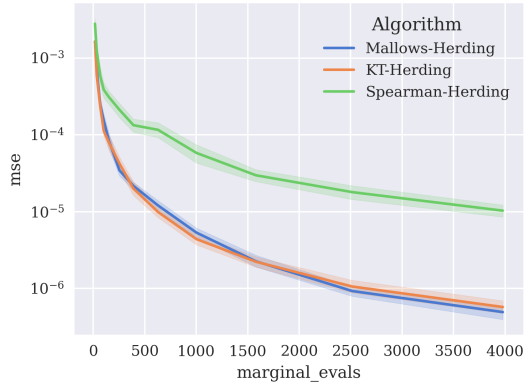
Figure 16: Varying argmax samples for herding algorithm ($\lambda = 4$) - Tabular datasets and GBDT models.



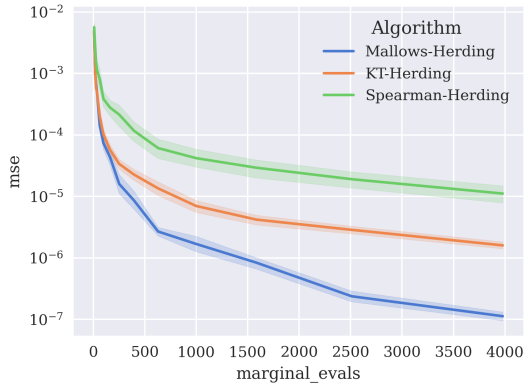
(a) *adult*



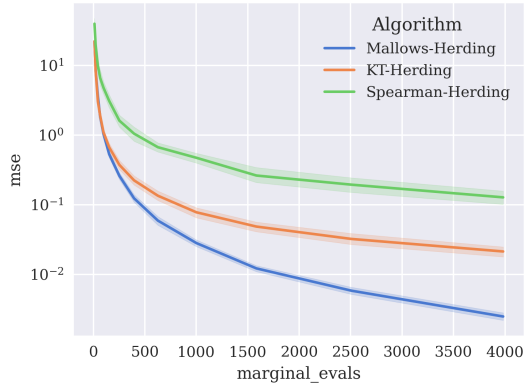
(b) *breast_cancer*



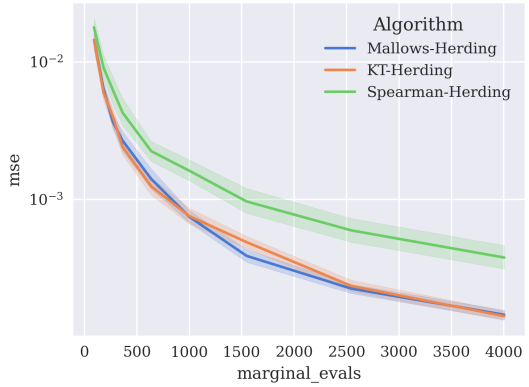
(c) *bank*



(d) *cal_housing*



(e) *make_regression*



(f) *year*

Figure 17: Comparing permutation kernels for kernel herding using tabular data and GBDT models.

Chapter 6

Bandwidth-Optimal Random Shuffling for GPUs

Bandwidth-Optimal Random Shuffling for GPUs

RORY MITCHELL*, Nvidia, USA and Waikato University, New Zealand
DANIEL STOKES*, Waikato University, New Zealand and Nyriad Ltd., New Zealand
EIBE FRANK and GEOFFREY HOLMES, Waikato University, New Zealand

Linear-time algorithms that are traditionally used to shuffle data on CPUs, such as the method of Fisher-Yates, are not well suited to implementation on GPUs due to inherent sequential dependencies, and existing parallel shuffling algorithms are unsuitable for GPU architectures because they incur a large number of read/write operations to high latency global memory. To address this, we provide a method of generating pseudo-random permutations in parallel by fusing suitable pseudo-random bijective functions with stream compaction operations. Our algorithm, termed 'bijective shuffle' trades increased per-thread arithmetic operations for reduced global memory transactions. It is work-efficient, deterministic, and only requires a single global memory read and write per shuffle input, thus maximising use of global memory bandwidth. To empirically demonstrate the correctness of the algorithm, we develop a statistical test for the quality of pseudo-random permutations based on kernel space embeddings. Experimental results show that the bijective shuffle algorithm outperforms competing algorithms on GPUs, showing improvements of between one and two orders of magnitude and approaching peak device bandwidth.

CCS Concepts: • **Computing methodologies → Massively parallel algorithms**.

Additional Key Words and Phrases: shuffling, GPU

ACM Reference Format:
Rory Mitchell, Daniel Stokes, Eibe Frank, and Geoffrey Holmes. 2018. Bandwidth-Optimal Random Shuffling for GPUs. 1, 1 (October 2018), 21 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Shuffling is a fundamental computer science problem, the objective of which is to rearrange a set of input elements into some pseudo-random order. The classical method of Fisher-Yates [19] (popularised by Knuth in [34]) randomly removes elements from an input buffer, one at a time, appending them to the output buffer. This algorithm runs optimally in $O(n)$ time, outputs uniformly distributed permutations, has a simple in-place variant, and is straightforward to implement. As such, sequential shuffling has long been considered a solved problem. However, modern computing hardware such as GPUs offer massively parallel computation that cannot be effectively exploited by the standard sequential shuffling approach due to dependencies inherent in Fisher-Yates type algorithms. The work presented in this paper was motivated by a gap in GPU-oriented parallel primitive libraries, such as Thrust [6], Cub [41], and Boost.Compute [62], which aim

*Both authors contributed equally to this research.

Authors' addresses: Rory Mitchell, ramitchellnz@gmail.com, Nvidia, 2788 San Tomas Expressway, Santa Clara, CA, USA, 95051 and Waikato University, Hillcrest Road, Hamilton, New Zealand, 3240; Daniel Stokes, Waikato University, Hillcrest Road, Hamilton, New Zealand, 3240 and Nyriad Ltd., New Zealand; Eibe Frank; Geoffrey Holmes, Waikato University, Hillcrest Road, Hamilton, New Zealand, 3240.

to implement the C++ standard library in equivalent form for GPUs, that prior to the method presented in this paper all lacked shuffling algorithms. The primary contributions of this paper are:

- 'Bijective shuffle': A new shuffling algorithm, carefully optimised for GPUs.
- A novel statistical test for shuffling algorithms, based on kernel space embeddings.

Some applications motivating this work are: permutation Monte Carlo tests [21], bootstrap sampling [45], approximation of Shapley values for feature attribution [44], differentially private variational inference [50], and shuffle operators in GPU accelerated dataframe libraries [63]. More generally, whenever shuffling forms part of a high-performance GPU-based algorithm, performance is lost when data is copied back to the host system, shuffled using the CPU, and copied back to the GPU. Host-to-device copies are limited by PCIE bandwidth, which is typically an order of magnitude or more smaller than available device bandwidth [16].

Informally, an ideal parallel algorithm would be capable of assigning elements of the input sequence randomly to unique output locations without contention, communicate minimally between processors, evenly distribute work between processors, use minimal working space, and have deterministic run-time. To achieve this goal, we utilise pseudo-random bijective functions, defining mappings between permutations. These bijective functions allow independent processors to write permuted elements to an output buffer, in parallel, without collision. Critically, we allow bijective functions defining permutations of larger sets to be applied to smaller shuffling inputs via a parallel compaction operation that preserves the pseudo-random property. The result is an $O(n)$ work algorithm for generating uniformly random permutations. We perform thorough experiments to validate the empirical performance and correctness of our algorithm. To this end, we also develop a novel test on distributions of permutations via unique kernel space embeddings.

We begin in Section 2 by introducing notation, summarising existing parallel shuffling algorithms, and describing GPUs. Section 3 introduces the idea of bijective functions, explaining their connection to shuffling and proposing two candidate bijective functions. Section 4 explains how compaction operations can be combined with bijective functions to generate bijections for arbitrary length sequences, forming the 'bijective shuffle' algorithm. Section 5 describes a statistical test comparing the kernel space embedding of a shuffling algorithm's output with its expected value. This test is used to evaluate the quality of the proposed shuffling algorithms and to select parameters. Finally, in Section 6, we evaluate the runtime and throughput of the proposed GPU shuffling algorithm.

2 BACKGROUND

Before discussing existing work, we briefly describe some notation. The order of elements in the shuffled version of an input sequence can be defined as a permutation. We refer to the symmetric group of permutations of n elements as $\mathfrak{S}_n$. The permutation $\sigma \in \mathfrak{S}_n$ assigns rank j to element i by $\sigma(i) = j$. For example, given the permutation written in one-line notation:

$$\sigma = \begin{pmatrix} 0 & 3 & 1 & 2 \end{pmatrix}$$

and the list of items

$$(x_0, x_1, x_2, x_3)$$

the items are reordered such that x_i occupies the $\sigma(i)$ coordinate

$$(x_0, x_2, x_3, x_1).$$

Note that we are using non-traditional zero-based indexing of permutations for notational convenience: this simplifies description of the functions used for our shuffling approach later in this paper.

The product of permutations $\sigma\tau$ refers to the composition operator, where the element i is assigned rank $\sigma(\tau(i)) = j$.

Shuffling is a reordering of the elements of an array of length n by a random permutation on the finite symmetric group, $\sigma \in \mathfrak{S}_n$, such that each possible reordering is equally likely, i.e., $p(\sigma) = \frac{1}{n!}, \forall \sigma \in \mathfrak{S}_n$.

2.1 Existing Work on Parallel Shuffling

We analyse the computational complexity of shuffling algorithms in terms of the work-time framework [31], where an algorithm is described in terms of a number of rounds, where p processors perform operations in parallel. GPUs are assumed to be *data parallel* processors such that there is an independent processor for each data element. Algorithms are evaluated based on the notion of work complexity, or the total number of operations performed on all processors. As the Fisher-Yates shuffle runs in optimal $O(n)$ work on a single processor, a parallel algorithm is said to be *work efficient* if it achieves $O(n)$ work complexity.

A parallel divide-and-conquer shuffling algorithm is independently proposed by Rao [53] and Sandelius [57] (we refer to this algorithm as RS). The input is recursively divided into subarrays by selecting a random number $0 \cdots k-1$ until each subarray contains one element. The final permutation is obtained via an in-order traversal of all subarrays. Bacher et al. [4] provide a variant of this algorithm named *MergeShuffle*, taking a bottom-up approach where the input is partitioned into k subarrays that are randomly merged with their neighbours until the entire array is shuffled. The parallel work complexity of these recursive divide-and-conquer approaches is $O(n \log(n))$. The *RS* and *MergeShuffle* algorithms have strong parallels to integer sorting algorithms. In particular, *RS* is equivalent to a most significant digit (MSD) radix sort [5] on a sequence, where each element in the sequence is an infinite length random bit stream. Likewise, *MergeShuffle* has strong parallels to merge sort, differing in the definition of the merge operator for each pass. We provide experiments in Section 6 for divide-and-conquer style shuffling algorithms using GPU sorting.

Anderson [2] proves that each of the swaps in the Fisher-Yates algorithm can be reordered without biasing the generated permutation, assuming parallel processors implement an atomic swap operation and that atomic swaps are serialised fairly. Note that current GPU architectures do not serialise atomic swaps fairly, as no assumptions can be made as to the ordering of operations [47], so such an approach would be unsuitable for GPUs. Shun et al. [60] take a similar approach, proving that the Fisher-Yates algorithm has an execution dependence graph with a depth of $O(\log(n))$ with high probability, allowing parallel execution of the swaps, while generating the same output as the sequential approach.

The literature also has divide-and-conquer shuffling algorithms that focus on distributed environments. Sanders [58] describes an algorithm for external memory and distributed environments where each item is allocated to a separate processor and permuted locally. A prefix sum is then used to place each element in the final output buffer. Langr et al. [35] provide a concrete extension of Sander's algorithm for use with the MPI library. Gustedt [25] also expands on this approach, describing a method of constructing a random communication matrix that ensures work is distributed fairly between all processors.

Reif [54] describes an algorithm that assigns each element a random integer key between 1 and p, where p represents the number of processors. Elements are then assigned to processors by a sorting operation, where they are shuffled by the sequential Fisher-Yates method. The work complexity of the integer sort can be achieved in $O(\log(n))$ time using $n/\log(n)$ processors using the sort algorithm described by Reif [54]. A limitation of this approach is non-deterministic

load balancing. One processor may receive significantly more work than others, limiting the algorithm to the speed of the slowest processor.

Alonso and Schott [1] define a custom representation with an associated total ordering that can be 'sorted' using a variant of merge-sort to produce a random permutation, falling into the class of divide-and-conquer algorithms. Their method forms a bijection between a so-called 'lower-exceeding sequence' of length n and a permutation of n elements. It can be shown that there are exactly $n!$ such sequences of length n. Given a uniformly selected random lower-exceeding sequence, the algorithm outputs the corresponding random permutation.

Czumaj et al. [14] make use of network simulation to define a shuffling algorithm. Their work describes two methods for randomly generating a network mapping input elements to a random output location. The first constructs a network representing a random Fisher-Yates shuffle. The network is processed to produce n distinct keys. After sorting these keys, elements are efficiently mapped to output locations. All pre/post-processing steps require $O(\log(n))$ time using $n/\log(n)$ processors in linear space. The second method proposed by Czumaj et al. [14], composes n-way 'splitters' to construct a network randomly permuting the input array. Each splitter randomly divides the input into two equal sized groups. For each splitter, the first group is ordered before the second in the final output. Each group is recursively split until the size of each group is one. At this stage, the network defines a random permutation, and is traversed to find the output position of each element. The algorithm runs in $O(c \log\log(n))$ time using $n^{1+1/c\log\log(n)}$ processors, for an arbitrary positive constant c. Granboulan and Pornin [22] use this method to generate a bijective function over an arbitrary input domain using $O(\log(n))$ space and $O(\log(n))$ time, but find the cost of the hyper-geometric random number generator used to construct splitters prohibitive.

'Dart-throwing' algorithms place input items randomly in an array of size $O(n)$ until each item is placed in a unique location. A compaction operation is applied to place items in a dense output array. When a placement collides with an occupied space, the process is retried until the element is placed successfully. Reif and Miller [43] and later Reif and Rajasekaran [52] describe a simple method in which a parallel prefix sum is used to perform the compaction. Matias and Vishkin [39] describe a method using the canonical cycle representation to perform the compaction step, and Hagerup [26] provides an alternative method for computing the min prefixes used when generating the cycle representation. Dart-throwing approaches achieve parallel work complexity of $O(n)$ in expectation, however, non-determinism makes them unsuitable for many practical applications. In our work, we similarly utilise prefix sum to perform compaction, but show how to obtain permutations in a completely deterministic way, using pseudo-random bijective functions in place of dart-throwing.

Cong and Bader [13] compare four divide-and-conquer, integer sorting, and dart-throwing algorithms and evaluate their performance relative to a sequential Fisher-Yates shuffle on up to 12 processors. They find that the sorting based approach is substantially worse than the other approaches, with Anderson's [2] approach tending to perform the best.

Closest to our work is the linear congruential generator (LCG) based approach of Andrés and Pérez [3], where permutations are generated by an LCG of full period. This approach is limited by the need to find viable LCG parameterisations for arbitrary length inputs, where the subset of available parameters is dramatically smaller than the space of permutations. In this paper we describe how any bijective function can be applied to the process of generating random permutations; in particular, we look at a more flexible use of LCGs and the use of n-bit block ciphers with much stronger pseudo-random properties.

The above algorithms all suffer from one or more drawbacks with respect to an ideal GPU implementation. The divide-and-conquer approaches have sub-optimal $O(n \log n)$ work complexity. The Rao-Sandelius [53] additionally suffers from load balancing issues, where the partitions generated by the divide-and-conquer process can be unevenly sized. The

parallel Fisher-Yates algorithm of [2] is non-deterministic and relies on fairly serialised atomics for correctness (GPU atomics are not fairly serialised). The network simulation algorithm of [14] has a complicated implementation difficult to adapt to GPUs. Dart-throwing algorithms are non-deterministic both in terms of shuffle output and runtime. The simple LCG approach of [3] offers no solution for arbitrary length sequences and suffers from poor quality pseudorandom outputs. In subsequent sections, we develop a new approach addressing these issues.

2.2 Graphics Processing Units

GPUs are massively parallel processors optimised for throughput, in contrast to conventional CPUs, which optimise for latency. GPUs in use today consist of processing units with single-instruction, multiple-thread (SIMT) lanes that efficiently execute instructions for groups of threads operating in lockstep. In the CUDA programming model, execution units called "streaming multiprocessors" (SMs), have 32 SIMT lanes. The corresponding group of 32 threads is called a "warp". Warps are generally executed on SMs without order guarantees, enabling latency of global memory loads to be hidden by switching between warps [47].

Large speed-ups in the domain of GPU computing commonly occur when problems are expressed as a balanced set of vector operations with minimal control flow, and coalesced memory access patterns. Notable examples are matrix multiplication [10, 17, 27], image processing [9, 46], deep neural networks [11, 12, 48], and sorting [23, 59].

Implementation of shuffling algorithms poses a particular challenge for GPUs. The SIMD architecture favours a data-parallel approach, where work is evenly distributed among threads belonging to an execution unit — if any one thread is slow to complete, the entire execution unit is stalled. This rules out approaches from Section 2.1 with uneven work distribution per processor. Dart-throwing algorithms may be implemented for GPUs, using atomic compare and swap operators available in CUDA, although collisions for atomic compare and swap operations imply global synchronisation and are extremely costly. Of the existing approaches, sort-based approaches appear the most promising, given the availability of state-of-the-art GPU sorting. Highly optimised algorithms exist for both merge-sort and LSB radix sort via the Thrust [6] library. As mentioned in Section 2.1, the problem of shuffling can be expressed as a sort over a list whose elements are infinite length random bit strings. In fact, the keys do not need to be infinite, only long enough to break any ties in comparisons between elements. One baseline GPU algorithm we consider is to generate random integer sort keys of machine word length (assumed to be 64 bits) and perform a key-value sort. GPU sorting algorithms such as merge-sort and LSB radix sort perform several passes ($O(\log n)$ for merge-sort and $O(k)$ for radix sort, with k proportional to the sort key size in bits) scattering elements in memory. These scatter passes are particularly expensive for random keys, as memory writes cannot be coalesced together efficiently, and represent the largest performance bottleneck for this shuffling algorithm.

While the above algorithms are candidates for implementation on GPUs, we may improve shuffle throughput and device utilisation significantly by devising a new algorithm tailored to the architecture. As the shuffle operation must at minimum reorder elements using a gather or scatter operation, we posit that the maximum bandwidth of any random shuffle algorithm implemented for GPUs is that of a random gather or scatter. Figure 1 demonstrates a parallel random gather operation, where threads read from noncontiguous memory locations in the input buffer and write to contiguous memory locations in the output buffer. A scatter operation is the inverse, where threads read from contiguous memory locations and write to noncontiguous memory locations. Scatter/gather operations for GPUs are discussed in detail in [29]. As our experiments show that gather operations have a higher bandwidth than scatter operations, we focus on the former.

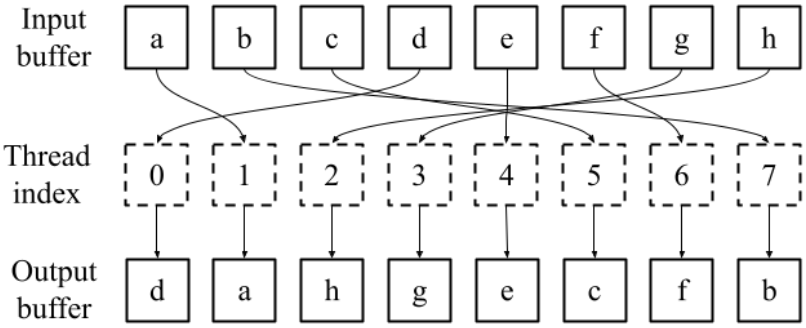


Fig. 1. Parallel random gather — Threads read from non-contiguous memory locations and write to contiguous memory locations.

Having established an upper bound on GPU bandwidth, we develop an optimised shuffling algorithm operating in a single gather pass, to approach the theoretical peak performance of the device.

3 SHUFFLING WITH BIJECTIVE FUNCTIONS

Our proposed approach is based on applying bijective functions. The symmetric group $\mathfrak{S}_n$, defined over any set of n distinct items, contains all bijections of the set onto itself, or all distinct permutations of n elements. A bijection is a one-to-one map between the elements of two sets, where each element from the first set is uniquely paired with an element from the second set. A bijection between two sets X, Y with $|X|, |Y| = n$ is characterised by some function $f_n : X \rightarrow Y$. The bijective function f_n admits a corresponding inverse $f_n^{-1} : Y \rightarrow X$, which is also a bijection. For any two such functions (f_n, g_n), their composition $f_n \circ g_n$ is also a bijection. As we are dealing with bijections from the symmetric group $\mathfrak{S}_n$, we define functions of the form $f_n : X \rightarrow X$. Without loss of generality, consider the set of nonnegative integers $X = \{0, 1, \cdots, n-1\}$. An example of a bijection $\sigma \in \mathfrak{S}_4$ is

$$f_4(0) = 2$$
$$f_4(1) = 3$$
$$f_4(2) = 1$$
$$f_4(3) = 0$$

If each of the n functions can be executed on a processor p_i, independently of any other processor, without communication and in reasonable time, then this defines a simple parallel algorithm for reordering elements. If f_n furthermore exhibit suitable pseudo-random properties such that $p(\sigma) = \frac{1}{n!}, \forall \sigma \in \mathfrak{S}_n$ (or close enough for practical application), then we have an effective parallel shuffling algorithm. We now discuss potential candidate bijective functions.

3.1 Linear Congruential

A first candidate is based on the common linear congruential random number generator (LCG). Given constants a, c, and n, the LCG outputs

$$y = (ax) + c \bmod n \tag{1}$$

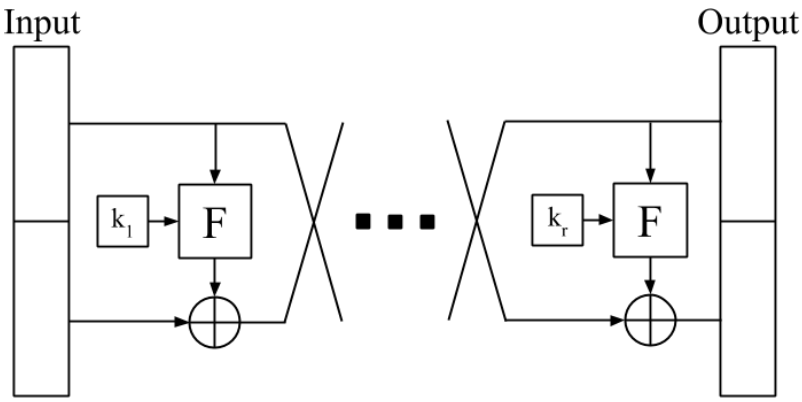


Fig. 2. Feistel Network construction — Successive rounds of the function F are applied to the input, each time with a unique key.

If a and n are co-prime, it is well-known that the input $x \in 0, 1, \cdots, n-1$ maps to a unique output location in the integer ring $\mathbb{Z}/n\mathbb{Z}$. Therefore, Equation 1 defines a bijective function and may be used to create permutations over inputs of length n. For now, assume n is fixed at the length of the input sequence. To find a bijection for a fixed n, the task reduces to finding some a co-prime to n (c is unrestricted). Finding a is trivial if n has certain properties, for example, any $a < n$ where n is prime, or odd $a < n$ where n is a power of two. We will later show how to modify the length of the input sequence such that co-prime a is always easily available.

The method described above is simple to implement and computationally inexpensive; however, linear congruential generators are known to have weaknesses as random number generators [51]. For our use case in particular, assuming some n fixed relative to input length and varying a and c from the above, we can achieve at most n^2 unique permutations — significantly less than the $n!$ possible permutations. Hence, permutations generated by this method may be appropriate for basic applications, but a more robust permutation generator is desirable.

3.2 Feistel Network

A second candidate bijective function is a block cipher construction known as a Feistel network [18]. A cryptographic block cipher, accepting an encryption key and a b-bit plain-text block, provides a bijective mapping to a b-bit cipher-text block. A perfect block cipher outputs cipher-text computationally indistinguishable from a random bit string, when the key is unknown. Thus, for a random choice of key, the output should be computationally indistinguishable from a random permutation. The Feistel network construction is a core component of many modern encryption algorithms, such as the Data Encryption Standard (DES) and so represents a significant improvement over the LCG in terms of the quality of its pseudo-random outputs [7].

The construction for a b-bit Feistel network consists of multiple applications of a round function as shown in Figure 2. In round i, the input is split into a $\lfloor b/2 \rfloor$-bit binary string L, representing the left half of the input block and a $\lceil b/2 \rceil$-bit binary string R representing the right half of the input block. A round function F is applied to R and a round key k_i. F does not need to be bijective. The first $\lfloor b/2 \rfloor$-bit output from F is combined with L using the exclusive-or operation to give the right half for round $i + 1$. The right half from round i becomes the left half for round $i + 1$. Thus, the ith round of the Feistel Network is defined as

$$f_i(L|R) = R|(L \oplus F(R, k_i)), \tag{2}$$

with $\oplus$ the bitwise exclusive-or operation. Consequently, a block cipher constructed using a Feistel Network construction with r rounds can be defined as

$$g(L|R) = f_r \circ \cdots f_2 \circ f_1(L|R) \tag{3}$$

Luby and Rackoff [37] proved that a three-round Feistel network is a pseudo-random permutation if the round function F is a pseudo-random function. Therefore, a Feistel Network with $rounds \geq 3$ can be used to generate a pseudo-random permutation for any set of size $n = 2^b$.

Feistel networks have been shown to be effective for parallel random number generation. Salmon *et al.* [56] describe two hardware-efficient pseudo-random number generators (PRNGs), named Philox and Threefry, based on simplifications of cryptographic block ciphers. These PRNGs are sometimes called *counter-based*, as generation of the i'th random number x_i is stateless, requiring only the index i and a random seed. Advancing the series does not require x_{i-1}, and so may be performed trivially by parallel threads. This is in contrast to the canonical Mersenne twister [40], which requires a state of 2.5kB and cannot advance the series arbitrarily from x_i to x_{i+m} in constant time with respect to m. The Philox and Threefry generators are shown to produce at least 2^{64} independent streams of random numbers, with a period of 2^{128} or more, and pass BigCrush [36] statistical tests. Figure 3 shows one step of the Philox PRNG, which differs from the standard Feistel network in the addition of the function B_k, which is strictly a bijection. Philox makes use of fast integer multiplication instructions, where multiplication by a carefully chosen, odd constant, yields upper bits forming F and lower bits modulo 2^w form the bijection B. B is guaranteed to be a bijection because the odd multiplicand is always coprime to the integer ring modulo 2^w.

Although these PRNGs happen to have the bijective property, the connection between bijections and the symmetric group $\mathfrak{S}_n$ is not exploited in [56]. In this paper, we adapt the Philox cipher to generate bijections with lengths of arbitrary powers of two instead of 2^{64}, while preserving the invertibility of the process, terming our modification *VariablePhilox*. For example, shuffling a sequence of size 2^7 results in halves with sizes $|L| = 3, |R| = 4$ and the bijective property is lost for the standard Philox cipher. Figure 4 shows a modified construction, where b is the odd bit and $|L| = |R|$. F_k is a pseudo-random key-dependent function, and B_k is a key-dependent bijective function. G is a function mixing bit b into R', where for $G(B_k(L), b) = (R', b')$, b is inserted in the least significant position of $B_k(L)$, yielding R'. Then the most significant bit of R' is removed and returned as b'. G is clearly invertible such that $G^{-1}(R', b') = (B_k(L), b)$. Defining $\oplus$ as bitwise exclusive-or, *VariablePhilox* encodes inputs as

$$\begin{aligned} L' &= F_k(L) \oplus R \\ (R', b') &= G(B_k(L), b) \end{aligned}$$

This process is inverted to retrieve (L, R, b) by computing (in order):

$$\begin{aligned} (B_k(L), b) &= G^{-1}(R', b') \\ L &= B_k^{-1}(B_k(L)) \\ R &= F_k(L) \oplus L' \end{aligned}$$

Invertibility guarantees the bijective property, so *VariablePhilox* is a bijection for arbitrary power-of-two length sequences. C++ reference code is given in Listing 1.

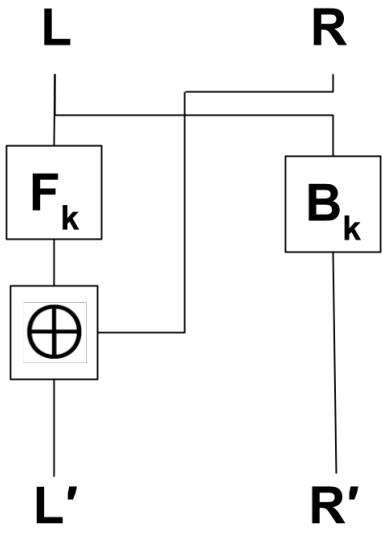


Fig. 3. Philox cipher — Bijective function B_k is added to the standard Feistel cipher

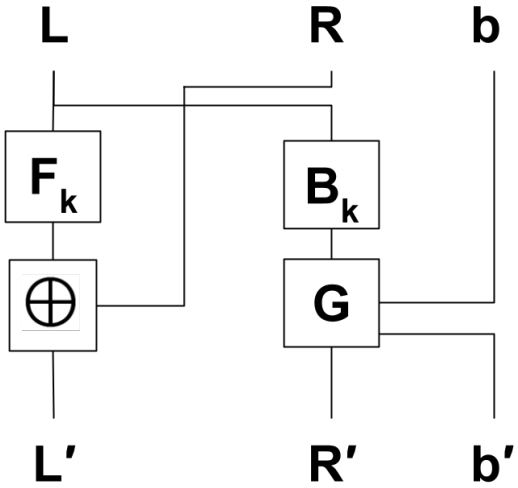


Fig. 4. VariablePhilox cipher — the odd bit b (if it exists) is mixed into R' by function G.

We have now described several methods of generating bijections with pseudo-random properties, but these bijections do not apply to arbitrary input lengths. In particular, the Feistel bijective functions are available for sequences of power of two length. We now show how to efficiently extend these to arbitrarily sequences.

4 ARBITRARY LENGTH BIJECTIONS

The key observation for generating bijections of arbitrary length is this: given an input vector $X = [0, 1, 2, \cdots, m-1]$ of length m, there may not be a readily available bijective function f_m of the same length, however, we can find the nearest applicable f_n such that $m \leq n$ and apply the bijective function to the padded vector $\hat{X} = [0, 1, 2, \cdots, n-1]$. The output is a vector W of length n, containing the randomly permuted input indices. By 'deleting' all $w \geq m$ from W, we obtain a permutation Y of length m. Using this process, we form an algorithm generating longer permutations of length $n \geq m$, then compacting them to form permutations of length m. This compaction of a longer random permutation is still an unbiased random permutation as per the following proposition.

Proposition 1. *Define the function $t : \mathfrak{S}_n \rightarrow \mathfrak{S}_m$ removing elements of $\sigma \in \mathfrak{S}_n$ where $\sigma(i) > m$, returning the permutation $\tau \in \mathfrak{S}_m$ of length m. If $p(\sigma) = \frac{1}{n!}, \forall \sigma \in \mathfrak{S}_n$, then $p(\tau) = \frac{1}{m!}$.*

Proof. The function $t(\sigma) = \tau$ is a surjective map from $\mathfrak{S}_m \rightarrow \mathfrak{S}_n$. For each $\tau \in \mathfrak{S}_n$, there exists a non-overlapping subset $\Pi_\tau \subseteq \mathfrak{S}_m$ such that $\forall \pi \in \Pi_\tau,\ t(\pi) = \tau$. Clearly $|\Pi_\tau| = \frac{m!}{n!}, \forall \tau \in \mathfrak{S}_n$ and so, if $p(\sigma) = \frac{1}{m!}, \forall \sigma \in \mathfrak{S}_m$, then $p(\tau) = \frac{1}{n!}$. □

To achieve the compaction in parallel, we use the work-efficient exclusive scan of Blelloch [8]. Flagging each element of the extended permutation W with 0 if $w_i \geq m$ or 1 if $w_i < m$, the output of the exclusive scan over these flags gives

```
uint64_t VariablePhilox(const uint64_t val) const
{
  static const uint64_t M0 = UINT64_C(0xD2B74407B1CE6E93);
  uint32_t state[2] = { uint32_t(val >> right_side_bits),
    uint32_t(val & right_side_mask)
  };
  for (int i = 0; i < num_rounds; i++)
  {
    uint32_t hi;
    uint32_t lo = mulhilo(M0, state[0], hi);
    lo = (lo << (right_side_bits - left_side_bits)) |
      state[1] >> left_side_bits;
    state[0] = ((hi ^ key[i]) ^ state[1]) & left_side_mask;
    state[1] = lo & right_side_mask;
  }
  // Combine the left and right sides together to get result
  return (uint64_t) state[0] << right_side_bits |
    (uint64_t) state[1];
}
```

Listing 1. VariablePhilox implementation — M0 is a constant selected in [56], mulhilo performs 64 bit integer multiplication, returning the result as the upper and lower 32 bits.

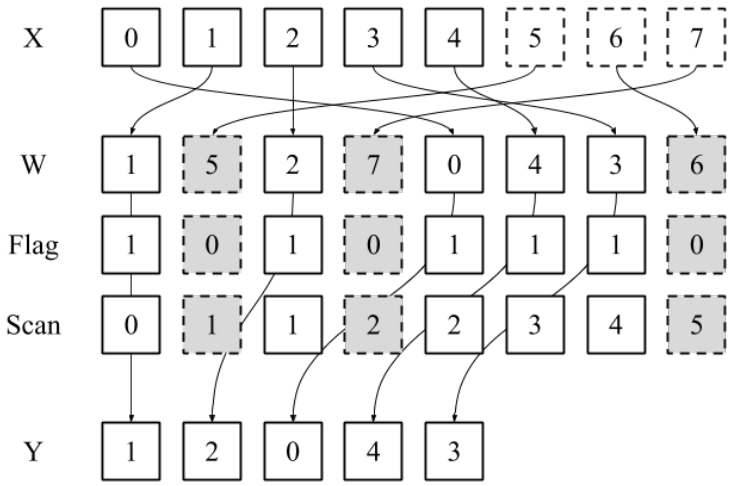


Fig. 5. Shuffle compaction — Dashed boxes represent dummy elements used to extend the sequence to a power of two. The bijective function is evaluated for each thread, yielding W. We flag any $W > 4$, then the scan of these flags provides the destination address for each $X[W_i]$, where $W_i \leq 4$.

Algorithm 1: Bijection Shuffle

input : m, n, X, f_n
output: Y
for $i = 0$ **to** $n - 1$ **in parallel do**
  `// Evaluate bijective function`
  $b = f_n(i)$;
  $flag = 1$;
  if $b \geq m$ **then**
    $flag = 0$;
  `// Find output location of valid elements`
  $out\_idx = exclusive\_scan(flag)$;
  `// Gather elements to output`
  if $b < m$ **then**
    $Y[out\_idx] = X[b]$;

the scatter indices into the shuffled output vector Y. This process is shown in Figure 5 for $m = 5, n = 8$, and pseudocode is given in Algorithm 1. Given $p = n$ independent threads, the exclusive scan operation has work complexity $O(n)$. Our bijective function, executed in parallel for each element, also has work complexity $O(n)$, and so the final algorithm has optimal work complexity $O(n)$. Note that, using the bijective functions described in Section 3, the padded array of length n will have at most $2m$ elements (the nearest power of two length), so the work complexity holds regardless of input length.

Algorithm 1 can be implemented easily in several steps using a GPU parallel primitives library, for example, evaluating the bijective function, gathering elements to an output buffer, then applying a stream compaction algorithm. Achieving "bandwidth optimality", i.e., a single global memory read and write per element, is more challenging, requiring the fusion of all operations into a single GPU kernel. The standard GPU parallel prefix sum of [28] uses two passes through global memory, the first to perform block-level scans and the second to propagate partial block-level results globally. This can be improved upon by using the specialised single-pass scan implementation of [42], using a technique termed *'decoupled look-back'* to achieve non-blocking communication between thread blocks, achieving optimal n global memory reads/writes. In Section 6, we evaluate three versions of our GPU shuffling algorithm, incorporating varying levels of kernel fusion and demonstrating the effectiveness of optimising global memory read/writes for the shuffling problem.

5 STATISTICAL TESTS

We now consider statistical tests for pseudo-random permutations to assess the suitability of the methods discussed in Section 3 and 4 for practical applications, and to select the number of rounds for the *VariablePhilox* cipher. Testing the distribution of permutations is challenging as the sampling space grows super-exponentially with the length of the permutation. In particular, $21! > 2^{64}$, so any computer algorithm for shuffling that uses a 64 bit integer seed cannot generate all permutations for $n \geq 21$.

Our first test considers the distribution of random permutations at $n = 5$, where $5! = 120$, using a standard χ^2 test with 119 degrees of freedom, under the null hypothesis that permutations are uniformly distributed, i.e., each permutation occurs with probability $p = \frac{1}{120}$. Figure 6 shows the change in the χ^2 statistic as the number of rounds in

the *VariablePhilox* cipher is increased. Each data point is computed from 100,000 random permutations. We also plot the acceptance thresholds for $\alpha = 0.05$, corresponding to the probability of observing a value of the χ^2 statistic above the line indicated on the figure, if the null hypothesis is true. Considering the results shown in the figure, it is unlikely that samples from *VariablePhilox* with less than 20 rounds are drawn from a uniform distribution. This is somewhat surprising given that only 10 rounds are recommended for the original Philox cipher in the PRNG setting [56]. The LCG bijective function generates a χ^2 statistic in the region of 500,000, so it clearly fails the test and is not included in this figure.

The χ^2 test is useful for small n, but is intractable otherwise. We develop another test statistic suited to larger n, based on the maximum mean discrepancy (MMD) in a reproducing kernel Hilbert space (RKHS). Two-sample hypothesis tests using MMD are developed in [24]. In a similar spirit, we derive a one sample test comparing the uniform distribution of permutations against a finite sample generated by a shuffling algorithm. The MMD^2 between two distributions $p(X)$ and $q(Y)$ in a RKHS $\mathcal{H}$, equipped with positive definite kernel K, is defined as

$$\mathrm{MMD}^2(p, q) = \mathbb{E}_{x,x'}[K(x, x')] - 2\mathbb{E}_{x,y}[K(x, y)] + \mathbb{E}_{y,y'}[K(y, y')]. \tag{4}$$

If the kernel K is a *characteristic kernel*, the mean embedding of a distribution induced by the kernel is injective [20]. In other words, the mean embedding of a distribution is unique to that distribution. As a consequence, $\mathrm{MMD}(p, q) = 0$ if and only if $p = q$. Thus, a strategy for statistical testing is to form the null hypothesis that $p = q$, compute the sample estimate $\hat{\mathrm{MMD}}^2(p, q)$ as the test statistic, and evaluate the probability of obtaining a sample estimate greater than some threshold, assuming $p = q$, using a concentration inequality. If the observed test statistic is sufficiently unlikely, this provides evidence that $p \neq q$.

To implement this idea, a characteristic kernel measuring the similarity of two permutations is needed. The Mallows kernel, for $\lambda \geq 0$, is defined for permutations as

$$K_M^\lambda(\sigma, \sigma') = e^{-\lambda n_{\mathrm{dis}}(\sigma,\sigma')/\binom{n}{2}}.$$

where

$$n_{\mathrm{dis}}(\sigma, \sigma') = \sum_{i<j} [\mathbb{1}_{\sigma(i)<\sigma(j)}\mathbb{1}_{\sigma'(i)>\sigma'(j)} + \mathbb{1}_{\sigma(i)>\sigma(j)}\mathbb{1}_{\sigma'(i)<\sigma'(j)}].$$

We (somewhat arbitrarily) use the parameter $\lambda = 5$ throughout this paper. The Mallows kernel is introduced in [32] and shown to be characteristic in [38]. It may be implemented in time $O(n \log n)$ using the procedure presented in [33]. In the following, we make use of the fact [44] that the expected value of the Mallows kernel under a uniform distribution of permutations is

$$\forall \sigma \in \mathfrak{S}_n, \quad \mathbb{E}_{\sigma'}[K(\sigma, \sigma')] = \prod_{j=1}^{n} \frac{1 - e^{-\lambda j/\binom{n}{2}}}{j(1 - e^{-\lambda/\binom{n}{2}})}.$$

The Mallows kernel is right-invariant in the sense that $K(\sigma, \sigma') = K(\tau\sigma, \tau\sigma')$ for any $\tau \in \mathfrak{S}_d$ [15]. Also note that the uniform distribution of permutations is invariant to composition. Using right invariance in conjunction with $\tau = \sigma^{-1}$, and assuming p denotes the uniform distribution over permutations, then

$$\begin{aligned}
\mathbb{E}_{\sigma'\sim p}[K(\sigma, \sigma')] &= \mathbb{E}_{\sigma'\sim p}[K(\tau\sigma, \tau\sigma')] \\
&= \mathbb{E}_{\sigma'\sim p}[K(I, \tau\sigma')] \\
&= \mathbb{E}_{\sigma'\sim p}[K(I, \sigma')]
\end{aligned}$$

with I the identity permutation. Using the above, and assuming that p is a uniform distribution of permutations, the MMD^2 from (4) is simplified as

$$\begin{aligned}\mathrm{MMD}^2(p,q) &= \mathbb{E}_x[K(I,x)] - 2\mathbb{E}_x[K(I,x)] + \mathbb{E}_{y,y'}[K(y,y')] \\ &= \mathbb{E}_{y,y'}[K(y,y')] - \mathbb{E}_x[K(I,x)] \\ &= \mathbb{E}_{y,y'}[K(y,y')] - \prod_{j=1}^{n} \frac{1-e^{-\lambda j/\binom{n}{2}}}{j(1-e^{-\lambda/\binom{n}{2}})}.\end{aligned}$$

Replacing q with a sample of random permutations Π, of size $|\Pi|$ and with $|\Pi| \equiv 0 \pmod 2$, we obtain an unbiased sample estimate via

$$\hat{\mathrm{MMD}}^2(p,\Pi) = \frac{2}{|\Pi|}\sum_{i}^{|\Pi|/2} K(\Pi_{2i-1},\Pi_{2i}) - \prod_{j=1}^{n} \frac{1-e^{-\lambda j/\binom{n}{2}}}{j(1-e^{-\lambda/\binom{n}{2}})}, \tag{5}$$

where $\mathbb{E}[\hat{\mathrm{MMD}}^2(p,\Pi)] = 0$ if and only if $p = q$. We derive two tests based on the $\hat{\mathrm{MMD}}^2(p,\Pi)$ statistic. The first is a distribution-free bound. Applying Hoeffding's inequality [30], with the fact that $0 \le K(\sigma,\sigma') \le 1$, we have

$$P(|\hat{\mathrm{MMD}}^2(p,\Pi)| \ge t) \le 2\exp(-|\Pi|t^2)$$

Let α_H be the level of significance under the Hoeffding bound. With null hypothesis $p = q$, the acceptance region of the test is

$$|\hat{\mathrm{MMD}}^2(p,\Pi)| < \sqrt{\frac{\log(2/\alpha_H)}{|\Pi|}}. \tag{6}$$

The second test uses the central limit theorem, implying that $\hat{\mathrm{MMD}}^2(p,\Pi)$ approaches a normal distribution as $|\Pi| \to \infty$. The variance of the Mallows kernel may be computed directly from its expectation as

$$\begin{aligned}\mathrm{Var}(K(\sigma,\sigma')) &= \mathbb{E}[K(\sigma,\sigma')^2] - \mathbb{E}[K(\sigma,\sigma')]^2 \\ &= \mathbb{E}\big[e^{-2\lambda n_{\mathrm{dis}}(\sigma,\sigma')/\binom{n}{2}}\big] - \mathbb{E}\big[e^{-\lambda n_{\mathrm{dis}}(\sigma,\sigma')/\binom{n}{2}}\big]^2 \\ &= \prod_{j=1}^{n} \frac{1-e^{-2\lambda j/\binom{n}{2}}}{j(1-e^{-2\lambda/\binom{n}{2}})} - \left(\prod_{j=1}^{n} \frac{1-e^{-\lambda j/\binom{n}{2}}}{j(1-e^{-\lambda/\binom{n}{2}})}\right)^2.\end{aligned}$$

So we have

$$\mathrm{Var}(\hat{\mathrm{MMD}}^2(p,\Pi)) = \frac{2\cdot\mathrm{Var}(K(\sigma,\sigma'))}{|\Pi|}.$$

The factor of two arises because the sum in (5) is of size $|\Pi|/2$. According to a normal distribution with mean 0 and variance as above,

$$P(|\hat{\mathrm{MMD}}^2(p,\Pi)| \ge t) = 1 - \mathrm{erf}\left(\frac{t}{\sqrt{2\mathrm{Var}(\hat{\mathrm{MMD}}^2)}}\right),$$

where erf is the canonical error function. The acceptance region for α_N is

$$|\hat{\mathrm{MMD}}^2(p,\Pi)| < \sqrt{2\mathrm{Var}(\hat{\mathrm{MMD}}^2)}\,\mathrm{erf}^{-1}(1-\alpha_N). \tag{7}$$

The threshold for α_H should be used for small $|\Pi|$ (i.e. < 100), as it makes no assumptions on the distribution of $\hat{\mathrm{MMD}}^2(p,\Pi)$. For larger sample sizes, the asymptotic α_N threshold is significantly tighter and should be preferred.

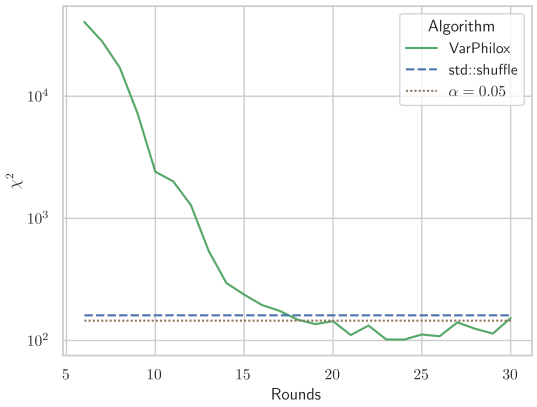


Fig. 6. χ^2 statistic vs. rounds, $|\Pi|$ =100,000

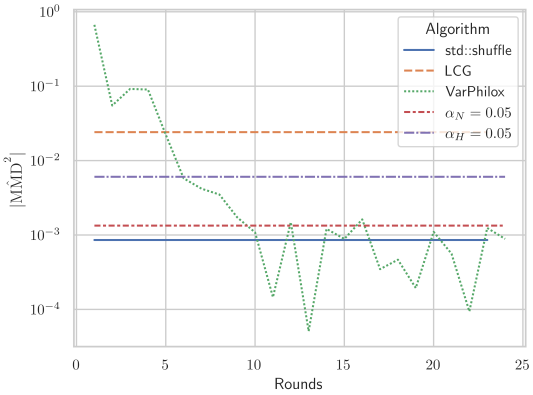


Fig. 7. $|\hat{\text{MMD}}^2(p, \Pi)|$ statistic, $n = 5$, $|\Pi|$ =100,000

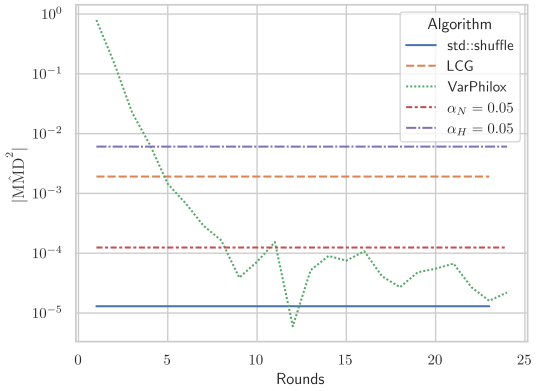


Fig. 8. $|\hat{\text{MMD}}^2(p, \Pi)|$ statistic, $n = 100$, $|\Pi|$ =100,000

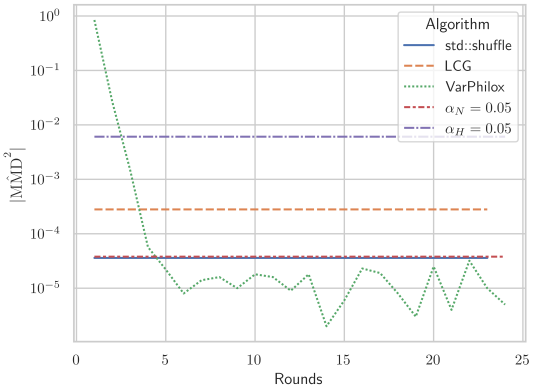


Fig. 9. $|\hat{\text{MMD}}^2(p, \Pi)|$ statistic, $n = 1000$, $|\Pi|$ =100,000

Comparing Figures 6 and 7, we see the MMD tests with the asymptotic acceptance threshold roughly coincide with the χ^2 test at n = 5, where some tests fail for VariablePhilox with fewer than 20 rounds, but VariablePhilox with 24 rounds or more passes all tests. Figures 7, 8, and 9 plot the $|\hat{\text{MMD}}^2(p, \Pi)|$ statistic for VariablePhilox, LCG, and std::shuffle using $|\Pi|$ =100,000, for varying permutation lengths. These experiments lead us to recommend a 24 round VariablePhilox cipher for random permutation generation, and we use this configuration in all subsequent experiments.

6 EVALUATION OF THROUGHPUT

We now evaluate the throughput of our bijective shuffling method, where throughput refers to (millions of keys)/time(s). Unless otherwise stated, experiments are performed on an array of 64-bit keys of length $2^w + 1$, where w ranges from 8 to 29. This represents the worst-case scenario, where our bijective shuffle algorithm must redundantly evaluate $2^w - 1$ elements. For all throughput results, we report the average of five trials.

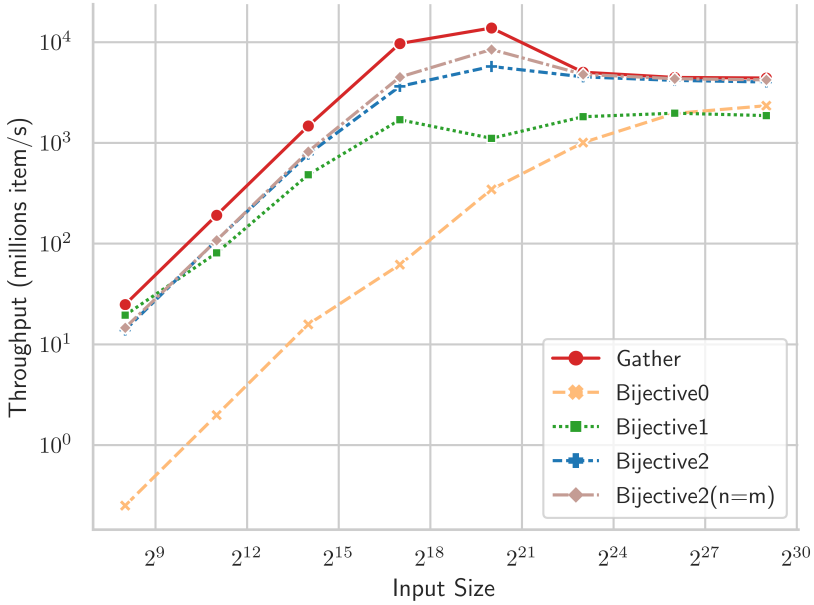


Fig. 10. GPU kernel fusion for bijective shuffle — increasing kernel fusion decreases memory transactions and approaches the gather throughput upper bound.

To consider the effect of code optimisations, we evaluate three CUDA implementations of Algorithm 1 with varying levels of GPU kernel fusion:

- *Bijective0*: The transformation ($b = f_n(i)$), stream compaction, and gather are implemented in separate passes.
- *Bijective1*: The transformation, stream compaction, and gather are fused into a single scan operation. The two-pass scan algorithm of [28] is used.
- *Bijective2*: The transformation, stream compaction, and gather are fused into a single scan operation. The single pass scan algorithm of [42] is used, for m total global memory reads/writes.

Figure 10 plots the throughput of these variants, on a Tesla V100-32GB GPU, using VariablePhilox as the bijective function. For reference, the line labeled "gather" shows an upper bound on throughput for n random gather operations in global memory. The optimised algorithm *Bijective2* closely matches the optimal throughput of random gather, and is equivalent in performance for sizes $> 2^{21}$. *Bijective2(n=m)* indicates the best-case performance of the algorithm, where the sequence is exactly a power of two length. The best-case and worst-case performance do not differ greatly because redundant elements do not incur global memory transactions, and are therefore relatively inexpensive to evaluate. The performance of random gather peaks around 2^{20}, where there is sufficient L2 cache to mitigate the effects of uncoalesced reads/writes. At this size, there is a gap between *Bijective2* and random gather due to arithmetic operations required in evaluating multiple rounds of the VariablePhilox function, and because the prefix-sum must be redundantly evaluated up to the nearest power of two (although no global memory transactions occur for these redundant elements). This gap disappears at larger sizes when the runtime of the kernel becomes dominated by memory operations.

Table 1. GPU shuffling throughput (millions items/s) - Tesla V100

Input size	Gather	VarPhilox	LCG	DartThrowing	SortShuffle
$2^8 + 1$	24.64	14.61	16.21	2.459	1.845
$2^{11} + 1$	188.8	115.4	124.8	15.83	8.191
$2^{14} + 1$	1471	810.6	851.4	97.3	26.17
$2^{17} + 1$	9696	3836	3968	145.	69.26
$2^{20} + 1$	13830	5800	5642	159.8	127.3
$2^{23} + 1$	5036	4527	4476	150.9	132.6
$2^{26} + 1$	4465	4172	4143	133.8	118.4
$2^{29} + 1$	4409	4018	4011	123.2	111.8

Table 2. GPU shuffling throughput (millions items/s) - GeForce RTX 2080

Input size	Gather	VarPhilox	LCG	DartThrowing	SortShuffle
$2^8 + 1$	35.63	13.73	14.19	2.98	2.16
$2^{11} + 1$	273.70	109.07	112.40	18.10	8.54
$2^{14} + 1$	2183.41	776.85	846.42	66.74	30.62
$2^{17} + 1$	12225.35	3649.39	3759.00	95.38	71.17
$2^{20} + 1$	4187.73	3641.61	3437.88	89.55	80.63
$2^{23} + 1$	2238.91	2232.61	2260.85	80.67	75.03
$2^{26} + 1$	2097.25	2096.47	2105.51	71.44	67.35

We now proceed to a comparison of different shuffling algorithms. Table 1 and Figure 11 show the throughput in millions of items per second, for the Tesla V100-32GB GPU, and Table 2 and Figure 12 show data for the GeForce RTX 2080 GPU. LCG and VariablePhilox implement optimised bijective shuffling using the methods described in Section 3. The dart-throwing algorithm uses CUDA atomic exchange instructions to attempt to place items randomly in a buffer of size $2n$, a stream compaction operation is then applied to the buffer to provide the final shuffled output. SortShuffle applies the state-of-the-art radix sort algorithm of [41] to randomly generated 64-bit keys. As discussed in Section 2.1, many existing work on shuffling reduce to sorting algorithms on infinite length keys. Thus, SortShuffle is representative of a wide class of divide-and-conquer algorithms.

The bijective shuffle algorithms with the VariablePhilox and LCG functions achieve near-optimal throughput at large sizes, where performance is dominated by global memory operations. Throughput for these methods is more than an order of magnitude higher than the DartThrowing or SortShuffle methods. The lesser throughput of DartThrowing can be explained by the overhead of generalised atomic instructions in the CUDA architecture, as well as contention among threads when multiple threads attempt to write to the same memory location. SortShuffle relies on radix sort, where prefix sum is applied to 4 bits in each pass, requiring 16 passes over the data to fully sort 64-bit keys. In comparison, bijective shuffle is designed to require only a single prefix sum operation. Bijective shuffle is also fully deterministic, unlike DartThrowing, and requires no global memory for working space, whereas DartThrowing and SortShuffle use memory proportional to $O(n)$.

We also evaluate the performance of our shuffling method using 2x Intel Xeon E5-2698 CPUs, with a total of 40 physical cores. Results are presented in Table 3 and Figure 13. The bijective shuffle method is not designed for CPU

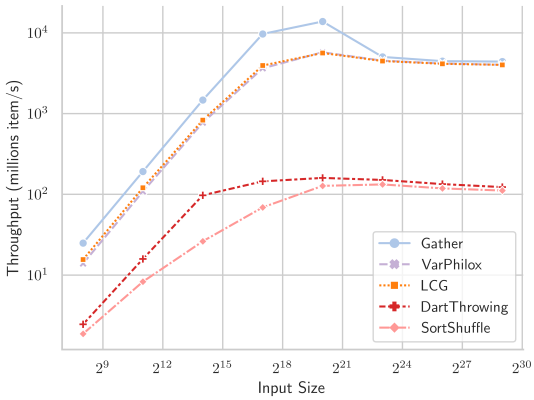


Fig. 11. GPU shuffling algorithms - Tesla V100

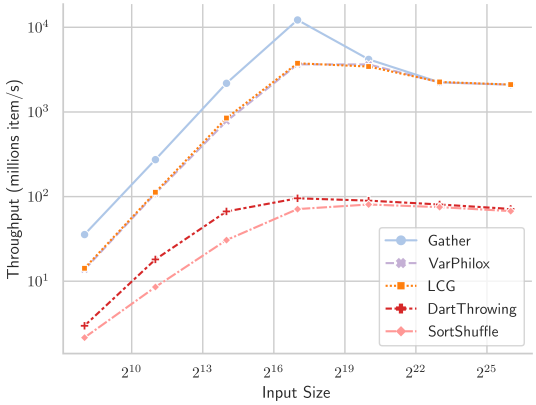


Fig. 12. GPU shuffling algorithms - GeForce RTX 2080

Table 3. CPU shuffling throughput (millions items/s) - 2x Xeon E5-2698

Input size	Gather	VarPhilox	DartThrowing	std::shuffle	RS	MergeShuffle	SortShuffle
$2^8 + 1$	128.03	2.81	0.09	40.77	41.18	46.24	2.45
$2^{11} + 1$	635.04	17.65	0.69	63.83	63.82	64.95	6.26
$2^{14} + 1$	935.55	89.88	4.76	68.05	68.00	67.85	8.30
$2^{17} + 1$	434.46	207.94	21.69	63.05	78.71	52.03	18.60
$2^{20} + 1$	353.72	246.79	39.53	59.38	99.76	32.94	79.30
$2^{23} + 1$	108.23	157.88	21.37	45.67	108.78	23.44	60.11
$2^{26} + 1$	64.32	49.02	16.04	24.93	111.15	18.38	47.45
$2^{29} + 1$	54.12	44.06	15.63	21.66	104.99	15.03	39.23

architectures, but the results are informative nonetheless. The Gather, VariablePhilox (bijective shuffle) and SortShuffle algorithms are implemented using the Intel Thread Building Blocks (TBB) library [55]. The Rao-Sandelius (RS) [53] and MergeShuffle [4] algorithms use the implementation of [4], and std::shuffle is the single-threaded C++ standard library implementation. VariablePhilox has the highest throughput for medium-sized inputs, although it is outperformed by RS when $n > 2^{24}$. Curiously, RS has higher throughput at large sizes than the TBB gather implementation. This could indicate that RS has a more cache-friendly implementation, despite having a worse computational complexity of $O(n \log n)$, or simply a suboptimal implementation in TBB. Further work is needed to properly investigate these effects for CPU architectures.

7 LIMITATIONS AND FUTURE RESEARCH

While the presented algorithm for GPUs is highly effective in terms of throughput and the quality of its pseudorandom outputs, its parameterisation differs from that of typical standard library implementations of shuffle. For example, the C++ standard library implementation accepts any 'uniform random bit generator' [61], such as the common Mersenne-twister generator [40]. Our implementation is instead parameterised by the selection of a bijective function, and a key of length $\log(n)k$ bits, where n is the length of the sequence to be shuffled, and k is the number of rounds in the cipher.

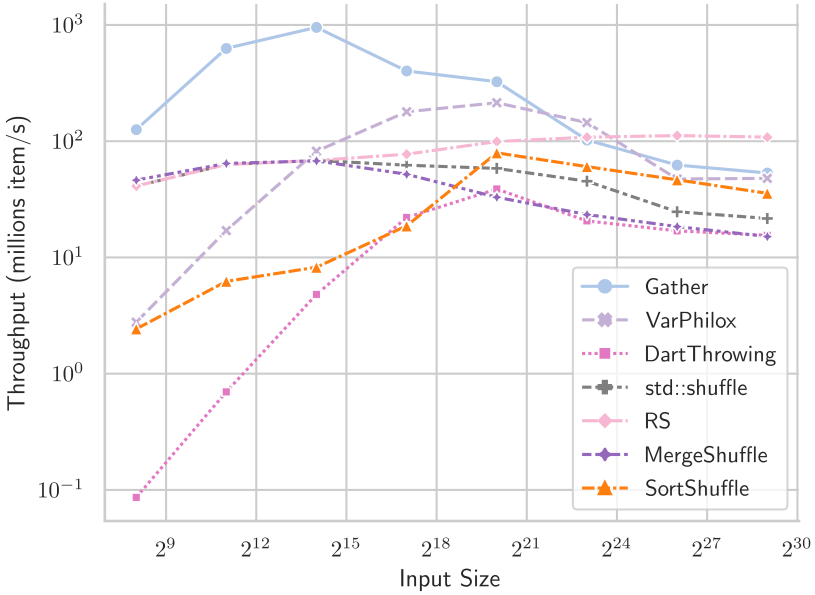


Fig. 13. CPU shuffling algorithms - 2x Intel Xeon E5-2698

Future work may explore other useful constructions of bijective functions, beyond the LCG and Feistel variants discussed in this paper.

It also should be noted that the Fisher-Yates and Rao-Sandelius algorithms may operate in-place, and our proposed GPU algorithm does not, requiring the allocation of an output buffer. This is not unexpected, as there are few truly in-place GPU algorithms that reorder inputs ([49] is a notable exception). For future work, it would be interesting to consider if an in-place parallel shuffling algorithm is possible for GPU architectures.

8 CONCLUSION

We provide an algorithm for random shuffling specifically optimised for GPU architectures, using modified bijective functions from cryptography. Our algorithm is highly practical, achieving performance approaching the maximum possible device throughput, while being fully deterministic, using no extra working space, and providing high-quality distributions of random permutations. An outcome of this work is also a statistical test for uniform distributions of permutations based on the Mallows kernel that we expect to be useful beyond shuffling algorithms.

REFERENCES

[1] Laurent Alonso and René Schott. 1996. A parallel algorithm for the generation of a permutation and applications. *Theoretical Computer Science* 159, 1 (1996), 15 – 28.

[2] R. Anderson. 1990. Parallel Algorithms for Generating Random Permutations on a Shared Memory Machine. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures* (Island of Crete, Greece) *(SPAA '90)*. Association for Computing Machinery, New York, NY, USA, 95–102.

[3] David Miraut Andrés and Luis Pastor Pérez. 2011. Efficient Parallel Random Rearrange. In *International Symposium on Distributed Computing and Artificial Intelligence*, Ajith Abraham, Juan M. Corchado, Sara Rodríguez González, and Juan F. De Paz Santana (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–190.

[4] Axel Bacher, Olivier Bodini, Alexandros Hollender, and Jérémie O. Lumbroso. 2015. MergeShuffle: A Very Fast, Parallel Random Permutation Algorithm. *CoRR* abs/1508.03167 (2015). arXiv:1508.03167 http://arxiv.org/abs/1508.03167

[5] Axel Bacher, Olivier Bodini, Hsien-Kuei Hwang, and Tsung-Hsi Tsai. 2017. Generating Random Permutations by Coin Tossing: Classical Algorithms, New Analysis, and Modern Implementation. *ACM Trans. Algorithms* 13, 2, Article 24 (Feb. 2017), 43 pages.

[6] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, 359–371.

[7] Alex Biryukov and Christophe De Cannière. 2005. *Data encryption standard (DES)*. Springer US, Boston, MA, 129–135.

[8] Guy E. Blelloch. 1990. *Prefix Sums and Their Applications*. Technical Report CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

[9] Bo Fang, Guobin Shen, Shipeng Li, and Huifang Chen. 2005. Techniques for efficient DCT/IDCT implementation on generic GPU. In *2005 IEEE International Symposium on Circuits and Systems*. 1126–1129 Vol. 2.

[10] Changhao Jiang and M. Snir. 2005. Automatic tuning matrix multiplication performance on graphics hardware. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 185–194.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. arXiv:1410.0759 [cs.NE]

[12] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. 2013. Deep Learning with COTS HPC Systems. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (Atlanta, GA, USA) *(ICML'13)*. JMLR.org, III–1337–III–1345.

[13] Guojing Cong and David A. Bader. 2005. An Empirical Analysis of Parallel Random Permutation Algorithms ON SMPs. In *Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems, September 12-14, 2005 Imperial Palace Hotel, Las Vegas, Nevada, USA*, Michael J. Oudshoorn and Sanguthevar Rajasekaran (Eds.). ISCA, 27–34.

[14] A. Czumaj, P. Kanarek, M. Kutylowski, and K. Lorys. 1998. Fast Generation of Random Permutations Via Networks Simulation. *Algorithmica* 21, 1 (1998), 2–20.

[15] Persi Diaconis. 1988. *Group representations in probability and statistics*. Institute of Mathematical Statistics, Hayward, CA. vi+198 pages.

[16] Rob Farber. 2011. *CUDA application design and development*. Elsevier.

[17] K. Fatahalian, J. Sugerman, and P. Hanrahan. 2004. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Grenoble, France) *(HWWS '04)*. Association for Computing Machinery, New York, NY, USA, 133–137.

[18] Horst Feistel. 1973. Cryptography and Computer Privacy. *Scientific American* 228, 5 (1973), 15–23.

[19] Ronald A Fisher and Frank Yates. 1943. *Statistical tables for biological, agricultural and medical research*. Oliver and Boyd Ltd, London.

[20] Kenji Fukumizu, Francis R. Bach, and Michael I. Jordan. 2004. Dimensionality Reduction for Supervised Learning with Reproducing Kernel Hilbert Spaces. *J. Mach. Learn. Res.* 5 (Dec. 2004), 73–99.

[21] Phillip I Good. 2006. *Permutation, parametric, and bootstrap tests of hypotheses*. Springer Science & Business Media.

[22] Louis Granboulan and Thomas Pornin. 2007. Perfect Block Ciphers with Small Blocks. In *Fast Software Encryption*, Alex Biryukov (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 452–465.

[23] Oded Green, Robert McColl, and David A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing* (San Servolo Island, Venice, Italy) *(ICS '12)*. Association for Computing Machinery, New York, NY, USA, 331–340.

[24] Arthur Gretton, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. 2012. A Kernel Two-Sample Test. *J. Mach. Learn. Res.* 13 (March 2012), 723–773.

[25] Jens Gustedt. 2003. Randomized permutations in a coarse grained parallel environment. In *Proceedings of the 6th European Conference on Computer Systems*.

[26] Torben Hagerup. 1991. Fast parallel generation of random permutations. In *Automata, Languages and Programming*, Javier Leach Albert, Burkhard Monien, and Mario Rodríguez Artalejo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 405–416.

[27] Jesse D Hall, Nathan A Carr, and John C Hart. 2003. Cache and bandwidth aware matrix multiplication on the GPU. (2003).

[28] Mark Harris, Shubhabrata Sengupta, and John D Owens. 2007. Parallel prefix sum (scan) with CUDA. *GPU gems* 3, 39 (2007), 851–876.

[29] Bingsheng He, Naga K Govindaraju, Qiong Luo, and Burton Smith. 2007. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. 1–12.

[30] Wassily Hoeffding. 1963. Probability Inequalities for Sums of Bounded Random Variables. *J. Amer. Statist. Assoc.* 58, 301 (1963), 13–30.

[31] Joseph JéJé. 1992. *An introduction to parallel algorithms*. Reading, MA: Addison-Wesley.

[32] Yunlong Jiao and Jean-Philippe Vert. 2015. The Kendall and Mallows kernels for permutations. In *International Conference on Machine Learning*. PMLR, 1935–1944.

[33] William R. Knight. 1966. A Computer Method for Calculating Kendall's Tau with Ungrouped Data. *J. Amer. Statist. Assoc.* 61, 314 (1966), 436–439.

[34] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[35] Daniel Langr, Pavel Tvrdik, Tomas Dytrych, and J. Draayer. 2014. Algorithm 947: Paraperm-Parallel Generation of Random Permutations with MPI. *ACM Trans. Math. Software* 41 (10 2014), 5:1–5:26.

[36] Pierre L'Ecuyer and Richard Simard. 2007. TestU01: AC library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)* 33, 4 (2007), 1–40.
[37] Michael Luby and Charles Rackoff. 1988. How to Construct Pseudorandom Permutations from Pseudorandom Functions. *SIAM J. Comput.* 17, 2 (1988).
[38] Horia Mania, Aaditya Ramdas, Martin J Wainwright, Michael I Jordan, Benjamin Recht, et al. 2018. On kernel methods for covariates that are rankings. *Electronic Journal of Statistics* 12, 2 (2018), 2537–2577.
[39] Yossi Matias and Uzi Vishkin. 1991. Converting high probability into nearly-constant time - With applications to parallel hashing. *Proc. 23rd Ann. ACM Symp. on Theory of Computing* (01 1991), 307–316.
[40] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (Jan. 1998), 3–30.
[41] Duane Merrill. 2015. Cub. *NVIDIA Research* (2015).
[42] Duane Merrill and Michael Garland. 2016. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002* (2016).
[43] G. L. Miller and J. H. Reif. 1985. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. 478–489.
[44] Rory Mitchell, Joshua Cooper, Eibe Frank, and Geoffrey Holmes. 2021. Sampling Permutations for Shapley Value Estimation. *arXiv preprint arXiv:2104.12199* (2021).
[45] Rory Mitchell and Eibe Frank. 2017. Accelerating the XGBoost algorithm using GPU computing. *PeerJ Computer Science* 3 (2017), e127.
[46] Kenneth Moreland and Edward Angel. 2003. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (San Diego, California) *(HWWS '03)*. Eurographics Association, Goslar, DEU, 112–119.
[47] NVIDIA Corporation. 2020. CUDA C++ Programming Guide. Version 11.1.
[48] Michelle Perry, Harrison B Prosper, and Anke Meyer-Baese. 2014. GPU Implementation of Bayesian Neural Network Construction for Data-Intensive Applications. *Journal of Physics: Conference Series* 513, 2 (jun 2014), 022027.
[49] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. 2009. Fast in-place sorting with cuda based on bitonic sort. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 403–410.
[50] Lukas Prediger, Niki Loppi, Samuel Kaski, and Antti Honkela. 2021. d3p–A Python Package for Differentially-Private Probabilistic Programming. *arXiv preprint arXiv:2103.11648* (2021).
[51] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing* (3 ed.). Cambridge University Press, New York, NY, USA.
[52] Sanguthevar Rajasekaran and John H. Reif. 1989. Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms. *SIAM J. Comput.* 18, 3 (06 1989), 594–14. Copyright - Copyright] © 1989 © Society for Industrial and Applied Mathematics; Last updated - 2012-07-02.
[53] C. Radhakrishna Rao. 1961. Generation of Random Permutations of Given Number of Elements Using Random Sampling Numbers. *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)* 23, 3 (1961), 305–307.
[54] J. H. Reif. 1985. An optimal parallel algorithm for integer sorting. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. 496–504.
[55] James Reinders. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.".
[56] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. 2011. Parallel random numbers: As easy as 1, 2, 3. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.
[57] Martin Sandelius. 1962. A Simple Randomization Procedure. *Journal of the Royal Statistical Society. Series B (Methodological)* 24, 2 (1962), 472–481.
[58] Peter Sanders. 1998. Random permutations on distributed, external and hierarchical memory. *Inform. Process. Lett.* 67, 6 (1998).
[59] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 351–362.
[60] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2015. Sequential Random Permutation, List Contraction and Tree Contraction Are Highly Parallel. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Diego, California) *(SODA '15)*. Society for Industrial and Applied Mathematics, USA, 431–448.
[61] Bjarne Stroustrup. 2013. *The C++ programming language.* Pearson Education.
[62] Jakub Szuppe. 2016. Boost.Compute: A Parallel Computing Library for C++ Based on OpenCL. In *Proceedings of the 4th International Workshop on OpenCL* (Vienna, Austria) *(IWOCL '16)*. Association for Computing Machinery, New York, NY, USA, Article 15, 39 pages.
[63] RAPIDS Development Team. 2018. *RAPIDS: Collection of Libraries for End to End GPU Data Science.* https://rapids.ai

A RUNTIME OF GPU AND CPU SHUFFLING ALGORITHMS

Figures 14 and 15 reproduce Figures 11 and 13, reporting runtime in seconds instead of throughput ((millions of keys)/time(s)).

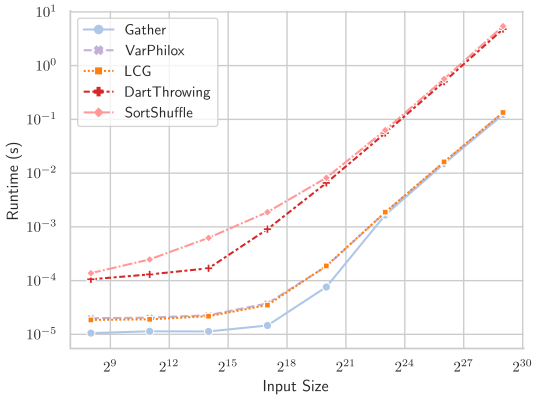


Fig. 14. GPU shuffling algorithms runtime - Tesla V100

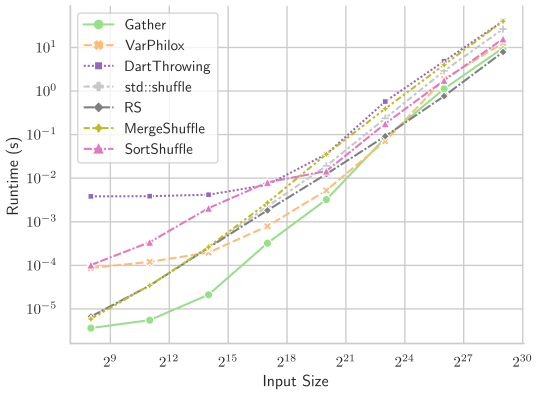


Fig. 15. CPU shuffling algorithms runtime - 2x Intel Xeon E5-2698

Chapter 7

Conclusion

This thesis introduces a set of improvements to widely used machine learning algorithms from a theoretical and software optimisation perspective. The techniques discussed provide significant improvements in throughput in a number of application domains.

7.1 Thesis Summary

The thesis consists of four primary contributions addressing different subproblems relating to computationally demanding tasks in the machine learning pipeline. These contributions are related by the common thread of software and algorithm optimisation techniques shared between chapters. Common themes explored in pursuit of throughput improvements are the application of non-standard GPU-based techniques to achieve large throughput improvements for unusual problems, the provision of open-source software as a contribution to benefit industry and research, and the use of reproducing kernel Hilbert space techniques in the domain of permutations. The specific contributions are as follows.

Chapter 3 discusses methods for density estimation and approximate quantiles based on extremely compact sketches containing sample moments. Our empirical analysis compares methods of fitting probability density functions to sample moments, including the method of maximum entropy, and orthogonal

series in a polynomial or trigonometric basis. We show that these sketches are highly practical if care is taken to avoid loss of numerical precision, and also show their suitability for execution on GPUs via tree reduction algorithms.

Chapter 4 introduces GPUTreeShap, an adaptation of interpretability algorithms for decision trees to GPU architectures. We apply a novel scheduling technique using bin-packing to efficiently map a nontrivial recursive algorithm to efficient hardware execution units on the GPU. Our open-source software package provide speed ups of between 15-340x over existing CPU based systems, allowing practitioners to solve compute intensive interpretability problems significantly faster.

Chapter 5 addresses an important NP-hard class of interpretability problems from the perspective of quasi-Monte Carlo methods. We investigate reproducing kernel Hilbert spaces over the symmetric group and use them to establish convergence results for quasi-Monte Caro methods based on these kernels. We also examine another family of quasi Monte Carlo sampling methods based on relations between the symmetric group and the n-sphere and compare them empirically to existing Shapley value approximations.

Finally, in Chapter 6, we develop a shuffling algorithm specially designed for GPUs, filling a gap in current parallel primitive algorithms. Our bijective-shuffle algorithm borrows from cryptography and pseudo random number generation, resulting in a parallel shuffling algorithm that is work efficient, accurate with respect to the distributions generated, and orders of magnitude faster than existing algorithms for GPUs or multicore CPUs.

In summary, revisiting the thesis statement, the above mentioned improvements successfully utilise specialised algorithm design and graphics processing units to significantly improve the throughput of real-world machine learning pipelines.

7.2 Future Work

Broadly speaking, opportunities remain to leverage algorithmic improvements and compute hardware to improve the outcomes of machine learning projects. The continued development of open-source software ecosystems is likely to play a key role in advancing the field — not only due to the provision of useful software tools for practitioners, but also as an efficient mechanism for later authors to build upon, and as a means of addressing a reproducibility crisis in machine learning [48, 45]. Furthermore, GPU-computing is still a subfield in its infancy. While many embarrassingly parallel problems have been convincingly solved, there remain a number of important emerging applications for which no effective GPU-acceleration techniques have been developed. Below, we highlight some more specific unanswered questions or opportunities encountered in individual chapters of this thesis.

The sketching methods introduced in Chapter 3 have further applications in decision tree algorithms. In particular, the mean absolute error criterion is difficult to optimise effectively for decision tree models [5], as the calculation of the value of any given split requires the median of the respective partitions of labels. This can be addressed in the sequential algorithm using data structures for the streaming median calculation. In a parallel decision tree induction algorithm (such as [37]), it is unclear how a similar approach would be possible. The moment-based quantile sketching methods [39] offer a data structure capable of estimating the median to reasonable accuracy, while also being compact and forming an associative operator. Thus, the quantile sketches can be used in a prefix sum computation in parallel decision tree induction algorithms such that the mean absolute error criterion can be optimised in parallel. This insight may extend to other regression error criteria beyond absolute error that do not make assumptions of normality.

In Chapter 4, we discuss methods for computing second-order feature interactions in decision tree models. As the number of features grows, the matrix of second-order interactions grows quadratically such that the computation in-

creases dramatically and the usefulness of manual inspection of Shapley values diminishes. A useful improvement would be an algorithm capable of finding and generating the top-k interactions only, at a reduced complexity than that of simply enumerating and filtering all interactions.

Another possible extension would be an in-place shuffling variant of the algorithm presented in Chapter 6. Most GPU algorithms that require rearrangement of inputs, such as sorting, are not in-place (a notable exception is bitonic sorting [44]). This is also true for the parallel shuffling methods discussed in Chapter 6. An open question is whether an effective in-place GPU shuffling algorithm can be formulated. Regardless of whether this is the case, it would be useful to establish the best-case time complexity for such an algorithm under the CUDA programming model.

References

[1] ACM. Fathers of the deep learning revolution receive acm a.m. turing award, March 2019.

[2] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*, pages 359–371. Elsevier, 2012.

[3] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[4] Leo Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, August 1996.

[5] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[6] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[7] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2017.

[8] Andreas Christmann and Ingo Steinwart. Universal kernels on non-standard input spaces. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[9] Shay Cohen, Gideon Dror, and Eytan Ruppin. Feature selection via coalitional game theory. *Neural Computation*, 19(7):1939–1961, 2007.

[10] Ian Covert, Scott Lundberg, and Su-In Lee. Explaining by removing: A unified framework for model explanation. *arXiv preprint arXiv:2011.14878*, 2020.

[11] Xiaotie Deng and Christos H Papadimitriou. On the complexity of cooperative solution concepts. *Mathematics of operations research*, 19(2):257–266, 1994.

[12] Richard M Dudley. *Real analysis and probability*. CRC Press, 2018.

[13] Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and experience*, 24(3):327–336, 1994.

[14] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, August 1997.

[15] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

[16] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *The Journal of Machine Learning Research*, 13(1):723–773, 2012.

[17] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[18] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[19] David Harrison Jr and Daniel L Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of environmental economics and management*, 5(1):81–102, 1978.

[20] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *Boosting and Additive Trees*, pages 337–387. Springer New York, New York, NY, 2009.

[21] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[23] Joseph JéJé. *An introduction to parallel algorithms*. Reading, MA: Addison-Wesley, 1992.

[24] Yunlong Jiao and Jean-Philippe Vert. The kendall and mallows kernels for permutations. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1935–1944. JMLR.org, 2015.

[25] Kaggle. Kaggle machine learning & data science survey, 2020. data retrieved from, `https://www.kaggle.com/c/kaggle-survey-2020`.

[26] Margot E Kaminski. The right to explanation, explained. *Berkeley Tech. LJ*, 34:189, 2019.

[27] Zohar Karnin, Kevin Lang, and Edo Liberty. Optimal quantile approximation in streams. In *2016 ieee 57th annual symposium on foundations of computer science (focs)*, pages 71–78. IEEE, 2016.

[28] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[29] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[30] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. From local explanations to global understanding with explainable ai for trees. *Nature Machine Intelligence*, 2(1):2522–5839, 2020.

[31] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

[32] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017.

[33] Horia Mania, Aaditya Ramdas, Martin J Wainwright, Michael I Jordan, and Benjamin Recht. On kernel methods for covariates that are rankings. *Electronic Journal of Statistics*, 12:2537–2577, 2018.

[34] Pedro J Martín, Luis F Ayuso, Roberto Torres, and Antonio Gavilanes. Algorithmic strategies for optimizing the parallel reduction primitive in cuda. In *2012 International Conference on High Performance Computing & Simulation (HPCS)*, pages 511–519. IEEE, 2012.

[35] Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002*, 2016.

[36] Charles A Micchelli, Yuesheng Xu, and Haizhang Zhang. Universal kernels. *Journal of Machine Learning Research*, 7(12), 2006.

[37] Rory Mitchell and Eibe Frank. Accelerating the xgboost algorithm using gpu computing. *PeerJ Computer Science*, 3:e127, 2017.

[38] Rory Mitchell, Eibe Frank, and Geoffrey Holmes. Gputreeshap: Fast parallel tree interpretability, 2020.

[39] Rory Mitchell, Eibe Frank, and Geoffrey Holmes. An empirical study of moment estimators for quantile approximation. *ACM Transactions on Database Systems (TODS)*, 46(1):1–21, 2021.

[40] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[41] NVIDIA Corporation. CUDA C++ programming guide, 2020. Version 11.1.

[42] Jochen Papenbrock, Peter Schwendner, Markus Jaeger, and Stephan Krügel. Matrix evolutions: synthetic correlations and explainable machine learning for constructing robust investment portfolios. *The Journal of Financial Data Science*, 3(2):51–69, 2021.

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[44] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place sorting with cuda based on bitonic sort. In *International Conference on Parallel Processing and Applied Mathematics*, pages 403–410. Springer, 2009.

[45] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program). *Journal of Machine Learning Research*, 22, 2021.

[46] Lukas Prediger, Niki Loppi, Samuel Kaski, and Antti Honkela. d3p–a python package for differentially-private probabilistic programming. *arXiv preprint arXiv:2103.11648*, 2021.

[47] Alexandre Quemy. Two-stage optimization for machine learning workflow. *Information Systems*, 92:101483, 2020.

[48] Edward Raff. A step toward quantifying independently reproducible machine learning research. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[49] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988.

[50] Amazon Web Services. Machine learning with Amazon SageMaker. `https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-mlconcepts.html`. Accessed: 2021-07-18.

[51] Lloyd S Shapley. A value for n-person games. *Contributions to the Theory of Games*, 2(28):307–317, 1953.

[52] Erik Strumbelj and Igor Kononenko. An efficient explanation of individual classifications using game theory. *J. Mach. Learn. Res.*, 11:1–18, March 2010.

[53] Vasily Volkov. *Understanding latency hiding on GPUs*. PhD thesis, UC Berkeley, 2016.

[54] Ulrike Von Luxburg and Bernhard Schölkopf. Statistical learning theory: Models, concepts, and results. In *Handbook of the History of Logic*, volume 10, pages 651–706. Elsevier, 2011.

[55] Erik Štrumbelj and Igor Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowl. Inf. Syst.*, 41(3):647–665, December 2014.

Appendix A

Co-authorship Forms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Co-Authorship Form

Postgraduate Studies Office
Student and Academic Services Division
Wahanga Ratonga Matauranga Akonga
The University of Waikato
Private Bag 3105
Hamilton 3240, New Zealand
Phone +64 7 838 4439
Website: http://www.waikato.ac.nz/sasd/postgraduate/

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in your appendices for all the copies of your thesis submitted for examination and library deposit (including digital deposit).

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.
Chapter 3 - An Empirical Study of Moment Estimators for Quantile Approximation

Published in ACM Transactions on Database Systems

Nature of contribution by PhD candidate	Conceived the idea, performed experiments, wrote the paper
Extent of contribution by PhD candidate (%)	70

CO-AUTHORS

Name	Nature of Contribution
Eibe Frank	Supervision, discussion, paper revision
Geoffrey Holmes	Supervision, discussion, paper revision

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and

Name	Signature	Date
Eibe Frank	Eibe Frank	20 July 2021
GEOFF HOLMES		21/7/21

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Co-Authorship Form

Postgraduate Studies Office
Student and Academic Services Division
Waharoa Ratonga Matauranga Akonga
The University of Waikato
Private Bag 3105
Hamilton 3240, New Zealand
Phone +64 7 838 4439
Website: http://www.waikato.ac.nz/sasd/postgraduate/

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work**. Completed forms should be included in your appendices for all the copies of your thesis submitted for examination and library deposit (including digital deposit).

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.
Chapter 4 - GPUTreeShap: Massively Parallel Exact Calculation of SHAP Scores for Tree Ensembles

Submitted to ACM Transactions on Parallel and Distributed Systems 23-Jun-2021

Nature of contribution by PhD candidate	Conceived the idea, performed experiments, wrote the paper
Extent of contribution by PhD candidate (%)	70

CO-AUTHORS

Name	Nature of Contribution
Eibe Frank	Supervision, discussion, paper revision
Geoffrey Holmes	Supervision, discussion, paper revision

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and

Name	Signature	Date
Eibe Frank	Eibe Frank	20 July 2021
GEOFF HOLMES		21/7/21

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Co-Authorship Form

Postgraduate Studies Office
Student and Academic Services Division
Wahanga Ratonga Matauranga Akonga
The University of Waikato
Private Bag 3105
Hamilton 3240, New Zealand
Phone +64 7 838 4439
Website: http://www.waikato.ac.nz/sasd/postgraduate/

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in your appendices for all the copies of your thesis submitted for examination and library deposit (including digital deposit).

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.
Chapter 5 - Sampling Permutations for Shapley Value Estimation

Submitted to Journal of Machine Learning Research 24-04-2021

Nature of contribution by PhD candidate	Conceived the idea, performed experiments, wrote the paper
Extent of contribution by PhD candidate (%)	70

CO-AUTHORS

Name	Nature of Contribution
Eibe Frank	Supervision, discussion, paper revision
Geoffrey Holmes	Supervision, discussion, paper revision
Joshua Cooper	Theorem 2 proof, paper revision, discussion.

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and

Name	Signature	Date
Eibe Frank	Eibe Frank	20 July 2021
GEOFF HOLMES		21/7/21
JOSHUA COOPER		24/7/21

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Co-Authorship Form

Postgraduate Studies Office
Student and Academic Services Division
Waharanga Ratonga Mafauranga Akonga
The University of Waikato
Private Bag 3105
Hamilton 3240, New Zealand
Phone +64 7 838 4439
Website: http://www.waikato.ac.nz/sasd/postgraduate/

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in your appendices for all the copies of your thesis submitted for examination and library deposit (including digital deposit).

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.
Chapter 6 - Bandwidth-Optimal Random Shuffling for GPUs

Submitted to ACM Transactions on Parallel Computing 19-05-2021

Nature of contribution by PhD candidate	Conceived core ideas, performed experiments, wrote the paper
Extent of contribution by PhD candidate (%)	60

CO-AUTHORS

Name	Nature of Contribution
Eibe Frank	Supervision, discussion, paper revision
Geoffrey Holmes	Supervision, discussion, paper revision
Daniel Stokes	Conceived core ideas, performed experiments, contributed to paper

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and

Name	Signature	Date
Eibe Frank	Eibe Frank	20 July 2021
GEOFF HOLMES		21/7/21
Daniel Stokes	Dstokes	22/7/21