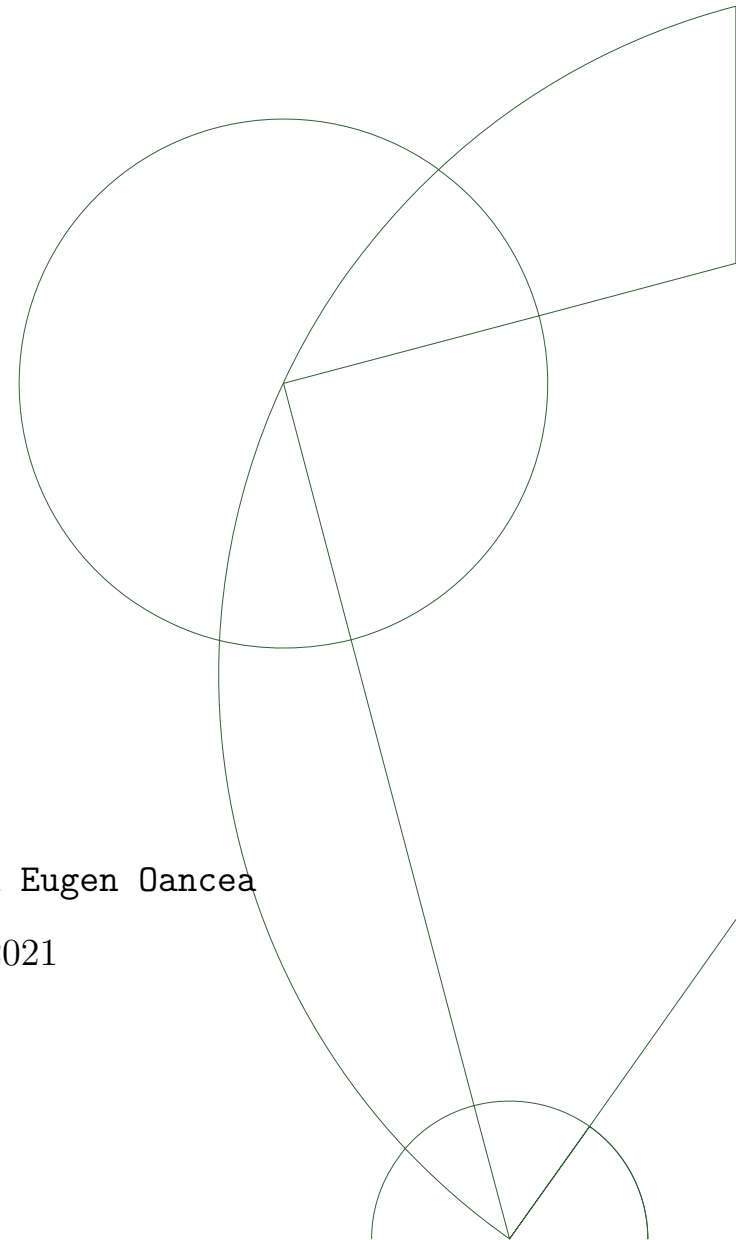
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Parallel implementations of machine learning algorithms

Gradient boosted decision trees

Kristian Høi
KU-ID:tcw684



supervisor: Cosmin Eugen Oancea

March 1, 2021

Abstract

Over the last decade, the popularity of GPUs has steadily increased. Now, all popular machine learning frameworks use the GPU to accelerate their performance. The Gradient Boosted Decision Trees(GDBT) is one of the popular machine learning algorithms which benefits from the power of the GPU. This requires an implementation in a low-level GPU programming language. Many of these implementations only target a specific platform. An alternative is to use a high-level GPU programming language to write the algorithm and rely on a compiler to generate efficient GPU code. Futhark is a high-level functional programming language specializing in compiling parallel array operations into efficient GPU code.
This thesis describes the algorithm for creating a GBDT model and translates the algorithm into using parallel building blocks. The parallel algorithm is created to fit the constraints of Futhark and flattening is used to achieve the maximum parallelism available. The Futhark implementation is compared to one of the most popular GDBT frameworks, XGBboost. It is shown the implementation achieves similar accuracy as the XGBoost. The XGBoost histogram-based implementation is x2 times faster than the futhark implementation. As Futhark is under development and is improving The Futhark implementation supports also the OpenCL platform whereas most GDBT frameworks only support CUDA.

Contents

1 Introduction

Gradient boosted decision trees (GBDT) is a highly popular and successful machine learning algorithm. It is frequently used in winning solutions for Kaggle competitions either as the standalone model or combined with other machine learning models. As Big Data continues to grow the demand for faster algorithms rises. The increasing amount of data processing required forces the need for faster implementations. Currently, the CPU takes a long time to create a decision tree for large datasets. To do create trees faster the algorithm translated so it supports parallel execution on the GPU.
The levels of a decision tree are built one level at a time making the algorithm an ideal candidate for parallel execution when creating the level. The question rises of how to translate the sequential algorithm so it uses parallel operations and how to implement this in a high-level programming GPU language compared to implementing the algorithm in a low-level GPU programming language.

In this thesis we present an implementation of the XGBoost algorithm used for creating gradient boosted decision trees. We present how this can be translated for parallel execution and is implemented in Futhark.
The specific contributions of this thesis is:

1. The mathematical theory behind gradient boosting and the XGBoost algorithm is derived. Using the mathematical theory an algorithmic framework is presented for finding the optimal decision tree.

2. The algorithm is translated into pseudo-Futhark using the parallel building blocks. These building blocks allows to achieve the maximum parallelism available.

3. Using flattening of the irregular parallelism is removed, so the algorithm can be implemented in futhark. This allows the compiler to generate code maximum the parallelism used.

4. The implementation is further improved by using histograms to build the trees. This allows for further elimination of irregular parallel and increases the speed of implementation

5. An empirical evaluation of the implementation and an comparison with the XGBoost framework

This thesis builds on the previous work of [1–3] which has been of great help in understanding the main algorithmic steps and the mathematical theory behind them. I would further like to acknowledge [1] for the development of the XGBoost framework.

Thesis structure

chapter 2: Background information about GPU programming and the challenges it presents. The chapter presents the high-level GPU programming language Futhark and the parallel building blocks it uses to maximise performance.

chapter 3: Gives an introduction to decision trees and how the are used. The chapter also shows how decision trees can be used with gradient boosting further improve their predictive strength.

chapter 4: Introduces algorithm for creating an gradient boosted decision tree. Herein it is shown the mathematical background for finding optimal decision tree and the GBDT algorithm is presented.

chapter 5: This chapter presents the parallel translation of the GDBT algorithm and how the parallel constructs from futhark is used. The usage of irregular parallelism is highlighted and how the implementation mitigates it by using flattening.

chapter 6: The Futhark implementation details for finding the optimal decision tree is presented. Herein implementation details about how decision trees are implemented and how flattening is used to create segmented parallel operations maximizing performance.

chapter 7: This chapter presents an improvement to the algorithm by using histograms when finding the optimal decision tree. These histograms reduces the time it takes to crease a decision tree.

chapter 8: shows the results of empirical evaluation on a large dataset. The accuracy and speed is compared against the popular XGBoost framework and the implementations performance using CUDA and OpenCL is discussed.

The code for the implementations presented in this thesis is made available on Github and be accessed by the following url: `https://github.com/KristianMH/FutharkGBDT`.

2 Background

General-purpose computing on graphics processing units (GPGPU) is increasing in popularity. The growth in single-core frequencies for CPUs has halted and programmers cannot rely on them to solve problems faster. The advance of Big Data has programmers turning their attention towards GPU programming to accelerate the performance of their algorithms. A GPU delivers thousands of computing cores running at a lower frequency than CPUs. This allows the GPU to execute many smaller operations in parallel to produce results faster. This gives massive speedup on tasks where the computations is done in parallel e.g. matrix multiplication.
The performance of matrix multiplication has a huge impact on neural networks as they rely heavily on matrix multiplication. Therefore every neural network requires a GPU to get a good performance.

Writing a GPU program is different than sequential programming on the CPU. The most common challenge is to avoid GPU memory becoming the bottleneck. Two primary platforms for GPU programming: CUDA and OpenCL.
CUDA provides an API for parallel computing on Nvidia GPUs. This platform is developed by Nvidia and proprietary. Programs written in CUDA can only be executed on a Nvidia GPU. OpenCL is an open-source API for parallel programming and is supported by the two major GPU vendors Nvidia and AMD. Programs written in OpenCL can be run on either hardware vendor.

Most algorithms use the GPU to accelerate the computation heavy tasks and the CPU doing the rest of the tasks. As the CUDA API is proprietary most frameworks have had to chose which platform they decide to use in their implementation.
Updates to CUDA might cause the algorithm implementation to rely on an unsupported operation. Now the programmer has to choose either to use a deprecated CUDA version or change to the OpenCL API. To convert the program to OpenCL may require to rethink the implementation to fit the API. Is it often easier to adopt the implementation to the updates instead.
Therefore most machine algorithm implementations target the CUDA platform. This includes popular frameworks such as XGBoost, Tensorflow, and PyTorch. Users of these algorithms are forced to obtain a Nvidia GPU to use the GPU accelerated implementations.

An alternative approach for GPU programming is to rely on a compiler to generate efficient GPU code. The programs can now be executed on any platform supported by the compiler.
Futhark is a high-level programming language with a compiler generating GPU code for CUDA and OpenCL [4]. The performance is now heavily reliant on the compiler to generate efficient GPU code but the algorithm is no longer limited to the CUDA API.

2.1 Futhark

Programs using the CUDA or OpenCL API are written in a low-level language and requires extensive knowledge of GPUs to create efficient programs. The GPU and CPU have different memory spaces so a copy is needed from CPU to GPU. the derives uses its own memory space for the computations

The GPU program receives input data from the host(CPU) and computes a result sending before it back to the host. It is difficult to transfer the data between host and device(GPU) efficiently and make sure the computations are done efficiently on the device.
To this, the programmer must know the pitfalls of data transfers and write optimal code for both the host and the device. Memory is often a performance bottleneck and using the right memory access to the cache is crucial to prevent this.

The goal of Futhark is to hide the difficulties of GPU programming: memory management and programming model. This reduces the burden of GPU programming from the programmer onto the compiler.
Futhark is a statically typed, data-parallel, and purely functional array language with a heavily optimized ahead-of-time compiler [4, 5]. The ahead-of-time compiler generates GPU code using calls to the CUDA or the OpenCL API. The language focuses on accelerating parallel array computations e.g. matrix computation. Futhark is inspired by Guy Blellochs' NESL [6] and focuses on achieving performance from nested parallelism and parallel building blocks.

Data-parallelism is when the application of a function to each member of a collection is done in parallel.
A flat data-parallel language does apply the function on the collection in parallel and the function is executed sequentially. A nested data-parallel language also applies the function in parallel for every member and the function is executed in parallel. This is useful in matrix multiplication as it is often implemented using three for-loops.

```
1 let matmul [N][M][L] (X: [N][M]i32) (Y:[M][L]i32) : [N][L]i32 =
2 map (\ xrow ->
3     map (\ ycol ->
4         reduce (+) 0 ( map2 (*) xrow ycol )
5     ) (transpose Y)
6 ) X
```

Figure 1: Matrix multiplication in Futhark

line 1: Function declaration of the function matmul. It takes two matrices as input and returns a single matrix. The [N][M][L] annotation is used to specify the sizes of input arrays. This allows the compiler to optimize the allocation in GPU memory.

line 2-5 The `map` operation operates on every row of X in parallel. The nested `map` operation operations on every column on Y. The dot product of the row and the column are done in parallel using `map` and `reduce` on line 4.

Second-Order Array Combinators (SOACs) are the most important building blocks of Futhark. They are highlighted with red in code examples. SOACs mirrors the behavior of higher-order functions found in other functional languages. They will have sequential semantics but parallel execution.
A list of important SOACs used in this thesis:

- `map f xs` : $(a \to b) \to [n]a \to [n]b$.
  The `map` operation takes a two arguments: a function f and an array. `map` applies f to each element in the array in parallel returning a new array of type b. This operator is commonly used for array transformations.

- `reduce op ne xs` : $(a \to a \to a) \to a \to [n]a \to a$
  The `reduce` operation takes three arguments as inputs: an operator $\oplus$, neutral element ne, an array. The operation returns $ne \oplus xs[0] \oplus xs[1] \oplus ... \oplus xs[n]$ in parallel.
  In order to do the aggregation in parallel the operator must be associative and it has a neutral element ne.

- `scan op ne xs`: $(a \to a \to a) \to a \to [n]a \to [n]a$
  The `scan` operation is also called *generalised prefix sum*. The `reduce` operation only produces one result for the input array. A `scan` operation produces the result for every prefix of the input array. A common use for the scan operation is to calculate the a cumulative sum: `scan (+) 0 [1,2,3] = [0+1, 0+1+2, 0+1+2+3]= [1,2,6]`.

- `scatter xs is vs`: $[m]a \to [n]i64 \to [n]a \to [m]a$
  `scatter` is used for in-place updates in parallel. *is* gives the write indices and *vs* is values to be written in the array *xs*. `scatter` [1,1,1] [1,2] [2,3] returns the result [1,2,3]. Here value 2 is written to index 1 and value 3 is written to index 2.

- `partition f xs`: $(a \to \text{bool}) \to [n]a \to ([]a, []a)$
  The `partition` operation evaluates the function f on every element in xs in parallel. All elements where $f(x)$ is true are returned in the first array. The elements where $f(x)$ is false are returned in the second array.

- `Reduce_by_index xs f ne is vs` : $[m]a \to (a \to a \to a) \to a \to [n]i64 \to [n]a \to [m]a$
  `Reduce_by_index` can be thought of a combination of `scatter` and `reduce`. The input *is* and *vs* are used as in `scatter`. Values written to same index are now aggregated using the function *f*. A common use of this SOAC is to calculate histograms [7]. Assume we have an array of [4,3,5,5] and would like to calculate a histogram counting the number of each values. The important operation is to find *is* such all elements with same value are written to same index.
  The command creating the histogram is:
  `reduce_by_index` [0, 0, 0] (+) 0 [0,1,2,2] [1,1,1,1] = [0+1, 0+1, 0+1+1] = [1,1,2].

2.1.1 Segmented operations

The compiler cannot flatten expressions using irregular nested parallelism, so every array in Futhark must be regular. For multi-dimensional arrays, this means all inner arrays must have the same shape. For example [[1,2,3], [4,5,6], [7,8,9]] is a valid array but the array [[1,2], [3,4,5], [6,7,8,9]] is not a valid array. The multi-dimensional array is instead represented using a shape array and the original data in a flat array [1,2,3,4,5,6,7,8,9]. The previous illegal matrix has shape [2,3,4]. This means the first two elements are the flat array is the first row of the multi-dimensional array.

Segmented operations allows for the use of SOACs on flat arrays. A segmented reduce will give the reduction of each entry : [1,2,3,4,5,6,7,8,9] with shape [2,3,4] and operator + returns [0+1+2, 0+3+4+5, 0+6+7+8+9] = [3, 12, 30]. Similarly a segmented scan will return [1,3,3,7,12,6,13,21,30]
To use the parallel construct on each segment, segmented functions take a flag array as input. The flag array for previous example is [T,F,T,F,F,T,F,F,F]. Here true represents the start of a new segment.

```
1  let segmented_scan [n] 't (op: t -> t -> t) (ne: t)
2                            (flags: [n]bool) (as: [n]t): [n]t =
3      let xs = zip flags as --combine elements and segment flag
4      -- define segmented operator which takes two elements as entries
5      -- it applies x op y if y flag is false i.e. same segment
6      --- else y i.e. start of new segment
7      let f (x_F, x) (y_F, y) =
8          (x_F || y_F, if y_F then y else x `op` y)
9      let res = scan f (false, ne) xs
10     in
11     (unzip res).1
```

Figure 2: implementation of segmented scan

line 1: The function now takes the additional flag array as input. The rest of the input is the same as `scan`.

line 3: Here the segment flags and array entries are combined. This is done so the segmented scan is done with a single scan operation.

line 7-8: f is a function which takes two elements as input. Each element is an array entry and segment flag. If the segment flag for the second element is true i.e. start of new segment then y is returned. If the second flag is false then we are still in a segment and x $\oplus$ y must be calculated.

line 9: The segmented scan is done using `scan` operation where the function now support segments. The neutral element is now (false, ne).

line 11: unzip the tuple array and returns the result of the segmented scan.

Indexing into to tuples are done with the dot notation in Futhark.
Consider a tuple: `let x =(3,4)`. To first element is accessed by x.0 and the second by x.1. Tuples are zero indexed. Figure 3 shows the important helper function for creating an flag array.

```
1 let mkFlagArray 't [m] (shp: [m]i64) (zero: t)
2                        (flag_val: t) (r: i64) : [r]t =
3   let shp_ind = scanExc (+) 0 shp
4   let vals = replicate m flag_val
5   in
6   scatter (replicate r zero) shp_ind vals
```

Figure 3: Function for creating the flag array based on the input shape

line 1: The function takes the shape of the irregular array is input. The zero and flag value are used in the resulting flag array. r denotes the length of the flag array.

line 3: The indices of where each segment starts are calculated with an exclusive scan operation. This correspond the exclusive prefix sum.

line 4: Allocate the number of flag values to be written

line 6: Write the segment flags into the locations given from the exclusive prefix sum.

It should be noted `mkFlagArray` function does not support representation of segments with no elements e.g. a shape array with [2,4,0,3]. For this array the created flagArray would be wrong as the zero element segment is marked to have a single element.

3 Decision trees

The decision tree learning method is a supervised machine learning strategy. Given a series of training instances and target values a model is created. The model will predict the target value based on the training instance. The prediction is obtained by traversing until a leaf(end-node) is reached. The traversal consists of the decisions made at the inner-nodes starting at the root.
A node within a decision tree consists of an index to the feature dimension and a split value. This could for example be height and 190. Here the values in the feature dimension named height are compared to 190. Based on the evaluation of the statement either left or right child is visited. The leaves of the tree contain the output value (prediction) of the model and their value depends on the learning objective.

Instance id	Age	Income	Plays football
0	15	10.000	0
1	54	512.000	1
2	42	320.000	1
3	30	423.000	0
4	65	1.000.000	0
5	80	80.000	0
6	25	50.000	1

Table 1: Example training instances

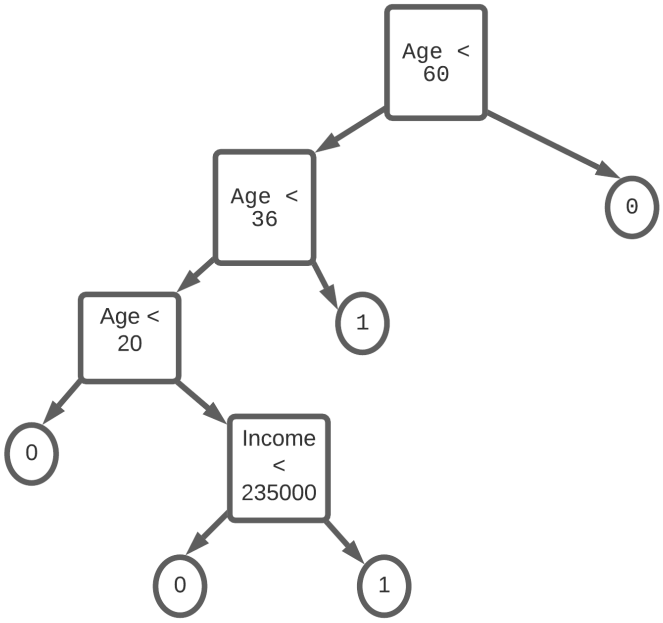


Figure 4: Example decision tree matching data in table 1

Figure 4 shows a decision tree based on the data in table 1. The tree does predict for each instance whether the instance plays football or not. This is represented with a binary variable 0 or 1, thus the tree is a classification tree.

To classify instance 0, the traversal starts at the root and compares the age (15) to the split value (60). As 15 is less than 60, the left child node is visited. Here the age is below 36 so the left child is visited again. The income of instance 0 is 10.000 which is less than 35.000, so the traversal ends in the left child leaf giving the prediction 0.

Two types of decision trees are commonly used in practice: classification and regression trees:

- Classification tree refers to the case an training instance is assigned to the predicted class. Consider an instance with age 20 and an income of 40.000. This instance will be classified to play football based on the traversal of the tree in figure 4
- Regression tree is when the predicted value is a real number.

The goal is to create a decision tree with achieves the lowest error based on the training instances. For classification trees, the error is typically a measure of the number of misclassifications.

3.1 Creating a decision tree

Creating a decision tree requires finding all the nodes and leaves. This process starts at the root and continues until every path ends in a leaf or a stop criteria is met. A common stop criteria is to limit the depth of the tree.
The tree in figure 4 is created by starting at the root and assign all training instances to it. A tree with only a root node will predict the same value all instances For a classification tree the prediction of a leaf is the most frequent class in the node. This means the root will predict class 0(Does not play football). The goal is the find a dimension index and split value such, by splitting the node the tree gets a lower prediction error. The training instances have two feature dimensions; age and income. Either feature can be used for splitting the instances, so all splits in each dimension has to be considered.
First, all the possible splits of instances by age is considered. The first possible split sends instance 0 to the left child and the rest to the right child. The tree then classifies instance 0 correctly. A split value for this split is chosen to between 15-25. Next split uses a split value between 25-30 and this sends instances 0 and 6 to the left child and rest to the right child. The best split found is Age < 60 as the tree then classifies instances 4 and 5 correctly. These instances are sent to the right child. It would not be beneficial to split the right child as both instances are of class 0. Therefore the node is turned into a leaf with prediction value 0.
The left child contains instances 1,2,3 and 6. Here the best split is found to Age < 36 as it will classify instance 1 and 2 correctly. This method of finding splits is used on every subset created from splitting a node.

The choice of an optimal split is hard to evaluate. Choosing a sub-optimal one may yield a better split in the next level of the tree improving the overall performance. Choosing an optimal split requires enumeration of every possible sub-tree created from the instances at a node. The number of possible trees are large and often infeasible to compute for larger datasets. Tree algorithms typically chose splits greedily. The split achieving the highest

quality is greedily chosen at every node. Depending on the objective task many different split quality measures can be used. For example, Gini impurity is for classification which gives the likelihood of misclassifying an instance. Here splits with the lowest Gini impurity value are chosen.
The values belonging to every feature are sorted and every split is enumerated within each feature. Assuming that n training instances with d features are assigned to a node and $\forall i = 1..n$, the split quality is evaluated by splitting at each instance i. This gives each feature $n \cdot d$ possible splits.

The decision tree model consists of a single decision tree and tends not to achieve good performance when trained on a large number of instances. The model may create a highly complex tree that overfits the training data and will have bad performance on validation data. Another reason is the locally optimal split chosen does not give a globally optimal tree. These disadvantages are typically mitigated by using an ensemble.

3.2 Decision trees ensembles

An ensemble is a collection of trees "collectively" improving the accuracy and robustness of the model. Each tree contributes to the final prediction allowing each tree to fit different patterns of the data. The two most commonly used ensemble methods for decision trees are; boosted trees and bootstrap aggregated trees.

- The bootstrap aggregated trees method builds multiple decision trees. Each tree trains on a random subset of the data allowing them to learn different patterns. This will result in different trees giving different predictions. The single over-complex tree might get a prediction wrong for an instance where the ensemble will have different predictions for the same instance. As the final prediction is done collectively, other ensemble members can "correct" the prediction made by the tree who learned the same pattern as the complex single tree.

- The boosted trees method builds an ensemble iteratively. The first tree will have some erroneous predictions and the next tree created tries to correct these predictions. The same approach is used for the third tree and so on. Combining the predictions from all trees will correct the errors from the predictions of the first tree.

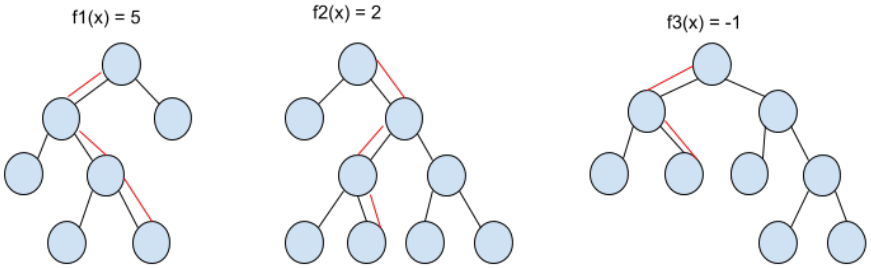


Figure 5: Illustration of predictions for instance x in the ensemble

Figure 5 shows how the prediction of each member in the ensemble is obtained. All the predictions are used for the collective prediction. The input instance x traverses through every tree to obtain a prediction. The traversal through each tree given by the red path in the figure.
For a bootstrap aggregated ensemble solving a regression task the collective prediction is $\frac{5+2-1}{3} = 2$. For classification tasks the collective prediction is typically the majority vote between predicted classes. The prediction for a boosted trees ensemble differs as each tree corrects the erroneous predictions from the previous tree. For figure 5 the ensemble would predict $5 + 2 - 1 = 6$ for regression.

3.3 Gradient boosting

Gradient boosting combines weak "learners" into a single strong leaner in an iterative fashion. The goal is to "teach" the model $\mathbf{F}(x)$ to predict target values. Here the strong learner $\mathbf{F}(x)$ is an ensemble and the individual trees are the "weak" learners. A weak learner is a tree that performs poorly and is considered slightly better than random guessing. The problem of creating an ensemble is represented as gradient descent. Each iteration of gradient descent we create tree which minimizes the given loss function. The loss function must be differentiable and it measures the performance of the model(ensemble) on the training set.
The boosting algorithm executes T boosting iterations to learn the strong learner $\mathbf{F}(x)$ which gives predictions $\mathbf{F}(x) = \hat{y}$ minimising the loss function $L(y, \hat{y})$

x_i	training instance i
y_i	target value for training instance i
$\hat{y}_i$	predicted value for training instance i
f_t	tree created at iteration t during boosting
$\mathbf{F}_t$	Ensemble of trees created up to boosting iteration t

Table 2: Notation used in the following sections

At each iteration a new estimator $f_t(x)$ is added to correct the prediction y for all training instances:

$$\mathbf{F}_{t+1}(x) = \mathbf{F}_t(x) + f_t(x) = \hat{y} \Rightarrow f_t(x) = y - \mathbf{F}_t(x)$$

This will fit the model $f_t(x)$ to the residuals $y - \mathbf{F}_t(x)$ for the current boosting iteration. $f_t(x)$ is the decision tree approximating the residuals.

This iterative process applies gradient descent by adding the prediction from tree f_t. Assuming the loss function used is squared error:

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

The loss over all training instances is given by:

$$E = \sum_{i=0}^{n} L(y_i, \mathbf{F}(x_i))$$

Each iteration seeks to minimize E by adjusting $\mathbf{F}(x_i)$. The gradient for x_i is given by:

$$\frac{\partial E}{\partial \mathbf{F}(x_i)} = \frac{\partial L(y_i, \mathbf{F}(x_i))}{\partial \mathbf{F}(x_i)} = \mathbf{F}_t(x_i) - y_i$$

Gradient descent will add the negative gradient in each iteration:

$$-\frac{\partial L(y, \mathbf{F}(x))}{\partial \mathbf{F}(x)} = -(\mathbf{F}_t(x) - y) = y - \mathbf{F}_t(x) = f_t(x)$$

So by adding the prediction from a decision tree f_t approximating the residuals corresponds to taking the gradient step.

For figure 5 each tree will correspond to the approximation of the gradient step. Assume a instance x has the target value y. The ensemble is created by finding the tree f_t at each iteration. Initially $\hat{y}_0 = 0$.
The first tree f_1 is created to approximate the residual $y - \mathbf{F}_0(x) = y$. The prediction for the first tree $f_1(x) = 5$. Next tree created now approximates the residual $y - \mathbf{F}_1(x) = y - (5 + 0)$. This gives prediction $f_2(x) = 2$.
The output of the ensemble is $\hat{y} = \hat{y}_0 + \sum_t f_t(x) = 0 + (5 + 2 - 1) = 6$ and since each tree approximates the gradient step, the ensemble implements gradient boosting.

4 Gradient boosting decision tree

In this section, the mathematical background for XGBoost algorithm [1, 2] is derived for creating a *gradient boosting decision tree* ensemble (GDBT) and a algorithmic overview is presented.
The XGBoost algorithm shows how to build each tree f_t during boosting. Instead of optimizing squared error, an objective function $\mathcal{L}$ with two parts l and Ω is used. l is the function measuring the ensemble performance on the individual training instance. Ω is the regularization term penalizing the complexity of the ensemble. The regularized objective is:

$$\mathcal{L}(\mathcal{X}) = \sum_{i=0}^{n} l(y_i, \mathbf{F}(\mathbf{x}_i)) + \sum_{t=0}^{E} \Omega(f_t), \;\; f_t \in \mathbf{F} \tag{1}$$
$$\text{where } \Omega(f) = \gamma M + \frac{1}{2}\lambda \mathbf{w}^T\mathbf{w}$$

- $\mathcal{X}$ is the set of training instances $\{\mathbf{x}_0 ... \mathbf{x}_n \mid \mathbf{x} \in \mathbb{R}^d\}$
- $\mathbf{F}$ is the ensemble with prediction $\mathbf{F}(\mathbf{x}) = \hat{y}$. The ensemble has size E.
- M is the number of leafs in the tree f_t. For each tree f_t the vector $\mathbf{w}_t$ holds the predictions of f_t. Each entry in $\mathbf{w}_t$ corresponds to prediction from a leaf.
- γ, λ is regularization parameters chosen by the user. γ increases the penalty of adding leaves to tree f_t.
  λ is used to penalize extreme weights i.e extreme predictions. These parameters allows the user to reduce the potential overfitting by the ensemble on the training instances.
- f_t is an ensemble member. We define the traversal of the tree f_t as $f_t(\mathbf{x}) = \mathbf{w}_{q(\mathbf{x})}$. where $q(\mathbf{x}) : \mathbb{R}^d \to M, \mathbf{w} \in \mathbb{R}^M$. q is a function which takes an input instance $\mathbf{x}$ and returns the leaf index of traversing the tree f_t. This is index is used to obtaining the prediction for f_t by using it as an index to $\mathbf{w}_t$
- l is a differential convex loss function and the total loss for the ensemble is the sum of the element-wise loss between target value y and prediction $\hat{y}$. For example sqaured error: $\frac{1}{2}(y - \mathbf{F}(x))^2$

The idea of this regularized objective is to obtain a model where each tree is as simple and predictive as possible. At each boosting iteration, it is the goal to find a minimal complex tree that simultaneously minimizes $l(y_i, \hat{y}_i)$.
The objective function at iteration t in T boosting rounds is given by:

$$\mathcal{L}^{(t)} = \sum_{i=0}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t) \tag{2}$$

Recall $\hat{y_i}^{t-1}$ is the prediction from the ensemble at the previous iteration. At iteration t we add the gradient step f_t, so the goal is find the tree f_t which minimizes the objective function most.

Using Taylor expansion allows for optimization of the objective function in a general setting:

$$\mathcal{L}^{(t)} \simeq \sum_{i=0}^{n} \left[l(y_i, \hat{y_i}^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right] + \Omega(f_t) \tag{3}$$

where by definition of Taylor series

$$g_i = \frac{\partial}{\partial \hat{y}^{(t-1)}} \, l(y_i, \hat{y_i}^{(t-1)}) \tag{4}$$

$$h_i = \frac{\partial^2}{\partial \hat{y_i^{(t-1)}}^2} \, l(y_i, \hat{y_i}^{(t-1)}) \tag{5}$$

Equation 4 and 5 returns the values of evaluating value from the first and second-order derivative evaluated at y and $\hat{y}$. The user chooses l, the first and second order derivative. For squared error we have: $g_i = 2(y - \hat{y})$ and $h_i = 2$. g_i and h_i is evaluated for each instance using y_i and $\hat{y}_i$.
As we optimize with respect to f_t, the loss from ensemble predictions are constant and can be ignored. The complexity of all previous trees in the ensemble is also constant and is ignored. This gives the simplified objective:

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=0}^{n} \left[g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right] + \Omega(f_t) \tag{6}$$

Recall $\mathbf{w}_t$ is the vector of leaf predictions from the tree f_t with length M
Let $I_j = \{i \mid q(\mathbf{x}_i) = j, i \in \{1...n\}\}$ be the set of training instances mapped to leaf j, $j \in [0..M]$.
All training instances in I_j have prediction w_j. By expanding the complexity term $\Omega(f_t)$ from equation 1, the objective function is rewritten to sum over number of leaves M.

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=0}^{n} \left[g_i \cdot w_{q(\mathbf{x}_i)} + \frac{1}{2} h_i \cdot w_{q(\mathbf{x}_i)}^2 \right] + \gamma M + \frac{1}{2} \lambda \sum_{j=1}^{M} w_j^2 \tag{7}$$

$$= \sum_{j=0}^{M} \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma M \tag{8}$$

$$= \sum_{j=0}^{M} \left[G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma M \tag{9}$$

where

$$G_j = \sum_{i \in I_j} g_i \qquad\qquad H_j = \sum_{i \in I_j} h_i$$

G_j and H_j is the sum of all first and second derivatives for all instances assigned to leaf j.

Now we need to find the weight w_j^* that for each leaf j minimizes the objection function. The optimal weight is given by taking the derivative of equation 9 and setting it to zero.

$$0 = \frac{\partial}{\partial w_j}\tilde{\mathcal{L}}^{(t)} = G_j + (H_j + \lambda)w_j^* \tag{10}$$

$$0 = G_j + (H_j + \lambda)w_j^* \tag{11}$$

$$w_j^* = \frac{-G_j}{H_j + \lambda} \tag{12}$$

Using w_j^* in equation 9 the objective function for finding the best tree f_t becomes:

$$\tilde{\mathcal{L}}^t = \sum_{j=0}^{M}\left[G_j \cdot -\frac{G_j}{H_j + \lambda} + \frac{1}{2}\cdot(H_j + \lambda) + \left(\frac{-G}{H + \lambda}\right)^2\right] + \gamma M \tag{13}$$

$$= \sum_{j=0}^{M}\left[-\frac{G_j^2}{H_j + \lambda} + \frac{1}{2}\frac{G_j^2}{(H_j + \lambda)^2}\cdot(H_j + \lambda)\right] + \gamma M \tag{14}$$

$$= -\frac{1}{2}\sum_{j=0}^{M}\frac{G_j^2}{H_j + \lambda} + \gamma M \tag{15}$$

Equation 15 gives the objective value for a tree. For a large number of training instances is it infeasible to create every possible tree and calculate equation 15. Therefore the tree f_t is created greedily.

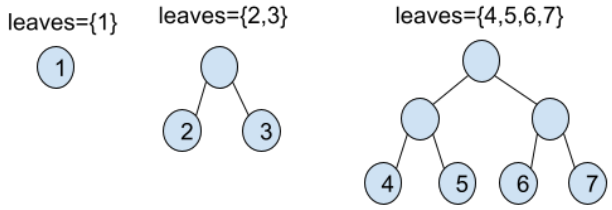


Figure 6: Growing the tree f_t

Figure 6 shows the growing strategy. We start with a root node meaning our tree structure has a single leaf. Using equation 15 to measure the objective value of a single leaf:

$$\tilde{\mathcal{L}}_j^{(t)} = -\frac{1}{2}\frac{G_j^2}{H_j + \lambda} + \gamma \tag{16}$$

If we split the root $I_j = I_{jL} \cup I_{jR}$ where I_{jL} and I_{jR} are the index sets of training instances. I_{jL} is the instances sent to left child and I_{jR} is the instances to right child. The objective function if we split leaf j is:

$$\tilde{\mathcal{L}}_{js}^{(t)} = -\frac{1}{2}\left(\frac{G_{jL}^2}{H_{jL} + \lambda} + \frac{G_{jR}^2}{H_{jR} + \lambda}\right) + 2\gamma \tag{17}$$

Thus we check if a split at leaf j decreases the objective function by combining equations 16 and 17 ($\tilde{\mathcal{L}}_j^{(t)} - \tilde{\mathcal{L}}_{js}^{(t)}$):

$$Q_j = -\frac{1}{2}\frac{G_j^2}{H_j+\lambda} + \gamma - \left(-\frac{1}{2}\left(\frac{G_{jL}^2}{H_{jL}+\lambda} + \frac{G_{jR}^2}{H_{jR}+\lambda}\right) + 2\gamma\right) \tag{18}$$

$$= -\frac{1}{2}\frac{G_j^2}{H_j+\lambda} + \frac{1}{2}\left(\frac{G_{jL}^2}{H_{jL}+\lambda} + \frac{G_{jR}^2}{H_{jR}+\lambda}\right) - \gamma \tag{19}$$

$$= \frac{1}{2}\left(\frac{G_{jL}^2}{H_{jL}+\lambda} + \frac{G_{jR}^2}{H_{jR}+\lambda} - \frac{G_j^2}{H_j+\lambda}\right) - \gamma \tag{20}$$

Now we can use equation 20 to measure the quality of a split. Recall we have to consider every possible split of training instances in each leaf. Therefore the quality is calculated for every possible split of the root in figure 6.
This is done for each feature dimension of the training instances. First all the instances are sorted according to the feature value. The split operator is used $<$ and by sorting we can check all possible split in a linear search. The first possible split sends 1 value to the left child and the rest to the right child. The next possible split sends two instances to the left and the rest to right.
This allows us to calculate the values G_{jL} and H_{jL} as a cumulative sum. Value G_{jR} is found using the property $G_j = G_{jL} + G_{na} + G_{jR} \Rightarrow G_{jR} = G_j + G_{jL} + G_{na}$. The analogous holds for H_{jR}.
This allows for efficient calculation of the split quality in equation 20.

Once the best split is found, the root is split if the quality is positive. This gives two new leaves (2 and 3) and splits are searched within the children. Figure 6 shows the strategy and which leaves are searched for splits for each level. We continue to grow the tree while splits with a positive quality are found. This strategy gives the tree f_t which minimizes the objective function i.e. the gradient step.

4.1 Algorithm overview

This section presents the algorithm for creating a GDBT ensemble. The algorithm 1 does creates the ensemble over T boosting iterations. In each iteration the optimal tree f_t is found. To find the optimal tree f_t algorithm 2 is used. Algorithm 2 describes process of how f_t is created.
Algorithm 1 returns the GDBT ensemble from T boosting iterations

Algorithm 1: Gradient Boosting training

Input: Training data $\mathcal{X} = \{x_1.....x_n\}$
Target Labels $\mathcal{Y} = \{y_1.....y_n\}$
Regularization parameters λ, γ, η
Maximum depth of trees D
Maximum number of iterations T
Objective function $\mathcal{L} : l$ and Ω
Output: Ensemble of f_t minimizing the objective $\mathcal{L}$ over T iterations

```
1  Ŷ ← {0.5..0.5}  /* Each instance has initial prediction of 0.5      */
2  for t ← 1 to T do
3      G ← Gradients(Y, Ŷ)  /* Equation 4 for each instance            */
4      H ← Hessians(Y, Ŷ)  /* Equation 5 for each instance             */
5      f_t ← FindOptimalTree(X, G, H, λ, γ, D)
6      Ŷ ← Ŷ + η · Predict(f_t, X)  /* add gradient step              */
7  end
8  return ∑_t^T f_t
```

line 6: The algorithm assigns 0.5 as the initial prediction for each instance. This is used for calculating the first and the second-order derivative values for $\hat{y}_{i0}$ for every $i \in [0..n]$

line 8: `Gradients` calculates the first-order derivative value by evaluating equation 4 for each instance.

line 9: `Hessians` calculates the second-order derivative value by evaluating equation 5 for each instance

line 10: `FindOptimalTree` finds the optimal tree f_t at boosting iteration t. It uses algorithm2 which uses the method described in section 4.

line 11: Adds the new predictions from the tree to each instance prediction. The `Predict` function does the tree traversal for each instance calculating $f_t(\mathbf{x})$.
η is a user variable used for controlling the loss convergence. A too high η value will result in stepping past to optimal objective value and a too low value will result in slow convergence.

Algorithm 2 builds the decision tree f_t using the method described in section 4. The algorithm loops over each level of the tree until the maximum depth is reached or no leaves can be split. For each level, every leaf is searched for a split using equation 20 and if the split quality is positive the leaf is split.
Many implementations of GBDT allow for the provision of missing values in the training instances. They are given with a NaN value. These values cannot be used as a split candidate. Algorithm 2 does calculate the split quality of either sending the missing values to the left or right child. This is done by adding G_{na} and H_{na} to left and right gradient/hessian sum respectively. G_{na} and H_{na} is the sum of gradient/hessian values for instances with a Nan value The direction for missing values is represented with a boolean flag: true for the left child and false for the right child.

Algorithm 2: FINDOPTIMALTREE: Finding the optimal tree structure f_t

Input: Training data $X = \{x_1.....x_n\}$
gradients $G = \{g_1.....g_n\}$, hessians $H = \{h_1.....h_n\}$
Regularization parameters λ, γ
Maximum depth of tree D

Output: Optimal tree f_t structure minimizing the objective $\mathcal{L}_t$

$I \leftarrow [\{1.....n\}]$ `/* Assign all instances to root */`
$m \leftarrow 0$ `/* number of leaves found */`
$(S, S_n) \leftarrow (\{0\}, \emptyset)$ `/* S:leaves at current level,` S_n`:leaves at next level */`
Nodes $\leftarrow \emptyset$, Leaves $\leftarrow \emptyset$
while $d <= D \wedge S \neq \emptyset$ **do**
  for $j \in S$ **do**
    $X_j \leftarrow \{x_i \mid i \in I[j]\}$ `/* instances in leaf j */`
    $G_j \leftarrow \sum_{i \in I_j} g_i$, $H_j \leftarrow \sum_{i \in I_j} h_i$
    $(d^*, v^*, flag^*) \leftarrow (-1, 0, False)$ `/* optimal split so far */`
    foreach *feature* d **do**
      $G_L \leftarrow 0$, $H_L \leftarrow 0$
      $I_{na} \leftarrow \{i \in I[j] \mid$ `is_NaN(`$\mathbf{x}_i[d]$`)}/* missing value instances */`
      $G_{na} \leftarrow \sum_{i \in I_{na}} g_i$, $H_{na} \leftarrow \sum_{i \in I_{na}} h_i$
      K $\leftarrow$ sort the set $\{k \mid k \in I[j] \setminus I_{na}\}$ by the corresponding value $\mathbf{x}_k[d]$
      for $i \in K$ **do**
        $G_L \leftarrow G_L + g_i$, $H_L \leftarrow H_L + h_i$ `/* cumulative sum */`
        $G_R \leftarrow G_j - G_L - G_{na}$, $H_R \leftarrow H_j - H_L - H_{na}$
        $Q_j(d, v, T) \leftarrow \frac{1}{2}\left[\frac{(G_L+G_{na})^2}{H_L+H_{na}+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda}\right] - \gamma$ `/* Equation 20 */`
        $Q_j(d, v, F) \leftarrow \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{(G_R+G_{na})^2}{H_R+H_{na}+\lambda} - \frac{G^2}{H+\lambda}\right] - \gamma$
        `/* Check if we should update optimal split */`
        if $Q_j(d, v, True) > Q_j(d^*, v^*, flag^*)$ **then**
          $(d^*, v^*, flag^*) \leftarrow (d, v, True)$
        else if $Q_j(d, v, False) > Q_j(d^*, v^*, flag^*)$ **then**
          $(d^*, v^*, flag^*) \leftarrow (d, v, False)$
        end
      end
    end
    if $Q_j(d^*, v^*, flag^*) > 0$ **then**
      $I_{na} \leftarrow \{i \in I[j] \mid$ `is_NaN(`$_i[d^*]$`)}`, $I_L \leftarrow \{i \in I[j] \mid \mathbf{x}_i[d^*] < v^*\}$
      $I_R \leftarrow I[j] \setminus (I_L \cup I_{na})$
      if *flag** **then**
        $I[m+1] \leftarrow I_L \cup I_a,\ I[m+2] \leftarrow I_R$
      else
        $I[m+1] \leftarrow I_L,\ I[m+2] \leftarrow I_R \cup I_{na}$
      end
      $S_n \leftarrow S_n \cup \{m+1, m+2\}$, Nodes $\leftarrow$ Nodes $\cup \{(d^*, v^*, flag^*)\}$, $m \leftarrow m+2$
    else
      Leaves $\leftarrow$ Leaves $\cup \frac{-G_j}{H_j+\lambda}$ `/* add prediction weight by equation 12 */`
    end
    $d \leftarrow d+1$, $S \leftarrow S_n$, $S_n \leftarrow \emptyset$
  end
end
return tree (Nodes, Leaves)

lines 1-3: The `FindOptimalTree` algorithm takes the set of training instances and the set of gradient values and hessian values as input. For instance i the gradient and hessian value are given by g_i and h_i where $i \in [0..n]$. Regularization parameters λ and γ are set by the user to reduce overfitting the training instances. D is the maximum depth of tree created.

lines 4-7: I is a vector, where each entry holds the indices of instances assigned to the node. For the root, $I[0]$ holds the indices for all instances. The variable m is used to count the number of leaves so far. This is initially set to 0.
S is the set of leaf indices for a level in the tree. This set is used when searching for splits. S_n is the set of leaf indices for the next level. Here entries are added as we split leaves at the current level. At the end of each level $S = S_n$ and $S_n = \emptyset$.
Nodes is the set of decision nodes created for the tree f_t. Leaves is the set of leaf predictions for f_t.

line 8 Start of the while-loop going over each level in the tree. This is done until the maximum depth is reached. It also terminates if there is no leaves left to split. This happens if $S = \emptyset$.

line 9 Start of the for-loop that checks every leaf at the current level and split them if is beneficial according to equation 20.

lines 10-13: X_j is the index set for all the instances in leaf j. G_j and H_j are the sum of gradient/hessian values for all the instances the leaf.
$(d^*, v^*, flag^*)$ is the optimal split information found so far for leaf j. d^* is the optimal feature to split, v^* is the best split value for feature d^*. $flag^*$ is the best direction for instances with missing values in dimension d^*.
They are reset to (-1, 0, False) for every leaf j. This default value correspond to no split found since we defined $Q(-1, 0, False) = -\infty$

line 13 Start of the for-loop going over each feature of the values in X. This loop searches for the optimal split in every feature d.

lines 14-17: The G_L and H_L is the variable for cumulative sum of gradient/hessian values for the left child for every split.
I_{na} is the set of instances with a missing value in feature d. The `is_NaN` function checks if the dimensional value is a NaN value. G_{na} and H_{na} is the gradient/hessian sums for instances with a missing value. These are used for determining the optimal direction for missing values as their value cannot be used as a split value.
Line 17 creates the index set by sorting instances according to their feature value. This is needed as we would like to enumerate every possible split in a linear search. Indices to instances with a NaN value are removed for this set: $I[j] \setminus I_{na}$.

lines 18-27: Calculates the quality of every split of instances in leaf j for dimension d. As the instances are sorted, splits are done between each instance left to right.

Therefore G_L and H_L is given by a cumulative sum.
G_R and H_R is calculated using property $G_R = G_j - G_L - G_{na}$.
$Q_j(d, v, T)$ represents the quality of the split on feature d with split value v and sending missing values to the left child. The quality is given by using equation 20.
The If-statements at lines 23-27 updates our optimal split information if we found a better one by comparing the quality of each split.

lines 30-41: Now the best split for leaf j is found and has the values $(d^*, v^*, flag^*)$. If the quality of the split $Q_j(d^*, v^*, flag^*)$ is positive we split the node otherwise we add the prediction from equation 12 to the leaves set.
If the node the is split, the index sets I_L, I_{na}, I_R is calculated. Depending on the flag the index set for missing values are either added to the left or right child.
The index set for left child is written to $I[m+1]$ where m is the total number of leaves found so far. The index set for the right child is written to $I[m+2]$. Indices m+1 and m+2 is added to S_n and the split information is added to the set of decision nodes. This means leaves m+1 and m+2 is added to S_n and we add the split information to nodes.

line 45: Returns the decision tree f_t made up of decision nodes and the leaves predictions. Section 6.1 explains the actual implementation of decision trees and how leaves and nodes are added to the tree.

Algorithm 2 is defined to be exact since it considers every possible split within each feature. This requires a huge amount of computations when a large number of training instances is in each leaf. This dominates the amount of work required to find the best split and thereby the time to create the tree f_t.
Another approach is to only use a subset of training instances when searching for a split. This gives a speed increase as only a subset of possible splits is considered, but it reduces the accuracy of the ensemble predictions.

5 Translating algorithms to parallel constructs

The challenge of implementing GBDT in futhark is to leverage as many of the SOACs as possible to utilize the maximum parallelism available. In this section, we describe the process of translating algorithms 1 and 2 into pseudo Futhark to get an overview of how the parallel constructors are used while ensuring the algorithm correctness for creating the ensemble.

```
1  let gradient_boost [n][d] (X: [n][d]f32) (Y: [n]f32) (T: i32)
2          (g: f32 -> f32 -> f32) (h: f32 -> f32-> f32) =
3  let Y_hat= replicate n 0.5
4      Loop (Y_hat, F) for t < T do
5          let gradients = map2 g Y Y_hat
6          let hessian = map2 h Y Y_hat
7          let ft = FindOptTree X gradients hessian
8          let preds = map (\xi -> predict ft xi) X
9          let Y_hat = map2 (+) Y_hat new_preds
10         let F = F ∪ ft -- saves the tree in the ensemble.
11         in
12         (Y_hat, F)
```

Figure 7: Pseudo Futhark of algorithm 1

line 1: Gradient boosting takes input instances X, target values Y, first-order derivative g and second-order derivative function h as inputs.

line 2: Set initial prediction to 0.5 for every instance.

lines 4-12: Sequential loop doing T boosting iterations. Each boosting iteration is done sequentially as iterations depends the predictions from the previous.

line 5: Calculates the individual first-order derivative value for each instance. This is done in parallel using a `map2` operation.

line 6: Calculates the individual second-order derivative value for each instance. This is done in parallel using a `map2` operation.

line 7: Finds the optimal tree f_t using algorithm 2. The parallel translation of algorithm 2 is shown in figure 8

line 8: Calculates new predictions for each instance. `Predict` does the tree traversal obtaining the prediction. This is done in parallel with a `map` operation since each instance traverses the tree independently. It takes a training instance and a tree as input. Section 6.1 explains the implementation.

line 9: Adds the new predictions in parallel with a `map` operation

line 10: Saves the found tree f_t to the ensemble $\mathbf{F}$. How this is done is described in section 6.1.

In order to achieve full parallelism every for-loop in algorithm 2 should be implemented using parallel constructs. Figure 8 shows the pseudo Futhark of the translation. This translation relies on irregular nested parallelism.

```
1  let findOptTree [n][d] (X: [n][d]f32) (G: [n]f32) (H: [n]f32)
2      -- all instances are assigned to root.
3      let (X, G, H) = ([X], [G], [H])
4      loop (X, G, H, l, N, L) for l <= D and !(null X) do
5      -- irregular nested parallelism, each entry in X differs in size
6          let splits = map3 (\Xj Gj Hj ->
7                      let (q, d, v, flag) = search_best_split Xj Gj Hj
8                      in (q, d, v, flag)
9                      ) X G H
10
11          -- irregular partition : split_info=(q,d,v,flag)
12         let (active_data, terminal_data) =
13             partition (\(split_info, _, _, _) -> split_info.0 > 0)
14                 (zip4 splits X G H)
15         let (_, _, terminal_G, terminal_H) = unzip4 terminal data
16         -- irregular map calculating prediction weights.
17         let L = L ++ map2 weight terminal_G terminal_H
18         let (act_splits, act_X, act_G, act_H) = unzip4 active_data
19         let N = N ++ act_splits
20         -- irregular partition, returns the irregular distribution
              of points in the new level
21         let X, G, H = map4 split_leaf act_splits act_X act_G act_H
              |> flatten |> unzip3
22         in
23           (X, G, H, l+1, N, L)
24     in
25     (N, L)
```

Figure 8: Pseudo-implementation of algorithm 2

line 1: The function takes the training instances X, gradient value G and hessian values H as input. Each instance have corresponding gradient/hessian value so all arrays have length n.

line 3: Initializes the irregular 2D arrays of instances assigned to each leaf at the level. At first all instances are assigned to the root. The irregular arrays for the instance distribution over leaves is initialized. For the root only a single leaf exists. For each level, we have $X = [X_1, ..X_M]$ where M is number at leaves at current level. $X_1..X_M$ may differ in size thus the array is irregular. Same holds for G and H. In the implementation these are flattened to use a flat representation. A shape array gives the number of instances in each leaf.

line 4: The sequential loop runs over each level of the tree and all the leaves at current level is processed in one iteration. The variables N and L corresponds to the decision nodes and leaf predictions found so far. The loops runs over each level until maximum depth or no more leaves with possible splits exists.

lines 6-9: An irregular `map` over the leaves at current level. The function `search_best_split` searches in parallel for the best split over all features. This function is shown in figure 11. The `map` returns the best split information for every leaf.
The `map` contains nested irregular parallelism should be flattened. To flatten this requires the use of segmented operations. The `search_best_split` function uses `radix_sort` for sorting. The sorting function should be lifted so it becomes a segmented sort, but this process is too tedious and not done in this thesis. Therefore this `map3` operation is implemented as a sequential loop over the leaves at the current level.

lines 11-14: Irregular `partition` over each leaf. This returns an active part and a terminal part. The active part contains the instances and gradient/hessian values for leaves which are split. The terminal part contains the instances and gradient/hessian values for leaves that are not split.

lines 17: The prediction value for terminal leaves are calculated using a `map` with the function `weight` on the terminal gradient/hessian values. The weight function is shown in figure 9. It calculates the prediction weight for the leaf.

line 21: The irregular `map` operation does the split of active leaves at the current level. This returns two new leaves for each entry. The function `split_leaf` is used to do the partition. This returns the instances sent to the left and to the right child. The function `split_leaf` returns a irregular 2D array containing the left and the right child. The result of `map` operation is flattened into a 2D array where each entry correspond to the new leaf introduces. Figure 10 shows the `split_leaf` function.
This `map` operation contains irregular nested parallelism and cannot be efficiently flattened by the compiler. In the implementation X, G and H will be represented using a shape array and flat arrays. To get efficient GPU code the irregular map and `split_leaf` is flattened into a segmented partition. Section 6.3 shows the flattened implementation.

```
1    let weight [m] (G: [m]f32) (H: [m]f32) (lambda: f32) : f32 =
2    let G_sum = reduce + 0 G
3    let H_sum = reduce + 0 H
4    in -G_sum/(H_sum+lambda)
```

Figure 9: Calculates the weight for a leaf using equation 12

```
1 let split [n][d] (act_splits: (f32,i64,f32,bool) (act_X: [n][d]f32)
2     (G: [n]f32) (H: [n]f32) : [2][]([d]f32, f32, f32)
3     let (_, d, v, flag) =
4     let (left, right) =
5         partition (\e ->
6             let (xi, gi, hi) = e
7             in xi[d] < v || (f32.isnan xi[d] && flag)
8             (zip3 X G H)
9     in [left, right]
```

Figure 10: Splits each leaf based on the split value

line 1: The function takes the split information for an leaf, the instances in the leaf and their corresponding gradient/hessian value.

lines 4-8: Apply partition to split the leaf into a left and right child. The split is given by the split information found when searching for the optimal split. This sends instances with a feature value less than the split value to left child. `f32.isnan` is used to check for NaN value and let the boolean flag determine to correct child for instances with a NaN value.

line 9: The function returns a 2D irregular array for the entries to the left and to right child.

Figure 11 shows the pseudo-code used to search for the best split within each leaf. All input arrays are fully regular and every parallel operation contains regular nested parallelism. The regular nested parallelism is efficiently optimized by the Futhark compiler.

```
1  let search_best_split [m][d] (X: [m][d]f32) (G: [m]f32) (H: [m]f32)
2      : (f32, i64, f32, bool) =
3      let dim_splits =
4          map (\feature_vals ->
5              let (missing, rest) = partition (\x -> f32.isnan x.0)
                     (zip3 feature_vals G H)
6              let (_, missing_G, missing_H) = unzip3 missing
7              let G_na = sum missing_G
8              let H_na = sum missing_H
9              let sorted = radix_sort rest
10             let (sorted_V, sorted_G, sorted_H) = unzip3 rest
11             let GLs = scan (+) 0 sorted_G
12             let HLs = scan (+) 0 sorted_H
13              -- map  calculates equation 20 returns (Quality, flag)
14             let Qs =
15                map2 (\gl hl -> quality gl hl (sum G) (sum H) G_na
                       H_na) GLs HLs
16             let opt_i = arg_max Qs
17             let (best_quality, v, flag) =
18                     (Qs[opt_i].0, sorted_V[opt_i], Qs[opt_i].1)
19             in
20             (best_quality, v, flag)
21         ) transpose X
22     let dim = arg_max dim_splits
23     let (q, v, f) = dim_splits[dim]
24     in
25     (q, dim, v, f) -- best split over all dimensions
```

Figure 11: Pseudo implementation of search_best_split

line 1: The function takes m training instances as input and returns the best split between all m instances over all features. This returns the quality, feature dimension index, split value and missing flag for the optimal split within the leaf.

line 3-22: The `map` operation finds the best split within each feature in parallel. This is a done on fully regular arrays as missing values are represented with a NaN value.

lines 5-8: The partition operation is used to separate instances with a missing value into a separate array. Line 7 and 8 calculate the gradient/hessian sum for all these instances.

lines 9-12: Instances are sorted according to their feature value. Radix_sort are used to sort in parallel. In the implementation the unique values are found, since only consider possible splits between unique values. In this pseudo code we abstract way from the finding of unique values and accumulating the gradient/hessian values for all instances. The code and details regarding this is given in section 6.2.

In one parallel scan operation we can calculate every G_L and H_L sum for every split. The quality of every possible split is now calculated using a `map` operation. The function `quality` returns the quality and optimal direction for missing values in the split by using equation 20. G_R and H_R is found using $G_R = G - G_L - G_{na}$.

lines 17-21: The best split information is found from the best split by using `arg_max` on the quality measures calculated.

lines 23-26: Returns the best split over all features. `arg_max` on the result `dim_splits` gives the dimension index for the optimal split.

6 Implementation of exact gradient boosted decision trees

This section describes the implementation details of the pseudo-code given in in figures 7 and 8.
Section 6.1 explains how the ensemble of decision trees is implemented and how trees are added to the ensemble. The search for best split over all features is done in parallel and explained in detail in 6.2. The implementation of segmented partition is explained in section 6.3.

The irregular map on lines 6-9 in figure 8 is not flattened due to no segmented sorting function. So the `map` operation is converted into a sequential loop over the leaves at the current level. The consequence of the sequential loop is some potential parallelism is not utilized in the implementation.
The introduction of the loop for the code in figure 8 results in moving calculation of leaf weights into the loop. The calculation of leaf weights was done using a irregular map, but is easier to inside the loop once the split for the single leaf is found.

6.1 Decision tree layout

The decision trees are represented using a flat array. Each entry consists of a dimension index, split value, missing direction flag, and an integer i giving the location for the left child. The right child is at $i + 1$. Leaves are represented by setting the index to -1. The flat representation allows minimal memory usage of the tree found in algorithm 2.
The number of nodes/leaves depends on the maximum depth of the tree. Typically users choose a maximum depth between 5-14. To accommodate different sizes of trees found from algorithm 2 a buffer of 10.000 entries is used. This ensures we do not allocate unnecessary number of entries. The buffer is doubled if it becomes full during `findOptTree` in figure 8.

```
1       let new_entries =
2               map2 (\x i -> -- x = (d, v, flag)
3                       let (dim_id, value) = (x.0, x.1)
4                       let child = offset+num_nodes_in_level+i*2
5                       in
6                               (dim_id, value, x.2, child )
7                   ) active_splits (indices act_idxs)
```

Figure 12: Node entries in the decision tree

Figure 12 shows how the node entries are created. This entries correspond to the decision nodes in the tree and are done for every node that is split.
The child indices are then given at offset+num_nodes+2i. Offset is the number of entries written into the tree so far and num_nodes is the number of entries to be written for the current level.
The value i used for the relative ordering between node children. The first leaf split will

have a left child index at offset+num_nodes+$2 \cdot 0$. The index for the left child of the second leaf split is given at offset+num_nodes+$2 \cdot 1$.
The trees found from algorithm 2 for each boosting iteration may differ in size, so for the ensemble a flat representation is used. The sizes of the individual trees are given by the shape array. A pre-allocated number of node entries is used and expanded using the same technique in tree creation. The trees will be saved into ensemble by using the `scatter` operation.
The trees created from algorithm 2 will have the wrong indices for the children. Using the same technique as in figure 12, all decision-nodes have their children index corrected by adding an offset. The offset is the total size of all trees in the ensemble which is the sum of the shape array. Figures 13 show how the prediction is obtained from a decision tree and figure 14 shows the prediction of an ensemble is implemented.

```
1  let predict [d][m] (x: [d]f32) (tree: [m](i64, f32, bool, i64))
2              (start: i64) : f32 =
3    let (_, res, _) =
4      loop (i, value, at_node)=(start, 0, true) while at_node do
5        let (dim, v, missing_flag, child) = tree[i]
6        in
7        if child >= 0 then
8          if x[dim] < v || (f32.isnan x[d] && missing_flag) then
9            (child, value, at_node)
10         else
11           (child+1, value, at_node)
12       else
13         (i, v, false)
14   in
15   res
```

Figure 13: Predict function returning $f_t(\mathbf{x})$

line 1: The function takes a tree of size m and a d-dimensional input instance. The start index is used for ensemble predictions. In that case, the tree is the whole ensemble and the start is the index where the tree starts.

line 4: Start of tree traversal. The loop starts at the root and traverses the left or right subtree depending on the split value. The traversal terminates once a leaf is reached.

line 5: The split information for node i.

lines 7-13: The first if statement checks whether it is a leaf and stops the termination returning the value v which is the final prediction. Line 8 checks whether the left or the right child is visited.
If `x[dim]` is a NaN value then the statement `x[dim] < v` will return false and `f32.isnan` used to ensure the direction for missing values is determined by the boolean flag.

```
1 let predict_all [n][d][l][m] (X: [n][d]f32)
2     (trees: [m](i64,f32,bool,i64)) (shape: [l]i64) (bias: f32)
3      : [n]f32 =
4   let pred_trees = map (\xi ->
5                           map (\i -> predict xi trees i) shape
6                           |> reduce (+) 0.0)
7                       X
8   in
9   map (+bias) pred_trees
```

Figure 14: Segmented prediction

line 1: The function takes four inputs: the instances X, the flat ensemble, a shape array, and a bias as input. The bias input is the initial prediction 0.5 used when training the ensemble.

lines 4-7: The ensemble prediction for each instance is obtained with a `map` operation. For each instance, the prediction of each ensemble member is obtained with the inner `map` operation over the shape array. It calls predict with the offset to the root of each tree. The final prediction is calculated with a `reduce` operation.

line 8: Adds the initial prediction to the ensemble prediction for each instance.

6.2 Finding best split

The `search_best_split` function finds the best split between instances in the leaf. For each feature dimension, every possible split quality is evaluated between every unique value. A split can not happen between two instances with the same value since we use $<$ as the split operator. The gradient/hessian values are accumulated for each unique value using segmented reduce.

```
1 let (sorted_data, sorted_gis, sorted_his) =
2       radix_sort_float_by_key (.0) f32.num_bits f32.get_bit rest
3         |> unzip3
4 let unique_seg_starts = map2 (!=) sorted_data (rotate (-1)
     sorted_data)
5 let l = map i64.bool unique_seg_starts |> i64.sum
6 let unique_gis = segmented_reduce (+) 0f32 unique_seg_starts
     sorted_gis l
7 let unique_his = segmented_reduce (+) 0f32 unique_seg_starts
     sorted_his l
```

Figure 15: Finding unique values and their aggregating gradient/hessian values.

lines 1-3: Sorts the feature values, the gradient values and the hessian values according the feature value. This is done with Radix sort by key. (.0) denotes the key i.e. feature value.

line 4: Finds the start of each segment of unique values. For an sorted array of [1,1,2,2,3] this will return [T,F,T,F,T]. This is used as the flag array for segmented reduction.

line 5: Finds the number of unique values used for segmented reduction.

lines 6-7: Segmented reduce to calculate gradient sum and hessian sum. These sums are used for G_L and H_L with a `scan` operation.

In algorithm 2 the gradient sum for the left child at each value is calculated as:

$$
\begin{aligned}
G_{L,0} &= \sum_{\{i \in I_j \mid \mathbf{for each unique feature value} x_i[d] < v_1\}} g_i \\
G_{L,1} &= G_{L,0} + \sum_{\{i \in I_j \mid v_1 < \mathbf{x}_i[d] < v_2\}} g_i \\
&\vdots \\
G_{L,n} &= G_{L,n-1} + \sum_{\{i \in I_{aj} \mid v_{n-1} < \mathbf{x}_i[d] < v_n\}} g_i
\end{aligned}
$$

Here v_1 represents the value used for the first possible split and G_{L0} is the gradient sum for the left child in the first split.
The gradient sum for second split, is G_{L0} plus the sum of gradient values for training instances satisfying $v_1 < \mathbf{x}_i[d] < v_2$ where v_2 is the split value for the second split.
This sum corresponds exactly to the second entry in `unique_gis` as only instances with the second-lowest feature value satisfy the constitution. A parallel scan will then give the calculation of gradient sums for the left child required algorithm 1. The same argument holds for H_L.
Lines 3-23 in figure 8 returns the best split for each feature. `argmax` finds the optimal index i to split the values. As the `scan` operation is inclusive scan the split value must be between v_i and v_{i+1}. The actual split value is $\frac{v_i + v_{i+1}}{2}$ and the XGBoost [1] uses same calculation of split value

6.3 Segmented partition

This section describes how to flatten the irregular `map` used for splitting leaves in figure 8. The `split_leaf` function partitions of each leaf by the dimension and split value. The distribution of instances over the leaves at the current level are given by a shape array. Consider a the distribution in the flat array: [1,4,2,3,2,5,6,1] and with shape [5,3]. This means the first leaf contains 5 instances and the second one contains 3. The first leaf should be split with the value 2.5 and the second with the value 3.5. This will give the new array: [1,2,2,4,3,1,5,6]. The new shape is now [3,2,1,2].
Figure 16 shows the implementation of lifting the partition so it becomes segmented applying splits to each segment in parallel.

```
1  let partition_lifted_by_vals [n][l][d] 't
2      (conds: [l](i64, t, bool)) (ne: t) (op: t -> t -> bool)
3      (isnan: t -> bool)(shp: [l]i64)
4      (X: [n][d]t) (G: [n]f32) (H: [n]f32)
5      : ([n][d]t, [n]f32, [n]f32, [l]i64) =
6      let flag_arr = mkFlagArray shp 0 1 n
7      let seg_offsets_idx = scan (+) 0 flag_arr |> map (\x -> x-1)
8      let cs = map2 (\xi i ->
9          let (dim, cond_val, flag) = conds[i]
10         in op xi[dim] cond_val || (isnan xi[dim] && flag)
11         ) X seg_offsets_idx
12     let true_ints = map i64.bool cs
13     let false_ints = map (\x -> 1-x) true_ints
14     let bool_flag_arr = map bool.i64 flag_arr
15     let true_offsets = segmented_scan (+) 0 bool_flag_arr true_ints
16     let false_offsets = segmented_scan (+) 0 bool_flag_arr
           false_ints
17     let seg_offsets = scanExc (+) 0 shp
18     let num_true_in_segs = segmented_reduce (+) 0 bool_flag_arr
           true_ints l
19     let true_val_offsets =  map2 (\x i -> x + seg_offsets[i])
20                             true_offsets seg_offsets_idx
21     let false_val_offsets =
22         map2 (\x i -> x + seg_offsets[i] + num_true_in_segs[i])
23            false_offsets seg_offsets_idx
24     let idxs = map3 (\c iT iF -> if c then iT-1 else iF-1)
25                  cs true_val_offsets false_val_offsets
26   in
27   (scatter2D (replicate n (replicate d ne)) idxs X,
28    scatter (replicate n 0f32) idxs G,
29    scatter (replicate n 0f32) idxs H,
30    num_true_in_segs)
```

Figure 16: segmented partition: does the split of instances on X, G and H

lines 1-5: The function does apply segmented partition to X, G and H. The input `conds` is the split dimension, split value and missing direction flag of each split for every leaf. The input operator *op* set to be $<$ in the implementation. The function `isnan` determines whether the feature value is a NaN value. *shp* is the distribution of

instances for the current level. X, G and H is the flat representation of instance, gradient and hessian values.

line 6: Creates the flag array representing the segment start. Here 0 and 1 are used as flag values . This array is later used for calculating segment indices as it can be done using a scan operation. In the running example this would be [1,0,0,0,0,1,0,0].

line 7: Creates the array where each entry correspond to the segment they belong to e.g. [0,0,0,0,0,1,1,1] meaning the first five elements belong to first segment and the last three belongs to the second segment.

lines 8-9: Calculates the boolean array where each entry determines if the instance is sent to the left or the right child. This uses the segment indices for each element to access the correct split value and dimension for the operator. For example running this would be [T,F,T,F,T,F,F,T].

lines 12-14: These arrays are used for calculating the writing indices of instances within each leaf. The boolean flag array is used for the segmented scan and reduce.

line 15: This gives an array of indices. The indices are used when writing the instances which evaluated true within in each node.
For the running example: [1,1,2,2,3,0,0,1].

line 16: The array giving the write indices for instances which evaluated to false in line 8. For the running example: [0,1,1,2,2,1,2,2].

line 17-18: Calculates the segment offsets and the number of true evaluations for each node. For the running example: [0, 5] and [3, 1]

line 19: Calculates the writing indices for all the instances sent to left child for every node. The write index is segment offset given by line 14 + internal write index given by line 12. For the running example: [1,1,2,2,3,5,5,6]

line 21: It is analogous of line 19. Since it is for false evaluations we must also add the number of true evaluations for each leaf. This ensures the instances sent to right child are written after instances in the left child. For the running example: [3,4,4,5,5,7,8,8].

line 24: Depending on the evaluation of the split condition the write index for the instance is chosen from either `true_val_offset` or `false_val_offsets`. This gives the indices used for scatter For the running example: [0,3,1,4,2,6,7,5]

lines 27-30: Writes the distribution of instances for next level. `scatter2D` is used for 2D instances for better performance by ensure coalesced writing. `num_true_in_segs` is number of true evaluations for each node. This is used for calculating the new shape for the distribution. For the running example, `scatter` using the indices above and values [1,4,2,3,2,5,6,1] will return [1,2,2,4,3,1,5,6].

As segmented partition is a permutation of the individual leaf, a gather2D was tried out instead of scatter2D. The idea is gather2D would save the cost of calling replicate and perform a little faster. Scatter2D does benefit from coalesced writing and gather2D

should benefit from coalesced reading. Gather2D did not yield the expected performance increase. The usage of scatter2D gave an overall speed increase of 10% compared to gather2D. The overall running time is how long T boosting iterations takes, so a 10% increase is significant.

7 Searching using histograms

It is expensive and repetitious to check every possible split of training instances when searching for the best split within a leaf. For large datasets, this requires multiple expensive sorting operations for every leaf we check for splits. The steps needed for finding the best split for a dimension within a leaf is simplified to:

- Sort the feature values
- Find all unique feature values
- Aggregate gradient/hessian values for instances with same value
- `scan` to calculate the cumulative sum of G_L and H_L

These steps are required before we can calculate the quality of every possible split of the training instances for a single dimension in a leaf.
This process becomes very expensive on a large number of training instances. This can be optimized using histograms [3].

The idea is to partition the sorted instances into a number of intervals and we only calculate the split quality of splitting between these intervals. This reduces the number of possible splits and seeks to eliminate the sorting operation resulting in a speed gain. This approach does affect the accuracy slightly as the best split is now an approximation. By assigning instances to each interval we can aggregate the gradient/hessian values for every interval using a histogram. The `scan` operation is now used over the histogram entries giving the cumulative sum of G_L and H_L. For the new approach, the following steps are required when searching for the best split within a feature dimension:

- Calculate the feature histogram of instances
- The gradient/hessian value sums are entries in the histogram
- `scan` operation on the histogram to calculate the cumulative sum of G_L and H_L

Now the number of splits considered is limited to the size of the histogram. The expensive process of sorting and finding unique values is removed. The size of each histogram is controlled by the user allowing them control the accuracy. A higher number of bins results in more possible splits and a better approximation to the exact split is found giving higher accuracy.

In order to use this histogram approach a preprocessing step is required. For each feature dimension we need to assign every feature value to an interval. These intervals will correspond to the indices in the histogram.
For each dimension d we need to find a series of intervals (v_l, v_u): $\mathbf{x}_i[d] \in [v_l ... v_u]$. Here v_l is the lower bound of the interval and v_u is the upper bound of the interval. The goal is to find intervals such the training instances are evenly distributed over the intervals.

To find the intervals, the training instances are sorted and partitioned into $\frac{n}{b}$ partitions. Here n is the number of training instances and b is the number of intervals/bins. For

each interval we must find v_l and v_u. Using these boundaries we can assign every feature value to an index. This index corresponds to the histogram bin where the gradient/hessian values of instances should be aggregated. For each dimension, we map every feature value to the bin index for the feature histogram and we can use `Reduce_by_index` to efficiently calculate the histogram for the gradient and hessian sums. The interval boundary values are referred to as bin boundaries in the context of histograms. The process of finding the bin boundaries and how they are used to map feature values to bin indices are described in section 7.1. Section 7.2 gives an overview of the parallel constructs used for implementation the histogram based searching approach. Section 7.3 describes how the histograms for gradient/hessian values are efficiently calculated using `Reduce_by_index`.

7.1 Finding and mapping feature values to bins

The usage of histograms requires feature values to be replaced with the bin index the feature value belongs to. For each feature dimension, a histogram has to be calculated. The goal is to create an equal frequency histogram of the values in each dimension. Figure 17 shows an example of a equal frequency histogram.

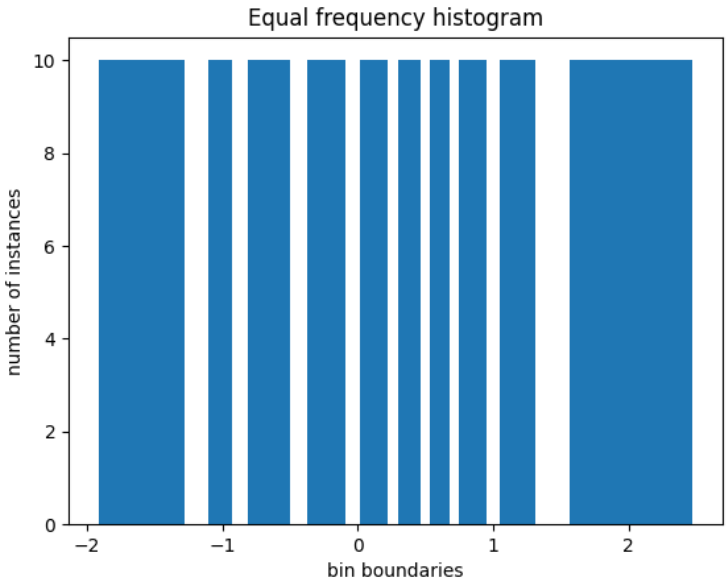


Figure 17: An example of equal frequency histogram

Figure 17 shows how 100 feature values have been assigned to 10 bins. Each bin has exactly 10 entries. The boundaries for each bin are given on the x-axis. The first bin has a lower bound value of -1.9 and an upper bound value of -1.3. To create the histogram we must find the bin boundary for each bin.

A boundary value v_b for a bin b is defined, so instances $\mathbf{x}_i$ ends up in the bin where the following holds: $v_{b-1} < \mathbf{x}_i[d] < v_b$. v_{b-1} is the boundary for the previous bin. For the first bin `f32.lowest` is used as lower bound.
Recall we can only split between unique values so the number of occurrences for each value

is found. From the number of occurrences, we have to find the bin boundaries resulting in a equal frequency histogram. The resulting histogram will be an approximation of the perfect equal frequency show in figure 17.

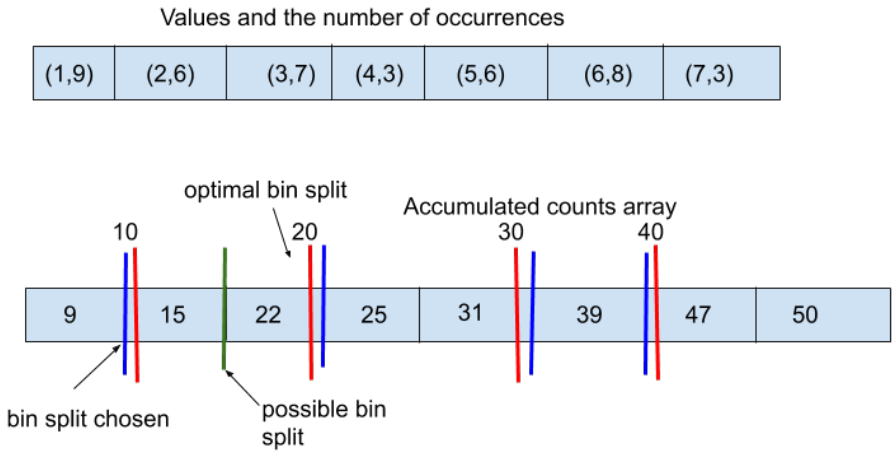


Figure 18: Splits chosen when creating the bins

Figure 18 shows the approach for finding the bin boundaries. The first array shows the values and the number of occurrences in the feature dimension.
The second array is the result of a `scan` operation over the number of counts. The goal is to split this array into b bins where each bin has the same amount of counts. Here we search for the splits giving 10 instances in each bin. The red partitions show the optimal bin sizes and the blue partitions show which bin sizes we choose.
The first bin will have 9 instances. The second bin will have 12 instances etc.

The unique values are found using radix sort as in figure 15. The count of each occurrence is calculated using a segmented reduce. If the number of bins is greater than the number of unique values then the boundaries are given by $\frac{v_i+v_{i+1}}{2}$. Here i is the index of every unique value. The boundary value is also the split value that is used for the decision tree. The median value is chosen so we can expect similar behavior when instances traverses the tree. This also ensures new instances with values between v_i and v_{i+1} is sent distributed evenly between the two bins.
If the number of unique values are greater than the number of bins, we need to find the indices for blue partitions in figure 18. This is done using a linear search over the cumulative sum of occurrences.

The average bin size $\bar{b}$ is 10 in the example. We start the linear search with one bin and then filling it by adding entries of the array from left to right. Every time we have $\bar{b} \cdot n \leq arr[i]$ we have filled up a bin. n is the number of bins found so far and is set to $n = 1$ at the beginning and incremented as we split the counts array. The first bin is full when we arrive at the second entry(arr[1]).
Now we have two possibilities either we split using the green or the blue partition in figure 18. We chose the index i where the value is closest to $\bar{b} \cdot n$. Here arr[0] is closest as

$|10-9| < |15-10|$. n is now incremented to 2 and the search for next bin split continues. The next bin is filled once we reach the third entry. Here we compare the distance between the optimal partition and the ones we can those. We choose to partition at i=2 since $|15-20| > |22-20|$. This continues until n=b. The indices found corresponds to the indices we split the sorted unique values. The bin boundary is $\frac{v_i+v_{i+1}}{2}$. For the last bin multiply the boundary value with two. For figure 18 we split at the indices [0, 2, 4, 5, 6] and this gives the bin boundaries: [1.5, 3.5, 5.5, 6.5, 14]. Figure 19 shows the implementation of `findBounds`. This function finds the bin boundaries for a feature dimension.

```
1  let findBounds [n][m] (vals: [n]f32) (num_bins: i64) (dest: *[m]f32)
2      : [m]f32 = -- input vals are sorted
3    let (_, rest) = partition (f32.isnan) vals
4    let num_vals = length rest
5    -- flag array with true entries gives an unique segment start
6    let unique_start = map2 (!=) rest (rotate (-1) rest)
7    let distinct_values = zip rest unique_start |> filter (\x -> x.1)
8                                |> unzip |> (.0)
9    let num_unique = length distinct_values
10   let bins_left= num_bins-1
11   let boundary_vals = if num_unique <= bins_left then
12         map2 (\v i -> if i == num_unique-1 then
13             v*2
14         else
15             (v+distinct_values[i+1])/2.0
16         ) (iota num_unique) distinct_values
17     else
18       search_bounds distinct_vals bins_left unique_start
19     in
20     scatter dest (indices boundary_vals) boundary_vals
21             with [m-1]=f32.nan
```

Figure 19: Function for finding bin boundaries.

line 1: `findBounds` function takes the sorted feature values, number of bins and array to write the values to as input. It returns the bin boundaries required to create an equal frequency histogram of the feature values.

line 4: Instances with a missing value will be assigned to the last bin. Therefore the last bin of the histogram correspond to the gradient/hessian sum for instances with a missing value for the given dimension. For finding the boundary values they are ignored.

line 6: Finds the flag array for unique segments. This is used in `search_bounds` for calculating the number of occurrences of each unique value.

lines 7-9: Finds the number of unique values and the array of unique values.

lines 11-18: The boundary values for the bins are found. If the number of unique elements is less than the number of bins, the boundary values are found using a `map2` operation

calculating the value. The function `search_bounds` finds the bin boundaries when we have more unique values than bins. This function is shown in figure 20 and uses the methods described in figure 18.

line 20: Writes the found bin boundaries the the array dest. The last value is set to `f32.nan`.

.

```
1  let search_bounds [n][l] (unique_vals: [l]f32) (num_bins: i64)
2      (unique_start: [n]bool) : [num_bins]f32 =
3      let one_arr = replicate n 1
4      let counts = segmented_reduce (+) 0 unique_start one_arr l
5      let mean_bin_size = f32.i64 n / num_bins
6      let sum_counts = scan (+) 0 counts |> map i64.f32
7      let bin_changes = map (\v -> f32.floor(v/mean_bin_size))
           sum_counts
8      let pos_splits = map (\i ->
9          if i == 0 then false
10         else i == l-1 then true
11         else bin_changes[i] != bin_changes[i-1]
12         ) (indices bin_changes)
13     let opt_bin_counts = map f32.bool pos_splits |> scan (+) 0 |>
          map (*mean_bin_size)
14     let splits_to_check =
15         filter (\i -> pos_splits[i]) (indices pos_splits)
16     let best_choices  =
17         map (\(_,i) -> -- i==0 is not present, removed in filter
18             let opt_count = opt_bin_counts[i]
19             let dist_right = f32.abs opt_count - counts[i]
20             let dist_left = f32.abs opt_count - counts[i-1]
21             in if dist_left < dist_right then
22                 i-1 else i
23         ) splits_to_check
24     let split_vals = map (\i ->
25         if i == l-1 then unique_vals[i]*2
26         else (unique_vals[i]+unique_vals[i+1])/2
27         ) best_choices
28     in
29         split_vals
```

Figure 20: finds bin boundary values for the optimal histogram

line 1: The function search_bounds takes the unique values, number of bins and the flag_array of unique segments as input. The function returns the bin boundaries for the optimal equal frequency histogram.

lines 3-4: Calculates the number of occurrences of each unique value. This is done using a segmented reduce on an array of ones using the flag_array.

lines 5-7: The average number of entries in each bin. This is used for determine if we should partition the array as in figure 18. By dividing each cumulative sum count with the average bin size we simulate the process filling bins. Each time the value change in

`bin_changes` we have filled a bin and we need to partition the cumulative sum of counts and use a new bin. For the example in figure 18 this will give [0, 1, 2, 2, 3, 3, 4, 5].

lines 8-12: Each time the value changes in `bin_change` we have the situation described in figure 18. This creates an flag array: [F, T, T, F, T, F, T, T].

line 13: Calculate the optimal accumulated counts for the bins

line 14: Removes all the false entries in `pos_splits`. Here we do not partition the instances and are ignored.

lines 16-25: The `map` operation finds the optimal indices where to split the data. It uses the cumulative sum of counts at the index i and at $i-1$. These two sums are compared against the optimal sum and the best index which is closest is chosen. For the example in figure 18 this gives : [0, 2, 4, 5, 6].

lines 24-27: The optimal split indices are now converted into split values using $\frac{v_i+v_{i+1}}{2}$ or if it the last unique value then it is multiplied with 2. This gives the resulting bin boundaries: [1.5, 3.5, 5.5, 6.5, 14].

Figure 21 combines the functions for finding bin boundaries and maps features values to bin indices. 21.

```
1 let binMap [n] (vals: [n]f32) (num_bins: i64)
2             : ([n]u16, [num_bins]f32) =
3   let dest = replicate num_bins f32.highest
4   let s_vals = radix_sort_float f32.num_bits f32.get_bit vals
5   let new_bounds = findBounds s_vals num_bins dest
6   let num_bins = u16.i64 num_bins
7   let mapped = map (\v -> value_to_bin v new_bounds num_bounds) vals
8   in
9   (mapped, new_bounds)
```

Figure 21: The function which finds bin and maps the feature values

line 1: The function `binMap` takes the feature values and number of bins as input. It returns the array of mapped features values with type u16 and the bin boundary for each bin.

line 3: allocate the array for the bin boundaries. This is done as `findBins` writes the boundaries found into this array.

line 4: Sorts the feature values, this is done for finding the unique values in `findBins`

line 5: The new bounds are found using `findBins`. This function finds the bin boundaries using the method as described earlier in the section.

line 7: The map uses `value_to_bin` to find the bin indices for each feature value. `value_to_bin` uses the bin boundaries to find which bin the feature value belongs to. Note the `map` operation is used on the original input of feature value to keep the ordering.

All the functions shown at figures 19, 20 and 21 all uses regular parallelism and no flattening is needed. The transformation of the whole dataset is then performed using a `map` operation over each dimension. For large datasets, this operation over each feature dimension calling `binMap` will result in high memory usage, since additional d sorted arrays of size n are created. Therefore it is implemented using a sequential loop over each dimension.The `binMap` function fully saturates the GPU for large datasets.

The input dataset is a [n][d]f32 matrix and is now converted into [n][d]u16 as a result of mapping to bin indices. The conversion does reduce the memory usage but the usages of u16 enforce an upper bound on the number of bins. An optimization would be to make type parametric, such it depends on the number of bins. If the user only wants to use 255 bins the dataset should be of type u8.

7.2 Histogram implementation overview

The introduction of histograms does change the futhark translation of algorithm 2. It is adjusted to creates the histogram for the feature values and searches them for the best split. In the implementation the processing step of mapping values to bins is added. This is done before starting the gradient boosting using `binMap` on every dimension. Figure 22 shows the futhark translation of algorithm 2 where histograms is used for searching.

```
1  let findOptTree [n][d][b] (X: [n][d]u16) (G: [n]f32) (H: [n]f32)
2      (bin_bounds: [d][b]f32)
3      -- all instances are assigned to root.
4      let(X, G, H) = ([X], [G], [H])
5      loop (X, G, H, l, N, L) for l <= D and !(null X) do
6          let histograms = create_histograms X G H shape
7          let splits = map (\dim_hists -> --[d][b](f32,f32)
8            let (q, d, v, flag) = best_hist_split dim_hists bin_bounds
9            in (q, d, v, flag)
10           ) histograms
11         -- irregular partition : split_info=(q,d,v,flag)
12         let (active_data, terminal_data) =
13             partition (\(split_info, _, _, _) -> split_info.0 > 0)
14                 (zip4 splits X G H)
15         let (_, _, terminal_G, terminal_H) = unzip4 terminal data
16         -- irregular map calculating prediction weights.
17         let L = L ++ map2 weight terminal_G terminal_H
18         let (act_splits, act_X, act_G, act_H) = unzip4 active_data
19         let N = N ++ act_splits
20         -- irregular partition, returns the irregular distribution
              of points in the new level
21         let X, G, H = map4 split_leaf act_splits act_X act_G act_H
22                           |> flatten |> unzip3
23         in
24           (X, G, H, l+1, N, L)
25   in
26   (N, L)
```

Figure 22: Pseudo code for histogram based version of algorithm 2

.

lines 1-2: The inputs are similar to `findOptTree` in figure 8, but the `findOptTree` now uses training instances where the values have bin mapped to bin indices. The bin indices are $0 \leq i \leq b$. For each bin, the corresponding boundary value is provided in the bin_bounds input. It has shape [d][b] as every dimension have a histogram of size b. The boundary value is the split value written into the decision nodes.

lines 5-23: The loop over levels in the tree built. Here the X G H is updated as splits are introduced into the leaves.

lines 6: The function `create_histograms` creates the histograms for each dimension for every segment. This function uses the irregular arrays X, G, Y.
This function is flattened to remove the irregular parallelism by creating a segmented histogram for each dimension. A segmented histogram contains the all the

histograms for each segment in a single one. The size of the segmented histogram is $s \cdot b$ for each dimension where s is the number of segments and b is the number of bins. The first b entries is the histogram for the first segment. The next b is for the second segment. This is segmented histograms is made into a [s][b] matrix. This gives a total of [d][s][b] histogram which now a fully regular arrays and the search for split is now nested regular parallelism which is flattened by the compiler. Section 7.3 explains the implementation of `create_histograms`

lines 7-10: The `map` operation over the irregular array containing the histograms. Each histogram entry is a [d][b](f32,f32) which gives the gradient/hessian sums for each bin in every dimension and `best_hist_split` will operate on fully regular arrays. `best_hist_split` now searches for the best split between each histogram bin over all dimensions. The search is simplified as the gradient/hessian sums for splits is given by the histogram. A scan over the histograms obtains the cumulative sum of G_L and H_L and the best splits are found similar to `search_best_split`. The details for function `best_hist_split` is shown in figure 23.

lines 12-20: Same lines of code as in 8. We partition the instances into an active and a terminal part. The terminal part is used for calculation of prediction values. The calculation of leaves predictions are now done by flattening the `map` on line 17. This is done by lifting the function in figure 9 into using segmented reduction and calculating the predictions for every terminal leaf at the current level.

line 21: The irregular `map` operation is flattened into a segmented partition for splitting the leaves at the current level. The implementation uses the segmented partition described in section 6.3.

line 23: Calculates the new shape array based on the splits applied to the active leaves.

```
1  let best_hist_split [d][b] (G_hists: [d][b]f32) (H_hists: [d][b]f32)
       (bounds: [d][b]f32) : (f32, i64, f32, bool) =
2      let best_split_dim =
3          map3 (\g_hist h_hist bin_bounds ->
4          let Gls = scan (+) 0 g_hist
5          let Hls = scan (+) 0 h_hist
6          let G_na = last Gls
7          let H_na = last Hls
8          let Qs = map2 (\gl hl -> gain gl hl (sum Gls) (sum Hls) G_na
                H_na)
9              GLs[:b-1] HLs[:b-1]
10         let opt_i = arg_max Qs
11         let(best_q , v, flag) =
12             (Qs[opt_i].0, bin_bounds[opt_i], Qs[opt_i].1)
13         in
14         (best_q, v, flag)
15         ) G_hists H_hists bounds
16     let d = arg_max best_split_dim
17     let (q, v, flag) = best_split_dim[d] |> unzip3
18     in
19     (q, d, v, flag)
```

Figure 23: Pseudo code for finding the best split using histograms

line 1: The function takes the two histograms as input: the gradient sums and the hessian sums. The histograms are given in a 2D matrix with shape [d][b]. The boundary values for each dimension are used to get the split value.

lines 2-15: Finds the best split within each dimension. This is done with a `map3` operation on G_hists, H_hists and bounds. This searches each dimension for the best split between the b possible ones.

lines 4-5: Calculates the sums G_L and H_L with a `scan` operation over each histogram. The properties explained in section 6.2 also holds when using histograms.

lines 6-7: The missing values are aggregated in the last bin of the histogram. This means the sums G_{na} and H_{na} are given by the last histogram entry. The layout of histograms are described in section 7.1.

line 8: Calculates the split quality of splitting between each bin. This is done using equation 20 and the function returns the quality and best direction for missing values for every split possible.

line 10-12: Finds the optimal split value. The value is given by the boundary value for bin `opt_i`.

line 16: Finds the optimal dimension to split the leaf.

7.3 Histogram creation

This section describes how the segmented histogram for each dimension is created. The idea is so use `Reduce_by_index` to create the histogram for a leaves. This will mean the histogram for a single dimension has $s \cdot b$ entries. Here s is the number of leaves and b is the number of bins. By using a `map` over each feature dimension where all the mapped feature values are processed at once. The use of `Reduce_by_index` reduces the n feature values to $s \cdot b$ histogram entries. Figure 24 shows how the segmented histogram is calculated for every dimension.

```
1  let create_histograms [n][d][s] (X: [n][d]u16) (G: [n]f32) (H: [n]
       f32)
2           (flag_arr: [n]i64) (num_segs: i64) (num_bins: i64)
3           : ([d][num_segs][num_bins]f32, [d][num_segs][num_bins]f32) =
4       let seg_offsets = scan (+) 0 flag_arr |> map (\x -> x-1)
5                         |> map (\x -> x *num_bins)
6       in
7       map (\dim_bins ->
8           let g_hist_entry =  replicate (num_segs*num_bins) 0.0
9           let h_hist_entry =  replicate (num_segs*num_bins) 0.0
10          let idxs = map i64.u16 dim_bins |> map2 (+) seg_offsets
11          let g_seg_hist = reduce_by_index g_hist_entry (+) 0.0 idxs G
12          let h_seg_hist = reduce_by_index h_hist_entry (+) 0.0 idxs H
13          in  ( unflatten num_segs num_bins g_seg_hist
14              , unflatten num_segs num_bins h_seg_hist)
15      ) (transpose X) |> unzip
```

Figure 24: Fully regular histogram calculation given data points, gradient and hessian values

lines 1-3: The function takes the three data arrays as inputs: X, G and H. The flag_arr gives the distribution of points over the segments. num_segs is the number of segments at the current level. num_bins is the number of bins in each histogram.
Every histogram is regular and the histograms are returned in a 3D matrix with shape [d][num_segs][num_bins]. This allows for searching for splits using nested regular `map` operations.

lines 4-5: The segment index for every instance is calculated using the scan over the flag array. Every segment index is then multiplied with b giving the indices of where each segment in the histogram starts. For example a segmented histogram with 3 bins they segment offsets are [0,0,0,0 3,3,3, 6,6,6,6].

lines 7-15 Calculates the segmented histogram for each dimension in parallel. `Reduce_by_index` takes the bin indices + segment offsets as aggregation indices and uses the gradient values to create the segmented histogram of gradient values for a single dimension. The `map` operation operates on the feature bins. This operation returns [d][num_segs][num_bins]

lines 8-9: Allocates the segmented histogram entry for the dimension. Each segmented histogram has the flat length of num_segs · num_bins.

line 10: Adds the segments offsets to aggregation indices for every training instance.

lines 11-12: Calculates the segmented histograms for the dimension. Each entry in `idxs` correspond to the bin where the gradient/hessian value for the instance should be aggregated. As we want to gradient/hessian sum we use (+) as our aggregation function.

lines 13-14: The segment histograms are transformed into two a 2D matrices with type: '[num_segs][num_bins]f32.

8 Experimental evaluation

In this section the two implementations are tested on a common benchmarking dataset used for gradient boosted decision trees. The Higgs dataset [1] is for classifying a signal process which produces Higgs bosons and a background process which does not.
The dataset is created through Monte carlo simulations. It contains 11 million instances each with 28 features. This dataset is commonly used for benchmarking a GDBT framework. [1, 3, 8]
The performance and accuracy of the two implementations is compared against release 1.21 of XGBoost [2]. The GPU hardware used is a Nvidia geforce RTX 2080 with 12 GB memory running Turing architecture with CUDA 11.0.

XGBoost is typically used in python however python introduces a overhead when doing function timings. Instead the XGBoost c-api[3] is used for measuring performance as it has minimal overhead.

8.1 Accuracy

The loss function used is logistic binary classification. For binary classification the predictions are compared to a threshold typically 0.5. If prediction is greater than 0.5 is belongs to class 1 otherwise zero Since the Higgs dataset has an imbalanced class distribution AUC(area under the ROC curve) is used an evaluation metric. The ROC curve shows the classification performance at every possible threshold for the predictions. AUC provides an aggregate measure of performance across all possible classification thresholds. If the model has 100% correct classifications an AUC score of 1 is achieved and 100% misclassifications gives 0.

500 boosting iterations are performed and the maximum depth of single tree is set to 6. λ is set to 0.5 giving it a slight regularization penalty. For gradient step, η 0.1 is used as higher values causes the model to over-fit. The number of bins is set to 256.

[1] `https://archive.ics.uci.edu/ml/datasets/HIGGS`
[2] `https://github.com/dmlc/xgboost/releases/tag/v1.2.1`
[3] `https://xgboost.readthedocs.io/en/latest/dev/c__api_8h.html`

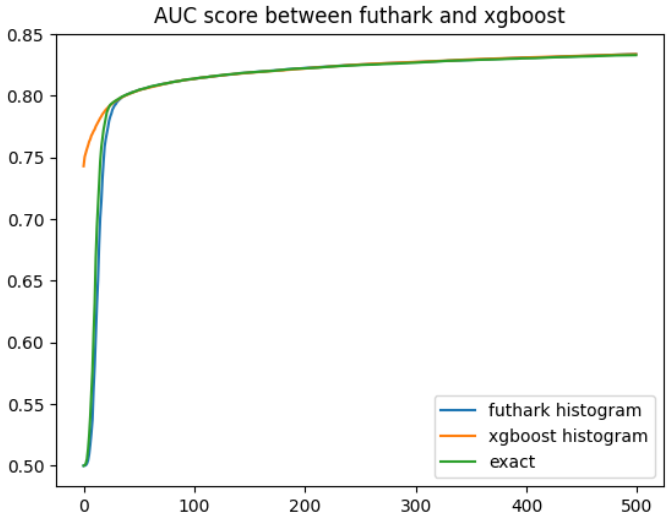


Figure 25: Auc score between XGBoost and futhboost, higher is better

Figure 25 shows the convergence of the AUC score for each boosting iteration. The futhark histogram and exact implementation both starts with a score of 0.5, since it is the base prediction and quickly convergences so it achieves similar performance as the xgboost histogram.
As the accuracy is identical between the exact and histogram implementations the histogram implementation have become the go to method for creating a GDBT ensemble.

8.2 Training time

The timings of XGBoost in C by is done by starting a timer before start of training. The timer is stopped as the training of the ensemble is done. Timings are done in μs but converted to seconds as the full model training takes several seconds.

The execution time for the Futhark implementation is measured using the `futhark bench` command. This command measures the actual processing time on the GPU by excluding the transfer time of data from RAM to GPU memory.
5 million data points are used and we do 100 boosting iterations to create our ensemble. The futhark implementation took on average 17.4s while the XGBoost library took on average 9.4s. Both implementations used 256 bins. The difference in execution time is a factor two for XGBoost and futhark when comparing over different numbers of training instances.

By profiling the execution we observe the implementation spends most of the time creating histograms. This is expected as the histograms were introduced to eliminate the need for sorting and finding unique values. 60% of the total time is spent on creating histograms. Smaller trees cause the segmented histograms to fit the local memory by `reduce_by_index` and no histogram aggregation is done in global memory. For trees with a larger depth, the segmented histogram is not guaranteed to fit local memory, and

access to global memory is required impacting the total execution time.
The cost of mapping feature values to bins is around 0.8s which is low compared to overall training time. It is unknown if XGBoost does the feature value mapping when the training data matrix is created with `XGDMatrixCreateFromFile` or if they are mapped just as the training of the ensemble begins.

If the user values better accuracy a higher number of bins is required. This will give a better approximation of the best exact split. The overall training time increases with the number of bins as larger histograms are needed and more possible splits are checked.

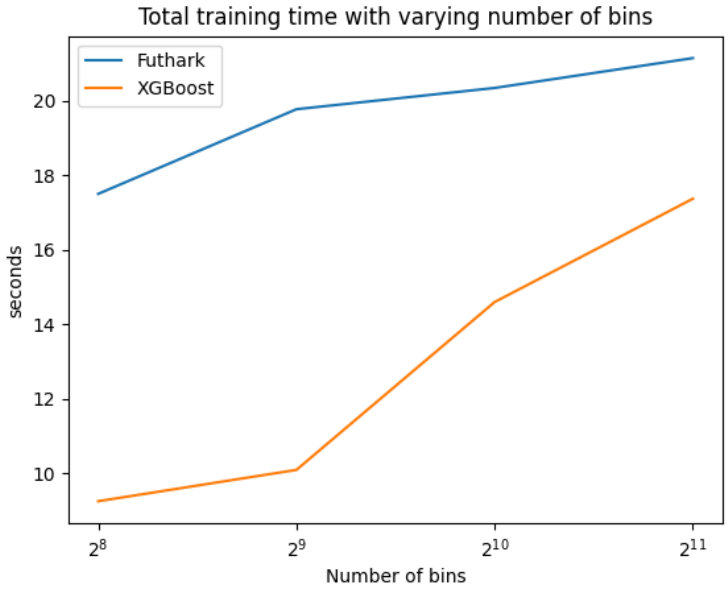


Figure 26: Training time different number of bins, notice y-axis starts at 9 seconds

Figure 26 shows the overall training time over 100 boosting iterations with a varying number of bins. The training data has 5 million training instances. The number of bins increases the time spent creating histograms. For 2048 bins 66% of time is spent creating histograms. The performance gap between XGBoost and futhark implementation reduces significantly as the number of bins increases.

The exact implementation were also benchmarked using 5 million data points. As there are many more possible splits than the histogram-based to consider we report the execution time per boosting iteration. The time per boosting iteration is an average over 100 boosting iterations.
The GPU version of the exact implementation uses 8.4 seconds per boosting iteration.
The multi-core CPU version of XGBoost framework uses 3.3 seconds per boosting iteration. The timing is done using a Intel Xeon E5-2650 CPU with 32 cores and is an average over 100 iterations.
The performance difference between the GPU and CPUis a consequence of not flattening the irregular map over leaves at line 6 in figure 8. Close to the maximum depth, the leaves left is likely to contain a fraction of the original number of training instances. This

fraction results in the implementation does not fully saturate the GPU and will cause a decrease in performance. The XGBoost framework did support an implementation of the GPU exact version [2] but was later removed as the multi-core CPUs did outperform the GPU version on large datasets [3]. Therefore most GDBT frameworks focuses on the histogram based algorithm for benchmarking and practical use.

8.3 OpenCL backend

Currently all the performance benchmarks are done with CUDA backend. This is because the atomic_add operations which considerably speeds up the histogram creation in CUDA compared to OpenCL.

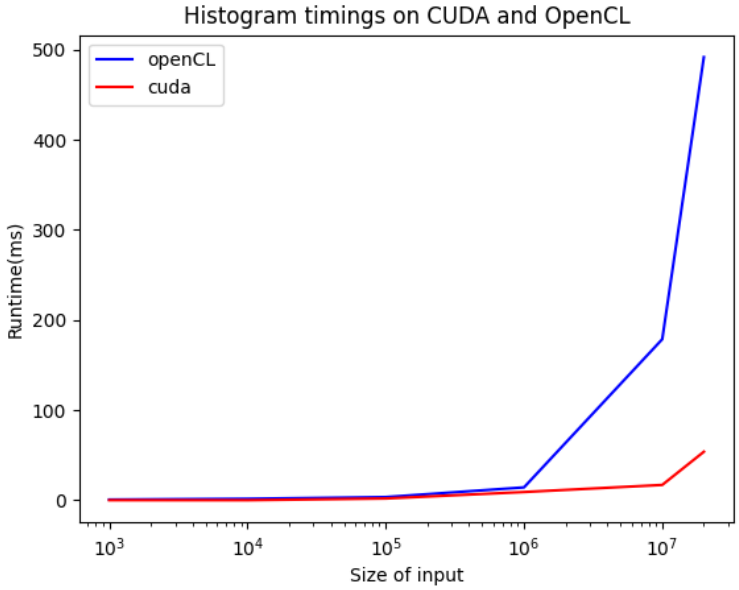


Figure 27: Timing histogram creation with different backend.

Figure 27 show the time used for calculating histograms with a varying number of training instances and segments. The timings are done using the CUDA and OpenCL backend. When the number of instances is above 1 million the CUDA backend starts to outperform the OpenCL. Here the missing atomic_add opertion causes additional global memory accesses for hte OpenCL backend resulting in a significant slowdown.

When replicating the timings from section 8.2, the implementation using the OpenCL backend is significantly slower. All the difference happens when the histograms are created. The rest of the implementation has timings similar to the CUDA backend but the histogram creation takes 0.95% of the total running time. Upgrades to the OpenCL backend will translate directly into a performance upgrade of the implementation.

9 Conclusion and future work

We have presented the XGBoost algorithm for creating gradient boosted decision trees and shown the mathematical basis for the decision tree created is the gradient step.
A algorithmic framework for finding the decision tree is shown and how it is translated into Futhark using parallel building blocks. As Futhark does not support irregular parallelism the flattening is used to fully utilize all the parallelism is available. The serach for the best split is further optimized using histograms and a parallel implementation is provided.
We benchmarked our implementations against the XGBoost framework on the Higgs dataset. We demonstrate the two implementations achieve similar accuracy and loss convergence as the XGBoost framework. The training speed was tested on 5 million training instances and the histogram-based implementation is found to x2 times slower than XGBoost implementation. We also have shown as the number of bins increases the gap between the two implementation decreases.

Future work:

- The radix sort should be lifted so it can sort segments in parallel. This will allow the exact implementation to fully utilize all parallelism available in `findOptTree`. This will result in better performance as the implementation will fully saturate the GPU for large trees.
- As mentioned in 6.1 the type of the mapped dataset should depend on number of bins chosen by the user. This will require translating the histogram-based implementation to be type polymorphic. This will require the use of module types.
- Further various techniques for creating gradient boosted trees can be explored: Sampling data points based on the gradient values as it is likely a split happens between two values with large gradient values.

  The growth strategy of trees is to grow level-wise. This means we search all leaves in the current level for the best split before moving onto next. Another strategy is to use leaf-wise growth. Here the leaf with the best split quality is chosen and split. Then the subtree is grown until no splits are left before growing the other subtree at the first split.

References

[1] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016.

[2] Rory Mitchell and Eibe Frank. Accelerating the XGBoost algorithm using GPU computing. *PeerJ Computer Science*, 3:e127, July 2017.

[3] Huan Zhang, Si Si, and Cho-Jui Hsieh. Gpu-acceleration for large-scale tree boosting, 2017.

[4] Troels Henriksen. *Design and Implementation of the Futhark Programming Language*. PhD thesis, 2017.

[5] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 53–67, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Guy E. Blelloch. Nesl: A nested data-parallel language (version 2.6). Technical report, USA, 1993.

[7] Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. Compiling generalized histograms for gpu. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[8] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 3146–3154. Curran Associates, Inc., 2017.